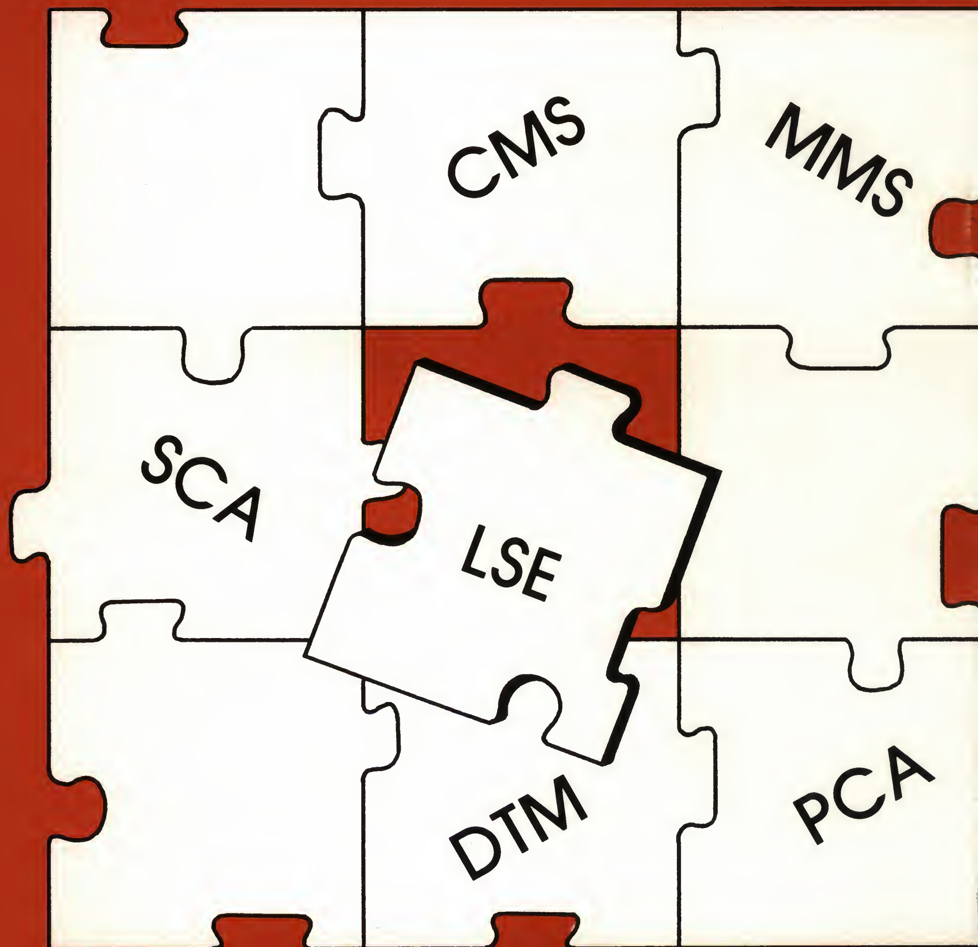
A Methodology for Software Development Using VMS Tools



digital

software

A Methodology for Software Development Using VMS Tools

Order Number: AA–HB16C–TE

April 1988

This manual describes how to use VAX Software Engineering Tools (VAXset) with other VMS facilities to create an effective software development environment.

Revision/Update Information: This manual supersedes *A Methodology for Software Development Using VMS Tools* (Order number AA–HB16B–TE).

Operating System and Version: VMS Version 4.6 or higher

Software Version: VAXset Version 6.0

digital equipment corporation
maynard, massachusetts

First Printing, December 1985
First Revision, April 1987
Second Revision, April 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1985, 1987, 1988 by Digital Equipment Corporation

All Rights Reserved.

The postpaid Reader's Comments forms at the end of this document request the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC
DEC/CMS
DEC/MMS
DECnet
DECsystem-10
DECSYSTEM-20
DECUS
DECwriter
DIBOL
EduSystem
IAS
MASSBUS
PDP
PDT
RSTS
RSX
UNIBUS
VAX
VAXcluster
VMS
VT

digital ™

ZK4708

Contents

PREFACE ix

CHAPTER 1 THE SOFTWARE DEVELOPMENT LIFE CYCLE 1–1

1.1 OVERVIEW OF THE DEVELOPMENT LIFE CYCLE 1–1

1.2 PROBLEMS OF SOFTWARE DEVELOPMENT 1–4

1.3 ADVANTAGES OF A FORMAL METHODOLOGY 1–7

1.4 TYPICAL COSTS OF DEVELOPMENT 1–8

1.5 USING TOOLS AT EACH STEP IN THE SOFTWARE LIFE CYCLE 1–9

CHAPTER 2 VAXSET TOOLS 2–1

2.1 VAXSET AS A PRODUCTIVITY TOOL 2–2
2.1.1 Additional VAXset Features 2–4
2.1.2 VAX DEC/Code Management System 2–4
2.1.3 VAX Language-Sensitive Editor 2–6
2.1.4 VAX Source Code Analyzer 2–8
2.1.5 VAX DEC/Module Management System 2–10
2.1.6 VAX DEC/Test Manager 2–11
2.1.7 VAX Performance and Coverage Analyzer 2–13

2.2 ADDITIONAL VMS TOOLS AND UTILITIES 2–15

CHAPTER 3 SETTING UP A SOFTWARE PROJECT 3–1

3.1 INITIAL PROCEDURES FOR A PROJECT LEADER 3–1
- 3.1.1 User Accounts and Directory Protection 3–2
- 3.1.2 Project Directory Structure and CMS Libraries 3–3
- 3.1.3 Build Directories 3–6
  - 3.1.3.1 MMS Description File • 3–8
  - 3.1.3.2 Reference Copy Area • 3–9
- 3.1.4 SCA Libraries 3–11
  - 3.1.4.1 Project-Wide Development • 3–12
  - 3.1.4.2 Incremental Development • 3–12
- 3.1.5 DTM Libraries 3–15
- 3.1.6 Project Standards 3–18
- 3.1.7 LSE Templates 3–21
- 3.1.8 LSE Logical Names 3–22
- 3.1.9 Communication Management and Report Mechanisms 3–23
  - 3.1.9.1 Communication Within Your Project • 3–24
  - 3.1.9.2 Communication Outside Your Project • 3–24
  - 3.1.9.3 Documentation Reviews • 3–26

3.2 ONGOING PROCEDURES FOR A PROJECT LEADER 3–26
- 3.2.1 Extending the Library Structure 3–27
- 3.2.2 Establishing Project and Personal Build Procedures 3–27
  - 3.2.2.1 Personal Build Procedures • 3–27
  - 3.2.2.2 Project Build Procedures • 3–28
  - 3.2.2.3 Access to SCA Libraries • 3–29
  - 3.2.2.4 Access to Source Files • 3–30
- 3.2.3 Setting Up Tests 3–32
- 3.2.4 Analyzing Performance and Coverage 3–35
- 3.2.5 Monitoring Project Progress 3–35
- 3.2.6 Using the VAX Software Project Manager 3–36
- 3.2.7 Tracking Reports During Field Testing 3–38
- 3.2.8 Final Steps in a Project 3–41

CHAPTER 4 USING TOOLS ON A SOFTWARE PROJECT 4–1

4.1 SETTING UP DIRECTORIES 4–1
4.1.1 Directory Structure 4–2
4.1.2 Creating Directories and Libraries 4–3
4.1.2.1 Restricting Access with ACLs • 4–5
4.1.2.2 Access Control for Large Projects • 4–6

4.2 MAINTAINING CMS SOURCE LIBRARIES 4–12
4.2.1 Storing Files in a CMS Library 4–12
4.2.2 Modifying Elements 4–13
4.2.3 Concurrent Access 4–14
4.2.4 Creating Classes 4–16
4.2.5 Retrieving Class Contents and Preparing a Build 4–17

4.3 SETTING DEFAULTS WITH A LOGIN.COM FILE 4–19
4.3.1 Environment File for LSE 4–20
4.3.2 Command Files 4–23

4.4 DEBUGGING SOURCE CODE 4–25

4.5 EDITING A SOURCE FILE WITH LSE 4–34

4.6 COMPILING AND LINKING A MODIFIED FILE 4–38

4.7 SETTING UP THE TEST SYSTEM 4–38
4.7.1 Setting Up a Noninteractive Test 4–39
4.7.2 Setting Up an Interactive Test 4–45
4.7.3 Verifying Your Test System 4–50
4.7.4 Changing Input for a Test 4–50

4.8 BUILDING THE SYSTEM 4–51

4.9 USING THE DTM REVIEW SUBSYSTEM 4–62
4.9.1 Selecting the Result File from DTM Review 4–63
4.9.2 Using the SHOW/DIFFERENCES Command 4–64
4.9.3 Invoking PCA from the REVIEW Subsystem 4–66

4.9.4 Invoking LSE from PCA 4–67
4.9.5 Using the Analyzer to Perform a Call Tree Analysis 4–69

4.10 MAINTAINING THE APPLICATION 4–72
4.10.1 CMS Provides History 4–72
4.10.2 SCA Provides Structural Information 4–73
4.10.3 MMS Simplifies Maintenance 4–74
4.10.4 CMS Used with MMS for Maintenance 4–75

INDEX

EXAMPLES

4–1 Sample LOGIN.COM File 4–24
4–2 Project Logical Definitions File 4–25
4–3 Sample Collection Prologue File — COLLECTION_PROLOGUE.COM 4–40
4–4 Sample Collection Epilogue File — COLLECTION_EPILOGUE.COM 4–41
4–5 TRANSLIT Test Template File — TRANSLIT_TEST.COM 4–43
4–6 A Build Procedure Using an MMS Description File 4–52
4–7 Flags and Macros Section of the Description File 4–55
4–8 Rules Section of the Description File 4–59
4–9 Targets Section of the Description File 4–60
4–10 Sample Annotated Source Code Listing 4–68
4–11 Static Call Tree Analysis 4–70
4–12 Dynamic Call Tree Analysis 4–71
4–13 Dependencies in an MMS Description File 4–75

FIGURES

1–1 Model of the Software Development Cycle — 1–2
1–2 Costs over the Software Development Life Cycle — 1–8
1–3 VMS Tools Used in the Software Development Life Cycle — 1–10
3–1 Initial Storage Areas for a Typical Project — 3–4
3–2 Build Directory Hierarchy — 3–7
3–3 Filling an SCA Library — 3–12
3–4 Physical versus Virtual SCA Libraries — 3–14
3–5 Directory Structure Showing DTM Libraries — 3–16
3–6 Initial Storage Areas for a Typical Project — 3–22
3–7 Working SCA Libraries for Developers — 3–29
3–8 Source Code Management — 3–31
3–9 Grouping Test Descriptions — 3–34
4–1 CMS Library with Two Generations — 4–13
4–2 Variants in a CMS Library — 4–15
4–3 Merging with CMS Libraries — 4–16
4–4 Classes in a CMS Library — 4–17
4–5 Description File as Part of a CMS Class — 4–19
4–6 Extracting a Token — 4–21
4–7 Creating an Environment File — 4–22
4–8 Steps to Implement Code — 4–26
4–9 Problem Source Code from the Debugger — 4–28
4–10 The EXAMINE Command in the Debugger — 4–29
4–11 Exiting the Debugger to LSE — 4–30
4–12 GOTO DECLARATION in SCA — 4–32
4–13 Navigating Based on FIND Results — 4–33
4–14 Using a Token with LSE — 4–35
4–15 Expanding an LSE Token — 4–36
4–16 Completing Changes to Code — 4–37
4–17 Sample SHOW/DIFFERENCES Output — Screen 0 — 4–64
4–18 Sample SHOW/DIFFERENCES Output — Screen 13 — 4–65

TABLES

3–1 Examples of Logical Names for Directories ____ 3–23
3–2 Keywords for a Sample QAR System ____ 3–40
4–1 Project's Directories ____ 4–2
4–2 MMS Description File Command Options ____ 4–62

Preface

This manual explains how to use VAX Software Engineering Tools (VAXset) with other VMS facilities to create or customize an effective software development environment. In particular, this manual highlights the integration features of the six tools that make up the VAXset package.

Intended Audience

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

Manual Structure

This manual consists of four chapters. The topics covered in each chapter are as follows:

- Chapter 1, The Software Development Life Cycle, describes the software development life cycle, associated problems, and the role of VAXset tools and VMS facilities during the life cycle.
- Chapter 2, VAXset Tools, provides an overview of VAXset and other VMS tools that support the VMS software development environment.
- Chapter 3, Setting Up a Software Project, describes how to integrate the VAXset tools and facilities into your software development environment to help manage development tasks.
- Chapter 4, Using Tools on a Software Project, describes how the VAXset tools can be used in a software development project.

Associated Documents

The following publications provide additional information about the VMS operating system and the VAXset tools discussed in this manual:

- The VAX/VMS documentation set
- The *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*
- The *Guide to VAX DEC/Code Management System*
- The *Guide to VAX DEC/Module Management System*
- The *Guide to VAX DEC/Test Manager*
- The *Guide to VAX Performance and Coverage Analyzer*
- The *Introduction to Application Development*
- The *Introduction to Database Development*

Conventions

Convention	Meaning
LSE> RESERVE	Interactive examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
[expression]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification.)
element	Italicized lettering indicates a term that is defined in the text.
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/E, CTRL/W.

Chapter 1

The Software Development Life Cycle

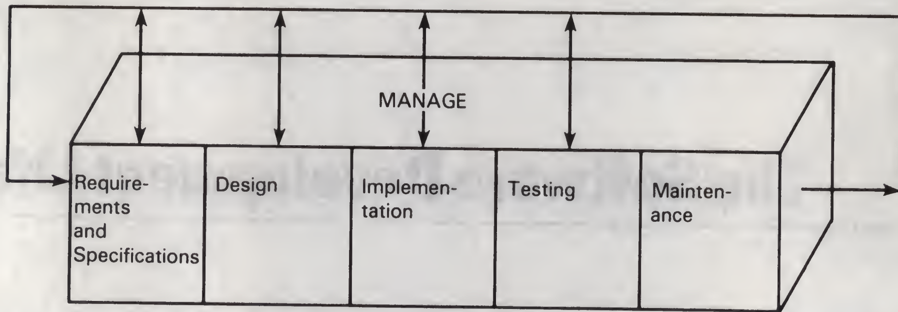
This chapter provides an overview of the software development life cycle and the problems typically encountered in the process of developing software. This chapter then describes the advantages to be gained by adopting a formal development methodology to deal with these problems, showing the VAXset and VMS tools that can be used at each stage in the software life cycle to automate many procedures and mitigate the problems.

1.1 Overview of the Development Life Cycle

Figure 1–1 shows a model of the software development process. The arrows demonstrate the iterative process involved in developing software, showing that the process involves repeated loops as previous work becomes obsolete and recycled.

The following sections describe the different phases of the development life cycle as generally practiced at DIGITAL and as summarized in Figure 1–1.

Figure 1–1: Model of the Software Development Cycle



Phase 1: Requirements and Specifications

This first phase in the software development life cycle defines the project, and it takes place 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 may 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 project's requirements depends heavily on feedback from outside the team — from customers, from any internal users, and possibly from marketing. Other groups within an organization also provide information — a central quality group, and engineering departments. When the team has completed the requirements step, they have defined project goals: essentially, what is to be built, but not 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 also 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 and, optionally, help develop usability requirements, if the team chooses to subject 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, has produced a preliminary functional specification, and must decide whether or not to go forward with the project.

Phase 2: Design

During the design phase, the project team determines more precisely 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 includes resource estimates as part of the design; these estimates approximate the time needed to implement a particular piece of functionality. As project designs evolve, the team can modify the design specification to include new developments. This design document makes it possible to keep the design plans in one location, 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 methodology 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 and distribute it.

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 fine tunes the application's code to optimize its performance. To aid in this refinement stage, the application may 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 general customers. In the final stages of this phase, the source code and documentation are frozen, and, if necessary, 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 may 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 Problems of Software Development

Projects get more complex every day. This can make a project team scramble just to keep up with the job. Without a good methodology and a strong support environment, a team can face problems like the following:

- Disorganized information
- Too many dependencies in the code and files to track
- Too many channels of communication
- Too many revisions
- Not enough people
- Not enough time
- Awkward, nonautomated development procedures

Complexity Problems

The increasing complexity and size of software applications can create a range of problems. These include the following:

- Applications consisting of many modules

  A large number of modules makes it difficult to verify that all the input modules for a portion of the design are available and that the system build incorporates the appropriate version of each.

- Code size

  Too many lines of code make a team develop and test the application in a series of repeated steps rather than writing all the code before testing any of it. This causes problems identifying which version of a module to use. Also, this makes it necessary to work on different parts of the product in parallel in order to meet deadlines. In this situation, programmers often concurrently modify the same module. This can cause conflicting or lost changes. Another complication that arises with a large number of code lines is that it becomes increasingly difficult for one person to know the whole application well.

- Dependencies among modules

  Changes in one module may require changes to other modules.

- Multiple products

  Frequently, a development team builds similar but not identical products as part of one project. For example, the team may need to build a screen management package to support different types of terminals, or an application that runs on more than one operating system. Problems can occur in identifying the appropriate version of each component when the team builds the product.

- Rising development costs

  The cost of developing increasingly sophisticated applications rises because of the greater commitments of time and resources.

Problems Related to Changes Over Time

The gradual development of an application and the process of upgrading the software can create problems. These include the following.

- Visibility of new work

  On large projects, team members may not have equal access to all parts of the application; modified modules may be made available only after periodic builds. Therefore, developers may be working on outdated versions.

- Obsolescence

  Initial modules and requirements specification documents often need to change in response to ongoing developments in modules that make up the application. Once obsolete, these modules must be either rebuilt in the correct order or modified in a manner that corresponds to the changes in the input modules. However, difficulties may arise in trying to find all the related modules to make the necessary changes.

- Increased maintenance

  Greater sophistication in the software combines with increasingly high expectations from customers to force greater maintenance commitments, both in terms of correcting errors and enhancing current products. Balancing performance against ease-of-use also presents testing and maintenance complications.

- New personnel

  Over time, new people who are not familiar with the code join a project team. It can become increasingly difficult to get new people up to speed with procedures and code that have grown complex.

Problems Related to People

Software development depends upon the work and creative interaction of people. Problems can arise by the very nature of these interactions. These include the following:

- Start-up time

  People who join a project do not become productive immediately. They must learn how a project is organized and what their tasks will be. They may even need to learn one or more of the languages used on the application, along with learning how to use productivity tools.

- Varied coding practices

  One programmer may vary from another in his or her familiarity with coding languages and with coding standards. Frequently, programmers do not all follow consistent coding standards, thereby increasing the difficulty of working on each other's code. This problem may be

compounded by turnover among members of the team that often takes place before the product is fully developed.

- Work procedures

  A project team must have control over the work procedures among its members. For example, several developers may need to change the same module simultaneously. One developer's work can affect another developer's work. As the number of these interactions increases, so does the complexity of the development effort.

- Mistakes

  People make coding errors when implementing a design. People can also damage project data by mistakes, such as modifying the wrong modules, deleting the wrong files, and so on.

1.3 Advantages of a Formal Methodology

One good way to handle the complexities and problems listed in the previous section is to adopt a formal development methodology. A planned strategy allows a team to automate some of the repetitive tasks of the life cycle and manage projects of increasing size and complexity more easily. Additionally, the software development team can begin to benefit consistently and fully from software productivity tools.

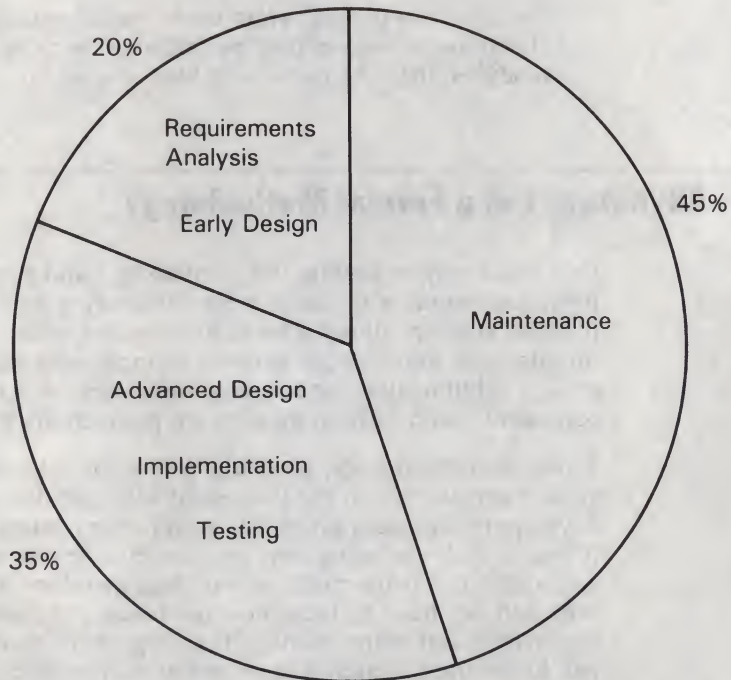
A formal methodology, particularly one shared across projects, provides greater consistency in the process of software development. A consistent development process gives the team better control over its own work. It is also easier to bring new people up-to-speed when the process is well-defined. Furthermore, when team members move to other projects, they will not have to learn new methods. Additionally, managers and supervisors can better monitor the progress of more than one team. They will know the milestones to expect as each project evolves.

Finally, a formal methodology allows a team to effectively control its management tasks; these include such things as source code control and tracking, system build procedures, problem reporting and tracking, and consistent testing. The increasing complexity of software projects makes these management tasks correspondingly difficult and time consuming. An effective and planned methodology prevents developers from wasting time and losing productivity. Many of the problems that occur in a project's later stages could be avoided by adequately establishing an overall mechanism for project development from the beginning.

1.4 Typical Costs of Development

Each stage of the development life cycle has costs associated with it that may vary from development team to development team. Figure 1–2 shows a typical view of the costs of software development.

Figure 1–2: Costs over the Software Development Life Cycle



As shown in Figure 1–2, advanced design, implementation, testing, and maintenance can account for up to 80 percent of a project's costs.

1.5 Using Tools at Each Step in the Software Life Cycle

The key to reducing the problems associated with the development cycle is to provide developers with a consistent support environment in which to work. The VMS tools environment helps developers manage the complexity of their project by automating many of the procedures. Figure 1–3 shows the integration of a number of VMS tools, mainly VAXset tools, with the software development life cycle. The black bars show where in the life cycle the corresponding tools have their primary purpose. The light bars show where the corresponding tools can be used.

Figure 1–3: VMS Tools Used in the Software Development Life Cycle

Requirements and Specifications phase

Design phase

Implementation phase

Testing phase

Maintenance phase

Runoff, EDT, TPU, DOCUMENT

VAX Notes, VMS Mail

Language-Sensitive Editor

DEC/Code Management System

DEC/Module Management System

Compilers, Linker

DEBUG

Performance and Coverage Analyzer

Source Code Analyzer

DEC/Test Manager

ZK-5937-HC

Chapter 2

VAXset Tools

This chapter summarizes the features of the VAX Software Engineering Tools (VAXset), highlighting how the tools address the problems associated with software development described in the previous chapter. Although each tool is described in its own documentation set, this book presents a special focus on the *tools integration*—how you can use the tools together and take advantage of numerous tool-to-tool links.

The VAXset tools, available individually and as part of the VAXset collection, are as follows:

- VAX Language-Sensitive Editor (LSE)
- VAX Source Code Analyzer (SCA)
- VAX DEC/Code Management System (CMS)
- VAX DEC/Module Management System (MMS)
- VAX DEC/Test Manager (DTM)
- VAX Performance and Coverage Analyzer (PCA)

These tools combine with the VMS operating system to make for an integrated software development environment.

2.1 VAXset as a Productivity Tool

Using the VAXset tools together, you can avoid many of the problems typically associated with software development. This section lists common problem areas along with the VAXset tool or tools that are designed to solve them.

Productivity

VAXset tools help improve productivity by automating tasks such as code or text entry (LSE), source retrieval (CMS), build procedures (MMS), and testing (DTM). SCA lets developers quickly move through a project's sources while searching out definitions and references to symbols. In addition, SCA helps new developers learn the internal structure of a project's code. Together, these tools help create applications more quickly and with lower development costs.

Software Reliability

Using VAXset tools, you can build predictable and reliable software that meets the intended requirements and specifications. LSE helps developers enter accurate code through its language support. SCA provides analysis, such as call checking. MMS provides a consistent means of building an application. DTM simplifies the testing of code and helps manage test systems, thus helping a team to avoid errors in code implementation. PCA helps you develop more effective tests by analyzing the coverage afforded by the test data.

Maintenance

VAXset tools help build software that is easier and less expensive to maintain. CMS stores the sources efficiently, making retrieval easy with its history tracking and class features. SCA helps maintainers quickly understand a system and the effects of any changes made to the code. MMS rebuilds the application accurately and easily for a maintenance team. With DTM, the team has the means to retest the application using the same tests developed by the implementation team, and to add tests for problems that are corrected.

Problem Identification

VAXset tools help identify problems early in the development life cycle through ongoing verification. Initially, the compilers and the VMS Debugger are used to identify the problem. PCA makes it easier to identify performance bottlenecks. DTM alerts a team to regressive developments in their software. SCA then becomes the means to trace the dependencies of that problem code, making clear any effects that apply across the application's modules.

Effective Project Management

VAXset tools simplify the managing, tracking, and controlling of your project. CMS provides a reliable storage area for your files while tracking all major transactions on library contents. Additionally, it allows developers to work concurrently on the same files, while ensuring that changes will not be lost. MMS consistently builds your application, while at the same time carrying out DTM test procedures automatically. By means of DTM epilogues, reports can be sent to team members informing them of the latest test results. The VAX Software Project Manager (PM), though not part of VAXset, provides a package of tools for planning, scheduling, estimating, and controlling software projects.

Program Compatibility

Multilanguage programs let you reuse existing code, or share code with another project even if the code is written in a different language. All the VAXset tools support applications written in more than one language.

Performance

Developers can more easily identify performance problems in the source code. PCA identifies bottlenecks in performance, and when used with DTM, provides an analysis of code coverage.

The following sections briefly describe the features of each of the VAXset tools.

2.1.1 Additional VAXset Features

The VAXset features also include the following:

- Support for most VAX languages.
- Access from the VMS Debugger and VMS Mail.
- Use of the DIGITAL Command Language (DCL) or the VAX DEC/Shell command language (a UNIX® -like interface to the VMS operating system) to invoke the individual VAXset tools.
- Compatibility with all VAX processors.
- Consistent online HELP.
- Adherence to the VMS calling standard.
- Consistent installation procedures through VMSINSTAL.
- Integration among tools. For example, LSE can obtain files from CMS; PCA can be used with DTM to ensure that tests cover important code paths; Debug can invoke LSE, and so on.

2.1.2 VAX DEC/Code Management System

The VAX DEC/Code Management System (CMS) provides a method for storing files in your project and tracking all changes to those files. Code management is especially important on large projects with long time spans and multiple versions of the developing software.

CMS works on any kind of file, such as files created by an editor, a compiler, or a linker. You can use CMS effectively to store documents, plans, specifications, status reports, object files, executable images, sixel files, or other records. (CMS cannot store directory files.) Thus, it is a tool that all team members can use — managers, system analysts, technical writers, and programmers.

CMS also supports the VAX Distributed File Service (DFS), allowing you to work with libraries on disks accessed by DFS.

® UNIX is a registered trademark of AT&T in the US and other countries.

Features of CMS

CMS performs the following functions:

- Keeps track of files at every stage of development by showing who made changes, when, and why.
- Monitors changes in files to avoid conflict.
- Allows different team members to work concurrently on the same file without the danger of losing the changes made by any team member, while reporting any conflicts.
- Conserves disk space as it stores your sources for documentation and code.
- Generates project activity reports.
- Maintains a history of library activity.
- Stores files from other software development tools.

CMS Libraries

CMS keeps your files in project libraries, which are single VMS directories. These directories store your project's files, or *elements*, as well as history information. As the project evolves, CMS tracks changes to a project library by storing only the changes made to a file with each reservation and replacement. Not only does this dramatically reduce the amount of disk space used for storing multiple versions of files, it allows CMS to reconstruct any previous version of a file, and to easily 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. By issuing CMS commands, team members can easily 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 4 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.

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 the documents at once from the library. In other words, groups allow 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 DIGITAL, 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, perhaps a field test version or a version ready for customer release.

CMS Integration

You can invoke CMS directly from within the VAX Language-Sensitive Editor (LSE), allowing you to access CMS elements. 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 VMS tools. CMS can store description files for the VAX DEC/Module Management System, and test files (prologue, template, epilogue) for the VAX DEC/Test Manager as well as the tests themselves, and results description files (benchmarks). These tools can also access elements in CMS libraries automatically. And as developers modify the files, CMS can track the changes. CMS has a callable interface and thus can be customized to assist in collecting project metrics.

2.1.3 VAX Language-Sensitive Editor

The VAX Language-Sensitive Editor (LSE) is a multilanguage, programmable editor specifically designed to help develop and maintain source code. LSE is layered on top of the VAX Text Processing Utility (VAXTPU), 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 the experienced programmer build syntactically correct programs faster and with fewer errors.

Features of LSE

The VAX Language-Sensitive Editor provides the following features:

- Syntax support for each of the VAX languages and products it supports (and support LSE):

  VAX Ada
  VAX BASIC
  VAX BLISS-32
  VAX C
  VAX CDD
  VAX COBOL
  VAX DATATRIEVE
  VAX DIBOL
  VAX DOCUMENT
  VAX FORTRAN
  VAX MACRO-32
  VAX Pascal
  VAXELN Pascal
  VAX PL/I
  VAX SCAN

- Language-specific source code templates to quickly and efficiently enter source code.
- Compiling, reviewing, and correcting of compile-time errors within a single editing session.
- Interactive editing capabilities during a debugging session.
- Ability to modify existing language environments or to define an environment.
- Integrated access to the cross-referencing features of SCA.
- Support for inspection of library elements based on SCA or diagnostics file information.
- Support for a package facility for defining your own subroutine call templates. LSE packages allow you to specify a subroutine and its calling sequence once, and then have tokens and placeholders for the subroutine and its parameters available for use by multiple LSE language environments.
- Support for user-written diagnostic files, enhancing support for user-modified or nonsupported compilers.

LSE Integration

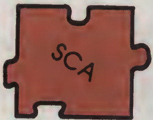
LSE is closely integrated with the VMS development environment. LSE works with supported languages to provide a highly interactive environment for source code development. Without ever leaving the LSE environment, you can create and edit code, compile and review that code, and correct compile-time errors. Furthermore, you can invoke LSE directly from the VMS Debugger to correct source errors found during a debugging session.

The VAX Language-Sensitive Editor has the ability to move among languages in different buffers. LSE determines the language you are using by the file type, thereby providing the proper interface for the appropriate compiler and calling up the language-specific templates and online HELP.

LSE is integrated with CMS to provide source code management. From LSE, you can issue 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 GOTO 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 4.2.2 for more details and an example of LSE's integration with CMS.

LSE is integrated with the VAX Source Code Analyzer (see Section 2.1.4 for more details). You can also invoke LSE from the Analyzer portion of the VAX Performance and Coverage Analyzer (see Section 2.1.7 for more details) or from VMS Mail. Chapter 4 provides additional examples of LSE's integration with other VMS tools.

2.1.4 VAX Source Code Analyzer



The VAX 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 allowing you to make inquiries about the symbols used in the project's code. With SCA, you can easily move through all your project's sources, quickly locating the definitions of any identifier, or any references made to that identifier.

Features of SCA

SCA's cross-referencing capabilities provide information about program symbols and source files for applications written in the following supported languages:

VAX Ada
VAX BASIC
VAX BLISS-32
VAX C
VAX FORTRAN
VAX MACRO
VAX Pascal
VAX PL/I

Cross-reference features are provided by the FIND command, and allow you to do the following:

- Locate names and the occurrences (uses) of names.
- Query a specified set of names (or partial names, using wildcard characters).
- Limit a query to specific characteristics, such as routine names, variable names, or source files.
- Limit a query to specific occurrences, such as the primary declaration of a symbol, read or write occurrences of a symbol, or occurrences of a file name.

For example, SCA's cross-referencing capability will quickly let you find all the locations where a symbol is used throughout an application. It will also allow you to better understand the implications of any changes to code using a specific symbol.

SCA's static analysis capabilities let you get information about program structure; that is, the interrelation of routines, symbols, and files. Features include the following:

- The display of routine call relationships relative to a specified routine.
- The analysis of routine calls for consistency, with specific regard to the numbers and data types of arguments passed and the types of values returned.

LSE/SCA Integration

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

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

In addition, SCA lets you access, from within LSE, all the sources for your project. Because these capabilities are available from within LSE, a developer saves considerable time when finding particular symbols, retrieving the related files from a reference copy area or a CMS library, and then editing the files.

2.1.5 VAX DEC/Module Management System

VAX DEC/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 wasted steps of recompiling and linking modules that have not changed. Once set up, MMS can build both small and large systems with one command.

Features of MMS

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.

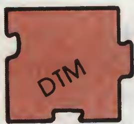
The description file is an ASCII text file that contains rules describing how the components of your application are related and the commands that MMS uses to build the application. Once you create your description file, you can use it every time you invoke MMS to build your system. In addition, this 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.

MMS Integration

Because MMS is part of the overall tools environment, 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 that 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 (VAX CDD) or forms stored in libraries for VAX Forms Management System (FMS).

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.

2.1.6 VAX DEC/Test Manager



The VAX DEC/Test Manager (DTM) organizes software tests and automates the way you run tests and evaluate test results. DTM uses the concept of 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, must be duplicated if the software is functioning properly. If the current results do not agree with the benchmark results, the modified software may contain errors. If this is the case, the software has regressed in that it deviates from previously established behavior. DTM allows you to quickly discover any regressive developments in your software.

Features of DTM

DTM provides the following features:

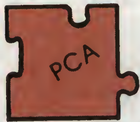
- Lets you create descriptions of software tests.
- Allows you to group these test descriptions into meaningful combinations for later runs.
- Executes specific tests, groups of tests, and combinations of test groups, either interactively or in batch mode.
- Compares the results of each executed test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

DTM Integration

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

By using DTM with the VAX Performance and Coverage Analyzer, you can measure the performance or coverage of tests run under DTM 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 4.

2.1.7 VAX Performance and Coverage Analyzer



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

- Pinpoints execution bottlenecks, and then allows you to determine the cause of them.
- Analyzes test coverage by measuring what parts of an application are or are not 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 the 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.

Features of PCA

The features of PCA are presented here in three categories: Collector features, Analyzer features, and productivity features.

- Collector features. The Collector gathers the following kinds of data:
  - Program counter sampling data. Provides a good overview of where your program consumes the most 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 VAX Record Management Services (RMS) calls in your program, helping you to understand your program's input/output behavior.
  - Exact execution counts. Tells you the exact number of times your program executes at 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 test run your program.
  - Tasking data. Shows all context switches in VAX 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.

- 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 allows 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 allows 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.
  - Lists hexadecimal dump of the contents of a file.

Additional PCA features include the following:

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

- Acceptable Non-Coverage. 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 non-covered. 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 allows 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.

PCA Integration

Used with DTM, 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 LSE's features, such as links to SCA and CMS.

2.2 Additional VMS Tools and Utilities

In addition to the VAXset tools, the VMS environment and other layered products are also important components of an integrated software development environment. Of particular use to developers is the VMS Debugger. This is an interactive, symbolic program debugger that supports the family of VAX languages. Because you can invoke LSE from the debugger, any errors you detect during a debugging session can be corrected in the original source code file without leaving the debugger. When you invoke LSE from the debugger, you are positioned in LSE at the line of source code that corresponds to your position in the debugging session. From here, you can make edits, use SCA, or reserve files from CMS. When you finish correcting the error, LSE returns you to the debugger at the position where you left off.

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

Related Software

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

The VAX Common Data Dictionary (CDD): The VAX Common Data Dictionary (CDD) can act as the central repository for data descriptions and definitions used by various VAX languages (including fourth generation languages), databases, and tools. The CDD provides the following benefits to a project:

- Storing data definitions within the CDD eliminates the need to define data within application modules. This applies to application modules written in VAX BASIC, VAX C, VAX COBOL, VAX DIBOL, VAX FORTRAN, VAX Pascal, VAX PL/I, and VAX RPG II.
- Storing data definitions in a central area reduces redundancy (multiple copies of the same data definitions) and inconsistency. To change a data definition that affects several application modules, you need to make the change only once, in the CDD, then recompile the affected modules.
- Multiple modules, although written in different languages, can share one or more definitions stored in the CDD.
- Using the VAX CDD history list feature, you can keep a record of each access to a VAX CDD directory or dictionary object.
- The VMS Lock Manager facility lets users access the VAX CDD concurrently without interfering with one another.

The CDD provides an efficient way to help manage and control definitions across the modules that make up an application. By planning for its use early in a project, a team can simplify its management tasks.

VAX DATATRIEVE: VAX DATATRIEVE is a tool for managing and manipulating data either interactively at a terminal or from an applications program. With a set of English-like commands and statements, you can interactively retrieve, store, modify, and report on data in meaningful ways. For applications programmers, VAX DATATRIEVE can save coding/debugging time and source space by handling the following functions:

- Finding and opening data files
- Performing input and output operations

- Formatting data
- Converting data types
- Handling error and end-of-file conditions

VAX SCAN: VAX SCAN, a string manipulation language, uses macros to generate the desired output text stream from an input stream of text. For example, you can use SCAN to create a filter to run with DEC/Test Manager. To avoid trivial conflicts between a test's output and its benchmark, SCAN can globally modify a text string (for example, a time output) to have the same output in both the test and its benchmark. SCAN is especially useful for building tools such as preprocessors, translators, and parsers.

VAX NOTES: VAX Notes is a computer conferencing system that lets you conduct online conferences or meetings. Using VAX Notes, you can communicate conveniently and economically with people in different geographic locations. A development team can take advantage of VAX Notes to discuss and exchange information on product design and development issues, particularly with people geographically separated from the immediate team.

VAX SOFTWARE PROJECT MANAGER: The VAX Software Project Manager (PM) is a tool that automates project management activities throughout the software development life cycle. You can use PM to plan, control, and estimate software projects. Using PM, you can record and modify project information, track project costs and resources, and produce project schedules, charts, and reports quickly.

VAX DOCUMENT: VAX DOCUMENT is a system for producing technical documentation and is designed to fully automate the creation of typeset-quality documentation from generically coded input files. VAX DOCUMENT is an integrated series of software processors that convert generically coded, device-independent source files into formatted output. Using VAX DOCUMENT, 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 common text editor. VAX DOCUMENT supports a wide variety of *document types*, ranging from interoffice memos to software reference manuals, which ensure consistency in format, typeset design, and basic organization. VAX DOCUMENT also provides a document type that meets the federal government's requirements for DOD-STD-2167 and templates for each of the DIDS Data Item Descriptors specified in that standard.

In summary, the VMS software environment provides powerful tools to aid in developing your applications. In order to use the tools effectively, you need to structure your project to maximize your output from these tools.

Chapter 3

Setting Up a Software Project

This chapter suggests ways to set up a software development project, and discusses the reasons for adopting a particular software development methodology. The chapter also discusses some of the tasks the project leader or supervisor should consider when beginning a software project, along with some options that are part of these tasks.

The goal of the chapter is to provide a framework on which to build your software development project, not to provide rigid guidelines for software development.

3.1 Initial Procedures for a Project Leader

The following sections explain the initial procedures a project leader or supervisor should follow to develop a software application. These include the following:

- Organizing a project directory structure and initial libraries to store key elements and build areas for the project.
- Establishing a proper protection scheme for all the project's directories and files.
- Setting up a procedure to meet build requirements.
- Implementing templates, standards, and logical names.
- Establishing a mechanism for documentation reviews.
- Establishing a mechanism for internal and external communication.
- Establishing a test system and a test storage area.

3.1.1 User Accounts and Directory Protection

A project leader must first determine what project information and resources team members need. Based on this, the project leader sets up user accounts and protection for these accounts.

VMS provides several tools to authorize and control the use of system resources by individual users. The *VAX/VMS System Manager's Handbook* describes these tools and discusses how and when to use them. The *VMS DCL Concepts Manual* gives a detailed explanation of protection procedures, User Identification Codes (UICs), and Access Control Lists (ACLs).

The first steps in resource planning should provide a basis for system accounting, file protection, and interprocess communication. You can do this as follows:

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 and enter those identifiers in the system rights list. For example, you may wish to make developers holders of the PROJECT_SOURCE identifier, and make technical writers holders of the PROJECT_SOURCE_READ identifier. Examples of using these rights identifiers are described in Section 4.1.2.2.
3. Set up UIC protection masks (use the DCL command SET PROTECTION) to further restrict outside access (system, owner, group, world) to member directories, certain project and library directories, and selected files. Additionally, you may want to define categories of access within the group (read, write, execute, delete).
4. Use Access Control Lists (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, as necessary, access to specific elements for users having other UIC numbers.

See Chapter 4 for an example of setting up protection for directories and libraries.

3.1.2 Project Directory Structure and CMS Libraries

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

- How much disk space do you have?
- 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?
- Is it possible to predict size requirements for each library in the system?
- 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?

Directories and libraries should be set up to optimize their use by both the team members and the tools. Figure 3–1 shows the first stages of an effective directory structure. It contains the directories and libraries that you will set up in the early stages of your project. You will add more directories and libraries to this hierarchy as your project develops, as discussed in later sections of this chapter.

The first storage areas you need include the following:

- A subdirectory for public access

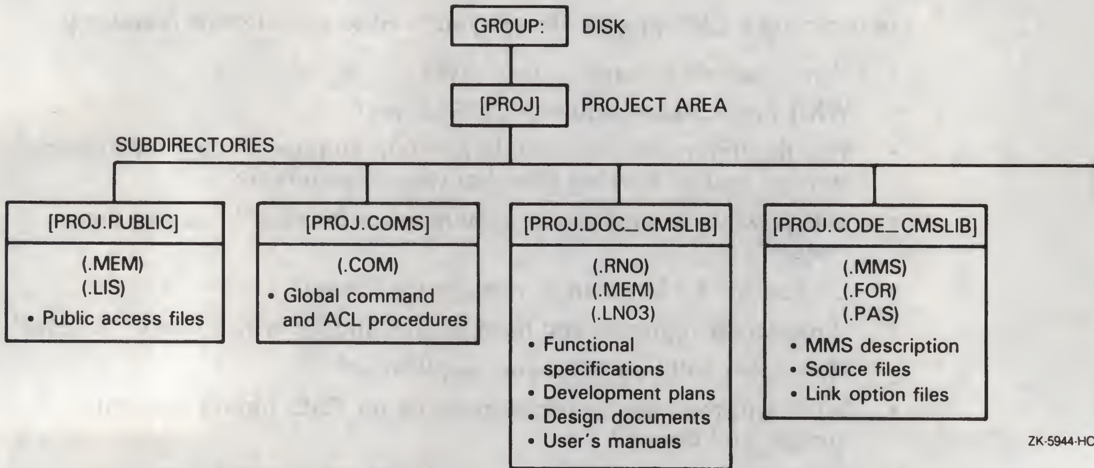
  This subdirectory, [PROJ.PUBLIC], contains information for users and interested parties (for example, printable documentation files). (Alternatively, this directory could be kept outside the project area.)

- A subdirectory for global command procedures

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

- A CMS library for source code

Figure 3–1: Initial Storage Areas for a Typical Project



This library, [PROJ.CODE_CMSLIB], 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).[1]

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.

- A CMS library for documentation

  The CMS library for documentation, [PROJ.DOC_CMSLIB], can contain files that document the application; for example, those produced for DIGITAL Standard Runoff (DSR) (.RNO files) or VAX DOCUMENT (.SDML files). You can keep the most recent printable files (.MEM or .LN03 files, for example) in the publicly accessible subdirectory, [PROJ.PUBLIC], or in your CMS library. But generally, you should not store files that can easily be reproduced by processing already stored files if disk space is a concern. Alternatively, you may want to store your document sources in more than one CMS library;

[1] Precompiler source files are for the RDBPRE precompiler, for example, before precompiling.

for example, using one CMS library for requirements documentation and another CMS library for specifications documentation.

Another alternative to setting up separate CMS libraries for the documentation sources would be to place them in the CMS code library. This way, you could fetch or reserve both the documentation and the source files with one generic file designation. For example:

```
$ CMS FETCH proj_file.*
```

This command retrieves files related by name but with different functions and different file types; for example, source code versus documentation source. Chapter 4 provides an example of setting up CMS libraries. See the *Guide to VAX DEC/Code Management System* for detailed explanations of CMS libraries and commands.

Considerations for Large Projects

For large projects, you may need additional CMS libraries for your source code. 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 probability that a library will be busy increases as the number of elements and users increases. The problems of having a large library with numerous users include the following:

- Access. CMS allows multiple simultaneous readers; that is, users of the library for tasks that do not modify the contents of the library (for example, SHOW and FETCH operations). However, CMS provides only single thread entry; that is, only one "door" for operations that modify the library's contents (for example, RESERVE, REPLACE, INSERT operations).

  The many developers involved with a large project are likely to 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 wild card 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 simply 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 do 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 difficulty, you can use the Search List facility of CMS to help with managing multiple libraries. As you subdivide libraries to improve performance, though, there will be a tradeoff between performance and ease of library management.

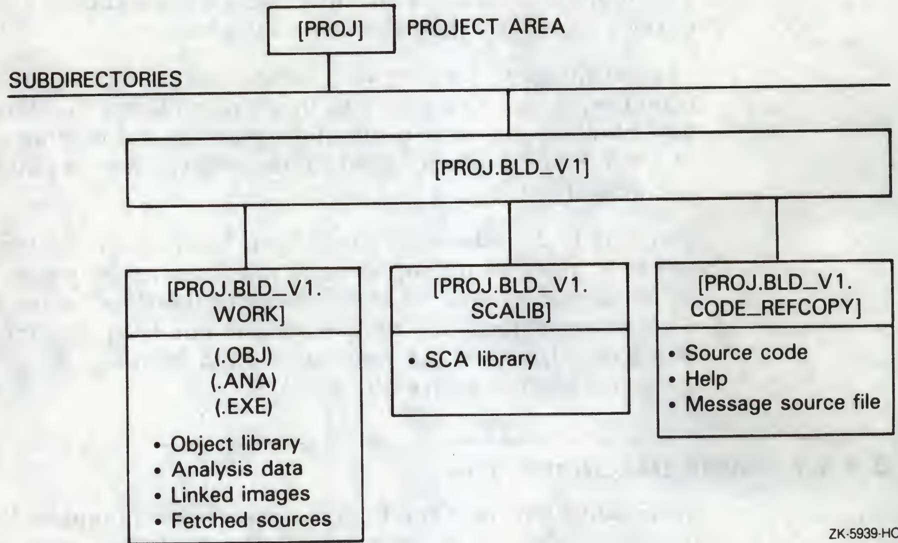
For operations that are common and need to be faster than is possible with a command procedure, you may want to write programs that use callable CMS to perform the operations. Refer to the *VAX DEC/Code Management System Callable Routines Reference Manual* for more information.

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. In effect, each facility of the project would function as its own "miniproject." The entire application will come together at major application milestones or base levels that represent agreed-upon stages of progress.

3.1.3 Build Directories

Although CMS can store nontext files, such as binary sources, you may choose to refrain from storing files that can be produced from other stored files (for example, object files that can be produced from source files). This saves disk space. You can set the build areas up as parallel subdirectories, which will correspond to the initial prototype build directory. Figure 3–2 shows the hierarchy that would result from this type of build structure.

Figure 3–2: Build Directory Hierarchy



Located on your group's disk are directories devoted to the periodic building of your application: GROUP:[PROJ.BLD_Vn], where n represents the version number of the software. As your project develops, you can create subdirectories to store successive versions: [PROJ.BLD_V1], [PROJ.BLD_V2], and so on.

You can use a directory [PROJ.BLD_V1.WORK] to carry out your project builds. Essentially, this directory [PROJ.BLD_V1.WORK] can store all the files that you do not want to store in a CMS library, but that you will reconstruct with each major version of the project. As a result of the builds, this directory will store the source files fetched from CMS, and the .EXE, .OBJ, and .ANA files that are produced during compilation and linking.

You may want to group all the .OBJ files into a object library (.OLB) in order to simplify your linking procedures. You can link against the object library rather than all the individual object files. See the *VMS Librarian Utility Manual* and the *VMS Linker Utility Manual* for more information about VMS 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 whereby 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 may find it advantageous to keep one major build that you frequently return to, perhaps for support and maintenance purposes. Intermediate builds do not have to be kept 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 re-create a build from any development stage using the original source files in CMS.

3.1.3.1 MMS Description File

MMS helps you with the building process by automatically accessing the source files you store in a CMS library. It follows the sequence of dependencies among your source files as described by you, 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 are likely to 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 snapshot of your files at each successive base level. Whenever you need to rebuild a class, for instance, an older base level, you have stored in that class the sources along with the MMS description file to build the application.

Your use of MMS is likely to 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. This kind of incremental building contrasts with a build procedure that uses all the sources in the application. Because of this feature, you may want to carry out nightly incremental builds, while leaving builds that start from all your sources and regenerate all intermediate files only for planned milestones during your project's development. The advantage to 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 4 contains an example of setting up an MMS description file. See the *Guide to VAX DEC/Module Management System* for a complete description of MMS.

Considerations for Large Projects

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

1. Determining stages of development (milestones or base levels) at which updated facilities contribute their sources.
2. Creating classes within each facility that represent the current sources for that facility.
3. Fetching from those classes to a storage area that has limited access beyond a person assigned the role of configuration manager.
4. Using a master build procedure to assemble the sources into the executable application itself.

The way a project implements these steps can vary depending on the demands of the project. For instance, the configuration manager may layer another CMS library on top of the facility libraries to monitor the fetched classes. This library would not be an active development library, but a storage facility with the added 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.

3.1.3.2 Reference Copy Area

A reference copy area is a storage area that you create as a separate subdirectory (for example, [PROJ.BLD_V1.CODE_REFCOPY] in Figure 3–2). Using a reference copy area can provide an alternative way to carry out your builds. Rather than having MMS fetch files directly from CMS, you can have it access the files in a reference copy area and build up from all the sources.

One advantage to building from a reference copy area is the increase in speed during the initial part of any build procedures. 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. Note that a CMS library can only update one reference copy area at a time. Should 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 or not you actually use the reference copy area for builds depends on the work procedures your project sets up. You also need clear copies of your sources for the debugger, 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. 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, however. If you choose to use a reference copy area for building, you should restrict access to this directory so that 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, thereby generating a history of the file transaction.

Another disadvantage of building from the reference copy area is that you cannot re-create 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. In other words, you cannot use this feature for variant source development. Despite the build limitations, the reference copy area provides a useful intermediary between the CMS library and individual work areas.

Advantages exist for 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 feature is particularly 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, carrying out the build with the new modules only. This is the build procedure followed in the scenario described in Chapter 4.

3.1.4 SCA Libraries

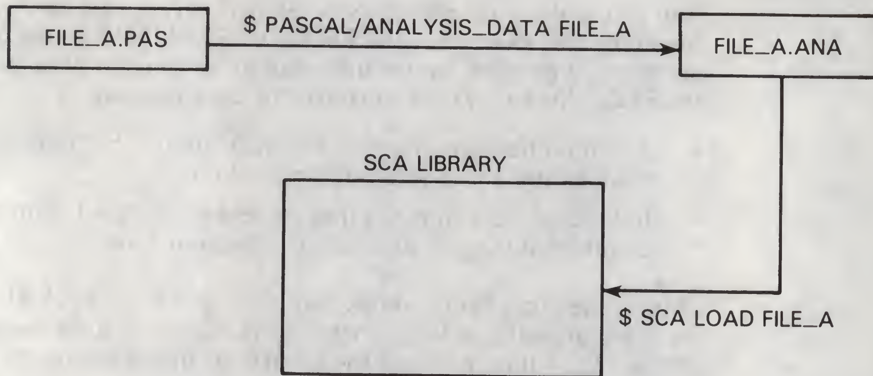
You can make your project-wide SCA library a part of the build tree hierarchy (for example, [PROJ.BLD_V1.SCALIB] in Figure 3–2. To provide an index of detailed source information for your project, you can arrange your SCA library system to consist of the following:

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

Figure 3–3: Filling an SCA Library



3.1.4.1 Project-Wide Development

By using the /SCA_LIBRARY qualifier on the MMS command line to initiate a project-wide build procedure, you can have MMS automatically compile your files with the /ANALYSIS_DATA qualifier and load 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.

Section 4.8 contains an example of an MMS description file that shows these steps done manually for purposes of illustrating them.

3.1.4.2 Incremental Development

Your team will benefit from an SCA library structure in which team members independently manage those libraries that are small and frequently changing. This becomes even more important for extremely large projects; consequently, individual developers benefit from updating only their local SCA libraries on a regular basis, and not the project-wide SCA library. It even may 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 carry out ongoing development. After compiling the files, they can use their local SCA library to store the newly created analysis data.

When actually 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). This can be done by using the SET LIBRARY command as follows:

```
$ SCA SET LIBRARY lib_1,lib_2
```

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

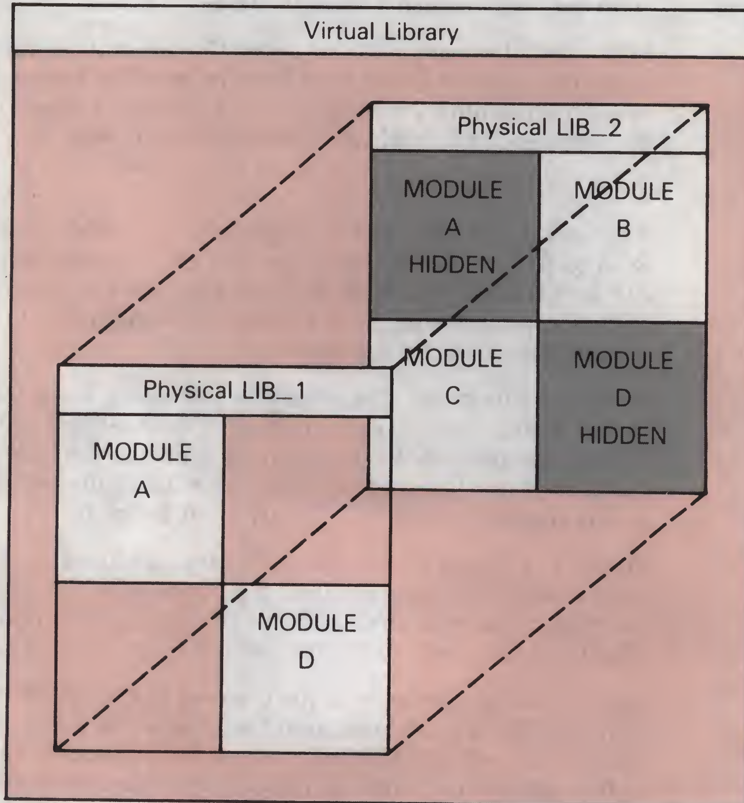
Every module of the first library in the search list is included for viewing in the virtual library. Each module in each subsequent library on the list is then compared with the modules previously selected. If a module is uniquely named, it is included for viewing in the virtual library; otherwise, it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

Figure 3–4 shows how a virtual library concatenates the physical libraries on a library list. As a result of the previous SET LIBRARY command, a developer creates a virtual library containing four modules: A and D from Physical LIB_1 together with C and B from Physical LIB_2.

When making queries, you have access to the information in any of these libraries. Since 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. Note that when SCA joins the physical libraries into a virtual library, only modules found in the first libraries will be accessed by SCA even when they are also stored in libraries further down on the library list. The modules down the list are hidden.

An added benefit of this procedure is that it prevents unintended changes to the project-wide SCA library: any updates to the SCA library take place only to the library on the list corresponding to the source modules being updated. When a developer updates an SCA library with the LOAD command, only that developer has access to the library until the load process is finished. If many people were using the same library, and that

Figure 3–4: Physical versus Virtual SCA Libraries



library were subject to frequent updating, access would 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 quite manageable. Note that an SCA library supports queries by multiple users.

As team members complete particular programming tasks, they will 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 can efficiently use SCA on immediate development tasks. Finally, when the team reaches predefined milestones and begins a major application build, they can re-create the entire project-wide library. In this way, the team can tie current build sources and development status to specific milestones. In all cases, the team can create or modify 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.

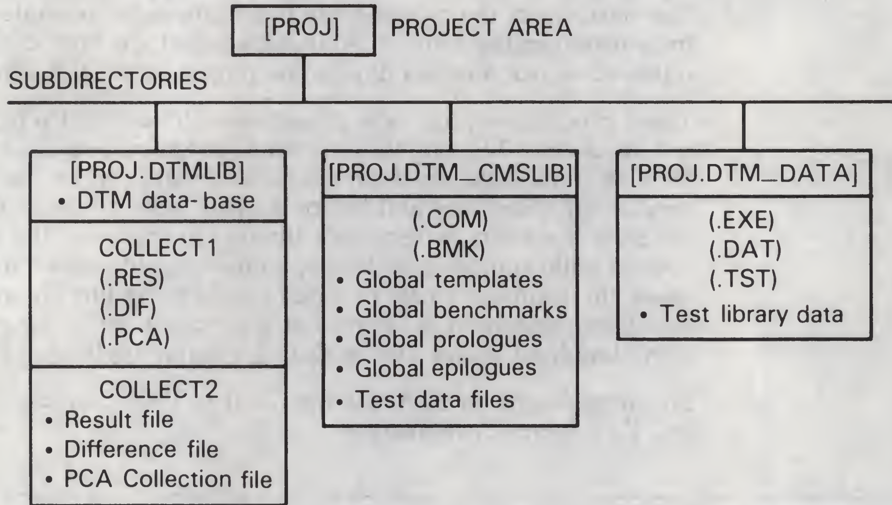
For more details on SCA, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer.*

3.1.5 DTM Libraries

As part of your directory organization, you should set up DTM storage space. Setting up test directories early in your project provides the following advantages:

- Your organizational directory structure is fairly 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.

Figure 3–5: Directory Structure Showing DTM Libraries



As shown in Figure 3–5, your DTM directory organization may require three storage areas:

- A DTM library for test results

  This library is a separate subdirectory (in Figure 3–5, designated as [PROJ.DTMLIB]). This library stores DTM test description information. This test information, located in a DTM database, provides DTM 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 DTM is to set up for each test or collection of tests while executing (DTM variables)
- Any associated filter procedures

The DTM library also stores the results of your tests; these results may also include PCA test coverage or performance results if DTM is used to control PCA data gathering. In this situation, since DTM controls the data collecting, it automatically places the PCA results in the DTM library. After you act upon DTM and PCA results, if problems do occur, you can purge the result files as part of your ongoing development process. Note that the collections shown in the DTM library ([PROJ.DTMLIB]) in Figure 3–5 are generated automatically by DTM.

Note that when you collect data using PCA by itself rather than through DTM, 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 DTM test files

  You can store templates, benchmarks, test data files, prologue files, and epilogue files in this library. (In Figure 3–5, 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, thereby ensuring the integrity of past and future test performances.

  As an alternative, you can create separate subdirectories for your DTM test files rather than using a CMS library. You can even create multiple subdirectories for this purpose corresponding 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 DTM subdirectory for test data

  Whether or not you need this particular subdirectory depends on your application. You may choose to use this directory to store input files needed for generating output files for tests. For example, a linker project may 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, you may find it an organizational aid to maintain this subdirectory for testing purposes. In Figure 3–5, [PROJ.DTM_DATA] stores these types of data files.

3.1.6 Project Standards

Project procedures should include establishing standards for your team to operate by, such as design standards, coding standards, and testing standards. By agreeing on these standards early in the project's life cycle and having all team members adhere to these standards, you avoid confusion and conflict later. An added benefit is that standards provide a consistent framework for new team members, enabling them to become knowledgeable about the project faster.

Design Standards

Establishing design standards means agreeing upon, and adhering to, the format of the application's designs. You need to decide on the content of your designs. The designs should consider not only the specifics of how you plan to implement functional areas, but also the context of your application. Examples of additional considerations include evaluating dependencies outside of the project, as well as within it, or deciding what languages you will use to implement a particular functionality.

To aid in making development decisions in the future, you should record not only design features you decide to keep, but also alternatives that you discard. The goal is to have workable and complete designs; consistency that your team agrees upon will make this more likely.

Coding Standards

You should set up coding standards in the early stages of your project. For instance, you will find that consistent naming standards provide the following advantages:

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

The following are examples of naming conventions for modules, routines, and variables in an application. See the *Guide to Creating VMS Modular Procedures* for more detailed information on coding standards.

- 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 facility prefix is the first part of any routine name. In this example, the PROJ_ is the facility prefix. The remainder of the name indicates the type and function of this routine. Other examples of functions might include GET, FIND, and MODIFY.

- File name: PROJ_DB.PAS

  This example uses the facility prefix, PROJ, and the DB designation to show which type of routines are in this file (in this case, database routines).

- Modules: PROJ_DB

  Module names are identical to file names except module names do not have extensions.

- Variables:

  - PROJ_GT_USERNAME

    This example has the prefix PROJ_. The letters, GT, indicate that this is a global text declaration, accessible to any module.

  - PROJ_K_MAXFILE_COUNT

    The K designates a constant, with the remainder of the name describing the function, in this case, a value for the maximum number of files to be used. Unlike the preceding example, this is a local constant because it does not have a G for GLOBAL immediately in front of the K.

A different example of coding standards is the use of coding formats. Built into LSE's 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 that they adhere to site-specific coding standards. This benefits not only your team during development, but also new programmers who may maintain or update your team's work in the future. See Chapter 4 for an example of using LSE to help set up coding standards on your project.

When you set up coding standards, you may want to set up periodic code review meetings. You can use these meetings to inspect sections of code to see that it meets the team's standards, and that the code not only works, but that it works efficiently.

Testing Standards

By establishing goals for testing, you can create a testing plan that will meet the particular needs of your project.

Realistic testing goals should include testing that starts early in the development process and runs through to the end of the life cycle. By testing as your project grows, you are less likely to omit a particular piece of code from testing as future work layers new functionality over it. Also, you are more likely to discover errors while the code is still fresh in people's minds; waiting to test could 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 do in fact 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 can be applied more effectively in advanced stages of development.

"White box," or unit testing, takes 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 more exhaustive testing, in combination with "black box" testing, can meet the needs of those projects that demand extensive testing.

A team involved with this type of rigorous testing can benefit from an added feature of DTM: when used in combination with PCA, it can produce an annotated listing of all code lines exercised by the test set.

Performance Standards

After writing a certain amount of code, programmers should be able to run a simple set of tests to compare against performance benchmarks. Although PCA does not have a facility for benchmarking, you can improve the overall performance of your software by using PCA to find performance bottlenecks in the cases where you find poor performance.

3.1.7 LSE Templates

You can use LSE templates to provide consistency in your coding and commenting conventions. You can modify constructs, menus, and descriptions within existing definitions. If you need a language that LSE does not support directly, you can create your own templates for that language. See Section 4.3.1 for procedures to create templates.

You may want to create LSE templates for your documentation; for instance, design or specification formats. These templates may 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 may make and save his or her own modifications, if desired, as well. These templates can be available to each developer individually, or you can have them become a permanent part of the environment files provided with your system. In the case of government contracts, document templates supplied with VAX DOCUMENT can save considerable time while enforcing consistent and complete documentation.

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

- Undefined tokens
- Undefined or unreferenced placeholders
- Placeholders defined with the same name as LSE package parameters
- Tokens defined with the same name as LSE package routines
- Routines or parameters defined with the same name in multiple packages
- Invalid help 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 VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

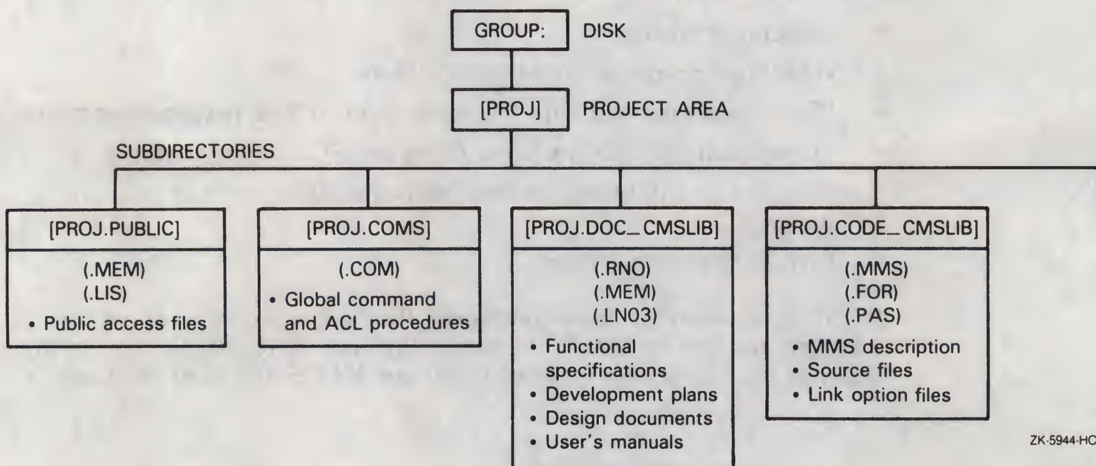
3.1.8 LSE Logical Names

As part of the initial setup of your project, you will want to have the context for the project implemented from a command file. You should store the file itself in an area such as [PROJ.COMS] in Figure 3–6.

Your logical name definitions are included in this command file. These are names you can use in place of a file specification, part of a file specification, or a directory. You can define names that your team uses frequently in this common command procedure that can, in turn, be invoked by the individual login procedures of your team members. Alternatively, you can have these logicals defined in the system startup file with the /SYSTEM qualifier so that they only need be defined once. See Section 4.3.2 for a specific example of this procedure.

As yet 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 *VMS DCL Concepts Manual* for more information on logical name tables.

Figure 3–6: Initial Storage Areas for a Typical Project



Logical names serve two main functions:

- You can define commonly used files, directories, and devices to have short, meaningful logical names. Such names are easier to remember and type than full specifications.
- Logical names safeguard the project against changes in the computing environment. For example, if the project moves to a new disk, or to another VMS system entirely, 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 many last-minute changes when installing the application on the production system for which it is designed.

Table 3–1 shows some examples of naming conventions for your directories.

Table 3–1: Examples of Logical Names for Directories

Example	Directory Purpose
PROJ_PUBLIC	Public access area
PROJ_CMS_CODE	CMS library for source code
PROJ_BLD	Current version of the build area
PROJ_DTM	DTM 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.

3.1.9 Communication Management and Report Mechanisms

You want to ensure that people within your group can communicate easily and effectively with one another, as well as with people outside of your working group. You also want to keep records of much of your communication. DIGITAL provides the VMS Mail Utility (MAIL) and VAX Notes to aid in communication and record keeping.

The VAX Software Project Manager (PM) is designed to handle tasks associated with scheduling and individual status updating as well, and is thus an important communications tool. Section 3.2.6 has more details on using PM.

3.1.9.1 Communication Within Your Project

Every team can benefit from project reviews and meetings. The frequency of these meetings will depend on the size of your team and the proximity of team members. The meetings can help each team member become familiar with the work of the other members of the team. With newer programmers, there is the added benefit of having more experienced programmers give advice to help solve obstinate coding problems.

In addition to face-to-face contacts and telephone conversations, you can use the VMS Mail Utility. By forwarding documents and memos to everyone in the team, you enhance team awareness and cohesiveness.

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].

Using MAIL, you can track the status of a project. Usually, team leaders require monthly reports from the team members. With MAIL, these reports can be consolidated and forwarded to managers who are tracking the project's schedule.

The VAX Software Project Manager is ideally suited for reporting the status of work items and generating written status and schedule reports, as well as charting and analyzing the team's progress.

3.1.9.2 Communication Outside Your Project

If your project has many members or if the members are not in the same location, you may want to use VAX Notes, a conferencing tool.

VAX Notes provides your team with another way of communicating, particularly with those outside your project. Unlike MAIL, VAX Notes allows many people to communicate information on the same topic.

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

VAX Notes conferences are open to a larger audience and promote communication to a larger group than MAIL. VAX Notes keeps a record of all entries and replies. At a later date, some of this online information can be extracted to document changes to the software as part of a quality review. In this way, VAX 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 VAX Notes conferences to facilitate your project's goals include the following:

- A Problem Forum Conference

  Users outside your group may list errors or problems that they have found. Someone from your group who monitors the conference can suggest remedies, or if necessary, make changes to the software. Other users who read the conference can themselves suggest remedies or contribute their own opinion about the problem.

- A Wish List Conference

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

- Restricted Conferences

  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 may create a private conference for your 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, another for specifications. Your reviewers can read the information online and respond within the framework of the conference. This reduces the need for you to transfer a hard copy to the reviewers. In addition, your group can extract comments from the conference and incorporate these comments into documentation.

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

3.1.9.3 Documentation Reviews

Since accurate and timely user documentation is critical to a product's success, the team must build in adequate review procedures. The form these review procedures take depends upon your team's size. Smaller teams can rely on informal means of review, although formal reviews still play a role. Larger projects may 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; this narrows their focus.

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

3.2 Ongoing Procedures for a Project Leader

In the early stages of your project, you implemented a number of important procedures that will aid you in effectively managing the project:

- Finalized design plans.
- Created project directory and library structures.
- Implemented design and code standards.
- Agreed on document templates.
- Wrote an MMS description file to carry out builds for your documentation and preliminary code.
- Set up communication paths and mechanisms.
- Set up a project management database for the project, using the VAX Software Project Manager.

If you have adequately planned your project in the early stages, you will find that your ongoing management decisions are easier. However, you will likely need to reevaluate some of the procedures while adding new procedures as your project creates additional demands on your team.

3.2.1 Extending the Library Structure

As your project moves into the implementation stage, you may need to reevaluate the status of your directory structure to see if you can improve it. For instance, you may find that a single CMS code library is inadequate for your needs. It is recommended that you start with one CMS library for source code and break it up if you find that you have library contention problems.

You may want to rearrange your directory structure to better reflect the ongoing development needs of your project. You may find that by adding new storage areas, you can better access specific groups of files. These are decisions that each team makes based on its own particular needs and methodology.

3.2.2 Establishing Project and Personal Build Procedures

Section 3.1.3 discussed 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 make builds.
- How thoroughly you want to automate your build procedure.

3.2.2.1 Personal Build Procedures

Individual developers need to make decisions about their own storage areas. These typically are less complicated than the overall project structure. For instance, individual developers are not likely to need a personal DTM library. Many developers rely on the project CMS libraries to provide a storage area, although some developers set up a personal CMS library to track and organize their own work. Developers have a private work area and possibly a separate build area; they may 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 should 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 SCA commands to bring a read only 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 command to reserve a generation of the corresponding element from the CMS code library.
6. Use LSE to modify the file.
7. Compile the modified file using LSE's COMPILE/REVIEW command.
8. Link the image.
9. Carry out any test procedures using the debugger and DTM.
10. Replace the modified file into the CMS code library.

Note that if there are substantial changes to code or to routine calling sequences, you might want to use LSE's COMPILE $/ANALYSIS_DATA command instead of COMPILE/REVIEW.

3.2.2.2 Project Build Procedures

Different members of your team will make varying degrees of progress on their work as they modify files. Since 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 DTM 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 DTM epilogue, you can have a full status report on the test collection forwarded to everyone on the team. The idea is to automate repetitive tasks while enhancing overall team communication. For examples of using these tools together, see Chapter 4.

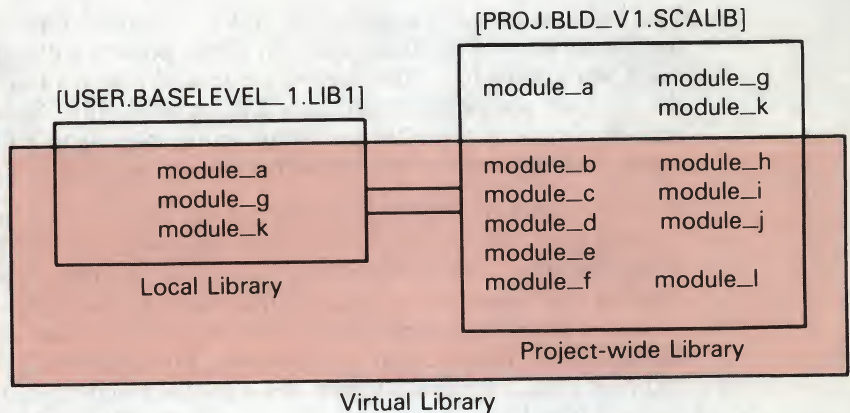
3.2.2.3 Access to SCA Libraries

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 CREATE LIBRARY command 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 3–7). The SET LIBRARY/AFTER command makes sure that the existing project-wide SCA library ([PROJ.BLD_V1.SCALIB] in Figure 3–7) is placed after the local library on the library search list. Note that all of these actions take place within the framework of the individual developer's work area. 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 3–7 shows the relationship between the two physical libraries in this example.

Figure 3–7: Working SCA Libraries for Developers



3.2.2.4 Access to Source Files

LSE, SCA, and CMS together provide ways to manage access to files. The goals are threefold:

- To easily display the source code corresponding to particular variables and routines
- To readily access modifiable files when you are ready to make changes
- To prevent inadvertent changes to sources

CMS provides the best repository for your sources. All CMS transactions generate useful history information, as well as keeping the sources themselves intact over their development life cycle.

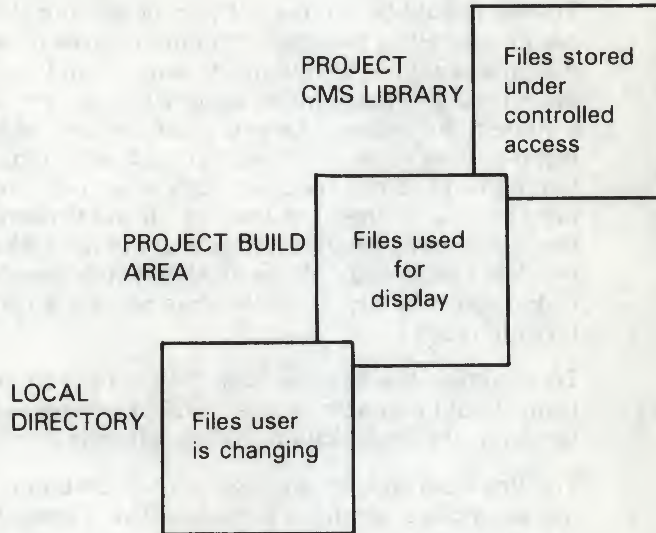
LSE can help manage source code by being the medium through which you access your storage areas, including your CMS library. LSE's file-access commands let you create a search list of directories. Typically, this search list might consist of, in this order, a local work area, a project build area, and,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

```
LSE> SET SOURCE_DIRECTORY [], proj_build, proj_cmslib
LSE> SET DIRECTORY/READ_ONLY 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 4.3.2 shows how to carry out a similar procedure using logicals in a LOGIN.COM file. Figure 3–8 represents the directories that correspond to the search list.

Note that with Version 3.0 of CMS, CMS objects may have Access Control Lists (ACLs) attached to them that should be taken into account when reserving and replacing elements and executing the commands themselves. Some project leaders may wish to use CMS ACLs to control access to library elements rather than requiring the use of the LSE SET DIRECTORY/READ_ONLY command. While the SET DIRECTORY /READ_ONLY command provides a useful mechanism for preventing the accidental modification of files in a directory or CMS library, this command does not provide the level of protection that ACLs can. By attaching VMS ACLs to directories and CMS ACLs to CMS libraries, you can protect objects that should not be modified (except by selected people). And last, ACLs give you an effective security mechanism that you can enforce for all users.

Figure 3–8: Source Code Management



CMS allows team members to access a file concurrently. When team members return the files, they can use CMS to merge the files, informing the most recent worker of any conflicting code that needs to be resolved.

Team members should use the Remark feature of CMS each time they do a transaction, which CMS logs in the CMS library. This feature lets each CMS transaction be documented more fully as it occurs, providing a history that gives the project leader a way to monitor not only ongoing changes to files but also the progress of programmers' work. If problems turn up later, the CMS history information lets the team know who made changes, when, and why. By providing informational remarks, team members can also 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.

3.2.3 Setting Up Tests

Testing should be an integral part of all your developers' work in the design as well as the implementation stages of your project. During the design stage, your development team should be designing tests and laying out a testing strategy in the same way they are designing and laying down 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. Since code develops in an iterative fashion, programmers need to be sure they run tests on all the versions of a file. Otherwise, new code may introduce errors that are not picked up. In effect, the code will have regressed. All the modified code should have tests stored in the CMS project library for DTM. (See Section 3.1.5 for information about DTM libraries.)

To maximize the benefits from your tests, you need to decide how your team should organize its tests. DTM 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 DTM. 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 DTM session file that exercises your software. Each test must have a template file.

Prologues and epilogues are command files associated with a specified test. Prologue files run before, and 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, for example, be used to perform clean-up operations or to filter the result file of run-dependent data.

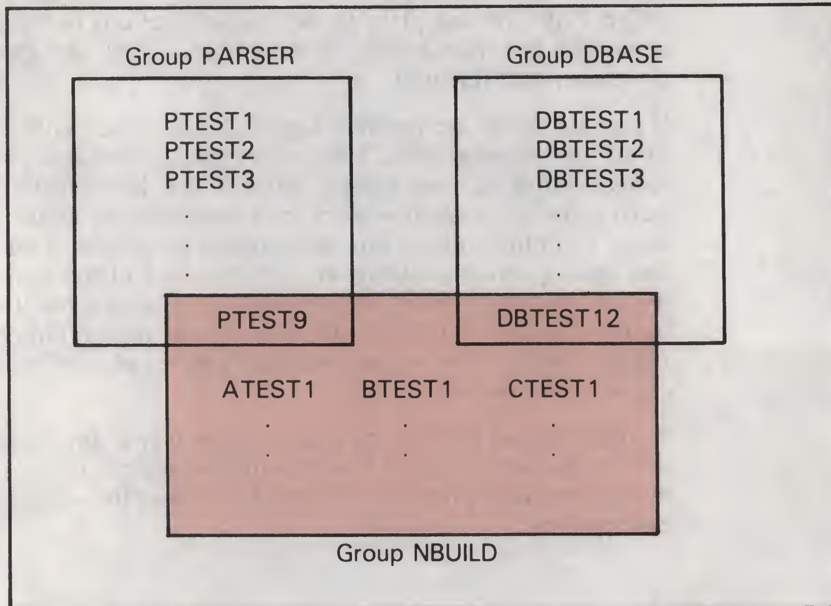
A variable in DTM is either a DCL symbol or a logical name. DTM stores variables and uses them when executing tests. You can use variables in templates, prologues, and epilogues. Variables provide a convenient way for you to tailor a single template, prologue, or epilogue file so that it can be used 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 DTM in a separate step, and also include them in each test description that uses them. Section 4.8 contains an example of using DTM variables.

You can organize your test descriptions in the DTM 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 you can group them by both developer and function.

If you are doing incremental nightly builds, you may want to run a subset of all the system's tests. This subset can, for example, consist of tests of software that accesses system services like the VMS Run-Time Library. Such software is often subject to numerous code modifications and bug fixes. 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. So 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 will be to fix.

Another use of test groups may involve forwarding tests to a quality review process in which a representative sample of tests rather than a full test system is appropriate. Figure 3–9 shows the concept of grouping test descriptions.

Figure 3–9: Grouping Test Descriptions



Grouping test descriptions simplifies the process of creating test collections—the mechanism by which you actually initiate the running of tests. The key is to create valuable 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. Section 4.8 explains how to use automated testing procedures.

3.2.4 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.

Performance problems are a frequent complaint. Tight deadlines can cause performance problems because software is released soon after coding is completed. The result is that teams find themselves evaluating performance intuitively rather than relying on quantitative data, and programmer intuition can often be wrong. Although PCA cannot solve problems associated with poorly designed software, PCA can help you improve performance by pinpointing performance bottlenecks, 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 non-covered. 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.

3.2.5 Monitoring Project Progress

Producing quality software requires that a project be carefully monitored throughout its life cycle. In order to carry out this task effectively, the project leader needs a development plan with realistic estimates of how long major milestones will take to implement. Project leaders need to track the progress of their teams to be sure that individual members of the team are maintaining their respective schedules. In turn, the project leader will report the team's progress to those supervisors responsible for fitting the project into overall company schedules.

Much of the work of monitoring a project depends on effective communication. Some of the tools and techniques that can help in this task have already been mentioned in previous sections:

- A realistic development plan available to the entire team and management
- Frequent team meetings

- Weekly or monthly reports by individual programmers and a collective project report for management
- History logging from CMS and DTM maintained in their respective libraries

The history logging from CMS and DTM is particularly useful for project leaders. For instance, changes made to elements in CMS are automatically documented, including the person making the change, the action taken, and the date of the transaction. Also, the report features of DTM (by means of an epilogue file) provide teams with additional means of monitoring a project.

3.2.6 Using the VAX Software Project Manager

If you have the VAX Software Project Manager (PM) present on your system, you will find it an especially useful tool for monitoring project process and for project management in general. PM is a package of tools that automate project management activities throughout the software development life cycle. You can use PM to plan, control, and estimate software projects. Using PM, you can record and modify project information, track project costs and resources, and produce project schedules, charts, and reports quickly. The PM tools are summarized in the following list:

Planning Tools: PM provides the following tools to plan your projects:

- Resource tool for establishing resources for your project, such as staff, equipment, supplies, and other elements needed to complete your project.
- Work Breakdown Structure (WBS) Composer and Scheduler for dividing the project into tasks and subtasks, and specifying the order and dependency of tasks and milestones. You can use the WBS Composer to create tasks and arrange them into a model of your software development project. This model, called a Work Breakdown Structure (WBS), is a tree structure composed of project tasks. Tasks at the highest level of the tree, called parent or interior tasks, represent organizational divisions of your project. Tasks at the lowest level, called child or leaf tasks, represent the actual project work activities.

You can create tasks and milestones with the Scheduler. These activities appear on the Scheduler screen. You can connect these activities to form a Precedence Network, which is a chronological map of your project, showing the order in which tasks must be completed. The Precedence Network begins and ends with milestones that may contain additional milestones that mark important events in your project, such as the end of a design stage. This Precedence Network must be in place before you create your project schedule.

- Calendar and Scheduler tools for automatically calculating a Project Schedule.

Controlling Tools: PM provides the following tools to report status and monitor projects:

- Scheduling tools to generate new schedules based on your current progress and to compare the results to a previous schedule.
- Status Updating tool to report effort and cost information for tasks between the dates work has begun and when it is completed. This information becomes the basis of the charts and reports PM produces to help you monitor your project.
- Gantt Chart for monitoring the progress of work on a task-by-task basis and seeing the amount of work remaining. The Gantt Chart shows schedule and status information for all project tasks.
- Rate Charts for identifying schedule and estimation errors, allowing you to compare actual against scheduled effort and costs.
- Reports tool for generating textual reports of work progress and costs. The Reports tool provides summary project schedule, precedence, and cost information for single tasks, or for all the project tasks and milestones. The reports produced help you monitor the status of parts of your project and of the project as a whole.

Estimating Models: PM provides the Estimator to make estimates of project effort and costs, building a model called an *Estimation Hierarchy*, based on Barry Boehm's COCOMO (Cost and Constructive Modeling) model. The Estimation Hierarchy is a tree structure, consisting of nodes that represent the deliverable components of the software system. The Estimator uses the information you supply to predict how much effort will be required to complete the project. After performing an estimation operation, each estimation node contains cost and effort estimates for your project. Information is rolled up, with interior nodes showing total cost and effort estimates for its subordinate nodes.

See the *Guide to VAX Software Project Manager* for more details on using PM.

3.2.7 Tracking Reports During Field Testing

Progress can be hampered by poor tracking as a project moves into its field test stages. At this point, 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 system to report, assign, and track problems (Quality Assurance Report (QAR) system)

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

Using VAX Notes

The built-in features of VAX Notes allow you to create a basic QAR system to suit the needs of many projects. For detailed information on using VAX Notes, refer to the *Guide to VAX Notes*.

You can use VAX Notes to organize feedback from the test sites. How you get that feedback from the test sites, and return your responses, depends on procedures set up at your work site. These feedback mechanisms may be written comment cards. In this case, members of the team will enter the comments into the QAR system. They will also enter into the QAR system their response to the test site comment, 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 will be sent to the test site by mail.

As an alternative to written comment cards, users can dial in to accessible accounts to directly input their comments to the QAR system. In this case, your team's responses would be entered directly into the QAR system. This method would require users to refer back to the online QAR system for a response to their comment. By having an online QAR system, an added benefit is that test site users can look in the QAR database for solutions to their problems and avoid entering duplicate problem reports.

VAX Notes organizes and speeds up the entire process. The QAR system relies on the ability of VAX Notes to organize input and replies. Further, it takes advantage of *keywords*, a feature of VAX Notes that allows you to group notes that concern a particular subject or do not have other attributes (such as title, author, or time of entry) in common. In this respect, it functions in a similar way to CMS groups.

A manageable QAR system using VAX Notes will have the following basic characteristics:

- A conference that is 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* (built-in feature of VAX 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 may include collecting responses from field test sites and adding them as topics in the conference. This person must also add appropriate keywords to the notes.

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 necessary 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 person acting as system monitor sends the problem report to the maintainer assigned the task of responding to the report. The response will take the form of a standard VAX 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 VAX Notes CREATE KEYWORD command. As stated previously, the person acting as system monitor adds the appropriate keywords (using the ADD KEYWORD command) to the notes. Once added, maintainers can retrieve all the notes to which a particular keyword has been added by using the VAX Notes commands: DIRECTORY/KEYWORD, SAVE/KEYWORD, or PRINT/KEYWORD. The following example results in a listing of those notes to which the keyword JONES (one of the maintainers) was added.

```
Notes> DIR /KEYWORD=JONES
```

Table 3–2 shows some categories and examples of potential keywords. Your project can use this feature to tailor its own cross-referencing system.

Table 3–2: Keywords for a Sample QAR 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

These keyword examples show that the organizational framework is easily tailored to specific project needs. By using the built-in features of VAX Notes, along with structured response procedures by the team, an effective QAR system can be established.

Using VAX DATATRIEVE

An alternative to using VAX Notes for a QAR system is to use VAX DATATRIEVE, VAX Forms Management System (FMS), and command procedures.

Benefits of VAX DATATRIEVE include the following:

- An English-like query language that allows easy access to data stored in VAX Rdb/VMS (Relational Database Management System), VAX DBMS (Database Management System), and VAX RMS files.
- An Application Development Tool that provides a simple, interactive means of defining record formats, VAX RMS files, and VAX DATATRIEVE procedures.
- A Report Writer that allows you to create formatted reports on any selection of data.
- A text editor (callable EDT) for easily changing record definitions or correcting syntax errors.
- Support for the forms management facilities of VAX FMS and VAX Terminal Data Management System (TDMS), which allow you to format the screen for data display or collection purposes.

3.2.8 Final Steps in a Project

The final steps in a project are most likely not the end of work on the software application. Maintenance may be an integral part of the application; in addition, future releases may follow. The procedures described in the software methodology should help people in the future to effectively maintain the software. The last few steps of a product release will contribute to this same goal.

Prepare Final Build

The procedure for the final build is really no different than those for earlier builds that marked major milestones in the product's life cycle. After completing 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 can store pertinent build information in the same CMS library. Use this library to provide information that will ease the work for those who follow. For example:

- Names of related classes in different CMS libraries
- List of product dependencies—which 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 DTM tests and results will certainly benefit the people who maintain or upgrade the project. SCA library information is useful, but can be rebuilt if necessary from the sources in CMS during compilation.

Permanent Storage

The sources for the code and documentation, along with any other information (for example, online HELP files), should be permanently stored in such a way as to prevent loss of the master files.

Chapter 4

Using Tools on a Software Project

This chapter describes how to use the VAXset tools in a coordinated fashion as they might be used on a hypothetical project. The examples used in this chapter are based on the so-called Transliteration project, and this chapter shows how an imaginary project team—the Transliteration project team—organizes its code libraries and builds and tests the transliteration software, using the VAXset tools, the VMS debugger, a VAX processor running the VMS operating system, and multiple VAX languages. The goal of these examples is to present techniques that will help you make the best use of the VAXset 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 broadly applicable to the problems associated with larger projects.

4.1 Setting Up Directories

The Transliteration application manipulates and substitutes text strings within a file. The project team, referred to throughout this book as the Transliteration project 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 will proceed efficiently, the team has planned a directory structure to support the tools and the project's storage and communication needs, shown in the following section.

4.1.1 Directory Structure

A project's directory structure should reflect the specific needs of the project along with the needs of the VAXset tools themselves. The project should also have a directory structure that organizes and eases access to the files of the application for those within the project. It is a good idea to provide public directories to allow those outside of the project to have access to nonrestricted files. It is also a good idea to create and maintain build directories for both the project and individual developers.

Table 4–1 shows the directories and libraries created by the transliteration project team. (Note that 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.

Table 4–1: Project's Directories

Directory	Logical Name	Function
General Directories		
[TRN.PUBLIC]	TRN_PUBLIC	Stores information for public users, for example, printable documentation.
[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.

Table 4–1 (Cont.): Project's Directories

Directory	Logical Name	Function
DTM 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 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.

4.1.2 Creating Directories and Libraries

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

Two steps are necessary to create a project library:

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

The Transliteration team implements protection through 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
```

Note that the CMS CREATE LIBRARY command uses a /REFERENCE_COPY qualifier 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 will put a copy of the new generation into the reference copy area, while deleting the previous version of the element from your area. (Section 3.1.3.2 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 DTM library in the same way. In all these examples, the CREATE LIBRARY command performs an implicit SET LIBRARY command, so that 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 DTM commands that access the library:

```
$ CMS SET LIBRARY TRN_CMS_DOC
```

4.1.2.1 Restricting Access with ACLs

After creating a library, the team may 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 fine-tune 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. Further, 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 *VMS DCL Concepts Manual* for a detailed explanation of protection procedures, user identification codes (UICs), and access control lists (ACLs). The *VMS Access Control List Editor Manual* contains information on using the ACL Editor, a VMS utility used to create and maintain ACLs. For more

information on CMS library security, see the *Guide to VAX DEC/Code Management System.*

4.1.2.2 Access Control for Large Projects

Because of the large number of people needing access to CMS sources, UIC mechanisms fall short of meeting all the needs of a larger project. To address the need to fine-tune the protection required in CMS, CMS provides its own ACL mechanism. Whereas VMS ACLs can protect access to VMS 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 VMS ACLs; however, they can give you greater control over your CMS libraries by allowing you to control access to the complete scope of CMS objects, and not just files and directories.

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

This section gives examples of creating ACLs for both CMS library elements and commands. In some cases, it may not be necessary to have ACLs on both elements and commands; how you design your protection mechanism depends on what your project needs are. For the purposes of showing examples of CMS ACLs, though, 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.

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

1. To establish four groups of users in the system rights list, each of whom having his or her own specific types of access to CMS objects. They are holders of 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/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 commands. This RED period helps the project leader prepare the PROJECT product root for a system build.
2. To first grant CONTROL access to the ACLs on 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, discussed later).
3. To create a mechanism that specifies 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. Conversely, 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. Thus, one simple method to secure all elements in a CMS library, for example, 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. The process of executing ACL commands from this point on, therefore, 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 ACLs to existing elements that have no ACLs.

6. To deny everyone access to the CREATE_ELEMENT and REPLACE commands when the project enters the RED period. This goal involves placing ACLs on commands, as opposed to simply library elements.
7. To use a CMS Event Notification ACE to cause the holder(s) of the CONFIG_MANAGER identifier to receive mail notification of any REPLACE command executed while the project is in its RED period.

At the end of this section, a brief discussion of the use of BYPASS is included.

NOTE

When creating CMS ACLs, take care in preparing the sequence of ACEs. As with VMS 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.

The sections that follow show how to address each of these goals.

Granting CONTROL Access to Library Elements

As mentioned earlier, 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. In the following command, the holder of CONFIG_MANAGER is given the following access types to 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 command:

```
$ CMS SET ACL/OBJECT=LIBRARY ELEMENT_LIST /ACL=(IDENTIFIER=CONFIG_MANAGER,-
  OPTIONS=DEFAULT,ACCESS=CONTROL+CREATE+DELETE+MODIFY+REPAIR+-
  REPLACE+RESERVE+UNRESERVE)
```

Specifying a Default ACE for Library Elements for 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 the sample command given in this section, holders of PROJECT_SOURCE are granted the following access types to the library element list:

CREATE
DELETE
MODIFY
REPAIR
REPLACE
RESERVE
UNRESERVE

The command to do this follows:

```
$ CMS SET ACL/OBJECT=LIBRARY ELEMENT_LIST /ACL=(IDENTIFIER=PROJECT_SOURCE,-
/OPTIONS=DEFAULT/ACCESS=CREATE+DELETE+MODIFY+REPAIR+REPLACE+RESERVE+UNRESERVE)
```

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 command grants these access types. This command uses the /DEFAULT qualifier to set this ACL to be the default for any new elements added to the CMS library.

```
$ CMS SET ACL/DEFAULT/OBJECT=ELEMENT *.* /ACL=(IDENTIFIER=PROJECT_SOURCE_READ,-
  ACCESS=ACCEPT+ANNOTATE+FETCH+REVIEW)
```

Copying CMS ACLs to Existing Objects

The ACLs shown in the previous examples will affect only those elements created after the ACL commands are issued. To have those ACLs apply to elements that existed previously, you should issue the following ACL command, which places those ACLs on previously existing library elements:

```
$ CMS SET ACL/DEFAULT/OBJECT=ELEMENT *.*
```

Placing ACLs on CMS Commands

The previous examples dealt with ACLs placed on library elements. The example in this section shows placing ACLs on 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 has to check when a command is issued. (CMS needs to check only the ACL on the command, rather than each ACL on each element.) This can result in improved performance and simplified maintenance. One of the goals of the Transliteration project team is to have a RED period during which the libraries must become stable in order to prepare for a system build. One way to achieve this is to deny access to the REPLACE and CREATE_ELEMENT commands during that RED period. The following command shows placing ACLs on REPLACE and CREATE_ELEMENT:

```
$ CMS SET ACL/OBJECT=COMMAND REPLACE -
_$ /ACL=(IDENTIFIER=PROJECT_SOURCE+PROJECT_RED,ACCESS=NONE)
$ CMS SET ACL/OBJECT=COMMAND CREATE_ELEMENT -
_$ /ACL=(IDENTIFIER=PROJECT_SOURCE+PROJECT_RED,ACCESS=NONE)
```

Using Event Notification ACLs

An Event Notification ACL can be used to send mail notification to someone when a specified event occurs. The Transliteration project team needs to create an Event Notification ACL that sends mail to CONFIG_MANAGER when anyone attempts to issue the REPLACE command when the Transliteration project is in its RED period. The following command accomplishes this:

```
$ CMS SET ACL/OBJECT=COMMAND REPLACE -
_$ /ACL=(NOTIFY=CONFIG_MANAGER,ACCESS=EXECUTE,IDENTIFIER=[*]+PROJECT_RED)
```

CMS allows you to write your own event handling program to take a specific action when an event occurs. See the *Guide to VAX DEC/Code Management System* for more details.

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 would not have a reason to access the product root.

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 that 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 would carry 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 group of users who are the holders of the PROJECT_SOURCE identifier. Since 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, thereby allowing normal development work to resume. To unlock the product root, the configuration manager removes the PROJECT_RED identifier from the system rights list. Following this procedure, all processes on the system lose the identifier.

As these examples demonstrate, protection procedures allow you to restrict access to the library as a whole, or parts of the library, or to CMS commands themselves, thereby yielding greater control during builds. Additionally, holding BYPASS or the existence of an ACE granting BYPASS allows someone to replace or unreserve elements on behalf of someone else. Holding a process BYPASS privilege circumvents CMS access checking, allowing your system manager or someone with system privileges to correct your ACLs if you have inadvertently locked yourself out of being able to correct them yourself.

As always, in order to further control library access, you will need to maintain consistent team working procedures. For more information on CMS security features, see the *Guide to VAX DEC/Code Management System*.

4.2 Maintaining CMS Source Libraries

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

4.2.1 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 4.1.2 for the procedures that set up this CMS library).

To invoke CMS and place these documents into the library, type the following commands:

```
$ CMS
CMS> CREATE ELEMENT SPEC.MEM
_Remark: Functional Spec
%CMS-S-CREATED, element TRN_DISK:[TRN.CODE_CMSLIB]SPEC.MEM created
```

Once inserted into the CMS library, the file is referred to as an *element*. When a developer places a file into a CMS library with the CMS CREATE ELEMENT command, that file becomes the first *generation* of the element. As developers modify the file, CMS will create subsequent generations of the element.

NOTE

In the above example command, a formatted DIGITAL Standard Runoff (DSR) file, or *.MEM* file, was inserted into the library. Although CMS can store both source and formatted DSR files (.RNO and .MEM, respectively), you may want to consider storing only the .RNO versions if disk space is a problem. If needed, you can rebuild the .MEM file using the .RNO file.

4.2.2 Modifying Elements

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

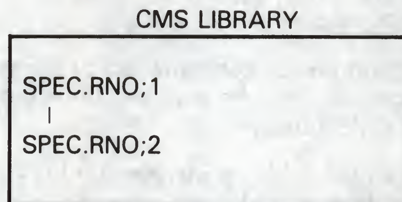
```
LSE> RESERVE SPEC.RNO
_Remark: Modify tasks system -- expand discussion of capabilities
generation 1 of element SPEC.RNO reserved
LSE>
```

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

```
LSE> REPLACE
_Remark: Functional Spec changed to add new functionality
generation 2 of element SPEC.RNO created
LSE>
```

The CMS library now contains two generations of the element. Figure 4–1 shows the contents of the CMS library.

Figure 4–1: CMS Library with Two Generations



4.2.3 Concurrent Access

CMS allows more than one user to access a file at the same time. CMS notifies you when you try to access a file when another user is working on that file. 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       29-FEB-1988 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 file after being notified that Mary had previously done so. Meanwhile, Mary replaces her copy. Later that day, she continues her work, developing another generation of the design specification.

When John replaces his copy, he indicates that his version is a variant, as follows:

```
$ CMS REPLACE TRANS_DESIGN.RNO/VARIANT=A  "Expand the design of user interface"
Element TRANS_DESIGN.RNO currently reserved by:
    (1)   Mary       1       29-FEB-1988 08:37:28 "Reserved for update"
Proceed?  [Y/N] (N): Y
%CMS-S-GENCREATED, generation 2A1 of element TRN_DISK:[TRN.CODE_CMSLIB]TRANS_DE
SIGN.RNO created
```

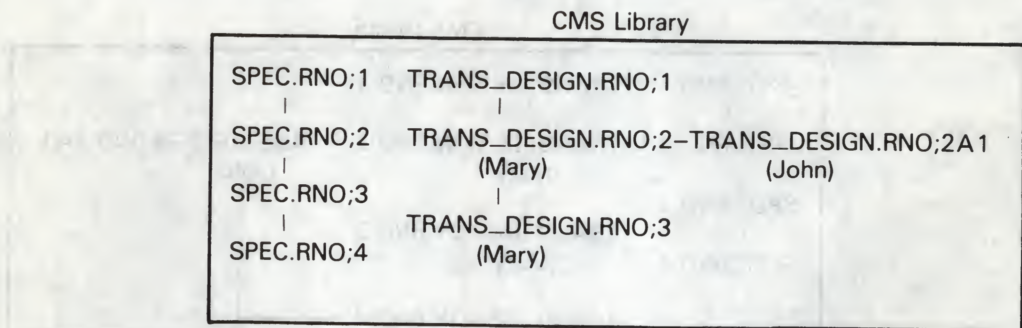
CMS prompts for confirmation to inform John that another version has been created since he reserved the element. Figure 4–2 shows the contents of the CMS library.

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, John would issue the following commands:

```
$ CMS RESERVE TRANS_DESIGN.RNO/MERGE=2A1 "Update on user input option"
%CMS-I-MERGECOUNT, 2 changes successfully merged with no conflicts
%CMS-S-RESERVED, generation 2 of element TRN_DISK:[TRN.CODE_CMSLIB]TRANS_DESIGN
.RNO reserved and merged with generation 2A1
```

Before John replaces the merged element, he verifies that the merged file is as expected. If the file is a code module, he would compile, link, and execute the code before actually replacing it.

Figure 4–2: Variants in a CMS Library

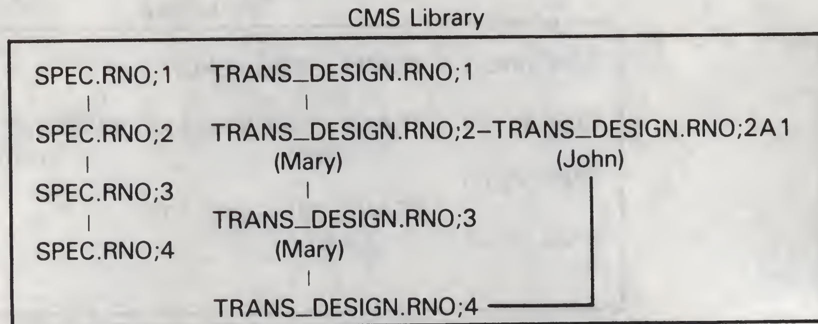


In this case, no conflicts occurred between the two versions. Had there been any, CMS would have flagged these, and the reserved element would have needed editing to resolve these conflicts. John then replaces the element with the following command:

```
$ CMS REPLACE TRANS_DESIGN.RNO "Merge generation 2A1"
%CMS-S-GENCREATE, generation 2 of element TRN_DISK:[TRN.CODE_CMSLIB]
TRANS_DESIGN.RNO created
```

Figure 4–3 shows the contents of the CMS library reflecting the merge.

Figure 4–3: Merging with CMS Libraries



For more information on merging variants of CMS elements, see the *Guide to VAX DEC/Code Management System*.

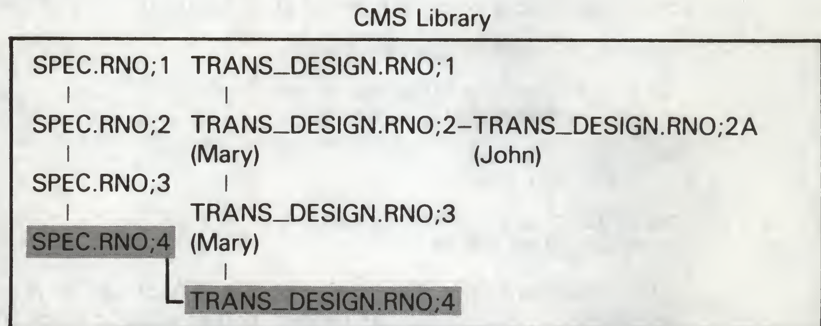
4.2.4 Creating Classes

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. The following commands show how to create a class and begin inserting generations into that class.

```
$ CMS
CMS> CREATE CLASS FIRST_STATUS
_Remark: Specs for first status report
%CMS-S-CREATED, class TRN_DISK:[TRN.CODE_CMSLIB]FIRST_STATUS created
CMS>INSERT GENERATION SPEC.RNO FIRST_STATUS "for first status report"
%CMS-S-GENINSERTED, generation 1 of element TRN_DISK:[TRN.CODE_CMSLIB]SPEC.RNO
inserted into class TRN_DISK:[TRN.CODE_CMSLIB]FIRST_STATUS
CMS> INSERT GENERATION TRANS_DESIGN.RNO FIRST_STATUS "for first status report"
%CMS-S-GENINSERTED, generation 1 of element
TRN_DISK:[TRN.CODE_CMSLIB]TRANS_DESIGN.RNO inserted into class
TRN_DISK:[TRN.CODE_CMSLIB]FIRST_STATUS
```

Figure 4–4 shows the contents of the CMS class FIRST_STATUS after the files have been inserted.

Figure 4–4: Classes in a CMS Library



4.2.5 Retrieving Class Contents and Preparing a Build

Developers can retrieve the contents of the class using either CMS FETCH or MMS. While CMS FETCH can fetch an element generation into the current default directory, MMS can fetch the element generation and also allow you to invoke RUNOFF (and other actions) on the files after fetching. MMS thus automates the building of documents and software systems (see Section 4.8 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 in order 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.

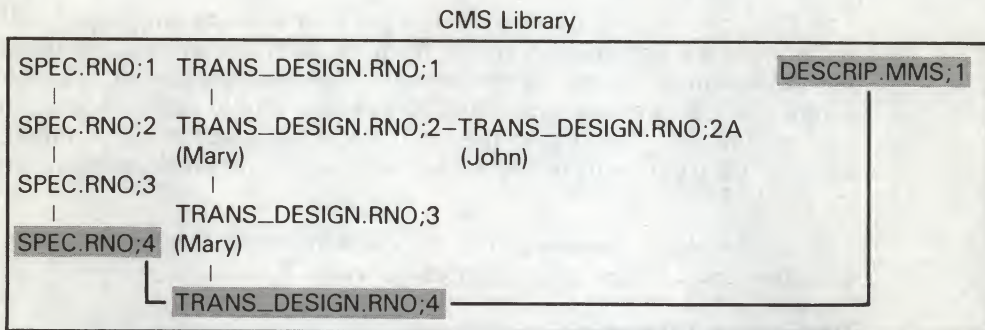
CMSFLAGS is a default macro provided by MMS that designates that the CMS FETCH command on an MMS action line should fetch the most recent generation of an element on the main line of descent. This macro can be redefined to indicate a specific element generation, or (as in our example) an element generation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

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

```
$ MMS/CMS
```

Without the /CMS qualifier, 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. Thus, if it becomes necessary in the future to rebuild these specifications, the team will be able to find the description file quickly. Figure 4–5 shows the final relationships in the CMS library.

Figure 4–5: Description File as Part of a CMS Class



4.3 Setting Defaults with a LOGIN.COM File

Another early task of the team is to establish work conditions and default values, in effect 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 a project's logical names.
- Set CMS, DTM, and SCA libraries.
- Set a source list to be used by LSE when accessing sources.

4.3.1 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 allows you to customize the environment file to redefine tokens, placeholders, or language definitions as well.

Redefining Tokens

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

1. Issue the GOTO BUFFER/CREATE command followed by a new buffer name to set up an empty buffer.
2. Issue the EXTRACT TOKEN, EXTRACT PLACEHOLDER, or EXTRACT LANGUAGE command, followed by the name of the token, placeholder, or language element you want modified, followed by the /LANGUAGE=language qualifier.
3. Edit the definition of the selected token, placeholder, or language.
4. Issue the DO command to execute the new definition.

NOTE

> If you want to save these modifications for future use, use the SAVE ENVIRONMENT command, described later in this section.

For example, to add a BEGIN/END construct to the default definition of a Pascal WHILE statement, use the EXTRACT command to begin modifying the default construct. After creating an empty buffer, issue the following command:

```
LSE> EXTRACT TOKEN WHILE/LANGUAGE=PASCAL
```

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

Figure 4–6: Extracting a Token

```
DELETE TOKEN WHILE -
    /LANGUAGE=PASCAL
DEFINE TOKEN WHILE -
    /LANGUAGE=PASCAL -
    /DESCRIPTION="WHILE expression DO statement" -
    /TOPIC="Statements WHILE"

    "WHILE %{expression}% DO"
    "     %{statement}%"

END DEFINE
[End of file]

 Buffer PASCAL.LSE                                    Write    Insert      Forward

Creating file TRN_DISK:[USER.WORK]PASCAL.LSE;
```

ZK-5979-HC

Edit the token by adding BEGIN and END statements, and the comment {WHILE} on the END statement. Once you have the definition the way you want it, press CTRL/Z to get the LSE> prompt. Then issue the DO command to execute the new definition. Now, each time you use the WHILE token in Pascal during the current editing session, LSE provides the new definition.

Saving Modified Definitions

You can save this modified WHILE definition so that it is available in subsequent work sessions. You need to create an environment file that can store the WHILE 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, issue 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

```
DELETE TOKEN WHILE -
    /LANGUAGE=PASCAL
DEFINE TOKEN WHILE -
    /LANGUAGE=PASCAL -
    /DESCRIPTION="WHILE expression DO statement" -
    /TOPIC="Statements WHILE"

    "WHILE %{expression}% DO"
    "     BEGIN"
    "     %{statement}%"
    "     END {WHILE}"

END DEFINE
[End of file]




 Buffer PASCAL.LSE                                 Write     Insert        Forward
LSE Command> SAVE ENVIRONMENT PASCAL.ENV
Creating file TRN_DISK:[USER.WORK]PASCAL.LSE;
```

ZK-5980-HC

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

```
$ LSEDIT/ENVIRONMENT=device:[directory]filename.ENV
```

In order to make the environment file available to the entire team, you can 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. The environment file provides coding standards, ensuring consistency among all the team's developers. Any further modifications to the environment file will supplement the team's standards.

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 qualifier every time they invoke LSE. To do this, they should add the following command to their LOGIN.COM files:

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

4.3.2 Command Files

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 can then execute.

Example 4–1 shows a LOGIN.COM file from one of the Transliteration project's developers.

Example 4–1: Sample LOGIN.COM File

```
$ DEFINE LSE$ENVIRONMENT          TRN_DISK:[TRN.COMS]PASCAL.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, TRN_CMS_CODE
$ 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
    .
    .
    .
```

This file makes available the environment file, PASCAL.ENV, and executes the logical name command file, TRN_LOGICALS.COM. It also sets this developer's CMS, DTM, 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 (user's default directory, project build area, and the 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 inadvertent changes to files when developers are using LSE and SCA to browse through different modules. As a result of this command, developers must reserve a module from CMS before making any changes, thereby 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$INIT. The file specification for the Analyzer initialization file is assigned to the logical name PCAA$INIT.

The logical name definitions file (TRN_LOGICALS.COM), accessed by the LOGIN.COM file, defines the various directories and libraries on the project. These logical names can be used in place of the full directory or library specifications. Example 4–2 shows the contents of this file.

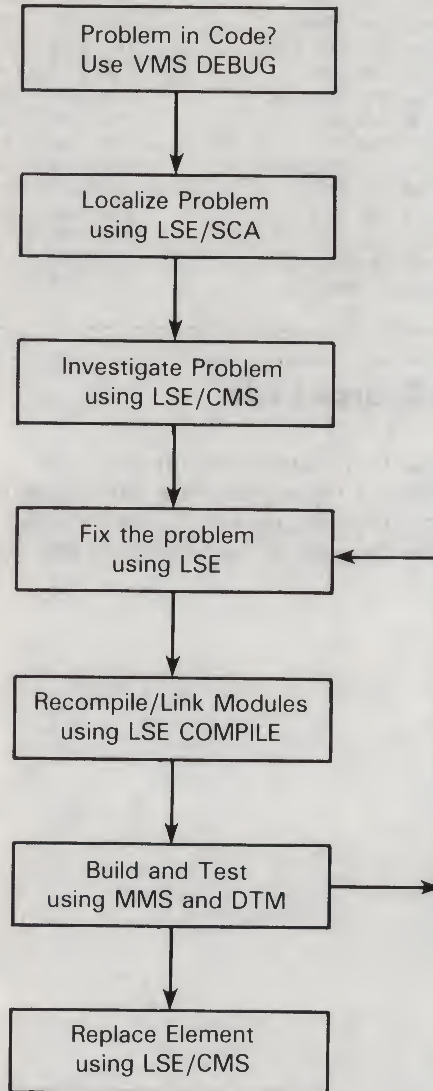
Example 4–2: Project Logical Definitions File

```
$ 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]
```

4.4 Debugging Source Code

This section, along with the ones that follow, shows how the team addresses a typical software development problem: debugging and editing source code. Figure 4–8 shows the main steps developers may follow, as well as the uses of the VMS tools that support these steps.

Figure 4–8: Steps to Implement Code



The Transliteration project has moved into active coding, creating an application named TRANSLIT. As a result, the team has already accumulated a body of code consisting of multiple modules stored in a CMS library, supplemented by a read-only reference copy area. As an aid to analyzing their code, the team has set up a project-wide SCA library; all developers also use local SCA libraries as necessary for small tasks.

A typical use of the TRANSLIT utility takes this form:

```
TRANSLIT file-spec  original-characters  [replacement-characters]
```

The following example replaces all lowercase letters (a to z) with their uppercase counterparts (A to Z).

```
TRANSLIT test.dat/OUT=test.out "a-z" "A-Z"
```

A recent enhancement to the TRANSLIT utility provides another way to describe the original characters that are to be replaced. A hyphen (-) at the beginning of the original character string causes the characters in quotation marks to represent all the characters that are not in the string. For example, "-A-Z" represents all characters except for the uppercase letters A to Z. In effect, a user can designate the original characters by a kind of complementary description.

Preliminary testing showed that the new enhancement worked as intended. The developer then returned the changed modules to the CMS library. Development continued, but subsequent testing using the full DTM test system showed that the code had regressed. In the process of enhancing the code, some of the original functionality was lost. Now, the TRANSLIT utility works properly only when the new, complementary description is used. When executing the previous simple test case, the TRANSLIT utility translated all lowercase letters to uppercase letters, but deleted all other characters, including line breaks.

Mary has been assigned to the problem, but does not know where in the code the problem exists. Because the application has already been compiled and linked with the VMS Debugger (debugger), she can begin to try to localize the problem by stepping through the code with the debugger. She has a sample test file that the TRANSLIT utility should modify using the following command:

```
$ TRANSLIT test.dat/OUT=test.out "a-z" "A-Z"
```

Figure 4–9: Problem Source Code from the Debugger

```
- SRC: module COPY_FILE -scroll-source
   26:

   27: {   COPY FILE copies the input file to the output file, transliterating
   28:     characters as it goes.  }
   29:
-> 30: [GLOBAL] PROCEDURE copy_file ( VAR in_file, out_file : TEXT;
   31:                                table : trans_table );
   32:
   33:     VAR
   34:         in_line : VARYING [max_record_len] OF CHAR;
   35:         out_line : PACKED ARRAY [1..max_record_len] OF CHAR;
- OUT -output





- PROMPT -error-program-prompt
DBG> Step/Into
%DEBUG-I-DYNMODSET, setting module COPY_FILE
DBG> 
```

ZK-5981-HC

Mary begins by stepping into the COPY_FILE procedure (a procedure in the TRANSLIT code). She can then watch characters move from the input file to the output file. Figure 4–9 shows her location in the code at this point.

After reading the first line from the input file, she examines the IN_LINE variable, in this case, "aBcDeFgHiJk". These characters are incorrectly translated to "ACEGIK" in the output file (they should translate to "ABCDEFGHIJK"). To investigate this symptom further, Mary steps through the translation loop the first time, and sees that "A" is written to OUT_LINE. She verifies this by using the EXAMINE command as follows:

```
DBG> EXAMINE out_line[out_index]
```

Figure 4–10: The EXAMINE Command in the Debugger

```
- SRC: module COPY_FILE -scroll-source------------------------------------------
    73:               THEN
    74:                   BEGIN
    75:                   WRITELN (out_file, SUBSTR (out_line, 1, out_index));
    76:                   out_index := 0;
    77:                   END;
    78:               END;
->  79:           copying := NOT table[code].compress;
    80:           END {WHILE};
    81:       END;
    82:
    83: END.
- OUT -output-------------------------------------------------------------------
COPY_FILE\IN_LINE:       'aBcDeFgHiJk'
COPY_FILE\OUT_LINE[1:1]:         'A'




- PROMPT -error-program-prompt--------------------------------------------------
DBG> Step
DBG> EXAMINE out_line[out_index]
DBG>
```

ZK-5982-HC

Figure 4–10 shows the results of her EXAMINE command.

The second time through the loop, the code fails to take the branch that writes the character to OUT_LINE. Mary examines the variable named CODE and finds that it translates to a value of 66. She verifies that this corresponds to the letter "B" by using the following command:

```
DBG> EVAL/DECIMAL 'B'
```

Mary then decides to examine the TABLE[CODE] variable. This tells her that TABLE[CODE].TRANS_VALUE has a value of 258. Since this value is greater than 255 (the upper limit for ASCII values), the code does not correctly branch to OUT_LINE. This information provides another piece to the puzzle.

Suspecting that the problem is near this point, Mary now leaves the debugger and enters LSE by issuing the following command:

```
DBG> EDIT/EXIT
```

Figure 4–11 shows the screen display as Mary issues this command.

Figure 4–11: Exiting the Debugger to LSE

```
- SRC: module COPY_FILE -scroll-source----------------------------------------
   67:              THEN
   68:                  BEGIN
   69:                  out_index := out_index + 1;
   70:                  out_line[out_index] := CHR (table[code].trans_value);
   71:                  END
-> 72:              ELSE IF table[code].trans_value = newline
   73:              THEN
   74:                  BEGIN
   75:                  WRITELN (out_file, SUBSTR (out_line, 1, out_index));
   76:                  out_index := 0;
   77:                  END;
- OUT -output-----------------------------------------------------------------
COPY_FILE\IN_LINE:      'aBcDeFgHiJk'
COPY_FILE\OUT_LINE[1:1]:        'A'
COPY_FILE\CODE: 66
66
COPY_FILE\TABLE[66]
    TRANS_VALUE:        258
    COMPRESS:   False
- PROMPT -error-program-prompt------------------------------------------------
DBG> EVAL/DECIMAL 'B'
DBG> EXAMINE table[code]
DBG> EDIT/EXIT
```

ZK-5983-HC

The EDIT/EXIT command places Mary in LSE at the same line of source code that she was viewing in the debugger. Mary is using a version of the application compiled and linked with the /DEBUG qualifier, built from sources in the project work area ([TRN.BLD_V1.WORK]). Because this directory is designated as read-only in Mary's LOGIN.COM file, LSE places a read-only version of the file into the LSE buffer. Now, from

within LSE, Mary has access to all the information previously stored in the project-wide SCA library.

To examine the TRANS_VALUE symbol, Mary uses the following SCA command:

```
LSE> GOTO DECLARATION TRANS_VALUE
```

SCA finds the file (in this case TYPES.PAS) that contains the declaration of TRANS_VALUE. LSE once again brings a read-only version of the file into a buffer. Figure 4–12 shows the display as it appears in the editing window.

Figure 4–12: GOTO DECLARATION in SCA

```
      first one is actually translated; subsequent ones are deleted.  }

   trans_table = ARRAY [code_value] OF RECORD
                                           trans_value : code_value;
                                           compress : BOOLEAN;
                                        END;

TYPE
   param_string = VARYING [256] OF CHAR;
 Buffer TYPES.PAS                          Read-only   Nomodify       Forward
         IF copying OR NOT table[code].compress
         THEN
             BEGIN
             IF table[code].trans_value <= 255
             THEN
                 BEGIN
                 out_index := out_index + 1;
                 out_line[out_index] := CHR (table[code].trans_value);
                 END
 Buffer COPYFILE.PAS                       Read-only   Nomodify       Forward
83 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]COPYFILE.PAS;1
1 occurrence found (1 symbol, 1 name)
59 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]TYPES.PAS;1
```

ZK-5984-HC

Now, Mary uses SCA to see how the TRANS_VALUE symbol is processed. She sees that TRANS_VALUE is of type CODE_VALUE. To learn more about the CODE_VALUE symbol, Mary positions the cursor on CODE_VALUE, and to display its declaration presses CTRL/D (that is, GOTO DECLARATION). The declaration shows that CODE_VALUE is defined to be 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. That value indicated that no translation was assigned to the character. To examine the TRANS_VALUE symbol further, Mary now uses the following SCA command to find all the places that TRANS_VALUE is assigned a value:

```
LSE> FIND trans_value/REFERENCE=WRITE
```

Figure 4–13 shows using the results generated by the FIND command.

Figure 4–13: Navigating Based on FIND Results

```
            END;
        FOR code := min_code TO max_code DO
            BEGIN
            IF table[code].trans_value = undef_code
            THEN
                BEGIN
                table[code].trans_value := replace_code;
                table[code].compress := TRUE;
                END;
            END;
 Buffer BUILDTABLE.PAS                               Read-only     Nomodify       Forward
  Symbol              Class        Module\Line             Type of Occurrence

TRANS_VALUE         component    BUILD_TABLE\76          write reference
                                 BUILD_TABLE\100         write reference
                                 BUILD_TABLE\107         write reference
                                 BUILD_TABLE\131         write reference

 Query 1   FIND TRANS_VALUE/REFERENCE=WRITE                                      Forward
1 occurrence found (1 symbol, 1 name)
4 occurrences found (1 symbol, 1 name)
137 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]BUILDTABLE.PAS;2
```

ZK-5985-HC

After examining a number of references to TRANS_VALUE, Mary decides that she can eliminate the problem by adding a FOR loop that originally had been part of the code. The new FOR loop must set the values in TABLE that correspond to characters that are not being translated. Mary speculates that, during the most recent enhancement work, the last developer may have used the original loop as a guide for the complement loop. In the process, he may have failed to reinsert the original loop back into the code.

To change the code, Mary needs to obtain a modifiable source file from CMS and edit it using LSE, described in Section 4.5.

4.5 Editing a Source File with LSE

LSE's integration with CMS allows Mary to reserve the source file directly from the current buffer in LSE, using the following command:

```
LSE> RESERVE
```

Note that LSE automatically reserves the current file from CMS.

Once the source module has been reserved from CMS, Mary can use the language-sensitive features of LSE to modify the code. To correct the error, Mary adds a FOR loop to properly set the values in TABLE for characters that are not being translated.

Mary modifies the code with LSE as follows:

1. Uses tokens to generate the FOR, BEGIN, and IF templates.
2. Fills in the loop-specific information.

Figure 4–14 shows the expanded FOR token.

Figure 4–14: Using a Token with LSE

```
        replace_code := del_code;
        FOR i := 1 TO orig_len DO
            BEGIN
            code := orig_vector[i];
            IF table[code].trans_value <> undef_code
            THEN
                signal_duplicate (code);
            IF i <= repl_len
            THEN
                replace_code := repl_vector[i];
            table[code].trans_value := replace_code;
            table[code].compress := compress AND (i >= repl_len);
            END;
        FOR %{control_var}% := %{value_expr}% %{TO | DOWNTO}% %{value_expr}% DO
            %{statement}%
        END;
    END {build_table};

END.
[End of file]
 Buffer BUILDTABLE.PAS                                 Write    Insert        Forward
137 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]BUILDTABLE.PAS;2
Generation 1 of element BUILDTABLE.PAS reserved
137 lines read from file TRN_DISK:[USER.WORK]BUILDTABLE.PAS;1
```

ZK-5986-HC

Figure 4–15 shows the loop partially completed: Mary has expanded the FOR token, filled in the specific code information (MIN_CODE and MAX_CODE), and expanded the BEGIN and IF tokens.

Figure 4–15: Expanding an LSE Token

```
            code := orig_vector[i];
            IF table[code].trans_value <> undef_code
            THEN
                signal_duplicate (code);
            IF i <= repl_len
            THEN
                replace_code := repl_vector[i];
            table[code].trans_value := replace_code;
            table[code].compress := compress AND (i >= repl_len);
            END;
        FOR code := min_code TO max_code DO
            BEGIN
            IF %{expression}%
            THEN
                %{statement}%
            %[ ELSE %{statement}% ]%;
            %[statement_list]%...
            END;
        END;
    END {build_table};
 Buffer BUILDTABLE.PAS                         Write    Insert        Forward
137 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]BUILDTABLE.PAS;2
Generation 1 of element BUILDTABLE.PAS reserved
137 lines read from file TRN_DISK:[USER.WORK]BUILDTABLE.PAS;1
```

ZK-5987-HC

Figure 4–16 shows the completed FOR loop: Mary has filled in the EXPRESSION and ASSIGNMENT statements, and deleted the ELSE and STATEMENT_LIST placeholders.

Figure 4–16: Completing Changes to Code

```
          IF table[code].trans_value <> undef_code
          THEN
             signal_duplicate (code);
          IF i <= repl_len
          THEN
             replace_code := repl_vector[i];
          table[code].trans_value := replace_code;
          table[code].compress := compress AND (i >= repl_len);
          END;
       FOR code := min_code TO max_code DO
          BEGIN
          IF table[code].trans_value = undef_code
          THEN
             table[code].trans_value := code;
          END;
       END;
    END {build_table};

END.
[End of file]
 Buffer BUILDTABLE.PAS                         Write    Insert        Forward
137 lines read from file TRN_DISK:[TRN.BLD_V1.WORK]BUILDTABLE.PAS;2
Generation 1 of element BUILDTABLE.PAS reserved
137 lines read from file TRN_DISK:[USER.WORK]BUILDTABLE.PAS;1
```

ZK-5988-HC

Having modified the code, Mary now wants to see if her changes were effective. To do this, she compiles the modified module directly from LSE to make sure that it compiles without error before attempting an updated build. The following LSE command allows Mary to compile and review any errors:

```
LSE> COMPILE/REVIEW
```

In this case, no errors appear. Mary then decides to perform a local build and test her program as part of the full system before returning the changed module to the CMS library.

4.6 Compiling and Linking a Modified File

Mary needs to compile and link the full application, including the modified module, using MMS. In this case, she wants to be sure that the application build will execute as intended, and that the DTM test set no longer shows any regressive effects. (See Section 4.8 for an explanation of the description file that executes the project build.)

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

```
$ MMS/CMS WORK_BUILD
```

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

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

```
$ CMS REPLACE BUILDTABLE.PAS "Fixed translation bug"
```

4.7 Setting Up the Test System

The Transliteration team needs to set up its test system. Larger projects benefit from the test system being set up early in the project, as explained in Section 3.2.3. The steps that the Transliteration team takes in setting up a test system are listed and explained in this section.

This section is divided into two topics: setting up noninteractive tests, and setting up interactive tests.

Noninteractive tests, described first, are adequate for testing software that does not have a terminal-oriented or menu interface. The TRANSLIT software described up until now fits this category, since the application simply accepts an input text file and gives you an output file with substituted characters.

However, to demonstrate the interactive testing capabilities of DTM Version 2.0, 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 4.7.2.

4.7.1 Setting Up a Noninteractive Test

The first step in setting up the test system is to create the following three separate storage areas. These were set up earlier, but they are repeated here:

A DTM library for test results	TRN_DISK:[TRN.DTMLIB]
A CMS library for DTM test files	TRN_DISK:[TRN.DTM_CMSLIB]
Two subdirectories for test data	TRN_DISK:[TRN.DTM_DATA] and TRN_DISK:[TRN.DTM_DATA_CMS]

The following list gives the additional steps the team needs to take to set up a noninteractive test:

- Set up the test collection prologue file.
- Set up the test collection epilogue file.
- Establish the default test template and benchmark file directory.
- Create the test template file.
- Create the test description.
- Set up DTM variables.
- Insert the test template file into the CMS library.

The following sections describe each of these steps in detail.

Setting up the test collection prologue file

The test collection prologue file is associated with one or more specified test descriptions and runs just before the test template file runs. Typically, the test prologue file is used as a setup file to establish any special environment the test requires. The collection prologue file, which the Transliteration project team creates, does two things:

1. Tests the DTM variable USE_PCA (defined and described later in this section) to determine whether or not to process the prologue file.

2. Defines the Collector initialization file according to the DTM variable USE_PCA_INIT_FILE, whose value is also defined and described later in this section.

Running the PCA Collector during a test is especially useful for determining which code paths are being exercised by the tests themselves. Section 4.9 shows an example of analyzing coverage data after tests have been executed.

The collection prologue file is shown in Example 4–3.

Example 4–3: Sample Collection Prologue File — COLLECTION_PROLOGUE.COM

```
$! Collection prologue file for running the Collector in batch mode
$!
$  SET VERIFY
$!
$  TRANSLIT:==$'F$LOGICAL("TRANSLIT")'
$!
$! Test DTM 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. The following command specifies the prologue file in Example 4–3 as the default (you do not need to have an existing prologue file to issue this command):

```
$ DTM SET PROLOGUE TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_PROLOGUE.COM
```

Note that the DTM SET PROLOGUE command requires full file specifications for the prologue file. In this example, the prologue file exists in the DTM CMS library.

Setting up the test collection epilogue file

Like the prologue file, the epilogue file is associated with one or more specified test descriptions and runs just after the test template file runs. Typically, the epilogue file is used to perform filtering and cleanup procedures. For example, the epilogue file can edit the results file to remove run-specific data, such as time stamps, or run information on the amount of memory used. The epilogue file can also be used to send mail notification to anyone on the project team when the tests are completed, giving test results, for example.

The epilogue file, which the Transliteration project team creates, sends the results of the test TRANSLIT_TEST to the project leader, Jones. This epilogue file, shown in Example 4–4, also makes use of the DTM-provided symbol DTM$COLLECTION_NAME.

Example 4–4: Sample Collection Epilogue File — COLLECTION_EPILOGUE.COM

```
$! 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 'dmt$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. The following command specifies the epilogue file in Example 4–4 as the default.

```
$ DTM SET EPILOGUE TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_EPILOGUE.COM
```

Note that the DTM SET EPILOGUE command also requires full file specifications for the epilogue file.

Establishing the default template and benchmark file directory

The following command establishes the default directory that DTM searches for the test template and benchmark files:

```
$ DTM SET TEMPLATE_DIRECTORY TRN_DISK:[TRN.DTM_CMSLIB]
$ DTM SET BENCHMARK_DIRECTORY TRN_DISK:[TRN.DTM_CMSLIB]
```

When you set the default directories for the template and benchmark files, DTM automatically looks for those files in those directories during the test. (DTM 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. By default, DTM assigns the values *test-name*.COM and *test-name*.BMK to those fields. It is good practice, though, to use the following guidelines:

- Let DTM 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. Additionally, 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 simply use the SET TEMPLATE_DIRECTORY and SET BENCHMARK_DIRECTORY commands instead.

Creating the template file for the test

The Transliteration team creates the template file, TRANSLIT_TEST.COM, shown in Example 4–5.

Example 4–5: TRANSLIT Test Template File — TRANSLIT_TEST.COM

```
$ 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 non-letters 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
```

Creating the test description

The following command creates the test description TRANSLIT_TEST, associating the template, prologue, and epilogue files shown earlier:

```
$ DTM CREATE TEST_DESCRIPTION TRANSLIT_TEST-
_$ /PROLOGUE=COLLECTION_PROLOGUE.COM/EPILOGUE=COLLECTION_EPILOGUE.COM
%DTM-I-DEFAULTED, benchmark file name defaulted to TRANSLIT_TEST.BMK
%DTM-I-DEFAULTED, template file name defaulted to TRANSLIT_TEST.COM
%DTM-S-CREATED, test description TRANSLIT_TEST created
```

Note that by default, DTM associates the template file TRANSLIT_TEST.COM and the benchmark file TRANSLIT_TEST.BMK with the test description.

Setting up the DTM variables

A DTM variable is a user-defined symbol or logical name that DTM 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 DTM variables:

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.

The following commands define these variables in the DTM library TRN_DISK:[TRN.DTMLIB]:

```
$ DTM CREATE VARIABLE TRANSLIT "TRN_DISK:[TRN.BLD_V1.WORK]TRANSLIT.EXE" -
_$ "Executable image to be used by test set"
$ DTM CREATE VARIABLE USE_PCA "FALSE" "Invoked as TRUE, gives PCA coverage"
$ DTM CREATE VARIABLE USE_PCA_INIT_FILE TRN_DISK:[TRN.COMS]MYINITFILE.PCAC"-
_$ "Provides default PCA init file"
```

Note that the DTM CREATE VARIABLE command accepts three arguments: the variable name, its value, and an associated remark.

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 TRN_DISK:[TRN.DTM_CMSLIB]:

```
$ CMS SET LIBRARY TRN_DISK:[TRN.DTM_CMSLIB]
$ CMS CREATE ELEMENT TRANSLIT_TEST.COM
```

The first comment sets the default CMS library, and the second command inserts the template file.

4.7.2 Setting Up an Interactive Test

To create an interactive test of the TRANSLIT variant that uses the menu-driven interface described in Section 4.7, the Transliteration project team uses the same library file, collection file, and epilogue file described earlier. Then, the team takes the following steps:

- Create the test description using the /RECORD qualifier.
- Invoke the Forms TRANSLIT variant application, FTRANSLIT, and run a sample session.
- Terminate the recording session.

The following sections describe each of these steps in detail.

Creating the Test Description, using /RECORD

As with noninteractive tests, the basic organizational unit within DTM is the *test description*. The /RECORD qualifier, used with the TEST_DESCRIPTION command, allows for the capturing of an interactive terminal session and produces a SESSION file as the test template. The SESSION file records all input and output. (The /PROLOGUE and /EPILOGUE qualifiers can be used with this command to invoke the collection prologue file shown in Section 4.7.1, provided you have identified their locations to DTM with the DTM SET PROLOGUE and SET EPILOGUE commands.)

Create the test description FTRANSLIT_TEST by entering the following command:

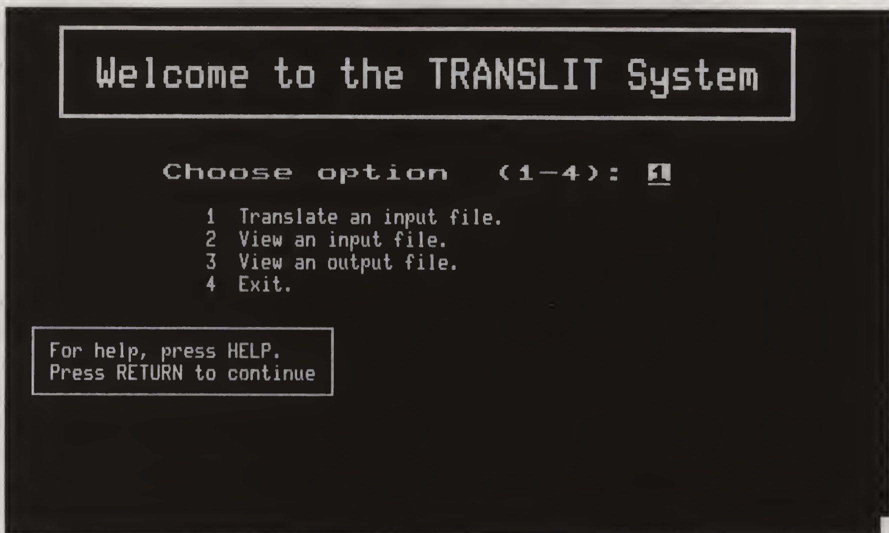
```
$ DTM CREATE TEST_DESCRIPTION /RECORD/PROLOGUE=COLLECTION_PROLOGUE.COM -
_$ /EPILOGUE=COLLECTION_EPILOGUE.COM FTRANSLIT_TEST
_Remark: DTM test of Forms TRANSLIT application
%DTM-I-DEFAULTED, benchmark file name defaulted to FTRANSLIT_TEST.BMK
%DTM-I-DEFAULTED, template file name defaulted to FTRANSLIT_TEST.SESSION
%DTM-I-BEGIN, your interactive test session is now beginning...
Type CTRL/P twice to terminate the session.
```

Invoking the Forms TRANSLIT application

Enter the following command to invoke the Forms TRANSLIT software:

```
$ RUN TRN_DISK:[USER]FTRANSLIT
```

The following screens show the Forms TRANSLIT application in its test session.



ZK-7447-HC

The user 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
```

ZK-7448-HC

The user selects option 1 to translate an input file, shown in the next screen.

```
Translate an Input File

Input File:          DISK1$:MYINPUTFIL.TXT
Input String:              A-Z0-9
Substitution string:       a-z9-0

Press RETURN to return to Main Menu
```

ZK-7449-HC

The user returns to the Main Menu, and then selects option 3, to 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
ABCDEfhgiy67583
aBcDeFgHiJk1234
AbCdEfGhIjK56789
tHIs iS ONly a tEst oF TRansLit
AAbbAbbbbCdEFHI
aaaAAAbb198976
abCdEfGhIjK56789
tHIs iS ONly a tEst oF TRansLit
AAbbAbbbbCdEFHI
```

ZK-7450-HC

The user returns to the Main Menu, and selects option 4 to exit.

Terminating the Recording Session

Terminate 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]FTRANSLIT_TEST.BMK;1
%DTM-S-RECORDED, test FTRANSLIT_TEST has been successfully recorded
       in file TRN_DISK:[TRN]FTRANSLIT_TEST.SESSION
%DTM-S-CREATED, test description FTRANSLIT_TEST created
```

At this point FTRANSLIT_TEST 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. The sample build at the end of this chapter relies on executing test collections in batch mode. Batch execution is preferred if you do not wish to tie up a terminal.

4.7.3 Verifying Your Test System

To check your DTM library to ensure that you have set your template, benchmark, prologue, and epilogue files correctly, you can use the DTM SHOW ALL command. The DTM SHOW ALL command displays the current directory specifications or CMS libraries containing these files, as well as the number of collections, test descriptions, groups, and variables in the library.

The following is an example of the DTM SHOW ALL command:

```
$ DTM SHOW ALL

 Description of DEC/Test Manager Library TRN_DISK:[TRN.DTMLIB]

 Default template  directory: TRN_DISK:[TRN.DTM_CMSLIB]  "Default template
directory"
 Default benchmark directory: TRN_DISK:[TRN.DTM_CMSLIB]  "Default benchmark
directory"
 Default collection prologue: TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_PROLOGUE.COM
""
 Default collection epilogue: TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_EPILOGUE.COM
""
 Number of collections:       3
 Number of test descriptions: 3
 Number of groups:            0
 Number of variables:         4
```

4.7.4 Changing Input for a Test

DTM provides the EXTRACT command to allow you to change the input that your 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 command 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 CREATE TEST_DESCRIPTION commands.

Later in this chapter, a section is included on using the DTM Review subsystem, Section 4.9. Part of that section discusses finding shortcomings in the test that was created and shown earlier in this chapter. The Transliteration project team needs to modify the test so that it tests for invalid input. Instead of rebuilding an interactive test from scratch, the project team uses the DTM EXTRACT command.

The following command shows an example of using the EXTRACT command to extract an INPUT file from FTRANSLIT_TEST.SESSION:

```
$ DTM EXTRACT FTRANSLIT_TEST.SESSION FTRANSLIT_INPUT.INP
```

The following command modifies the test description FTRANSLIT_TEST.SESSION, specifying FTRANSLIT_INPUT.INP as the INPUT file:

```
$ DTM MODIFY TEST_DESCRIPTION/INPUT=FTRANSLIT_INPUT.INP FTRANSLIT_TEST.SESSION
```

4.8 Building the System

The Transliteration team needs to build the following applications:

- Previous versions of their software
- A new stage in the project's development

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 a previous version 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.

In order to carry out these different tasks during the build, the following must be in place:

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

The MMS description shown in Example 4–6 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 (generally 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 would be appropriate during a project build using validated sources. This compilation would update the project-wide SCA library ([TRN.BLD_V1.SCALIB]).

A build from the local area would be 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 4–6 shows the complete MMS description file. This is followed by sections of the file with explanatory text.

Example 4–6: A Build Procedure Using an MMS Description File

```
!++
!                        BUILD Procedure
!--
!++++++++++++++++++
! FLAGS and MACROS
!------------------
! Create default flags and macros

DTM             = DTM
DELETE          = DELETE
DTMLIBRARY      = TRN_DTM                                ! Default DTM library
PCA             = FALSE
PCA_INIT_FILE   = TRN_DISK:[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
```

Example 4–6 Cont'd. on next page

Example 4–6 (Cont.): A Build Procedure Using an MMS Description File

```
TRACE           = TRACE
LIST            = LIST
DEBUG           = DEBUG
CHECK           = CHECK
OPTIMIZE        = NOOPTIMIZE
ANALYSIS_DATA   = ANALYSIS_DATA
PFLAGS          = /$(CHECK) /$(DEBUG) /NOOPTIMIZE /$(ANALYSIS_DATA)
CLDFLAGS        = / $(LIST)
MSGFLAGS        = / $(LIST)
DTMFLAGS        = /SUBMIT=(NOTIFY,LOG_FILE=[])-
                  /CLASS=(TEMPLATE:"$(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
TRANSLIT_MODULES= openfiles, types, buildtable, copyfile, expandstring, -
                  translit, translitc, translitm

!+++++++++++++
! RULES
!-------------
! Modify the Pascal rule to decide if analysis data files should be produced.

.pas.obj                ! a rule for compiling Pascal files
        IF USE_SCA .EQ. 0 THEN $(PASCAL) $(PFLAGS)/NOANALYSIS_DATA $(MMS$SOURCE)
        IF USE_SCA .EQ. 1 THEN $(PASCAL) $(PFLAGS) $(MMS$SOURCE)
        IF USE_SCA .EQ. 1 THEN $(SCA) LOAD $(MMS$TARGET_NAME)
        IF USE_SCA .EQ. 1 THEN $(DELETE) $(MMS$TARGET_NAME).ANA;
```

Example 4–6 Cont'd. on next page

Example 4–6 (Cont.): A Build Procedure Using an MMS Description File

```
.pas.pen                ! a rule for producing PASCAL pen files
        IF USE_SCA .EQ. 0 THEN $(PASCAL) $(PFLAGS)/NOANALYSIS_DATA $(MMS$SOURCE)
        IF USE_SCA .EQ. 1 THEN $(PASCAL) $(PFLAGS) $(MMS$SOURCE)
        IF USE_SCA .EQ. 1 THEN $(SCA) LOAD $(MMS$TARGET_NAME)
        IF USE_SCA .EQ. 1 THEN $(DELETE) $(MMS$TARGET_NAME).ANA;

!+++++++++++++++
! 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 "$(PCA)" .NES. "FALSE" THEN PCA_LINK = "/DEBUG=SYS$LIBRARY:PCA$OBJ"

! 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 -- used the same as work build, but has
!different target; allows .FIRST directive to cause Pascal not to generate .ANA files.

version_build   : work_build
                ! Version build completed at this point

!The translit executable

translit.exe : translit.olb($(translit_modules))  ! put modules into an object library
        $(link) $(linkflags) translit.olb/library/include=translit 'PCA_LINK'
        IF "$(DEBUG)" .EQS. "DEBUG" THEN $(link) $(linkflags)/debug/exe= -
        $(MMS$TARGET_NAME).debug translit.olb/library/include=translit
        set protection=w=re translit.*
```

Example 4–6 Cont'd. on next page

Example 4–6 (Cont.): A Build Procedure Using an MMS Description File

```
! 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 that follow create the needed .OBJ files which
! are the sources for the target TRANSLIT.EXE.

translit.obj       : translit.pas, openfiles.pen, types.pen
buildtable.obj     : types.pen
copyfile.obj       : types.pen
expandstring.obj   : types.pen
```

In the next three examples, the single description file in Example 4–6 has been broken into three main sections, each followed by explanatory text: Example 4–7, Flags and Macros; Example 4–8, Rules; and Example 4–9, Targets.

Example 4–7: Flags and Macros Section of the Description File

```
        !++
        !                       BUILD Procedure
        !--
        !++++++++++++++++++
        ! FLAGS and MACROS
        !------------------
        ! Create default flags and macros

❶ DTM             = DTM
    DELETE          = DELETE
    DTMLIBRARY      = TRN_DTM                            ! Default DTM library
    PCA             = FALSE
    PCA_INIT_FILE   = TRN_DISK:[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
```

Example 4–7 Cont'd. on next page

Example 4–7 (Cont.): Flags and Macros Section of the Description File

```
    TRACE          = TRACE
    LIST           = LIST
    DEBUG          = DEBUG
    CHECK          = CHECK
    OPTIMIZE       = NOOPTIMIZE
❷ ANALYSIS_DATA  = ANALYSIS_DATA
    PFLAGS         = /$(CHECK) /$(DEBUG) /NOOPTIMIZE /$(ANALYSIS_DATA)
    CLDFLAGS       = / $(LIST)
    MSGFLAGS       = / $(LIST)
❸ DTMFLAGS       = /SUBMIT=(NOTIFY,LOG_FILE=[])-
                     /CLASS=(TEMPLATE:"$(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
    TRANSLIT_MODULES= openfiles, types, buildtable, copyfile, expandstring, -
                      translit, translitc, translitm
```

Key to Example 4–7:

❶ This entire 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. 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 would then need to be included as part of the description file.

Members of the team may want to change the default names and macros on occasion. 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 DTM collection name (necessary if they have kept previous collections):

```
MMS/MACRO=(DTMCOLLECTION=NEW_NAME)
```

❷ The /ANALYSIS_DATA part of the PFLAGS macro causes the Pascal compiler to generate SCA data files (.ANA).

❸ The DTMFLAGS macro is used to submit a collection of tests (the tests themselves must already exist), and invokes the appropriate templates and benchmarks from their 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. Thus, by having previously set up a class called V1.0 in both the Code and DTM CMS libraries, the team can build and test V1.0 with the following command:

```
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") VERSION_BUILD
```

Note that this command does not update the SCA library because it designates the VERSION_BUILD option. This is described in the explanatory text of Example 4–9.

The DTMFLAGS macro also uses three DTM variables:

- /VAR=USE_PCA, which sets up a flag to specify coverage analysis for the test collection.
- /VAR=USE_PCA_INIT_FILE, which sets a default PCA Collector initialization file; at the same time, it lets the team choose a different initialization file when invoking MMS (command examples follow).
- TRANSLIT, which ensures that the DTM test collection is created based on the default work directory.

In order 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 DTM library. These variables were defined with the following DTM 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 DTM prologue file that sets up the PCA coverage and points to the PCA Collector initialization file.

- An image of the product to be tested, linked to invoke the PCA Collector; linking is done with the /DEBUG=SYS$LIBRARY:PCA$OBJ.OBJ qualifier. (The .FIRST part of Example 4–9 shows how this is done.)

An example follows of the PCA Collector initialization file. The DTM collection prologue file is shown in Section 4.7.

PCA Collector Initialization File: The initialization file that follows contains commands passed to the PCA Collector (the file itself is stored in the [TRN.COMS] directory). This file must contain the GO command as the last command; optionally, it may contain other Collector commands. The following example collects system service counts and measures test coverage by codepath over the entire program. In addition, it selects output verification and the collection of PC values from the VAX 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/PREVIOUS 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 and two correspondingly different MMS commands:

1. 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.)

   ```
   MMS/CMS WORK_BUILD
   ```

2. Build an executable application with a corresponding test collection and PCA test coverage for that test collection.

   ```
   MMS/CMS/MACRO=PCA=TRUE
   ```

   By adding to this command the additional qualifier, /MACRO=PCA_INIT_FILE=disk:[directory]filename.PCAC, other PCA initialization files can be designated, thereby providing the developer with greater flexibility.

Example 4–8: Rules Section of the Description File

```
!+++++++++++++
! RULES
!-------------
! Modify the Pascal rule to decide if analysis data files should be produced.
❶ .pas.obj              ! a rule for compiling PASCAL files
          IF USE_SCA .EQ. 0 THEN $(PASCAL) $(PFLAGS)/NOANALYSIS_DATA $(MMS$SOURCE)
          IF USE_SCA .EQ. 1 THEN $(PASCAL) $(PFLAGS) $(MMS$SOURCE)
          IF USE_SCA .EQ. 1 THEN $(SCA) LOAD $(MMS$TARGET_NAME)
          IF USE_SCA .EQ. 1 THEN $(DELETE) $(MMS$TARGET_NAME).ANA;

.pas.pen                     ! a rule for producing Pascal pen files
          IF USE_SCA .EQ. 0 THEN $(PASCAL) $(PFLAGS)/NOANALYSIS_DATA $(MMS$SOURCE)
          IF USE_SCA .EQ. 1 THEN $(PASCAL) $(PFLAGS) $(MMS$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 4-8:

❶ This section modifies the Pascal rule; this rule already exists since Pascal is an MMS-supported language. The result is to give the team the option of using SCA based on a specific compilation. For instance, for older versions, they may not want to generate analysis data to be loaded into the SCA library. On the other hand, if the build is assembling a new version, they are likely to want the SCA library updated.

NOTE

MMS provides built-in rules for using SCA and has a /SCA_LIBRARY qualifier to indicate that automatic SCA handling is desired. The rules shown here, however, are included to show how MMS rules can be manipulated by hand.

Example 4–9: Targets Section of the Description File

```
!+++++++++++++++
! TARGETS
!---------------

❶ .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 "$(PCA)" .NES. "FALSE" THEN PCA_LINK = "/DEBUG=SYS$LIBRARY:PCA$OBJ"

! The normal development build (The default if no other target is specified)

❷ 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 -- used the same as work build, but has
!different target; allows .FIRST directive to cause Pascal not to generate .ANA files.

version_build    : work_build
                  ! Version build completed at this point

!The translit executable

❸ translit.exe : translit.olb($(translit_modules))  ! put modules into an olb
        $(link) $(linkflags) translit.olb/library/include=translit 'PCA_LINK'
        IF "$(DEBUG)" .EQS. "DEBUG" THEN $(link) $(linkflags)/debug/exe= -
        $(MMS$TARGET_NAME).debug translit.olb/library/include=translit
        set protection=w=re translit.*

! 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 that follow create the needed .OBJ files which
! are the sources for the target TRANSLIT.EXE.

❹ translit.obj     : translit.pas, openfiles.pen, types.pen
buildtable.obj   : types.pen
copyfile.obj     : types.pen
expandstring.obj : types.pen
```

Key to Example 4-9:

❶ This section, in combination with the modified Pascal rule from Example 4–8, sets up the targets to update the SCA library. Two options exist:

1. 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 Pascal 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.

2. The SCA library is not updated; designate VERSION_BUILD when invoking MMS. This may be done when building an older generation, as explained in Example 4–7.

❷ These target instructions set up the MMS dependencies. Different types of executable images can be built depending on how the team invokes MMS (see the explanatory text of Example 4–7 for specific MMS invocations):

1. 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 qualifier) and the other is a nondebugger version that DTM uses for its test collection.

2. By choice, an image that represents previous versions of the software application as designated in a CMS class specified during the MMS command.

❸ In building the project executable image, the commands create an object library based on the previously defined macro, (TRANSLIT_MODULES).

❹ It is good practice to include the final source dependencies as part of the description file, although MMS would complete the build without this information. For additional information about MMS and DTM, see the *Guide to VAX DEC/Module Management System* and the *Guide to VAX DEC/Test Manager*.

Table 4–2 summarizes the different commands and their effects for this MMS description file.

Table 4–2: MMS Description File Command Options

Command	Effect
MMS/CMS	Builds current TRANSLIT.EXE with Test Collection, no PCA coverage, updates SCA library.
MMS/CMS WORK_BUILD	Builds current TRANSLIT.EXE with Test Collection, no PCA coverage, updates SCA library.
MMS/CMS/MACRO=PCA=TRUE	Builds current TRANSLIT.EXE with Test Collection, with PCA coverage on tests, updates SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") VERSION_BUILD	Builds previous class with corresponding Test Collection, no PCA coverage, no update to SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN= "V1.0") /MACRO=PCA=TRUE VERSION_BUILD	Builds previous class with corresponding Test Collection, with PCA coverage, no update to SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0")	Builds previous class with corresponding Test Collection, no PCA coverage, updates SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") WORK_BUILD	Builds previous class with corresponding Test Collection, no PCA coverage, updates SCA library.

4.9 Using the DTM Review Subsystem

The DTM Review subsystem allows you to examine the results generated by executing tests. When the TRANSLIT base level is built, DTM creates result and difference files when the test collection is executed. The Review subsystem allows you to examine these files as well as benchmark files for your tests.

The following command invokes the DTM Review subsystem to examine the test results in the collection file NEW_TEST:

```
$ DTM REVIEW NEW_TEST
Collection NEW_TEST with 1 test was created on  1-FEB-1988 10:05:10 by the
command:
        CREATE COLLECTION NEW_TEST
*/SUBMIT=(NOTIFY,NOPRINT,KEEP)/CLASS=(TEMPLATE:,BENCHMARK:)/VAR=(USE_PCA=TRUE,US
E_PCA_INIT_FILE=TRN_COMS:MYINITFILE.PCAC,TRANSLIT="TRN_DISK:[TRN.BLD_V1]TRANS
LIT.EXE;") "Test of build"
        Last Review Date =  5-FEB-1988 14:32:12
        Success count = 0
        Unsuccessful count = 1
        New test count = 0
        Updated test count = 0
        Comparisons aborted = 0
        Test not run count = 0
```

The following sections describe the following topics on the DTM Review subsystem:

- Selecting a results file
- Reviewing the differences between the results file and the benchmark file
- Invoking the PCA Analyzer from within DTM Review
- Invoking LSE from PCA

4.9.1 Selecting the Result File from DTM Review

After a collection has been executed and compared, each test is associated with a benchmark file, a difference file, and a result file. If the test was unsuccessful — that is, if differences were found between the result file and the benchmark file — DTM keeps the differences in the difference file, which you can examine with the SHOW/DIFFERENCE command.

From inside the DTM Review subsystem, the following SELECT command selects test results for NEW_TEST, the only test in the collection:

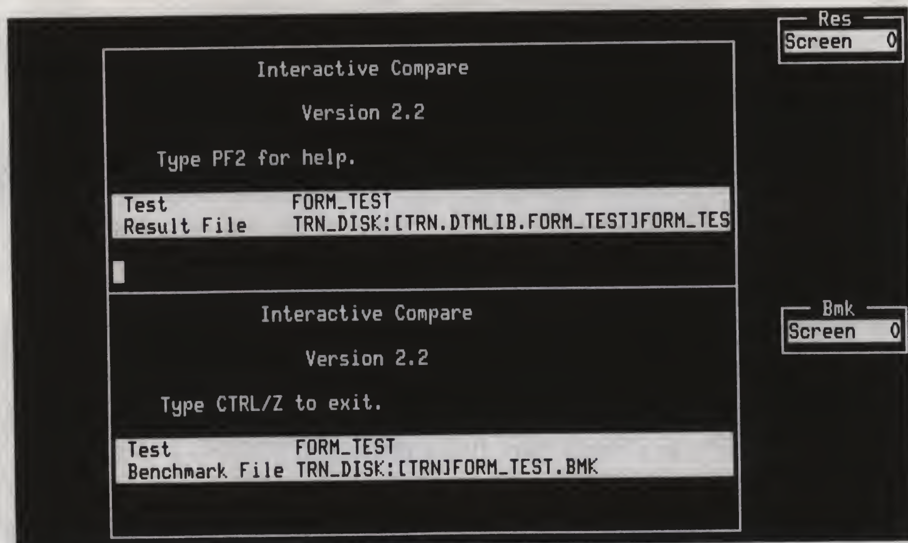
```
DTM_REVIEW> SELECT NEW_TEST
Result Description NEW_TEST       Comparison Status :  Unsuccessful
```

4.9.2 Using the SHOW/DIFFERENCES Command

The following SHOW/DIFFERENCES command displays the differences between the actual test results and the expected test results. In this example, two sample screens produced by SHOW/DIFFERENCES are included.

```
DTM_REVIEW> SHOW/DIFFERENCES
```

Figure 4–17: Sample SHOW/DIFFERENCES Output — Screen 0

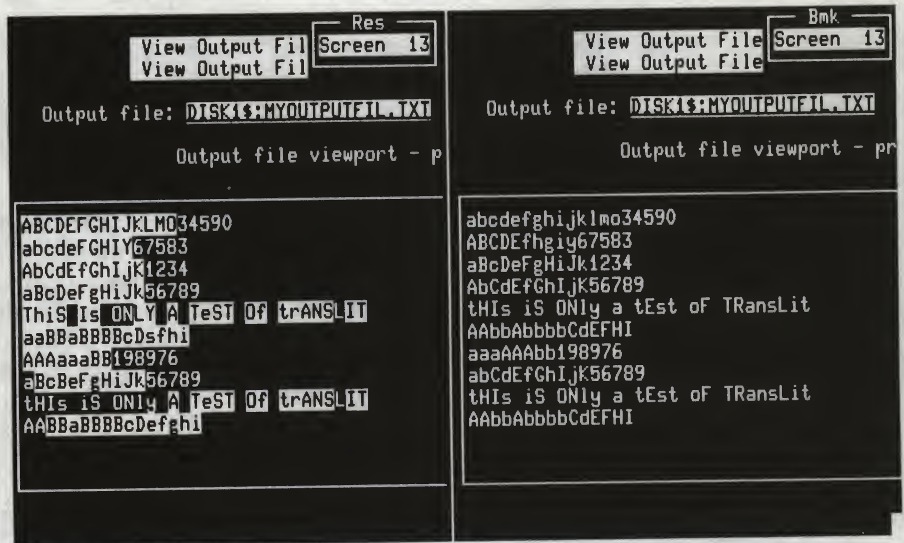


ZK-7452-HC

In Figure 4–17, DTM displays the name of the test and the locations of the Result and Benchmark files. To the right are two icons—one for the Result file, and the other for the Benchmark file—that show the current screen number. When you use SHOW/DIFFERENCES to obtain the differences screens, you can press the Keypad 0 key to move through the

differences screens. (Note that you can also press the Keypad PF2 key to get a help frame that shows what keys you can press to move through the differences screens and to present the information in different formats.)

Figure 4–18: Sample SHOW/DIFFERENCES Output — Screen 13



ZK-7451-HC

In Figure 4–18, you can see how DTM highlights the data in the Result file screen that differs from the corresponding information in the Benchmark file.

4.9.3 Invoking PCA from the REVIEW Subsystem

From the DTM Review subsystem, you can also invoke the PCA Analyzer using the PCA command. Using the PCA command allows you to examine performance and coverage data gathered by the PCA Collector while DTM was running the tests.

The PCA command, shown in the following example, invokes the Analyzer from within the Review subsystem of DTM. This command accepts no qualifiers or parameters. See the *Guide to VAX DEC/Test Manager* for more information on this command.

```
DTM_REVIEW> PCA
        VAX Performance and Coverage Analyzer Version 1.1
PCAA>
```

You must be in the DEC/Test Manager Review subsystem when you issue the PCA command. You also must be positioned at a DTM result description to issue this command. (The positioning commands are FIRST, LAST, NEXT, BACK, and SELECT.) When you issue the PCA command, DTM 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 your tests be used as input to the Analyzer.

DTM sets up an 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 your test system rather than on a test-by-test basis, you can cancel DTM_FILTER by issuing the Analyzer CANCEL FILTER command.

```
PCAA> CANCEL FILTER DTM_FILTER
```

If you want to examine data for a particular test in your DTM 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. See the *Guide to VAX Performance and Coverage Analyzer* or the *Guide to VAX DEC/Test Manager* for more information on using DTM and PCA together.

```
PCAA> SET FILTER DTM_FILTER RUN_NAME = testname
```

In this example, testname is the test containing data you want to examine next.

To obtain an annotated source listing from which you can locate code lines you may wish to edit, you can use the PCA PLOT/COVERAGE/SOURCE command from the DTM subsystem, as in the following example:

```
PCAA> PLOT/COVERAGE/SOURCE MODULE BUILD_TABLE BY LINE
```

This command line provides a source listing showing the coverage for each line in the module BUILD_TABLE. Additionally, by examining the output produced by this command, you can locate the lines of code you may want to edit. Invoking LSE from PCA is described in the following section.

4.9.4 Invoking LSE from PCA

When tuning an application, invoking LSE from PCA allows you to modify modules. If you want to edit the source code displayed by the most recent PLOT or TABULATE command, create an annotated source file listing using the /SOURCE qualifier, as shown in the previous section. You can then use the EDIT command to invoke LSE from PCA, and then go directly to the lines shown in the annotated listing.

Example 4–10 shows a portion of an annotated source listing created with the Analyzer /SOURCE qualifier:

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 in Example 4–10 that lines 88 and 99, which deal with error conditions, received no coverage. Referring to a listing of this file shown in Example 4–5, you can see that the tests cover only valid input, and no testing of error handling exists. By redesigning the tests to include testing for invalid input, the Transliteration team can rebuild TRANSLIT, and reexamine the coverage data to see if the tests are exercising the error conditions code.

If you wish to modify the source code shown in the annotated listing, the command syntax to invoke LSE from PCA is as follows:

```
PCAA> EDIT [/EXIT] [[module-name\] line number]
```

Example 4–10: Sample Annotated Source Code Listing

```
                    VAX Performance and Coverage Analyzer                 Page 2

                Test Coverage Data (784 data points total) - "*"

Percent   Count      Line
  0.1% ********       86:          IF repl_len > 1
                      87:          THEN
  0.0%                88:              lib$signal (IADDRESS (trnlit__repnotsin),
                       -:  0);
  0.1% ********       89:          IF repl_len = 0
                      90:          THEN
  0.0%                91:              replace_code := del_code
                      92:          ELSE
  0.1% ********       93:              replace_code := repl_vector[1];
  0.4% ********       94:          FOR i := 1 TO orig_len DO
                      95:              BEGIN
   -                  96:              code := orig_vector[i];
   -                  97:              IF table[code].trans_value <> undef_code
                      98:              THEN
  0.0%                99:                  signal_duplicate (code);
  0.1% ********      100:              table[code].trans_value := code;
                     101:              END;
  0.3% ********      102:          FOR code := min_code TO max_code DO
                     103:              BEGIN
   -                 104:              IF table[code].trans_value = undef_code
                     105:              THEN
                     106:                  BEGIN
  0.1% ********      107:                  table[code].trans_value := replace_co
                       -: de;
  0.3% ********      108:                  table[code].compress := TRUE;
                     109:                  END;
                     110:              END;
                     111:          END
```

If you are positioned at a source listing produced by the /SOURCE qualifier on a PLOT or TABULATE command, you use the EDIT command without parameters to invoke LSE.

This command spawns a subprocess to run LSE. The PCA Analyzer automatically positions LSE at the point in the source file displayed by the PLOT or TABULATE command. When you exit from LSE, the Analyzer session resumes.

If you use the /EXIT qualifier on the EDIT command, you terminate the Analyzer session and invoke LSE in the same process.

If you want to position LSE at a line or file different than the default, the EDIT command can take a module name and a line number as a parameter. In this example, EXPANDSTRING is a module name and 25 is a line number:

```
PCAA> EDIT EXPANDSTRING\25
```

If you omit the module name and backslash, LSE defaults to the module referenced by the PLOT or TABULATE command currently in effect.

4.9.5 Using the Analyzer to Perform a Call Tree Analysis

While 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. That is, the Analyzer can give you a runtime call tree analysis that allows you to examine the frequencies of how often each routine in your code is called. To perform a runtime 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 allows you to pinpoint the set of subroutine calls.

Example 4–11 shows a static call tree analysis file, produced by SCA on the routine READ_COMMAND_LINE.

Example 4–12, by contrast, shows a PCA dynamic call tree analysis of the same routine.

Example 4–11: Static Call Tree Analysis

```
TRANSLIT\READ_COMMAND_LINE calls
.   unknown\LIB$GET_FOREIGN
.   unknown\CLI$DCL_PARSE
.   unknown\CLI$GET_VALUE
.   OPEN_FILES\OPEN_IN calls
. .   unknown\OPEN
. .   unknown\RESET
.   unknown\IADDRESS
.   unknown\LENGTH
.   unknown\SUBSTR
.   EXPAND_STRING\EXPAND_STRING calls
. .   unknown\LENGTH
. .   unknown\IADDRESS
. .   unknown\ORD
. .   unknown\SUBSTR
. .   unknown\SUCC
.   unknown\LIB$SIGNAL
.   BUILD_TABLE\BUILD_TABLE calls
. .   unknown\LIB$SIGNAL
. .   unknown\IADDRESS
. .   BUILD_TABLE\SIGNAL_DUPLICATE calls
. . .   unknown\CHR
. . .   unknown\LIB$SIGNAL
. . .   unknown\IADDRESS
.   unknown\ODD
.   unknown\CLI$PRESENT
.   OPEN_FILES\OPEN_OUT calls
. .   unknown\OPEN
. .   unknown\REWRITE
```

Example 4–12: Dynamic Call Tree Analysis

```
                 VAX Performance and Coverage Analyzer                Page 1

           Program Counter Sampling Data (20218 data points total) - "*"

Percent   Count           Call Chain Name
  0.0%        1   Chain : TRANSLIT
 99.8%    20169   Chain : . READ_COMMAND_LINE
  0.1%       11   Chain : . . OPEN_IN
  0.0%        0   Chain : . . . SHARE$PASRTL
  0.0%        0   Chain : . . . . SHARE$PASRTL
  0.0%        6   Chain : . . . . . USER_OPEN
  0.0%        1   Chain : . . OPEN_OUT
  0.0%        0   Chain : . . . SHARE$PASRTL
  0.0%        0   Chain : . . . . SHARE$PASRTL
  0.0%        8   Chain : . . . . . USER_OPEN
  0.1%       14   Chain : . COPY_FILE

                 VAX Performance and Coverage Analyzer                Page 2

           Program Counter Sampling Data (20218 data points total) - "*"

                 VAX PCA Version 2.0-2       17-FEB-1988 16:46:33

TABULATE Command Summary Information:
Number of buckets tallied:                                     11

Program Counter Sampling Data - "*"

Data count in largest defined bucket:                       20169    99.8%
Data count in all defined buckets:                          20210   100.0%
Data count not in defined buckets:                              8     0.0%

Total number of data values collected:                      20218   100.0%

Command qualifiers and parameters used:
  Qualifiers:
    /PC_SAMPLING /NOSORT /NOMINIMUM /NOMAXIMUM
    /NOCUMULATIVE /NOSOURCE /ZEROS /NOSCALE /NOCREATOR_PC
    /NOPATHNAME /NOCHAIN_NAME /WRAP /NOPARENT_TASK
    /FILL=("*","O","x","@",":","#","/","+")
    /NOSTACK_DEPTH /MAIN_IMAGE
  Node specifications:
    CALL_TREE BY CHAIN_ROUTINE

No filters are defined
```

4.10 Maintaining the Application

All the procedures discussed in this document will make project maintenance easier. The following procedures provide an online knowledge base from which future developers can work. Several tools collect and provide access to this information:

- CMS, with its ability to accumulate a history of a project's evolution.
- SCA, with its ability to provide developers with 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.

4.10.1 CMS Provides History

CMS provides a history of a project that is useful to developers maintaining the application. CMS commands can generate the following informational reports:

- A list of elements in the system
- A list of elements of one file type
- The transactional history of the entire system
- The overall transactional history of a particular element, or specified for a time period
- The transactional history of certain kinds of operations

For instance, a new developer on the Transliteration project could list selected elements in the CMS code library by using the following command:

```
$ CMS SHOW ELEMENT *.PAS

Elements in DEC/CMS Library TRN_DISK:[TRN.CODE_CMSLIB]

    BUILDTABLE.PAS       "Creates translation table"
    COPYFILE.PAS         "Copies input file to output file while transliterating"
    EXPANDSTRING.PAS     "Expands string into code vector"
       .
       .
       .
```

A great deal of information is available to the developer with the SHOW HISTORY command. However, by itself, this command gives so much information that it generally is issued over selected parts of the library, for instance, only for specific elements, transactions, or times.

Perhaps a developer needs 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 command shows important element milestones. The following example shows this command.

```
$ CMS SHOW HISTORY EXPANDSTRING.PAS /TRANSACTION=REPLACE
History of DEC/CMS Library TRN_DISK:[TRN.CODE_CMSLIB]

4-DEC-1986 10:00:06 JOHN  REPLACE EXPANDSTRING.PAS "Changed condition handling"
8-DEC-1986 14:38:18 JOHN  REPLACE EXPANDSTRING.PAS "Modified character range loop"
```

By tailoring the CMS commands, individual developers can select the information that is most useful. Note that DTM also provides a similar history for your test set.

4.10.2 SCA Provides Structural Information

SCA provides cross-reference and static analysis information across the project. This eliminates the referencing barriers between modules, speeding access to different parts of the system. This means easier maintenance for a developer, particularly if that developer did not work on the project originally. In this situation, SCA can function as a learning aid by providing a means for a developer to walk through the sources of code using specific queries.

A developer new to a project can use SCA to learn the following:

- The definitional use of a data structure
- The declaration or call of a routine
- The coding standards
- Programming techniques

For example, by using the following command, a developer can locate calls to the routine BUILD_TREE.

```
LSE> FIND /REFERENCE=call build_tree
```

SCA's static analysis capabilities, particularly its VIEW CALL_TREE command, can also ease the tasks of maintenance. This command displays routine call information, as in the following example:

```
LSE> VIEW CALL_TREE routine_x /depth=2

   MODULE_X/ROUTINE_X calls
           MODULE_Y/ROUTINE_Y
           MODULE_X/ROUTINE_X1 calls
              MODULE_Z/ROUTINE_Z1
              MODULE_X/ROUTINE_X ( recursive )
           MODULE_Z/ROUTINE_Z2
           MODULE_Y/ROUTINE_Y1
%SCA-S-ROUTINES, 6 calls found (1 recursive, depth = 2)
```

By using commands like these, developers can educate themselves about an application quickly and independently.

4.10.3 MMS Simplifies Maintenance

Previous examples have shown how MMS automates the process of building an application (see Section 4.8). With an MMS description file in place for the complete application, people assigned to maintenance can rebuild the application. In addition, by examining the description file, developers can educate themselves as to the dependency relationships among modules.

Example 4–13 is an extract from Example 4–6, and shows the dependencies among modules in the Transliteration application. Here, essentially, are the procedures to build the entire application.

Example 4–13: Dependencies in an MMS Description File

```
!Modules in build

TRANSLIT_MODULES= openfiles, types, buildtable, copyfile, expandstring, -
                  translit, translitc, translitm

!The translit executable

translit.exe : translit.olb($(translit_modules))  ! put modules into an olb
        $(link) $(linkflags) translit.olb/library/include=translit 'PCA_LINK'
        IF "$(DEBUG)" .EQS. "DEBUG" THEN $(link) $(linkflags)/debug/exe= -
        $(MMS$TARGET_NAME).debug translit.olb/library/include=translit
        set protection=w=re translit.*

! Source code dependencies that follow create the needed .OBJ files which
! are the sources for the target TRANSLIT.EXE.

translit.obj     : translit.pas, openfiles.pen, types.pen
buildtable.obj   : types.pen
copyfile.obj     : types.pen
expandstring.obj : types.pen
```

4.10.4 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 considerable use in reconstructing a previous version.

Your team can rebuild the version of the software used by the customer site (assuming that you have not retained the working application as an intact build). After reproducing the problem, you can use SCA to help locate the source modules that are involved with the problem. These files can then be retrieved from CMS, modified, and returned to the CMS library. MMS can then rebuild an identical version of the customer's application with the exception of the new modules. You can then use DTM to regressively test the modified application before sending the updated version out to the customer.

Index

A

ACL (Access Control List)
  and large projects • 4–6
  use with new libraries • 4–5
  use with previously created libraries • 4–5
Application
  maintaining an • 4–72

B

Build directory
  See Directory structure
Build procedures
  automating • 3–28, 4–51
  final for maintenance • 3–41
  for individual developers • 3–27
  for large projects • 3–9
  frequency of • 3–7, 3–8
  MMS description file for • 3–8, 4–51
  planning • 3–7
  SCA libraries for • 3–11
  with CMS classes • 4–51
BYPASS privilege
  using with CMS • 4–11

C

Call trees
  dynamic • 4–71
  obtaining from PCA • 4–69
  obtaining from SCA • 4–69
CDD (Common Data Dictionary)
  benefits of • 2–16
CMS (Code Management System)
  classes • 2–6
    See also CMS library
    creating (figure) • 4–19
  features of • 2–4
  groups • 2–6
  history for maintenance • 4–72
  integration • 2–6
  libraries
    See also CMS library
  library for code • 3–3
  library for documentation • 3–4
  planning libraries for • 3–3
  REPLACE command • 4–38
  Search Lists • 3–6
  use with DTM • 2–12, 3–17
CMS Access Control Lists • 3–30
  event notification • 4–10
  identifier • 4–8
  placing on elements • 4–8
  using • 4–6
CMS elements
  reserving from LSE • 4–33
CMS library
  See also Libraries
  concurrent access • 3–31, 4–14
  creating • 4–4
  creating class • 4–16
  for DTM tests • 3–17
  for large projects • 3–5
  history tracking • 3–31
  merging elements into • 4–14
  modifying elements • 4–13

CMS library (cont'd.)
multiple libraries • 3–5 to 3–6
performance • 3–5
retrieving class • 4–17
storing elements • 4–12
using BYPASS with • 4–11
using Search Lists • 3–6
variants (figure) • 4–14
Code Management System
See CMS
Common Data Dictionary
See CDD
Communication management
outside project • 3–24
using VAX Notes • 3–24 to 3–25
to monitor progress • 3–35
within project • 3–24
using PM • 3–24
using VAX Notes • 3–24
using VMS Mail • 3–24
CONTROL access
granting in a CMS ACL • 4–7
CREATE LIBRARY command • 4–4

D

DATATRIEVE • 2–16, 3–41
Debugger • 4–27
EDIT command • 4–29
example • 4–27
DEC/Test Manager
See DTM
Defaults
setting in LOGIN.COM • 4–19
Description file
and large projects • 3–9
described • 3–8
example • 4–51
storing • 3–8
target dependencies • 4–17
target set-up • 4–59
to build documents • 4–17
with previous versions • 4–57
with unsupported languages • 4–59
Directory protection • 3–2
Directory structure
build directory • 3–6
Directory structure
build directory (cont'd.)
See also Reference copy area
build directory (figure) • 3–6
for DTM libraries • 3–15
for individual developers • 3–27
for sample project (table) • 4–2
initial requirements • 3–3 to 3–6
multiple libraries
need for • 3–27
planning • 3–3
planning (figure) • 3–3
public access • 3–3
storing command procedures • 3–3
DOCUMENT • 2–17
Documentation
reviews • 3–26
storing • 3–4
DTM (DEC/Test Manager)
epilogue file • 3–28, 4–41
features • 2–12
integration • 2–12
interactive test • 4–45
prologue file • 4–39
/RECORD qualifier • 4–45
setting up a test system • 4–38
SHOW ALL command • 4–50
SHOW/DIFFERENCES command • 4–64
test description • 3–16, 4–44
use with CMS • 2–12
variables • 4–44
DTM library • 3–15
See also Libraries
creating • 4–4
DTM Review subsystem
invoking PCA from • 4–66
using • 4–62
DTM subdirectory
and test data • 3–17
DTM tests
automating with MMS • 4–51
organizing • 3–32 to 3–33
DTM variable
explained • 3–32
in MMS description file • 4–57

E

EDIT command
  use from debugger • 4–29
EDT Editor • 2–6
Environment file
  access from LOGIN file • 4–23
  creating • 4–21
  LSE • 4–20
  using • 4–22
Epilogue file
  DTM • 4–41
EVE Editor • 2–6
Event notification ACLs
  in CMS • 4–10

F

Field tests
  tracking • 3–38
FIND command • 4–32, 4–73

G

GOTO DECLARATION command • 4–31

H

Human engineering testing • 1–2

I

Initialization file
  use with MMS for test coverage (example) • 4–58
Interactive test
  setting up for DTM • 4–45

L

Language-Sensitive Editor
  See LSE
Libraries
  controlling access • 4–5
    for large project • 4–6 to 4–12
Libraries (cont'd.)
  creating • 4–2
  planning • 3–3
  planning (figure) • 3–3
Life cycle
  costs related to phases (figure) • 1–8
  design • 1–3
  implementation • 1–3
  maintenance • 1–4
  model (figure) • 1–2
  phases described • 1–4
  phases of • 1–1
  relationship to tools (figure) • 1–7
  requirements and specifications analysis • 1–2
  testing • 1–4
  tools used in (figure) • 1–9
Logical names
  and LOGIN.COM • 4–23
  for build directories • 3–23
  for project setup • 3–22
LOGIN.COM
  setting defaults in • 4–19
LOGIN file
  and logical names • 4–23
  example • 4–23
  storing • 4–23
  to access environment file • 4–23
LSE (Language-Sensitive Editor)
  environment file • 4–20, 4–22
  expanding tokens • 4–34
  features of • 2–6
  integration • 2–8
  integration with CMS • 2–8
  integration with SCA • 2–8
  invoking from PCA • 4–67
  RESERVE command • 4–33
  reserving elements from CMS library • 4–33
  supported languages • 2–7
LSE SET DIRECTORY/READ_ONLY command • 3–30

M

Macros
  See MMS macros
Mail
  subdirectory • 3–24

Mail (cont'd.)
  use in environment • 2–15
Maintenance
  and CMS • 4–72
  and MMS • 4–74
  and permanent storage • 3–42
  and SCA • 4–73
  for an application • 4–72
  preparing for • 3–41
Merging CMS elements • 4–14
MMS (Module Management System)
  and maintenance • 4–74
  description file • 3–8
  features of • 2–10
  integration • 2–11
  /SCA qualifier • 3–12
MMS macros
  changing defaults • 4–56
  use in description file • 4–18, 4–56
Multiple CMS libraries
  See CMS library

N

Noninteractive test
  setting up • 4–39
Notes • 2–17
  for communication management • 3–24
  use for project communications • 3–24

P

PCA (Performance and Coverage Analyzer)
  call tree analysis • 4–69
  call tree analysis (example) • 4–71
  features of • 2–13
  integration • 2–15
  invoking from DTM Review • 4–66
  invoking LSE from • 4–67
  obtaining an annotated source file • 4–67
  using • 3–35
PCA data file
  storing • 3–17
Placeholders
  redefining • 4–20
PM (Software Project Manager) • 2–17
  as a communications tool • 3–24
PM (Software Project Manager) (cont'd.)
  features of • 3–36
Project
  setting up • 3–1
Prologue file
  DTM (sample) • 4–39
Protection
  and libraries
    for large projects • 4–6
  for libraries • 4–5
Prototypes
  building • 1–2

R

Reference copy area
  and CMS libraries • 3–10
  with builds • 3–9
REPLACE command • 4–13, 4–38
RESERVE command • 4–13, 4–33
Results file
  DTM • 4–63
Rights identifiers • 4–11
  creating • 3–2
  example • 4–7

S

Sample application • 4–27
SCA (Source Code Analyzer)
  cross-referencing • 2–9
  features of • 2–9
  FIND command • 4–32
  GOTO DECLARATION command • 4–31
  integration with LSE • 2–10
  static analysis • 2–9
  static call tree (example) • 4–69
  supported languages • 2–9
  use in maintenance • 4–73
SCA library
  access to • 3–29
    See also Libraries
  .ANA files • 3–11
  call tree analysis • 4–69
  creating • 4–4
  for daily work • 3–12
  loading • 3–13, 3–29

SCA library
loading (cont'd.)
automating with MMS • 4–56
local • 3–12
physical vs. virtual • 3–13
physical vs. virtual (figure) • 3–29
search list • 3–13 to 3–30
benefits of • 3–13
for directories (figure) • 3–30
updating • 3–11 to 3–13, 3–29
with builds • 3–11
SCAN • 2–17
use with DTM • 2–17
Schedule management • 3–36
Search Lists
CMS • 3–6
SET LIBRARY command • 3–13, 4–4
SHOW HISTORY command
CMS • 4–72
Software development
problems with • 1–4 to 1–7
Software Development Life Cycle
See Life cycle
Software Project Manager
See PM
Source Code Analyzer
See SCA
Source management
access to sources • 3–30
with CMS • 3–30
with LSE search list • 3–30
Standards
coding • 3–18 to 3–19
design • 3–18
performance • 3–20
project • 3–18 to 3–20
testing • 3–20
Status reporting • 3–36
System build
preparing for • 4–11
System rights list • 4–11

T

Target dependencies
See Description file
Templates
LSE • 3–21
Test
changing input for • 4–50
creating an interactive • 4–45
creating a noninteractive • 4–39
Test description
creating • 4–44
Testing
phase of life cycle • 1–4
Tests
See DTM tests
Test system
setting up • 4–38
Text Processing Utility • 2–6
Tokens
redefining • 4–20
Tools
benefits of • 2–2 to 2–3
relationship to cycle phases (figure) • 1–7

U

UIC (User Identification Code)
for large projects • 4–6
for library access • 4–5
Usability requirements
planning • 1–2
User accounts
setting up • 3–2

V

Variable
See DTM variable
VAX DOCUMENT
See DOCUMENT
VAX Notes
See Notes
for communication management • 3–25
for QAR system • 3–38
VAXset • 2–1
VAXset tools
acronyms for • 2–1
VAX Software Project Manager
See PM

VMS operating system
use in environment • 2–15

W

Work procedures
for individual developers • 3–27 to 3–31

Reader's Comments

A Methodology for Software Development Using VMS Tools
AA–HB16C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ________________________________

____

What I like best about this manual is ________________________________

____

What I like least about this manual is ________________________________

____

I found the following errors in this manual:

Page Description

____ ________________________________

____ ________________________________

____ ________________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ________________________ Dept. ____________

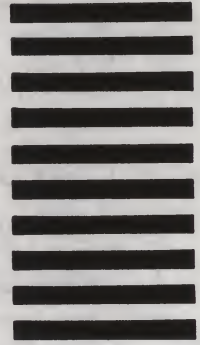
Company ________________________ Date ____________

Mailing Address ________________________________

________________________ Phone ____________

--- Do Not Tear - Fold Here and Tape ---

digital™



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line

Reader's Comments

A Methodology for Software Development Using VMS Tools
AA–HB16C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:
Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

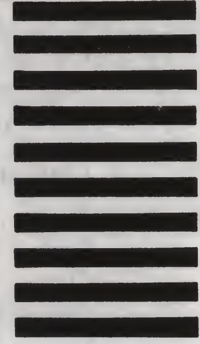
Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

-- Do Not Tear - Fold Here and Tape --

digital™



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here --

Cut Along Dotted Line

digital

AA-HB16C-TE