VAX Language-Sensitive Editor and VAX Source Code Analyzer User Manual

Order Number: AA–PAJLA–TK

December 1989

This manual describes the concepts and features of both the VAX Language-Sensitive Editor and the VAX Source Code Analyzer.

Revision/Update Information:	This document is a new manual.
Operating System and Version:	VMS Version 5.1 or higher for LSE VMS Version 5.2 or higher for SCA
Software Version:	VAX Language-Sensitive Editor Version 3.0 VAX Source Code Analyzer Version 2.0

digital equipment corporation
maynard, massachusetts

First Printing, December 1989

CDD/Plus	VAX Document	VAXstation
DATATRIEVE	VAX MACRO	VMS
DECforms	VAX Notes	VT
DECwindows	VAX SCAN	
VAX	VAXcluster	digital™
VAX DIBOL	VAXset	

ZK5311

Contents

Part 4 Customizing Functions

Chapter 13 Customizing Editing Functions

Chapter 14 Customizing LSE/DECwindows Menus

Tables

Preface

This manual explains how to use the VAX Language-Sensitive Editor and the VAX Source Code Analyzer on the VMS operating system.

Intended Audience

This manual is for experienced programmers, technical writers, and technical managers.

Document Structure

The *VAX Language-Sensitive Editor and VAX Source Code Analyzer User Manual* has 18 chapters.

- Chapter 1 provides an overview of both the VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA). It describes the concepts of each tool and explains how these tools can be used together to create an integrated, multilanguage software development environment. This chapter also briefly describes several other productivity tools that work with LSE and SCA.
- Chapter 2 provides an overview of LSE, including basic LSE features, the concepts of tokens and placeholders, how to invoke LSE, and how to compile source code. This chapter also provides a sample editing session that lets you experiment with the basic features of LSE.
- Chapter 3 describes the text-editing capabilities of LSE. This chapter provides information on multiple buffer and window support, file location and manipulation facilities, and LSE's code-viewing features.

- Chapter 4 provides an overview of the DECwindows LSE environment. It describes how to open files, perform basic editing tasks, review source code, and query source code with DECwindows. This chapter also provides a sample editing session that lets you experiment with the basic features of DECwindows LSE.
- Chapter 5 describes the language-sensitive features of LSE. This chapter provides information on designing, coding, compiling, and debugging source files.
- Chapter 6 provides an overview of SCA, including the key features of SCA and its integration with the LSE software development environment. This chapter provides information on how to invoke SCA, SCA concepts, libraries, and SCA commands.
- Chapter 7 provides a sample session that demonstrates the use of the basic SCA query commands and related LSE navigational commands.
- Chapter 8 describes advanced uses of the SCA FIND command with the SCA Query Language. This chapter includes overview information and a tutorial, which demonstrates the concepts and features of the SCA Query Language.
- Chapter 9 further describes the SCA Query Language by providing an encyclopedic summary of its rules, syntax, and components.
- Chapter 10 describes the structure, organization, and use of SCA libraries. This chapter provides library creation, manipulation, and maintenance information.
- Chapter 11 describes how to use SCA consistency checking and diagnostic capabilities. This chapter also demonstrates the features of the INSPECT command that allow you to tailor SCA for diverse programming styles.
- Chapter 12 provides a scenario for designing your own programs. In addition, it provides information on using tagged comments, generating design reports, and using the various design report formats.
- Chapter 13 describes how to customize your development environment. This chapter describes how to define keys, commands, and aliases; how to redefine language elements; and how to execute VAXTPU statements. The chapter also describes how to store your modifications and how to speed up LSE initialization.
- Chapter 14 describes how you can use the Menu Extension Service to add, modify, or delete menu entries from LSE pop-up and pull-down menus.

- Chapter 15 explains how to define your own environment files for text templates and programming-type languages. It describes how to save and store environment files, and how to define packages.
- Chapter 16 provides information on interfacing non-Digital language processors to the diagnostic review facility.
- Chapter 17 describes how to customize overviews for programming languages, including languages that Digital does not directly support.
- Chapter 18 describes how to customize reports.

Associated Documents

- The *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* contains reference material on how to use the VAX Language-Sensitive Editor and the VAX Source Code Analyzer.
- The *VAX Language-Sensitive Editor Installation Guide* contains instructions for installing LSE on VMS operating systems.
- The *VAX Source Code Analyzer Installation Guide* contains instructions for installing SCA on VMS operating systems.
- The *VAX Text Processing Utility Reference Manual* describes the VAX Text Processing Utility features, including the high-level procedural language available for use with LSE.
- *Using VAXset* describes how to use the VAXset products with other VMS software development facilities to create an effective development environment.

Convention	**Description**
RETURN	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, RETURN.
CTRL/*x*	The phrase CTRL/*x* indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/Y, CTRL/Z, CTRL/G.
KP*n*	The phrase KP*n* indicates that you must press the key labeled with the number or character *n* on the numeric keypad, for example, KP6, KP3.

Convention	Description
$ LSEDIT	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.
file-spec, . . .	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates additional parameters, options, or values you can enter.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term.
italic text	Italic text represents parameters, arguments, and information that can vary in system messages (for example, Internal error *number*), as well as book titles.
user input	The hardcopy version of this manual has interactive examples that show user input in red letters and system responses or prompts in black letters. The online version differentiates user input from system responses or prompts by using a different font.
UPPERCASE TEXT	Uppercase letters indicate the name of a command, a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1,MB2,MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (You can redefine the buttons.)

Chapter 1

Introduction

This chapter explains how the VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA) work together and with other VMS productivity tools to create an integrated, multilanguage software development environment.

LSE is a multilanguage text editor that speeds up writing and compiling source code. SCA is a multilanguage, interactive cross-reference and static analysis tool that provides detailed information about source code.

Individually, each of these tools allows you to take advantage of the multilanguage software development capabilities on VMS systems.

Together, these tools provide an integrated method for designing, creating, compiling, correcting, and inspecting your source code within a single editing session. You can include design information that can be processed, analyzed, and preserved throughout the software development cycle. You can review and, if necessary, modify the source code for your software project. Instead of working with each individual file, you can access all your project files through LSE.

Section 1.1 presents an overview of LSE, and Section 1.2 presents an overview of SCA. Section 1.3 describes how LSE and SCA work together. Section 1.4 describes the integration of LSE with VAX DEC/Code Management System (CMS), and Section 1.5 describes how LSE and SCA integrate with several other software development tools.

1.1 LSE

LSE is an advanced text editor with language-specific features. Users familiar with EDT or EVE will recognize the corresponding keypads in LSE and be able to use LSE immediately.

In addition to text-editing features, LSE provides the following language-specific support:

- Code compilation
- Diagnostic review
- Formatted language constructs
- Online language HELP
- Pseudocode entry support
- Code elision
- Documentation extraction

You can compile source files from within LSE. The VMS compilers write diagnostics that are read and displayed by LSE. LSE accesses and displays the corresponding source locations.

LSE supplies the formatted language constructs. These constructs, known as templates, include keywords, punctuation, and placeholders for most VAX programming languages. LSE can supply you with the appropriate syntax for any of the following VAX languages:

- VAX Ada
- VAX BASIC
- VAX BLISS
- VAX C
- VAX CDD
- VAX COBOL
- VAX DATATRIEVE
- VAX DIBOL
- VAX FORTRAN
- VAX Pascal
- VAX PL/I

- VAX SCAN
- VAXELN Pascal

See the release notes or the Software Product Description (SPD) for a more complete list of supported languages.

In addition to formatted language constructs, LSE provides templates for subroutine libraries, including the VMS System Service library and the Record Management System (VAX RMS).

LSE has an online HELP Facility for information on unfamiliar language constructs, routines, and text insertion. Help is also provided for all LSE commands and key definitions and all SCA commands.

LSE has features for editing comments efficiently, including word wrapping, paragraph fill, and comment alignment.

LSE provides for incremental program design and development by using features that allow you to write structured text, or pseudocode, in source files. You use this pseudocode to describe the design of the code before you write the actual program statements. You can process source files that contain pseudocode by using supported VMS compilers. LSE also provides functions for capturing this pseudocode in comments when you enter the actual source code.

LSE provides a facility that lets you view programs at various levels of detail. The concept is sometimes called outlining, holophrasting, or code elision. You can generate overviews of programs by collapsing lines of code to hide detail and expanding overviews to see more detail.

LSE is tightly integrated with the following VMS tools:

- VAX Source Code Analyzer (SCA)
- VAX DEC/Code Management System (CMS)
- VAX Text Processing Utility (VAXTPU)

All SCA commands are available within LSE. CMS commands are available within LSE by using the prefix *CMS* at the LSE command prompt. All VAXTPU commands are available within LSE by using the prefix *DO/TPU* at the LSE command prompt.

You can call LSE from the following VMS tools:

- VAX DATATRIEVE
- VMS Debugger
- VMS Mail Utility (MAIL)
- VAX Notes
- VAX Performance and Coverage Analyzer

See Sections 1.4 and 1.5 for more details on LSE's integration with VMS tools.

With LSE, you can customize your editing environment to meet your programming preference or style. You can also extend your editing environment to handle highly specialized editing needs.

1.2 SCA

The VAX Source Code Analyzer is a multilanguage, interactive cross-reference and static analysis tool.

SCA helps you understand large-scale software projects by allowing you to make inquiries about the symbols used in such projects. With SCA, you can quickly locate information about any identifier. Thus, SCA is useful during the implementation and maintenance phases of a project, regardless of your familiarity with the project.

SCA provides the following capabilities:

- **Cross-referencing**

  SCA provides a cross-referencing facility that gives you an index to information in your source code.

  SCA depends on supported VMS compilers for the generation of detailed source analysis data. Source analysis data consists of a collection of information relating to all of the symbols, files, and modules contained in the source. The information is loaded into an SCA library and used as a database for the SCA cross-reference query and static analysis features.

  The SCA query capabilities allow you to check for specific symbols, files, or modules. You can determine such things as declarations of program symbols, references to the symbols, and references to source files.

- **Static analysis**

  The SCA static analysis facility allows you to extract information about program structure and the relationship of routines, symbols, and files. You can display call tree information to determine the relationships between routines. You can also check whether routines are used in a consistent manner.

- **Library creation and maintenance**

  SCA takes the information generated by supported VMS compilers and merges these files together into libraries to create a picture of your entire project.

  Once a library has been set up for a particular software project, you can use the cross-referencing, query, and source analysis features of SCA on that software project. You can also set up a personal library, containing information on only those modules that you are working on, and use this library with the main library describing the rest of the system.

1.3 LSE/SCA Integration

Using LSE and SCA together gives you more power than using each product individually. With these tools, you can do the following:

- **Access your entire system from LSE**

  You can tell LSE where to look for your files and modules. SCA allows you to have full access to all sources for your project from within LSE. Then, you can browse through all your code to look for specific declarations or symbols or other pertinent information about your project.

- **Create a private SCA library for your local sources**

  You can modify modules in your own directory and also access other modules in your project-wide SCA library.

- **Write your source files in more than one language**

  LSE always provides you with the right language support for the file you are editing. SCA provides you with global navigation of your entire project regardless of the languages used in each module.

- **Query your source files**

  Using SCA and LSE together, you can point to an identifier with the LSE cursor and, with one keystroke, you can bring up the definition of that identifier in another window. You can also step through the results of an SCA query, looking at the actual source code corresponding to each reference found by SCA.

- **Generate reports**

  In addition to getting information directly from SCA queries, you can use the SCA REPORT command to produce a variety of reports from your SCA database. Reports provide information in a structured, organized way. Reports are implemented in TPU, and therefore require LSE. By changing or rewriting the TPU code, you can customize reports to suit your needs.

In general, the preferred way to use SCA interactively is through LSE. However, you can use SCA from the DIGITAL Command Language (DCL) level for batch jobs and time-consuming tasks, such as creating project-wide libraries.

1.4 VAX DEC/Code Management System Integration

LSE provides an interface to VAX DEC/Code Management System (CMS) to simplify program development and source file management. Using CMS with LSE provides the following capabilities:

- **Online storage library**

  You can create an online library to store your source files. LSE allows you to use all source files stored in the CMS library. Once LSE is made aware of the CMS library, you can locate and examine these files from within LSE.

- **Easy file access**

  You do not have to remember the names or locations of the files and modules contained in your project. LSE will locate your files automatically.

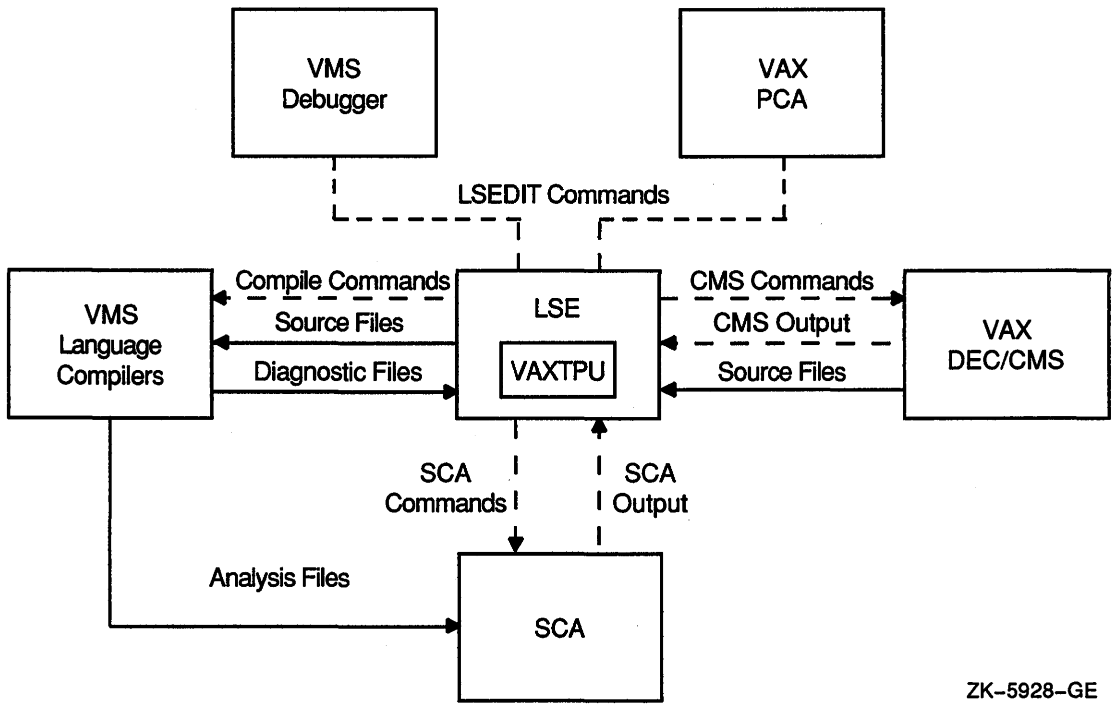
- **File manipulation**

  Using LSE's file manipulation commands, you can move a file from the CMS library, place it in your current buffer, modify it, and return it to the library without leaving LSE.

1.5 Integration with Other VMS Tools

LSE works with other VMS productivity tools to simplify the program development process. Figure 1–1 shows how LSE relates to each of these tools to make up the VAX Language-Sensitive Editor software development environment.

Figure 1–1: VAX Language-Sensitive Editor Software Development Environment



LSE has links to the following:

- **VMS compilers**

  Using LSE, you can write and compile your source code in one editing session. The supported VMS compilers can process placeholders and comments. In addition, compilers generate files that LSE reads when correcting syntax errors in source code. Thus, you can write and compile your code within a single editing session.

  LSE's multiwindow capability enables you to review your errors while examining the associated source code. Not only does this feature eliminate tedious steps in the error-correction process, it also helps ensure that you fix all the errors before recompiling.

  Some compilers, such as VAX C and VAX Ada, provide you with suggested corrections for syntax errors. With LSE, it is easy for you to incorporate the compiler's corrections or to make your own corrections.

In addition, the compilers generate files that contain detailed information about your source code that SCA uses when performing queries. Thus, your cross-reference information is available on an interactive basis.

- **VAX Performance and Coverage Analyzer (PCA)**

  LSE is integrated with the VAX Performance and Coverage Analyzer (PCA) to simplify the tuning portions of software development. You can invoke LSE from the analyzer portion of the VAX Performance and Coverage Analyzer.

- **VMS Debugger**

  You can invoke LSE from the VMS Debugger (debugger) so that errors detected during a debugging session can be corrected in the original source code file. This enables you to make corrections immediately. 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. When you finish making a correction, you are positioned back in the debugger where you left off.

- **VAX MAIL**

  You can use LSE as your default editor in MAIL by putting the following command in your LOGIN.COM file:

  ```
  $ DEFINE MAIL$EDIT CALLABLE_LSE
  ```

- **VAX Notes**

  You can specify LSE as the editor that you want to use in Notes by issuing the following Notes command:

  ```
  NOTES> SET PROFILE /EDITOR=(LSE,CALL)
  ```

- **VAX DATATRIEVE**

  You can specify LSE as the editor for VAX DATATRIEVE to use when you issue a DATATRIEVE EDIT command by putting the following command in your LOGIN.COM file:

  ```
  $ DEFINE DTR$EDIT LSE
  ```

See *Using VAXset* for a more detailed description of how these software tools work together to create an effective development environment.

Part 1 Using LSE

This part contains tutorial information on using the VAX Language-Sensitive Editor and includes the following:

- Performing editing tasks
  - Using windows and buffers
  - Working with files
  - Collapsing and expanding program source
- Using LSE/DECwindows
- Performing language-specific tasks
  - Using placeholders and tokens
  - Using pseudocode and comments

Chapter 2

Introduction to LSE

This chapter is a brief introduction to the VAX Language-Sensitive Editor (LSE). Section 2.1 provides an overview of LSE, including the features of LSE. Section 2.2 describes how to invoke LSE, enter source code, and leave LSE. Section 2.3 provides a sample editing session. Section 2.4 describes how to compile source code within LSE. Section 2.5 describes how to invoke LSE from the VMS Debugger and from the VAX Performance and Coverage Analyzer. Section 2.6 describes the format of the LSE command line.

2.1 Overview

LSE is a multilanguage, advanced text editor that is layered on the VAX Text Processing Utility (VAXTPU). LSE works with VAX languages and VMS productivity tools to enhance program development.

With LSE, you can control your editing environment and use LSE's knowledge of specific languages to develop programs quickly and accurately.

2.1.1 LSE Features

LSE provides the following features:

- **Error correction and review**

  The error correction and review feature allows you to compile, review, and correct compilation errors within a single editing session. LSE provides an interface to the supported VMS compilers so that you can perform compilations without leaving LSE. The compilers provide LSE with compilation diagnostics in a way that allows you to review compilation errors in one editing window while displaying the related source in another window.

In addition, LSE provides a mechanism for interfacing non-Digital language processors to the diagnostic review facility.

- **Language-specific templates**

  LSE accesses a collection of formatted language constructs, called **templates**, that provide keywords, punctuation, and placeholders for each supported VAX language. Templates provide a fast and efficient way to sketch design ideas and enter source code.

  LSE allows you to modify existing templates or define your own language or text templates.

- **Design support**

  LSE provides program design and development support. You can write structured text in the form of pseudocode that describes the design of the code before you write the actual program statements. LSE provides a mechanism to preserve this design information in comments when you enter the source code. In addition, LSE provides code-viewing features that allow you to see more or less detail at a particular point in a program. LSE and SCA provide a report tool that allows you to present the overviews you select in a structured manner.

- **Integrated programming environment**

  LSE is integrated into the VMS development environment. LSE is invoked by using the DIGITAL Command Language (DCL). LSE works with supported languages, SCA, VAX DEC/Code Management System (CMS), the VMS Debugger (debugger), and the VAX Performance and Coverage Analyzer (PCA) to provide a highly interactive environment. This environment enables you to design, create and edit code, view multiple source modules, compile programs, and review and correct compile-time errors in one editing session.

  You can invoke LSE directly from the debugger to correct source code problems found during debugging sessions. In addition, you can invoke LSE from the VAX Performance and Coverage Analyzer to correct performance problems found during analyzing sessions.

- **Online HELP Facility**

  LSE provides an online HELP Facility for information on unfamiliar language constructs and routines. Help is also provided for all LSE's commands and key definitions.

- **Source code analysis**

  LSE's integration with VAX Source Code Analyzer (SCA) allows you to search for specific information contained in your source files.

SCA is a source code cross-reference and static analysis tool that helps programmers familiarize themselves with complex systems. SCA accesses source information generated by supported VMS compilers. SCA allows you to move through this information and gain access to related source files as necessary. You can find out how a program symbol was declared, where a particular routine is called, or what module needs to be recompiled.

- **Source code management**

  An interface with VAX DEC/Code Management System (CMS) simplifies the functions of program development.

  You can issue all CMS commands within LSE. In addition, you can request to fetch or reserve files directly from a CMS library when you issue standard LSE file manipulation commands.

- **LSE customization**

  With LSE, you can extend your editing environment to handle highly specialized editing needs. LSE provides an interface to VAXTPU. VAXTPU is part of the VMS operating system. VAXTPU features include a compiler and an interpreter, and procedures for screen management and text manipulation. The VAXTPU language is block-structured and provides looping, conditional, case, and assignment statements, as well as many built-in procedures so you can perform more powerful editing tasks.

- **EVE/EDT keypads**

  LSE provides a SET MODE KEYPAD command that sets the key definitions to be similiar to EVE or EDT.

- **System Services and Run-Time Library templates**

  LSE provides templates for subroutine libraries, including the VMS System Service library, Run-Time Library (RTL) (LIB$, STR$, SMG$), and the Record Management System (VAX RMS). In addition, LSE allows you to define templates for packages of subroutine libraries.

2.2 Getting Started

This section presents some general LSE concepts, including the following:

- Using tokens and placeholders
- Issuing commands
- Invoking LSE
- Invoking the online HELP Facility

The best way to learn about LSE is to start using it. Section 2.3 guides you through a sample editing session to familiarize you with the basic features of LSE. You will learn how to invoke LSE, use templates to enter source code, and leave LSE.

2.2.1 Understanding LSE Concepts

Before you start using LSE, you need to understand the concepts of tokens and placeholders, which are language elements that have been predefined for each of the supported languages. You can expand these elements into templates for language constructs.

Tokens are reserved words or function names that you type into the editing buffer and expand to provide templates for corresponding language constructs. For example, you can type the keyword IF and then expand it into a complete skeleton IF statement, with consistent indentation and capitalization.

Placeholders are items surrounded by delimiters that are inserted into the editing buffer by LSE when you expand other placeholders or tokens. Placeholders are markers that indicate locations in the source code where you must provide additional program text or choose from indicated options.

Placeholders are either required or optional. Required placeholders, indicated by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets ([ ]), represent places in the source code where you can either provide additional constructs or erase the placeholder. For example, a required placeholder might look like this:

```
{compilation_unit}
```

You can expand, erase, or type directly over placeholders. When you type over a placeholder, the placeholder is automatically removed and the text you type is inserted into the buffer. When you erase a placeholder, the placeholder is automatically deleted. However, if you erase a required placeholder, LSE displays a message saying that the placeholder is required, and asks if you want to continue the erase operation. When you press the EXPAND key (CTRL/E) while the cursor is on a placeholder, one of three events occurs:

- The placeholder is replaced automatically with a template consisting of language constructs. This type of placeholder is called a **nonterminal placeholder** because it inserts a template into the buffer when expanded.

- Text appears in a separate window to aid you in supplying a value. This type of placeholder is called a **terminal placeholder** because it does not insert a template into the buffer when expanded. Instead, you must supply the necessary text. You can press the spacebar to remove the window.
- A menu appears that provides you with options that can be selected and expanded into templates. This type of placeholder is called a **menu placeholder**.

In any of these three cases, you may type the desired text over the placeholder, and the placeholder is erased automatically. When expanding a menu placeholder, you can move through the options by using the up and down arrow keys. To select an option, you press the EXPAND key, the RETURN key, or the ENTER key. To exit from the menu without selecting an option, you press the spacebar.

Some placeholders are automatically duplicated when expanded. These placeholders are called **list placeholders**.

In addition, LSE provides **pseudocode placeholders**. Pseudocode placeholders are placeholders that contain natural language text that expresses design information. Pseudocode placeholders, unlike regular placeholders, are not defined by LSE. LSE inserts pseudocode placeholder delimiters into the editing buffer when you press the ENTER PSEUDOCODE key (PF1-spacebar). You type the appropriate design information within the delimiters. You can move pseudocode placeholder text to program comments. See Chapter 5 for more details on using pseudocode.

You can construct a complete program by repeatedly expanding templates. You do not have to continuously expand templates until you reach a terminal placeholder. Rather, you may find it more appropriate to type in the desired value yourself at a higher level. See Chapter 5 for additional information on tokens and placeholders.

2.2.2 Issuing Commands

LSE provides the following two ways to issue commands:

- Keypad mode
- Command line mode

When you invoke LSE, you are in keypad mode. In keypad mode, text that you type is inserted into a buffer. Keypad, cursor, and control keys execute LSE functions. Thus, you can press keys to perform editing functions rather than typing commands on the command line. LSE binds commonly used commands to certain keys to simplify editing. LSE provides access to both the EDT and EVE keypads and commands. When you invoke LSE, the keypad mode is set to the EDT keypad. You use the SET MODE KEYPAD command to get the EVE keypad. If you are more comfortable with the EVE keypad, you can put the SET MODE KEYPAD command in your initialization file so that it will be set to EVE each time you invoke LSE. (See the SET MODE command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details.) The EDT key bindings are used in the examples in this manual.

Some LSE commands are not bound to keys. Therefore, they must be issued in command line mode. There are two command line prompts: LSE Command> and LSE>.

The LSE Command> prompt processes one command. After that command is processed, LSE returns to keypad mode. There are two ways to get the LSE Command> prompt:

- Press the DO key
- Press the COMMAND key (PF1-KP7)

The LSE Command> prompt appears near the bottom of the screen.

Alternatively, the LSE> prompt allows you to issue as many commands as you want. To get the LSE> prompt, press CTRL/Z. The LSE> prompt appears near the bottom of the screen. To return to keypad mode, you can press CTRL/Z again or issue the CONTINUE command.

The prompts are in either insert or overstrike mode. The setting defaults to the current setting of the terminal. In LSE/DECwindows, the default is overstrike. You can change this setting by including the following commands in an initialization file:

```
SET INSERT/BUFFER=$COMMANDS
SET INSERT/BUFFER=$PROMPTS
```

2.2.3 Invoking LSE

The format for invoking the LSE command line is as follows:

```
$ LSEDIT [/qualifiers] [file-spec]
```

LSEDIT invokes LSE; /qualifiers specify command qualifiers; and file-spec specifies the file to be edited. It must be a VMS file specification.

If you do not specify a file name or file type in your file specification, LSE uses the file name or file type specified in your last LSEDIT command if you issued the EXIT command to end that editing session. Otherwise, LSE creates a new buffer called $MAIN and prompts you for a file name when you exit from LSE if you have added text to the $MAIN buffer.

2.2.4 Getting Help

To get help at any time during your editing session, do any of the following:

- To see a diagram of the keypad, press the HELP key (PF2).
- To see a list of the keys and their descriptions, press CTRL/Z to get the LSE> prompt, and type the SHOW KEY command.
- To see a list of LSE commands and their explanations, press CTRL/Z to get the LSE> prompt, and type the HELP command.
- To see a list of all the predefined tokens or placeholders for the language of the current buffer, press CTRL/Z to get the LSE> prompt, and type the SHOW TOKEN or SHOW PLACEHOLDER command.
- To get language-specific help on a particular keyword or placeholder, position the cursor on the keyword or placeholder and press PF1-PF2. Help is not available for all keywords and placeholders.

2.3 Sample Session

The following sample editing session helps you experiment with LSE. This editing session uses a sample language, called EXAMPLE, that is supplied with LSE. The editing session highlights the following:

- Expanding nonterminal placeholders
- Deleting placeholders
- Typing over list placeholders
- Expanding menu placeholders

- Expanding tokens
- Expanding terminal placeholders
- Entering pseudocode placeholders
- Moving pseudocode placeholders to comments

All required placeholders are indicated by braces ({}) and optional language elements are indicated by brackets ([ ]). For this example, all commands are referred to by the command name and corresponding EDT key binding.

You can use several commands for manipulating tokens and placeholders. Table 2–1 lists these commands and their default key bindings. Table 2–2 lists the manipulation commands and their functions.

Table 2–1: Commands for Token and Placeholder Manipulation

Command	EDT Keypad	EVE VT100 Keypad	EVE VT200 Keypad
EXPAND	CTRL/E or CTRL//	CTRL//	CTRL//
UNEXPAND	PF1-CTRL/E or PF1-CTRL//	PF1-CTRL//	PF1-CTRL//
ERASE PLACEHOLDER/FORWARD	CTRL/K	CTRL/K	CTRL/K
UNERASE PLACEHOLDER	PF1-CTRL/K	PF1-CTRL/K	PF1-CTRL/K
GOTO PLACEHOLDER/FORWARD	CTRL/N	CTRL/N	CTRL/N
GOTO PLACEHOLDER/REVERSE	CTRL/P	CTRL/P	CTRL/P
ENTER PSEUDOCODE	PF1-spacebar	PF1-spacebar	PF1-spacebar
ENTER COMMENT/BLOCK	PF1-B	PF1-B	PF1-B
ENTER COMMENT/LINE	PF1-L	PF1-L	PF1-L

Table 2–2: Manipulation Commands and Their Functions

Command	Function
EXPAND	The EXPAND key (CTRL/E) replaces a placeholder at the current cursor position with the appropriate body of text or code. When you press the EXPAND key after typing a token name, the token expands in much the same manner as a placeholder.

(continued on next page)

Table 2–2 (Cont.): Manipulation Commands and Their Functions

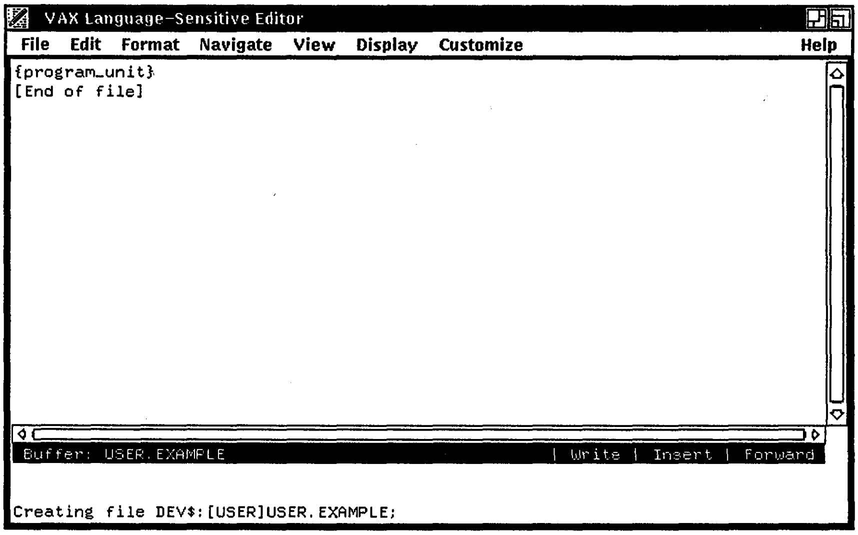
Command	Function
UNEXPAND	The UNEXPAND key (PF1-CTRL/E) reverses the effect of the last EXPAND command.
ERASE PLACEHOLDER/FORWARD	The ERASE PLACEHOLDER/FORWARD key (CTRL/K) allows you to remove optional placeholders that are not necessary for your program.
UNERASE PLACEHOLDER	The UNERASE PLACEHOLDER key (PF1-CTRL/K) restores the text deleted by the corresponding ERASE PLACEHOLDER command that you most recently executed.
GOTO PLACEHOLDER/FORWARD	The GOTO PLACEHOLDER/FORWARD key (CTRL/N) places you on the next placeholder.
GOTO PLACEHOLDER/REVERSE	The GOTO PLACEHOLDER/REVERSE key (CTRL/P) allows you to move back to the previous placeholder.
ENTER PSEUDOCODE	The ENTER PSEUDOCODE key (PF1-spacebar) allows you to enter pseudocode placeholders.
ENTER COMMENT/BLOCK	The ENTER COMMENT/BLOCK key (PF1-B) allows you to move text from a pseudocode placeholder into a block comment.
ENTER COMMENT/LINE	The ENTER COMMENT/LINE key (PF1-L) allows you to move text from a pseudocode placeholder into a line comment.

To invoke LSE and start the sample session on your screen, type the following:

```
$ LSEDIT USER.EXAMPLE
```

The placeholder {program_unit} appears at the top of your screen. This placeholder is called the **initial string** because it is the first language element that LSE puts into a newly created buffer. Figure 2–1 shows the initial string {program_unit}.

Figure 2–1: Initial String Placeholder in a New Buffer


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
{program_unit}
[End of file]
```

```
Buffer: USER.EXAMPLE                                | Write | Insert | Forward
```

```
Creating file DEV$:[USER]USER.EXAMPLE;
```

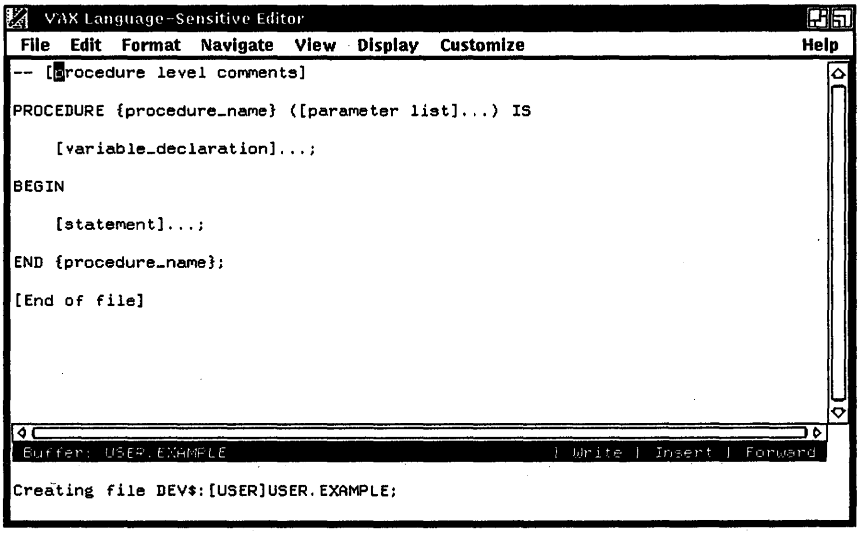
2.3.1 Expanding Nonterminal Placeholders

To experiment with placeholders and tokens, use the following steps:

1. Press CTRL/E (the EXPAND key) while the cursor is on {program_unit}.

   The initial string expands into a template of language constructs. The cursor is now positioned on [procedure level comments]. Figure 2–2 shows the resulting screen.

Figure 2–2: Expanding a Nonterminal Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE {procedure_name} ([parameter list]...) IS

    [variable_declaration]...;

BEGIN

    [statement]...;

END {procedure_name};

[End of file]
```

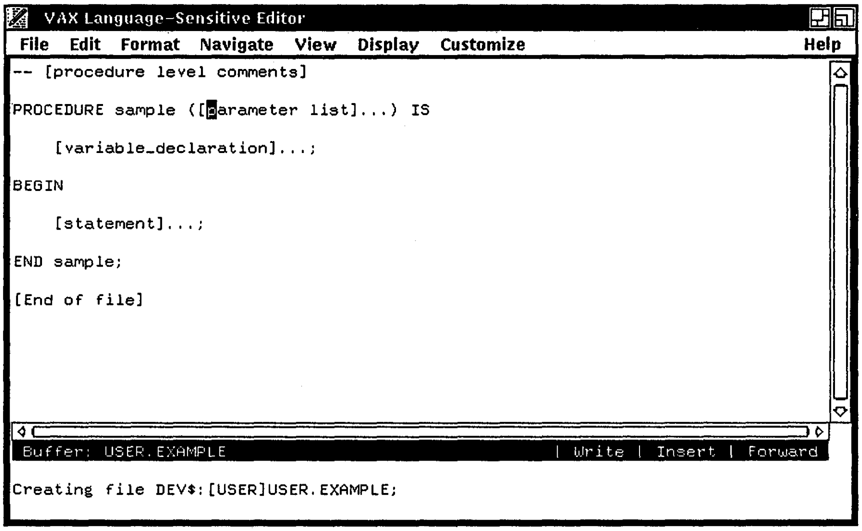
```
Buffer: USER.EXAMPLE                                  | Write | Insert | Forward
```

```
Creating file DEV$:[USER]USER.EXAMPLE;
```

2. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) to move the cursor to {procedure_name}.
3. Type the text *sample* over {procedure_name}.
4. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) to move the cursor to ([parameter list] . . .).

Notice that as soon as you move the cursor from the text, the second occurrence of {procedure_name} is replaced with the text *sample*. This is an example of the AUTO_SUBSTITUTE feature. (See the DEFINE PLACEHOLDER command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details.) Figure 2–3 shows the resulting screen.

Figure 2–3: Typing over a Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample ([parameter list]...) IS

    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]
```

```
Buffer: USER.EXAMPLE                          | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

2.3.2 Deleting Placeholders

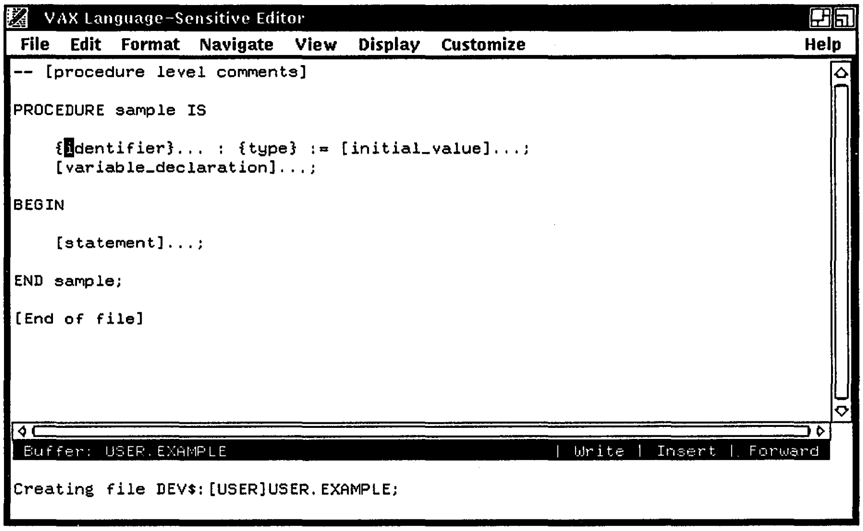
1. Press CTRL/K (the ERASE PLACEHOLDER/FORWARD key) to remove ([parameter list] . . .).

   The cursor is now positioned on [variable_declaration] . . . .

2. Press CTRL/E (the EXPAND key).

   Figure 2–4 shows the resulting screen.

Figure 2–4: Using a List Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample IS

    {identifier}... : {type} := [initial_value]...;
    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]
```

```
Buffer: USER.EXAMPLE                          | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

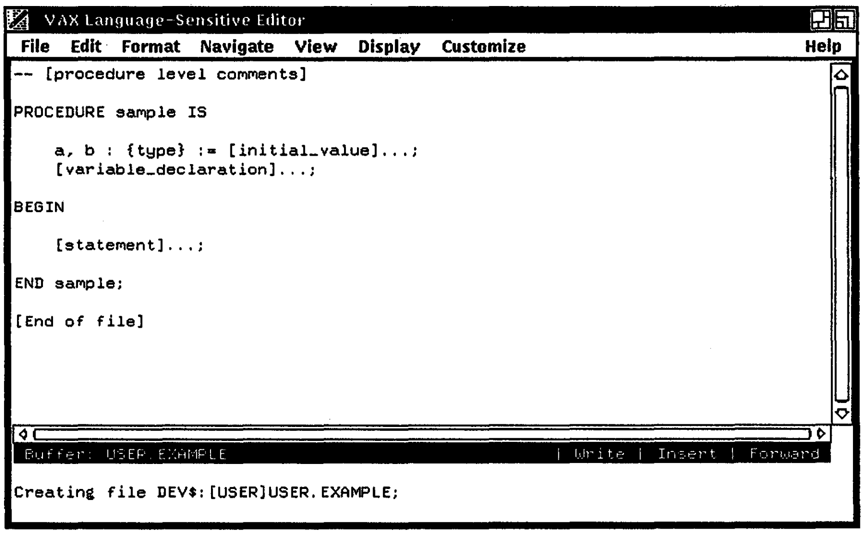
2.3.3 Typing over List Placeholders

The cursor is now positioned on the list placeholder {identifier} . . . . A list placeholder, indicated by the ellipsis (. . .), is automatically duplicated whenever you type over it or expand it.

1. Type the letter *a* over {identifier} . . . .
2. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) to move the cursor to [identifier] . . . .
3. Type the letter *b* over [identifier] . . . .
4. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) to move the cursor to [identifier] . . . .
5. Press CTRL/K (the ERASE PLACEHOLDER/FORWARD key) to remove [identifier] . . . .

   Figure 2–5 shows the resulting screen.

Figure 2–5: Typing over a List Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : {type} := [initial_value]...;
    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]
```

```
Buffer: USER.EXAMPLE                    | Write | Insert | Forward
```

```
Creating file DEV$:[USER]USER.EXAMPLE;
```

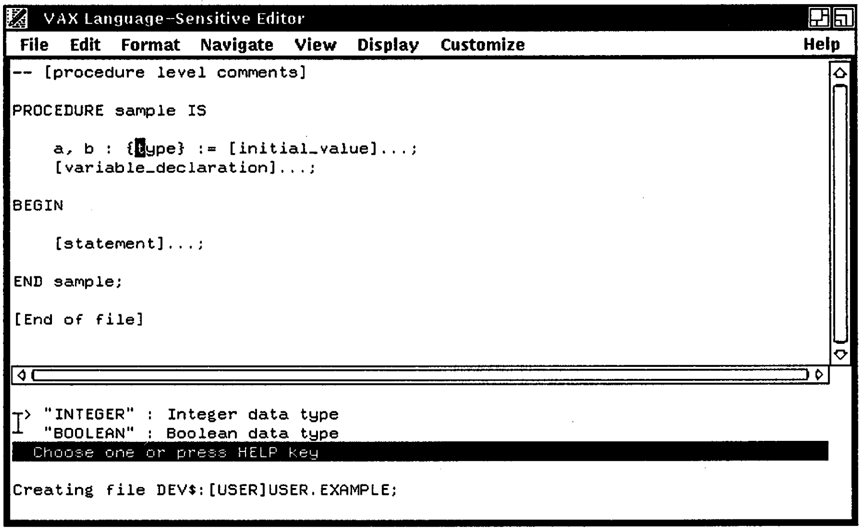
2.3.4 Expanding Menu Placeholders

The cursor is now on {type}.

1. Press CTRL/E (the EXPAND key).

   Figure 2–6 shows the resulting screen.

Figure 2–6: Using a Menu Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : {type} := [initial_value]...;
    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]
```

```
 > "INTEGER" : Integer data type
   "BOOLEAN" : Boolean data type
  Choose one or press HELP key

Creating file DEV$:[USER]USER.EXAMPLE;
```

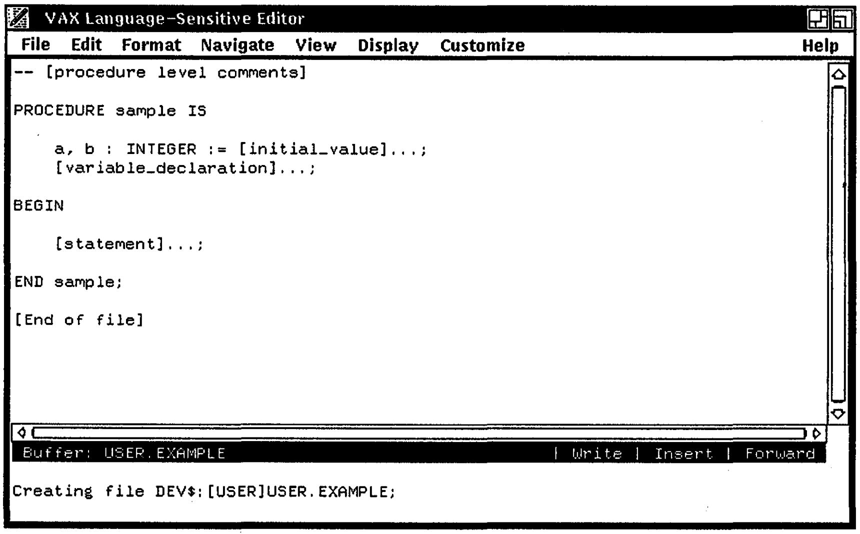
A menu of options appears at the bottom of the screen. The options are INTEGER and BOOLEAN. The text after each menu option describes what will be inserted into your buffer if you select that option.

The up and down arrow keys allow you to move the indicator to the desired option in the menu.

2. Press CTRL/E (the EXPAND key) or the ENTER or RETURN key to select INTEGER from the menu.

   The menu is removed, and option INTEGER is inserted into the buffer. The cursor is now positioned on [initial_value] . . . . Figure 2–7 shows the resulting screen.

Figure 2–7: Selecting a Menu Item


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

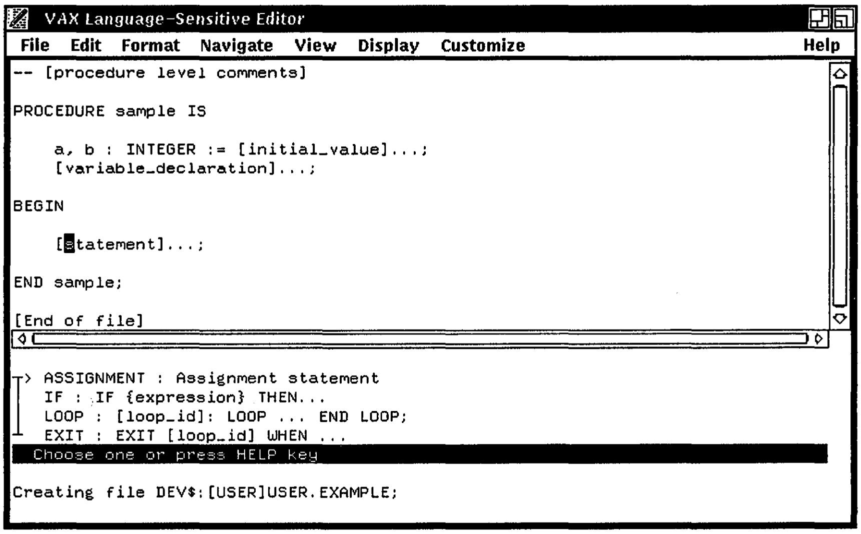
```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]
```

```
Buffer: USER.EXAMPLE                    | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

3. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) twice to move the cursor to [statement] . . . .
4. Press CTRL/E (the EXPAND key).

   Figure 2–8 shows the resulting screen.

Figure 2–8: Using Tokens in Menu Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

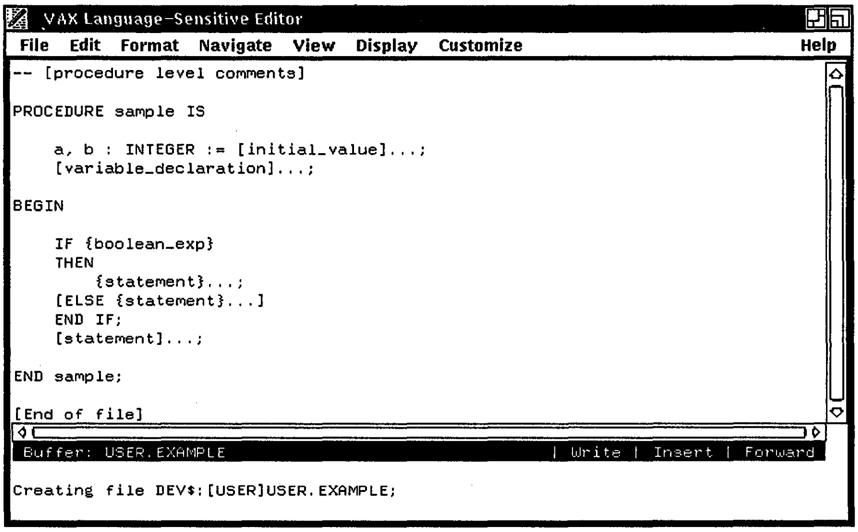
```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    [statement]...;

END sample;

[End of file]

-> ASSIGNMENT : Assignment statement
   IF : IF {expression} THEN...
   LOOP : [loop_id]: LOOP ... END LOOP;
   EXIT : EXIT [loop_id] WHEN ...
 Choose one or press HELP key

Creating file DEV$:[USER]USER.EXAMPLE;
```

A menu of options appears at the bottom of the screen. The options ASSIGNMENT and IF are tokens that you can expand into templates. The text after each token describes what will be inserted into your buffer if you select that token.

5. Use the down arrow key to move to the IF token.
6. Press CTRL/E (the EXPAND key) or the ENTER or RETURN key while the indicator is on the IF token.

   Figure 2–9 shows the resulting screen.

Figure 2–9: Selecting a Token from a Menu


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    IF {boolean_exp}
    THEN
        {statement}...;
    [ELSE {statement}...]
    END IF;
    [statement]...;

END sample;

[End of file]
 Buffer: USER.EXAMPLE                              | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

The cursor is now positioned on {boolean_exp}.

2.3.5 Expanding Tokens

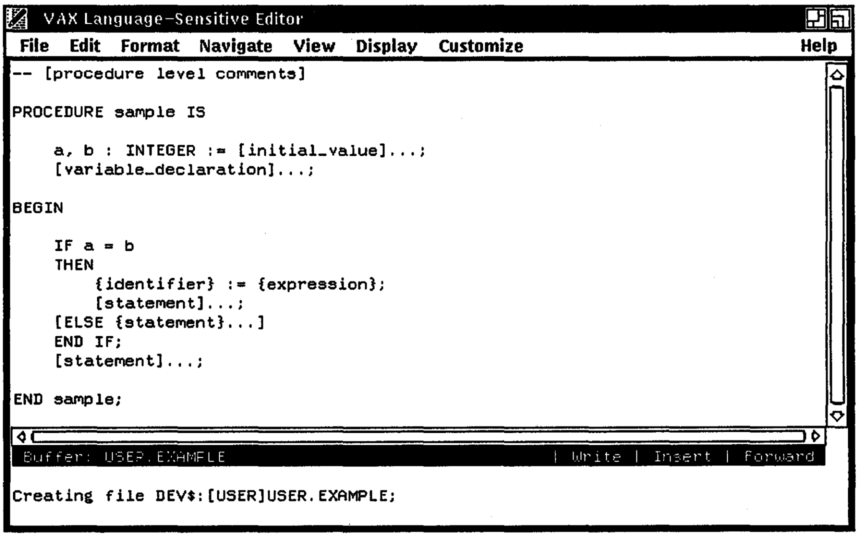
1. Type the expression *a* = *b*.
   This replaces {boolean_exp}.
2. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) to position the cursor on the first list placeholder {statement} . . . .
3. Type the text *assign*.
4. Press CTRL/E (the EXPAND key).

You do not have to type the entire token name ASSIGNMENT; with LSE, you can abbreviate token names.

In this case, you are typing the token *assignment* over the {statement} . . . placeholder. However, a token name does not have to be typed over a placeholder. You can type a token name anywhere in the editing buffer and press CTRL/E to produce the template for that token.

The separator text, a semicolon, is automatically placed after the assignment statement. Figure 2–10 shows the resulting screen.

Figure 2–10: Expanding a Token


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
-- [procedure level comments]

PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    IF a = b
    THEN
        {identifier} := {expression};
        [statement]...;
    [ELSE {statement}...]
    END IF;
    [statement]...;

END sample;
```

```
Buffer: USER.EXAMPLE                    | Write | Insert | Forward
```

```
Creating file DEV$:[USER]USER.EXAMPLE;
```

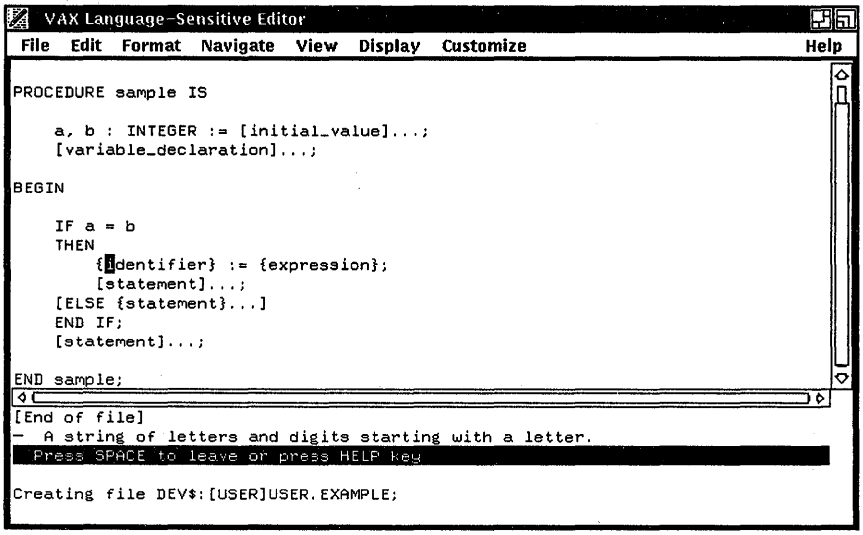
2.3.6 Expanding Terminal Placeholders

The cursor is now positioned on the terminal placeholder {identifier}. If you press CTRL/E while positioned on a terminal placeholder, LSE displays information to help you supply the necessary text.

1. Press CTRL/E (the EXPAND key) to see the information for {identifier}.

   Figure 2–11 shows the resulting screen.

Figure 2–11: Expanding a Terminal Placeholder


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    IF a = b
    THEN
        {identifier} := {expression};
        [statement]...;
    [ELSE {statement}...]
    END IF;
    [statement]...;

END sample;
[End of file]
-  A string of letters and digits starting with a letter.
  Press SPACE to leave or press HELP key

Creating file DEV$:[USER]USER.EXAMPLE;
```

2. Press the spacebar to remove the information about {identifier} . . . .

2.3.7 Entering Pseudocode

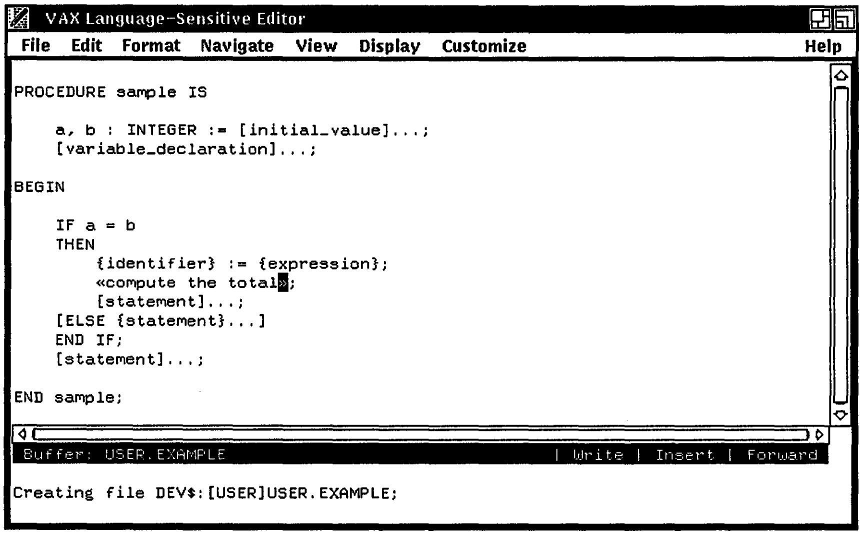
LSE provides pseudocode placeholders that enable you to enter design information into your source files. To enter pseudocode, do the following:

1. Press CTRL/N (the GOTO PLACEHOLDER/FORWARD key) twice to position the cursor on the [statement]... placeholder.
2. Press PF1-spacebar (the ENTER PSEUDOCODE key).

   LSE inserts the special brackets « and » into the buffer that delimit pseudocode placeholders and positions the cursor within the delimiters.
3. Type the text *compute the total* at the cursor position.

Figure 2–12 shows the resulting screen.

Figure 2–12: Typing Pseudocode


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    IF a = b
    THEN
        {identifier} := {expression};
        «compute the total»;
        [statement]...;
    [ELSE {statement}...]
    END IF;
    [statement]...;

END sample;
```

```
Buffer: USER.EXAMPLE                                | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

2.3.8 Moving Pseudocode to Comments

With LSE, you can move pseudocode placeholder text to program comments to preserve design information when implementing your programs. To move text from a pseudocode placeholder to a block comment, do the following:

1. Press PF1-B (the ENTER COMMENT/BLOCK key) while positioned on the pseudocode placeholder.

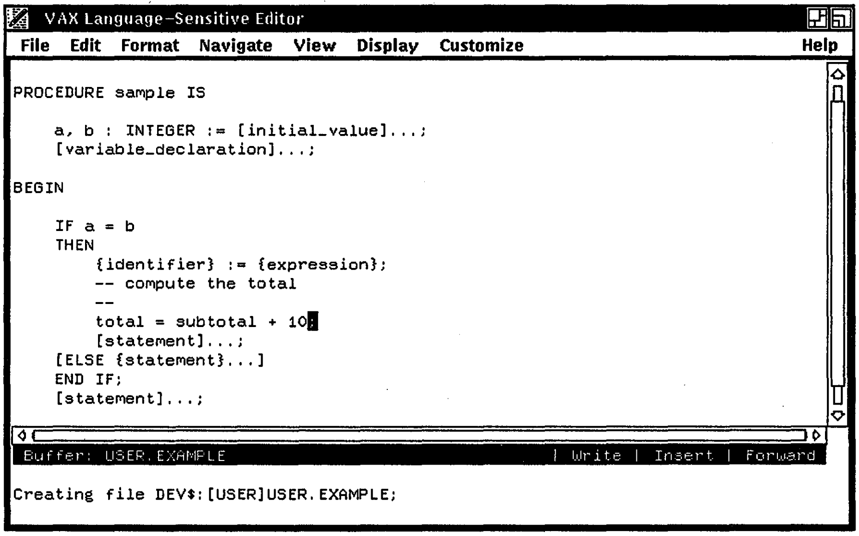
   LSE places the pseudocode placeholder text in a comment. In addition, LSE inserts the {tbs} placeholder into the buffer to provide an easy way to enter your code.

2. Type the text *total = subtotal + 10* on the {tbs} placeholder.

   Notice that the {tbs} placeholder is removed when you type over it.

Figure 2–13 shows the resulting screen.

Figure 2–13: Pseudocode to Comments


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
PROCEDURE sample IS

    a, b : INTEGER := [initial_value]...;
    [variable_declaration]...;

BEGIN

    IF a = b
    THEN
        {identifier} := {expression};
        -- compute the total
        --
        total = subtotal + 10
        [statement]...;
    [ELSE {statement}...]
    END IF;
    [statement]...;
```

```
Buffer: USER.EXAMPLE                      | Write | Insert | Forward

Creating file DEV$:[USER]USER.EXAMPLE;
```

2.3.9 Ending the Sample Session

To leave LSE, press CTRL/Z to get the LSE> prompt. If you want to save modifications to the file, type the EXIT command. If you do not want to save the file or any modifications, type the QUIT command.

When you use the EXIT command, LSE remembers the original input file specification and the current cursor position. Thus, you can return to editing a file at exactly the same position by typing the LSEDIT command without specifying any input file parameters. This information is lost, however, when you log out.

2.4 Compiling Source Code

While writing your program, you can use the COMPILE and REVIEW commands to compile your code and review compilation errors without leaving the editing session. Supported VMS compilers generate a file of compile-time diagnostic information that LSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier.

The COMPILE command issues a DCL command in a subprocess to invoke the appropriate compiler. LSE checks to see if the language supports diagnostics capabilities. If so, LSE appends the /DIAGNOSTICS qualifier to the COMPILE command.

For example, if you issue the COMPILE command while in the buffer TEST.ADA, the resulting DCL command is as follows:

```
$ ADA DEV:[DIRECTORY]TEST.ADA/DIAGNOSTICS=DEV:[DIRECTORY]TEST.DIA
```

LSE supports all of the compiler's command qualifiers as well as user-supplied command procedures. You can specify DCL qualifiers, such as /LIBRARY, when invoking the compiler from LSE. In addition, you can specify the /DESIGN qualifier to process designs. For example, to invoke the compiler with the /DESIGN qualifier, type the following command:

```
LSE>  COMPILE $/DESIGN
```

The REVIEW command displays any diagnostic messages that resulted from a compilation. LSE displays the compilation errors in one window, with the corresponding source code displayed in a second window. (See the REVIEW command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details.)

Multiwindow capability allows you to review your errors while examining the associated source code. This eliminates tedious steps in the error correction process and helps ensure that all the errors are fixed before looping back through the compilation process.

LSE provides several commands to help you review errors and examine your source code. Table 2–3 lists these commands and their default key bindings. Table 2–4 lists the review commands and their functions.

Table 2–3: Commands for Reviewing Compilation Errors

Command	EDT Keypad	EVE VT100 Keypad	EVE VT200 Keypad
COMPILE/REVIEW			
COMPILE			
REVIEW			
END REVIEW			
GOTO SOURCE	CTRL/G	CTRL/G	CTRL/G
NEXT STEP	CTRL/F	CTRL/F	CTRL/F
PREVIOUS STEP	CTRL/B	CTRL/B	CTRL/B

Table 2–4: Review Commands and Their Functions

Command	Function
COMPILE/REVIEW	The COMPILE/REVIEW command compiles the contents of a buffer and then displays a set of diagnostic messages that results from the compilation.
COMPILE	The COMPILE command compiles the contents of a buffer.
REVIEW	The REVIEW command selects and displays a set of diagnostic messages that results from a compilation.
END REVIEW	The END REVIEW command ends an LSE REVIEW session.
GOTO SOURCE	The GOTO SOURCE key (CTRL/G) displays the source corresponding to the current diagnostic item.
NEXT STEP	The NEXT STEP key (CTRL/F) moves the cursor forward to the next error.
PREVIOUS STEP	The PREVIOUS STEP key (CTRL/B) moves the cursor back to the previous error.

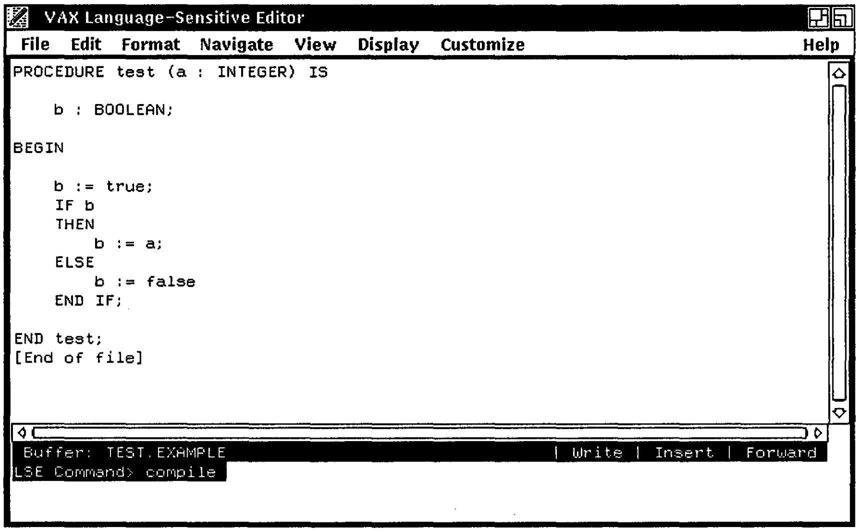
The following example demonstrates how to compile and review errors within your editing session. Note that this example and the sample language, called EXAMPLE, do not have compiler support. Therefore, when experimenting with the COMPILE command, you should construct your own example using one of the languages that has compiler support.

If you are editing a program and you want to compile it, use the following steps:

1. Press CTRL/Z to get the LSE> prompt and type the COMPILE command.

   Figure 2–14 shows the resulting screen.

Figure 2–14: Issuing the COMPILE Command


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
PROCEDURE test (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
        b := a;
    ELSE
        b := false
    END IF;

END test;
[End of file]
```

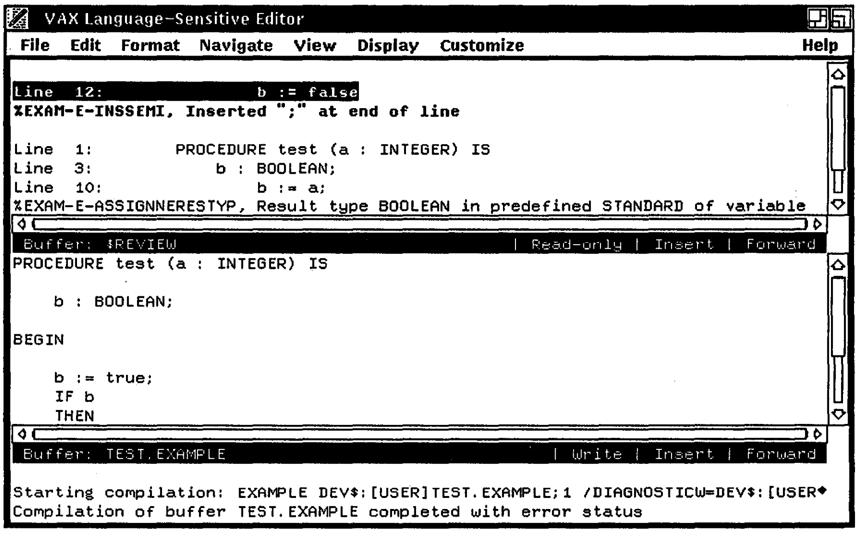
```
 Buffer: TEST.EXAMPLE                                  | Write | Insert | Forward
LSE Command> compile
```

   While the compilation is running, you may continue to use LSE to edit. When the compilation ends, a message appears in the message buffer.

2. Type the REVIEW command at the LSE> prompt.

   The REVIEW command instructs LSE to review the compilation errors generated by the COMPILE command. Figure 2–15 shows your screen with the compilation errors in the top window and the source code in the bottom window.

Figure 2–15: Result of Issuing the REVIEW Command


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  12:                  b := false
%EXAM-E-INSSEMI, Inserted ";" at end of line

Line  1:        PROCEDURE test (a : INTEGER) IS
Line  3:            b : BOOLEAN;
Line  10:               b := a;
%EXAM-E-ASSIGNNERESTYP, Result type BOOLEAN in predefined STANDARD of variable
 Buffer: $REVIEW                                   | Read-only | Insert | Forward
PROCEDURE test (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
 Buffer: TEST.EXAMPLE                                  | Write | Insert | Forward

Starting compilation: EXAMPLE DEV$:[USER]TEST.EXAMPLE;1 /DIAGNOSTICW=DEV$:[USER◆
Compilation of buffer TEST.EXAMPLE completed with error status
```

3. Press CTRL/Z to move from the LSE> prompt to the buffer and begin reviewing the errors.

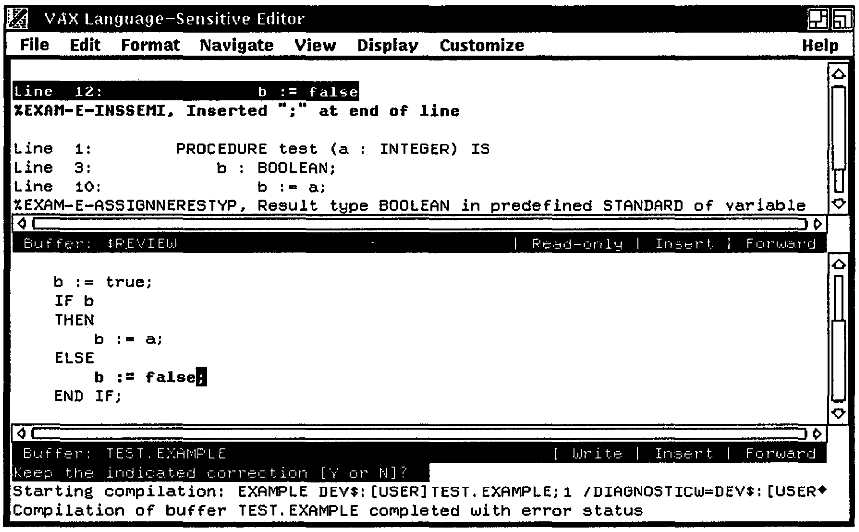
   You can use the NEXT STEP key (CTRL/F) and the PREVIOUS STEP key (CTRL/B) to step from error to error in the $REVIEW buffer. If you want to correct the source code associated with a particular error, press the GOTO SOURCE key (CTRL/G) while the cursor is positioned on that error.

   In addition to the NEXT STEP and PREVIOUS STEP commands, you can use any cursor movement keys to move around in the $REVIEW buffer. You can type the GOTO SOURCE command to go to the associated source code. Thus, you can examine other lines of code associated with an error.

4. Press CTRL/G (the GOTO SOURCE key) to correct the source code associated with the first error.

   Figure 2–16 shows the resulting screen.

Figure 2–16: GOTO SOURCE Command


VAX Language–Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  12:                    b := false
%EXAM-E-INSSEMI, Inserted ";" at end of line

Line  1:        PROCEDURE test (a : INTEGER) IS
Line  3:              b : BOOLEAN;
Line  10:                   b := a;
%EXAM-E-ASSIGNNERESTYP, Result type BOOLEAN in predefined STANDARD of variable
 Buffer: $REVIEW                              | Read-only | Insert | Forward

    b := true;
    IF b
    THEN
          b := a;
    ELSE
          b := false;
    END IF;

 Buffer: TEST.EXAMPLE                               | Write | Insert | Forward
Keep the indicated correction [Y or N]?
Starting compilation: EXAMPLE DEV$:[USER]TEST.EXAMPLE;1 /DIAGNOSTICW=DEV$:[USER◆
Compilation of buffer TEST.EXAMPLE completed with error status
```

Corrections accompany some errors. For example, in Figure 2–16, the compiler supplied a correction for the missing semicolon. You can accept or reject the correction.

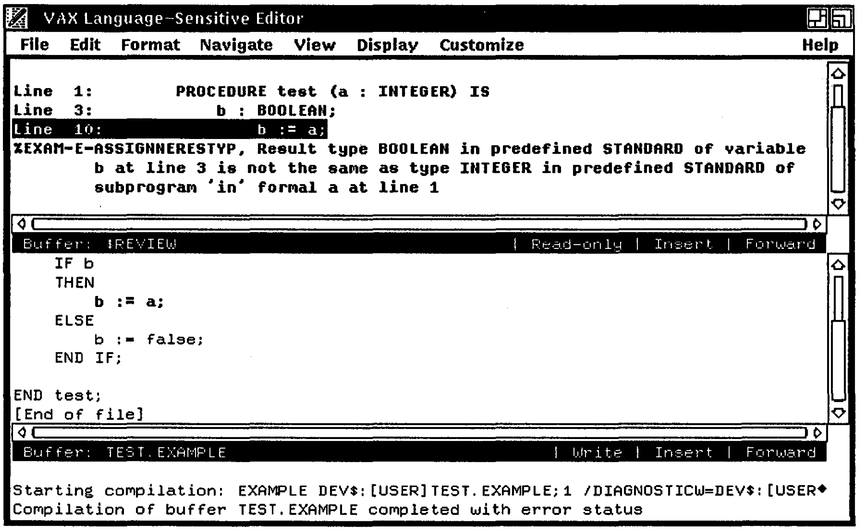
5. Type Y after the following prompt to accept the correction:

```
Keep the indicated correction [Y OR N]?
```

6. Press CTRL/F (the NEXT STEP key) to move to the next error.
7. Press CTRL/G (the GOTO SOURCE key).

   Figure 2–17 shows the resulting screen.

Figure 2–17: NEXT STEP Command


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  1:         PROCEDURE test (a : INTEGER) IS
Line  3:             b : BOOLEAN;
Line  10:                b := a;
%EXAM-E-ASSIGNMERESTYP, Result type BOOLEAN in predefined STANDARD of variable
        b at line 3 is not the same as type INTEGER in predefined STANDARD of
        subprogram 'in' formal a at line 1
```

```
 Buffer: $REVIEW                                  | Read-only | Insert | Forward
```

```
    IF b
    THEN
        b := a;
    ELSE
        b := false;
    END IF;

END test;
[End of file]
```

```
 Buffer: TEST.EXAMPLE                                 | Write | Insert | Forward
```

```
Starting compilation: EXAMPLE DEV$:[USER]TEST.EXAMPLE;1 /DIAGNOSTICW=DEV$:[USER◆
Compilation of buffer TEST.EXAMPLE completed with error status
```

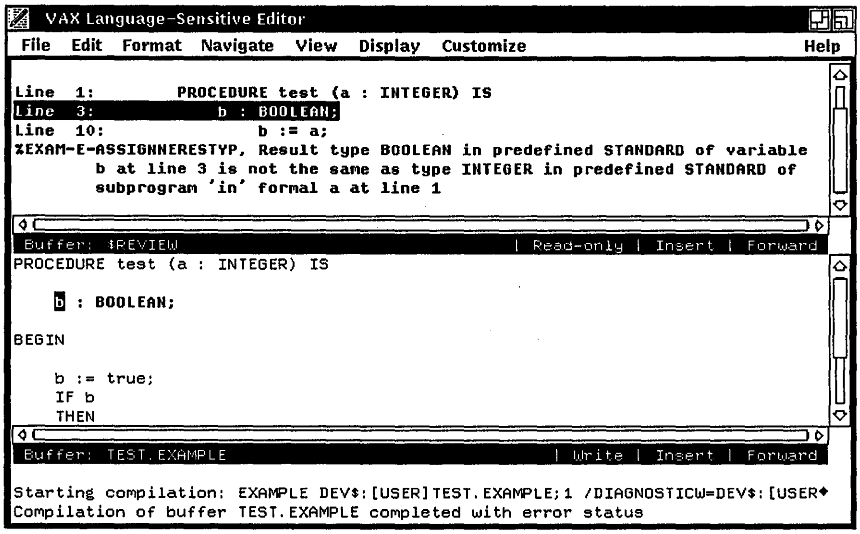
Notice that there are several source lines displayed with this error in the $REVIEW buffer.

8. Press PF1-up arrow (the PREVIOUS WINDOW key) to go to the $REVIEW buffer.
9. Press the up arrow key to move the cursor to line 3: b : BOOLEAN;.

   Notice that line 10 is highlighted and the cursor moved to the declaration of b.
10. Press CTRL/G (the GOTO SOURCE key). Line 3 is now highlighted.

   Figure 2–18 shows your screen with the cursor on the declaration of b.

Figure 2–18: GOTO SOURCE Command


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  1:        PROCEDURE test (a : INTEGER) IS
Line  3:            b : BOOLEAN;
Line  10:           b := a;
%EXAM-E-ASSIGNMERESTYP, Result type BOOLEAN in predefined STANDARD of variable
        b at line 3 is not the same as type INTEGER in predefined STANDARD of
        subprogram 'in' formal a at line 1
 Buffer: $REVIEW                              | Read-only | Insert | Forward
PROCEDURE test (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
 Buffer: TEST.EXAMPLE                             | Write | Insert | Forward

Starting compilation: EXAMPLE DEV$:[USER]TEST.EXAMPLE;1 /DIAGNOSTICW=DEV$:[USER◆
Compilation of buffer TEST.EXAMPLE completed with error status
```

When you finish correcting your program, you can return to a single window containing the source code.

11. Press CTRL/Z to get the LSE> prompt and type the END REVIEW command.

2.5 Invoking LSE from VMS Debugger and from VAX Performance and Coverage Analyzer

While in the VMS Debugger (debugger) or the VAX Performance and Coverage Analyzer (PCA), you can invoke LSE to edit your code.

The command syntax to invoke LSE from the debugger or PCA is as follows:

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

LSE positions the cursor at the line in the file that corresponds to the specified module and line in the debugger or the VAX Performance and Coverage Analyzer. The default file and line are taken from the current source display.

The rules for specifying the module-name and line-number qualifiers are as follows.

- If both module-name and line-number are specified, their values determine the module and line at which the file is positioned in LSE.
- If only line-number is specified, the module is assumed to be the module of the current source display.
- If neither module-number nor line-number is specified, the module is assumed to be the module of the current source display, and the line number is assumed to be the central line in the window of that display.

When you invoke LSE from the debugger or PCA, a subprocess is spawned for the editing session. Control automatically returns to the debugger or the VAX Performance and Coverage Analyzer after the editing is completed.

You use the edit command to tell LSE exactly what file you are editing. The EDIT command also checks the file in the debugger or the VAX Performance and Coverage Analyzer source display to see if it is the most recent version. LSE always edits the most recent version of the file. If the displayed version is not the most recent version, LSE issues an error message.

When debugging and editing code, it is faster to use the EDIT/EXIT command at the DBG> prompt rather than returning to the debugger and typing the EXIT command. This is useful when only minor editing must be done before recompiling.

2.6 LSE Command Line

This section describes the format of the LSE command line and includes detailed descriptions of each command line qualifier.

The LSEDIT command invokes LSE. The LSEDIT command has the following form:

```
LSEDIT [/qualifiers] [file-spec]
```

/qualifiers
Specifies LSEDIT command qualifiers.

file-spec
Specifies the file to be edited. It must be a VMS file specification. LSE uses the setting of the SET SOURCE_DIRECTORY command or the corresponding LSE$SOURCE logical name to resolve the file specification.

LSE reads the file into a buffer if the file exists. The buffer name is taken from the name and type of the file specification in the command line. The file type determines the language for the buffer. For example, .FOR is the file type for FORTRAN, .PLI is the file type for PL/I, and .PAS is the file type for Pascal. If the file does not exist, it is created when you use the EXIT command to leave LSE.

If you do not specify a file name or file type in your file specification, LSE uses the file name or file type specified in your last LSEDIT command, provided you issued the EXIT command to end that editing session. Furthermore, the cursor is positioned at the same place as when you last left LSE. The file name, type, and position are collectively called the **current file information**. The current file information is updated only when you use the EXIT command to leave LSE. If you use the /NOCURRENT_FILE qualifier, LSE does not use the file specification from the previous LSEDIT command as the input file specification. The QUIT command or CTRL/Y does not change the current file information.

2.6.1 LSE Command Line Qualifiers

You can use several command line qualifiers to provide additional information to LSE on how to handle your files. Table 2–5 lists these command line qualifiers. Detailed descriptions of the qualifiers and their defaults, indicated by (D), follow the table.

Table 2–5: LSE Command Line Qualifiers

Qualifier	Default
/[NO]COMMAND=file-spec	See text
/[NO]CREATE	/CREATE
/[NO]CURRENT_FILE	/CURRENT_FILE
/[NO]DEBUG	/NODEBUG
/[NO]DISPLAY	/DISPLAY=CHARACTER_CELL
/[NO]ENVIRONMENT=file-spec-list	See text
/[NO]INITIALIZATION=file-spec	See text
/[NO]INTERFACE	/INTERFACE=CHARACTER_CELL
/[NO]JOURNAL[=file-spec]	/JOURNAL

(continued on next page)

Table 2–5 (Cont.): LSE Command Line Qualifiers

Qualifier	Default
/LANGUAGE=language	
/[NO]MODIFY	
/[NO]OUTPUT[=file-spec]	/OUTPUT
/[NO]READ_ONLY	
/[NO]RECOVER	/NORECOVER
/[NO]SECTION=file-spec	/SECTION=LSE$SECTION
/START_POSITION=(line,character)	
/[NO]SYSTEM_ENVIRONMENT	/SYSTEM_ENVIRONMENT=LSE$SYSTEM_ ENVIRONMENT
/[NO]WRITE	

/COMMAND=file-spec
/NOCOMMAND

Specifies a file containing VAXTPU statements to be executed as part of LSE initialization.

If you specify the /NOCOMMAND qualifier, LSE does not use a VAXTPU initialization command file. (See the *VAX Text Processing Utility Reference Manual* for more information.)

You can define the logical name LSE$COMMAND to point to a file containing VAXTPU statements. If neither the /COMMAND nor /NOCOMMAND qualifier appears on the command line, LSE attempts to translate the logical name LSE$COMMAND. If it has a translation, that value is used in the same way as the /COMMAND qualifier value.

/CREATE (D)
/NOCREATE

Controls whether LSE creates a new file when the specified input file is not found. By default, LSE provides a buffer in which to create the file. When you exit from LSE or write out the contents of the buffer with the WRITE or COMPILE commands, LSE creates a new file with the input file specification in the appropriate directory.

When you specify the /NOCREATE qualifier on the LSE command line and the name of a file to edit, and the named file does not exist, LSE displays an error message and places you in a buffer called $MAIN.

/CURRENT_FILE (D)
/NOCURRENT_FILE
Specifies whether or not LSE uses the last file edited as the input file specification if no file is specified on the command line.

/DEBUG
/NODEBUG (D)
Specifies whether LSE loads, compiles, and executes a file implementing a VAXTPU debugger. If /DEBUG is specified, LSE reads, compiles, and executes the contents of a debugger file before executing the procedure TPU$INIT_PROCEDURE and before executing the command file. (For more information on VAXTPU's initialization sequence, see the *VAX Text Processing Utility Manual.*)

By default, LSE does not load a debugger. If you specify that a debugger is to be loaded but do not supply a file specification, LSE loads the file SYS$SHARE:LSE$DEBUG.TPU. (For more information on how to use the default VAXTPU debugger, see the *VAX Text Processing Utility Manual.*)

To use a debugger file other than the default, use the /DEBUG qualifier and specify the device, directory, and file name of the debugger to be used. If you specify only the file name, LSE searches SYS$SHARE for the file. You can define the logical name LSE$DEBUG to specify a file containing a debugger program. Once you define this logical name, if you use /DEBUG without specifying a file, LSE calls the file specified by LSE$DEBUG.

/DISPLAY=CHARACTER_CELL (D)
/DISPLAY=DECWINDOWS
/DISPLAY=screen_manager_filespec
/NODISPLAY
Specifies which screen manager you want to run.

The /DISPLAY command qualifier is optional. By default, LSE uses the character-cell screen manager. As an alternative to the /DISPLAY qualifier, you can define the logical name LSE$DISPLAY_MANAGER as DECWINDOWS, CHARACTER_CELL, or as a screen-manager file specification.

If you specify /DISPLAY=CHARACTER_CELL, LSE uses the character-cell screen manager, which runs in a DECterm (or VWS) terminal emulator or on a physical terminal.

If you specify /DISPLAY=DECWINDOWS, LSE uses the DECwindows screen manager, which creates a DECwindows window in which to run LSE.

You cannot use the /NODISPLAY qualifier if the logical name LSE$DISPLAY_MANAGER is pointing to the DECwindows window manager.

/ENVIRONMENT=file-spec-list
/NOENVIRONMENT(D)

Specifies the name of one or more binary environment files containing LSE language, token, placeholder, alias, or package definitions. LSE reads in these definitions as part of the LSE startup. If you specify more than one file, you must enclose the files in parentheses and separate them with commas.

If definitions or deletions of items appear in more than one file, the definition that appears in the file listed first takes precedence.

SYS$LIBRARY: is the default device. The default file type is .ENV.

The logical name LSE$ENVIRONMENT is an alternative to the /ENVIRONMENT command qualifier. If the /ENVIRONMENT or /NOENVIRONMENT qualifier does not appear on the command line, LSE attempts to translate the logical name LSE$ENVIRONMENT. If it has a translation, the value is used in the same way as the /ENVIRONMENT qualifier value. LSE translates the first ten indexes of the logical name LSE$ENVIRONMENT.

See the SAVE ENVIRONMENT command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for information on using environment files.

/INITIALIZATION=file-spec
/NOINITIALIZATION

Specifies the name of a file containing a sequence of LSE commands to be executed as part of the LSE startup. Usually this file contains the occurrences of the DEFINE KEY and DEFINE COMMAND commands.

The logical name LSE$INITIALIZATION is an alternative to the /INITIALIZATION qualifier. If /INITIALIZATION or /NOINITIALIZATION does not appear on the command line, LSE attempts to translate the logical name LSE$INITIALIZATION. If it has a translation, the value is used in the same way as the /INITIALIZATION qualifier value.

/INTERFACE=CHARACTER_CELL (D)
/INTERFACE=DECWINDOWS
/INTERFACE=screen_manager_filespec

Specifies which screen manager you want to run.

The /INTERFACE qualifier is optional. By default, LSE uses the character-cell screen manager. As an alternative to the /INTERFACE qualifier, you can define the logical name LSE$DISPLAY_MANAGER as DECWINDOWS, CHARACTER_CELL, or as a screen-manager file specification.

If you specify /INTERFACE=CHARACTER_CELL, LSE uses the character-cell screen manager, which runs in a DECterm (or VWS) terminal emulator or on a physical terminal.

If you specify /INTERFACE=DECWINDOWS, LSE uses the DECwindows screen manager, which creates a DECwindows window in which to run LSE.

/JOURNAL
/JOURNAL[=file-name] (D)
/NOJOURNAL
Enables buffer-change journaling and keystroke journaling. The /JOURNAL qualifier enables buffer-change journaling *only*. The /JOURNAL=file-name qualifier enables buffer-change journaling *and* keystroke journaling. One buffer-change journal file is created per buffer. One keystroke journal file is created for the entire editing session. The *file-name* argument specifies the name for the edit session journal file. The default file name is the file name from the input file. The default file type for journal files is .TJL.

If you do not want to create a journal file of either type, use the /NOJOURNAL qualifier.

/LANGUAGE=language
Sets the language for the current input file, overriding the language indicated by the input file's file type.

/MODIFY
/NOMODIFY
Specifies whether the buffer you create is modifiable or unmodifiable. If you specify the /MODIFY qualifier, the LSEDIT command creates a modifiable buffer. If you specify the /NOMODIFY qualifier, the LSEDIT command creates an unmodifiable buffer. If you do not specify either qualifier, LSE determines the buffer's modifiable status from the read-only/write setting. By default, a read-only buffer is unmodifiable and a write buffer is modifiable.

/OUTPUT[=file-spec] (D)
/NOOUTPUT
Specifies the name of the file LSE is to create when you exit from the editing session. Specifying a file specification on the /OUTPUT qualifier causes LSE to ignore the current file information. By default, LSE creates a new version of the input file.

Missing components of the file specification in the /OUTPUT qualifier take their values from the corresponding fields of the input file specification.

When you exit from the editing session, LSE writes other buffers to their associated files if the buffer contents have been modified during the session. If you specify the /NOOUTPUT qualifier, LSE prevents the writing back of the main buffer when you exit.

/READ_ONLY
/NOREAD_ONLY
Specifies that LSE create a read-only buffer for the input file. LSE does not create a new output file. Any changes to the file are lost when you exit from the editing session. This qualifier does not affect the writing back of other buffers to their associated files if they were modified during the editing session.

If the /[NO]READ_ONLY qualifier is not specified explicitly, the read/write status of the buffer for the input file is determined by the default settings of the SET DIRECTORY command or LSE$READ_ONLY_DIRECTORY logical name.

/RECOVER
/NORECOVER (D)
Directs LSE to use the latest version of the input file's corresponding journal file to recover changes that may have been lost during an abnormal LSE termination. LSE uses the buffer-change journal files as necessary.

When you recover a session, all files must be in the same state as they were at the start of the editing session that is being recovered. You must issue the LSEDIT/RECOVER command with the same qualifiers, initialization file, section file, and environment file as you did for the session being recovered. All terminal characteristics must also be in the same state as they were at the start of the editing session being recovered. If you changed the width or page length of the terminal, you must change it back to the value it had at the start of the editing session you want to recover. Check especially the following values by using the DCL command SHOW TERMINAL:

- Device_type
- Edit_mode
- Eightbit
- Page
- Width

See Chapter 3 for more details on recovering edits.

/SECTION=file-spec
/SECTION=LSE$SECTION (D)
/NOSECTION
Specifies whether LSE is to map a section file containing VAXTPU procedures, key definitions, and variables. By default, LSE maps section file LSE$SECTION. If you specify another file specification, LSE applies the default SYS$LIBRARY:.TPU$SECTION when it opens the file.

If you specify the /NOSECTION qualifier, LSE does not use a section file, and many LSE commands will not work. Therefore, when using the /NOSECTION qualifier, you should specify the /COMMAND qualifier. The command file should use only standard VAXTPU built-ins.

/START_POSITION=(line,character)
Specifies the starting line and character in the file (top-of-file is /START_POSITION=(1,1)). If you do not specify /START_POSITION, LSE starts either at the top of the file or at the position of the cursor when you last edited the file.

/SYSTEM_ENVIRONMENT=file-spec
/SYSTEM_ENVIRONMENT=LSE$SYSTEM_ENVIRONMENT (D)
/NOSYSTEM_ENVIRONMENT
Specifies the name of a system environment file. The difference between the file specified by this qualifier and the file specified by the /ENVIRONMENT qualifier is that definitions from the system environment file are not saved by a SAVE ENVIRONMENT command.

The default device is SYS$LIBRARY: and the default file type is .ENV.

/WRITE
/NOWRITE
Specifies that LSE create a new output file when you exit from the editing session. Any changes you make to the file are saved.

If the /[NO]WRITE qualifier is not specified explicitly, the read/write status of the buffer for the input file is determined by the default settings of the SET DIRECTORY command or LSE$READ_ONLY_DIRECTORY logical name.

2.7 Running LSE/DECwindows in a Separate Process

When you invoke LSE with the /DISPLAY=DECWINDOWS qualifier to run LSE/DECwindows, LSE runs in a DECwindows application window and not in a terminal window. To run LSE from a terminal session without tying up the terminal windows for the duration of the session, type the following command:

```
$   SPAWN /NOWAIT/INPUT=NL: LSEDIT/NOCURRENT/DISPLAY=DECWINDOWS
```

The /INPUT=NL: qualifier will prevent terminating the subprocess running LSE if you press CTRL/Y in the parent process. If you run LSE this way, you must avoid stopping the parent process.

You can also invoke LSE from FileView. The LSEDIT verb is automatically defined when LSE is installed, but you must add the verb to the menu where you want it.

In addition, you can run LSE from a detached processing by typing the following command:

```
$   RUN/DET/INP=LSE.COM /AUTHORIZE -
_$  sys$system:loginout.exe /OUT=lse.log
```

The file LSE.COM invokes LSE in DECwindows mode. A typical LSE.COM file might contain the following:

```
$ @LOGIN
$ SET DISP/CRE/NODE=name/TRANSPORT=LOCAL
$ LSEDIT/DISP=DECW/NOCURRENT
```

The LSE.LOG file is useful for diagnosing any problems that might arise.

If you change the transport to be DECnet, you can run LSE on a different node than the workstation that you are logged into and still have the LSE window on the workstation. If you have a workstation with little memory, this can give you significantly better performance. You need to add the node name and account of the non-local node to the list of authorized users using the Session Manager Customize/Security... menu item.

Chapter 3

Performing Editing Tasks

This chapter describes the editing capabilities of LSE. LSE provides you with many features, including multiple buffer and window support, that simplify the task of editing major documents or source files. The file location and manipulation facilities help you to access and modify documents or source files easily and quickly. In addition, LSE provides code elision features that allow you to view programs at various levels of detail.

Sections 3.1 and 3.2 provide details on LSE's multiple buffer and window support. Section 3.3 provides information on LSE's search and substitution features, and Section 3.4 provides information on file manipulation and directory searchlists. Section 3.5 describes how to recover edits when a system or editor failure occurs. Section 3.6 describes the code elision features, including how to expand and collapse source code, and how to edit overviews.

3.1 Using Buffers

A **buffer** is a temporary holding area that provides a workspace for editing text. You can create a new file or edit an existing file in a buffer. A buffer becomes visible when it is associated with a window that is mapped to the screen. Buffers exist only for the duration of your editing session. When you exit from LSE, the current buffer is discarded and the contents of the buffer are stored in a file.

With LSE, you can create multiple buffers. Thus, you can edit several different files in one editing session. You can create additional buffers to store portions of text that you might want to look at, but not edit, during your editing session.

System Buffers

Some buffers are used by LSE for special purposes. These are called system buffers. Unlike user buffers, system buffers do not correspond to files. You can edit a system buffer like any other buffer, but you should avoid changing its contents. By convention, system buffer names start with a dollar sign ($). The most frequently used system buffers are $DEFAULTS, $HELP, $MESSAGES, $REVIEW, and $SHOW. System buffers are not displayed by the SHOW BUFFER command unless you use the /SYSTEM_BUFFERS qualifier.

3.1.1 Buffer Attributes

Buffers have many attributes. This section provides details on buffer attributes and properties. You can use the SHOW BUFFER command to display the characteristics of one or more buffers.

Buffer Names

A buffer has a name that is displayed in the status line. Buffers are usually named by the name and type of their associated input file. The GOTO FILE and GOTO BUFFER commands can create buffers.

Insert/Overstrike

LSE has two text entry modes: insert and overstrike. In insert mode, text is inserted into the buffer at the cursor position. Text to the right of the cursor moves to the right. In overstrike mode, text typed at the cursor replaces text that is currently under the cursor.

When you start an editing session, the buffer is automatically placed in insert mode. To change the text entry mode, you can use the SET INSERT command, SET OVERSTRIKE command, or CHANGE TEXT_ENTRY_MODE command. (See Table 3–1 for buffer manipulation commands and their key bindings.)

Forward/Reverse

LSE maintains a current direction for each buffer. The current direction is displayed in the status line. This direction is used for SEARCH operations and most GOTO and ERASE commands. When you start an editing session, the buffer direction is set to forward. To set the current direction to forward, you use the SET FORWARD command. To set the current direction to reverse, you use the SET REVERSE command. Alternatively, you can use the CHANGE DIRECTION command to change the current direction. (See Table 3–1 for buffer manipulation commands and their key bindings.)

Input/Output

Buffers may have an associated input or output file. An input file is read into a buffer when the buffer is created. An output file indicates where LSE writes a buffer; this is usually a new version of an input file. You can change the output file name with the SET OUTPUT_FILE command. The GOTO FILE command creates a buffer and reads a file into it.

Read/Write

Buffers have either the read-only or write attribute. The read-only attribute indicates that the contents of the buffer is not written to a file when you exit from the editing session. The write attribute indicates that the buffer is written to a file when you exit from the editing session.

Usually, a file is associated with a buffer by the GOTO FILE command, which creates a buffer and fills it with the contents of a file. When the buffer is written, it is written to a new version of the file. If no file is associated with a buffer that has the write attribute, LSE prompts for a file specification when you exit from the editing session. Note that a buffer is written only if its contents have been modified.

Modifiable/Unmodifiable

Buffers are either modifiable or unmodifiable. Unmodifiable buffers protect the contents of a given buffer. You cannot change an unmodifiable buffer. You use the GOTO FILE/READ_ONLY and GOTO SOURCE/READ_ONLY commands to create unmodifiable buffers. If you want to modify an unmodifiable buffer, you must issue the SET MODIFY or SET WRITE command.

There are some relationships between the READ-ONLY/WRITE buffer attributes and the UNMODIFIABLE/MODIFIABLE buffer attributes. Given these attributes, a buffer may be in one of four possible states. The following list describes these states and explains how to create these states for a buffer.

- MODIFIABLE—WRITE

  The GOTO FILE/WRITE, GOTO SOURCE/WRITE, SET WRITE, and RESERVE commands set buffers to this state. It is also the default for the file specified in the LSEDIT command line. The buffer may be modified and is written when you exit from the editing session if it has been modified.

- MODIFIABLE—READ-ONLY

  This is the default for the GOTO BUFFER/CREATE command that you use to create a "scratch" buffer. The buffer may be modified, but it is not written when you exit from the editing session.

- UNMODIFIABLE—READ-ONLY

  The GOTO FILE/READ_ONLY and GOTO SOURCE/READ_ONLY commands create buffers in this state. The buffer cannot be modified. If you issue a SET MODIFY command on this buffer and modify the contents, LSE does not write the contents when you exit from the editing session unless you also issue the SET WRITE command for the buffer.

- UNMODIFIABLE—WRITE

  You can set a buffer to this state when you have completed a set of changes to a buffer in the MODIFIABLE—WRITE state and then issued a SET NOMODIFY command for the buffer. This protects the buffer from accidental change for the remainder of the editing session. LSE writes the file when you exit from the editing session if it has been changed during the session.

Languages

Buffers may have a language associated with them. This attribute determines which language is used for the language-sensitive features (see Chapter 5 for details). The file type of the input file associated with your current buffer determines the language LSE uses. Thus, you can move between different languages in different buffers, and LSE will provide the interfaces to the appropriate compilers. The SET LANGUAGE command associates a language with a buffer.

Overview

You can use buffers for overview operations. The SET OVERVIEW command enables the COLLAPSE, FOCUS, and VIEW SOURCE commands, and the use of the EXPAND command on an overview line.

Current Indentation and Tab Settings

LSE maintains two settings to control the action of the tab key: current indentation level and tab increment. When you are at the left margin, the tab key indents to the current indentation level. If you are not at the left margin, the tab key takes you to the next tab column based on the tab increment setting. The SET INDENTATION command sets the current indentation level; the SET TAB_INCREMENT command sets the size of the tab increment.

Wrap/Nowrap

Buffers have either the *wrap* or *nowrap* attribute. If the *wrap* attribute is set, LSE automatically performs a return to a new line and indent to the left margin when the text reaches the right margin.

AUTO_ERASE/NOAUTO_ERASE

Buffers have either the AUTO_ERASE or NOAUTO_ERASE attribute. If the AUTO_ERASE attribute is set, LSE erases the placeholder the cursor is on as soon as you insert a character over the placeholder.

Margins

Buffers have left and right margin attributes. The SET LEFT_MARGIN command sets the left margin for the indicated buffer. By default, the left margin is set at column 1. The SET RIGHT_MARGIN command sets the right margin for the indicated buffer to the column number you specify. By default, the right margin is set at column 80. The right margin controls where LSE wraps words when you type text into a buffer. The right margin also controls how the FILL command reformats text.

Table 3–1 contains the commands and their default key bindings used for manipulating buffers.

Table 3–1: Buffer Manipulation Commands

Command	EDT Keypad	EVE VT100 Keypad	EVE VT200 Keypad
CHANGE DIRECTION	F11	PF3	F11
CHANGE TEXT_ENTRY_MODE	F14 CTRL/A	ENTER CTRL/A	F14 CTRL/A
GOTO BUFFER			
GOTO FILE			
GOTO SOURCE	CTRL/G	CTRL/G	CTRL/G
SET AUTO_ERASE			
SET FORWARD	KP4		KP4
SET INDENTATION			
SET INSERT			

(continued on next page)

Table 3–1 (Cont.): Buffer Manipulation Commands

Command	EDT Keypad	EVE VT100 Keypad	EVE VT200 Keypad
SET LANGUAGE			
SET MODIFY			
SET NOAUTO_ERASE			
SET NOMODIFY			
SET NOWRAP			
SET OUTPUT_FILE			
SET OVERSTRIKE			
SET OVERVIEW			
SET REVERSE	KP5		KP5
SET TAB_INCREMENT			
SET WRAP			
SHOW BUFFER			

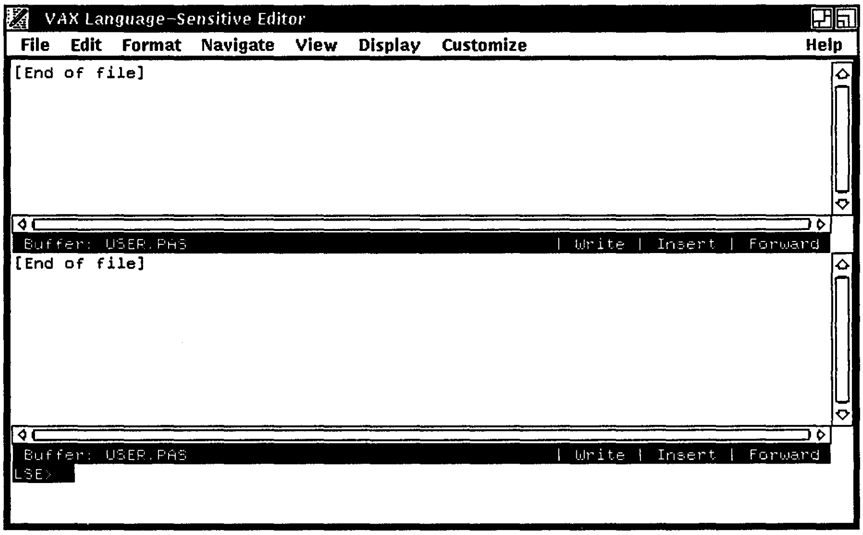
3.2 Using Windows

A **window** is a section of your work region that displays the contents of a buffer.

With LSE, you can split the work region into several windows, each mapped to a different buffer. By splitting the screen into multiple windows, you can view multiple buffers simultaneously.

Figure 3–1 shows the screen format for LSE. The screen consists of the following regions: a **work region**, a **message region**, and a **prompt** or **command region**.

Figure 3–1: Screen Format


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
[End of file]

Buffer: USER.PAS                                  | Write | Insert | Forward
[End of file]

Buffer: USER.PAS                                  | Write | Insert | Forward
LSE>
```

Each window in the work region has a **status line**. The status line is highlighted and provides information about the associated buffer. The status line tells you the name of the buffer, whether the buffer is a write or read-only buffer, whether you are in insert or overstrike mode, and whether the buffer is in a forward or reverse direction.

The message region is located at the bottom of the screen and displays broadcast messages and messages issued by LSE and SCA.

The prompt region appears above the message region with the LSE> or LSE Command> prompts, which prompt for commands or required parameters for commands. You can use LSE commands to manipulate the screen and its format.

Table 3–2 contains the commands and their default key bindings used for manipulating screens.

Table 3–2: Screen Manipulation Commands

Command	EDT VT200 Keypad	EVE VT100 Keypad	EVE VT200 Keypad
CHANGE WINDOW_ MODE	PF1-=	PF1-=	PF1-=
NEXT WINDOW	PF1-↓	PF1-KP0	PF1-E6
PREVIOUS WINDOW	PF1-↑	PF1-keypad period	PF1-E5
GOTO BUFFER			
GOTO FILE			
SET SCREEN			
SPLIT WINDOW			
TWO WINDOWS			
DELETE WINDOW			

For example, if you are editing a file called MODULE1.PAS, you could issue the following commands to move a procedure from the MODULE2.PAS file to the current file.

1. Press the CHANGE WINDOW_MODE key (PF1-=).

   This command splits the screen into two windows. Both windows contain the current buffer. The cursor is placed in the bottom window.
2. Type the command GOTO FILE MODULE2.PAS.

   This command puts the contents of the file MODULE2.PAS into the bottom window. Now that the two files are displayed on the screen, you can locate both the procedure you want to select and the location in the current file where you want the procedure placed.
3. Move to the procedure you want and press the SELECT key (keypad period). Use the arrow keys to select the entire procedure and press the CUT key (KP6) to capture the procedure.
4. Press the PREVIOUS WINDOW key (PF1-↑) to place the cursor in file MODULE1.PAS.
5. Press the PASTE key (PF1-KP6) at the location where the procedure should be placed.
6. Press the CHANGE WINDOW_MODE key (PF1-=) to return the screen to one window containing the current buffer.

3.3 Using the Search and Substitute Operations

LSE provides a SEARCH command that allows you to search for a string in a buffer. LSE also provides a SUBSTITUTE command that will replace a search string with another value.

3.3.1 Searching Through Buffers

The SEARCH command searches through the current buffer for a specific word, character, or short phrase. The SEARCH command searches in either a forward or reverse direction. The direction is determined by the current setting of the buffer.

If an occurrence of the search string is found, the cursor is positioned on the first character of the string and the search string is highlighted. If the string is not found, the cursor is not moved and an error message is displayed in the message buffer.

After you issue a SEARCH command, LSE remembers the search string. You can continue searching for the same string by using the SEARCH or SUBSTITUTE command. If you are an EDT keypad user, you can press the FNDNXT key (PF3), and LSE automatically uses the previous search string.

LSE supports VMS- and ULTRIX-style wildcard characters with the /PATTERN qualifier on the SEARCH command. For example, the asterisk (*) character may be used to match any number of characters within one line. The percent sign (%) character may be used to match any one character. These wildcard characters match the VMS format for wildcard representation. (For more details on wildcard characters, refer to the SEARCH command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*)

If you want to search through your file to locate where you reference VMS Run-Time Library virtual memory routines, you would issue the following command:

```
LSE>  SEARCH/PATTERN "LIB$*_VM"
```

If you want to search for an asterisk (*) or percent sign (%) as part of your pattern, you must include a backslash (\ ) before using the asterisk (*) or percent sign (%) in the pattern search string.

For example, if you want to search for 20%, 22%, and so on, in a buffer, you would issue the following command:

```
LSE>  SEARCH/PATTERN "2%\%"
```

3.3.2 Substituting Text Strings

The SUBSTITUTE command replaces occurrences of one text string with another text string. When you issue the SUBSTITUTE command, LSE prompts you for the search string and the value of the replacement string.

For example, you can use the SUBSTITUTE command to replace all occurrences of STR$APPEND with STR$PREFIX as follows:

```
LSE> SUBSTITUTE
_Search for:  STR$APPEND
_Replace with:  STR$PREFIX
```

LSE provides case-sensitive substitution. This enables you to specify that the case of the replacement string should be altered to match the case of the string located by a SEARCH operation.

For example, you can search for the STR$APPEND function name and replace it with STR$PREFIX by issuing the following command:

```
LSE> SUBSTITUTE/CASE_MATCHING
_Search for:  str$append
_Replace with:  str$prefix
```

Because you have included the /CASE_MATCHING qualifier on the command line, LSE alters the case of str$prefix to match the case of str$append found in your file.

LSE highlights each occurrence and prompts you for an action. *Yes* instructs LSE to replace the occurrence. *No* instructs LSE not to replace the occurrence but to search for the next occurrence. *Quit* ends the command without replacing the occurrence and stops the SUBSTITUTE operation. *Last* instructs LSE to replace the occurrence and then ends the command. *All* replaces the occurrence and all remaining occurrences without further prompting. If you use the /ALL qualifier, LSE replaces all occurrences it finds without prompting you for an action.

LSE provides the /PATTERN qualifier on the SUBSTITUTE command. The /PATTERN qualifier uses the same pattern expressions as does the /PATTERN qualifier on the SEARCH command. The asterisk (*) and percent sign (%) have the same meaning when used on the SUBSTITUTE command as they do on the SEARCH command.

An example of using the /PATTERN qualifier on the SUBSTITUTE command follows:

```
LSE> SUBSTITUTE/PATTERN "NAME_%_LENGTH" "NAME_B_LENGTH"
```

3.4 Working with Files

This section describes the basic file manipulation commands that you use to bring files into an editing buffer and then write these buffers to files. Section 3.4.1 describes locating, displaying, and editing source files; Section 3.4.2 describes locating files in multiple directories; Section 3.4.3 describes how to set default directories; and Section 3.4.4 describes how LSE provides access to files stored in VAX DEC/Code Management System (CMS) libraries.

3.4.1 Locating, Displaying, and Editing Source Files

LSE commands bring files into buffers and write contents of buffers to files.

Getting Files

LSE provides several commands for locating, displaying, and editing source files within your editing session. The commands are as follows:

- GOTO DECLARATION
- GOTO FILE
- GOTO SOURCE
- INCLUDE
- READ

The GOTO DECLARATION command displays the source file corresponding to the specified or indicated symbol declaration. The GOTO FILE command locates a file and reads it into a buffer.

The GOTO SOURCE command displays the source file corresponding to the current diagnostic or query item.

The INCLUDE command inserts a copy of a file at the current cursor position. The cursor position does not change.

The READ command inserts a copy of a file at the current cursor position. The cursor moves to the end of the inserted text.

Writing Files

LSE provides several commands for writing the contents of buffers into files. The commands are as follows:

- COMPILE
- EXIT
- WRITE

The COMPILE command first writes out the current buffer, if it has been modified, and writes out any other modified buffers associated with the same language. It then compiles the file associated with the current buffer.

The EXIT command ends an editing session and writes out buffers that have been modified, provided they are not marked read-only buffers. Buffers that are read-only are not written out by a COMPILE or EXIT command. (If you do not want to save your modifications, you can use the QUIT command to end your editing session.)

The WRITE command writes out your current buffer or a specified buffer.

3.4.2 Locating Files in Multiple Directories

With LSE, you can specify a list of directories for LSE to use when locating files.

The SET SOURCE_DIRECTORY command specifies a searchlist of directories to be used to find source files. LSE searches the directories in the order specified on the SET SOURCE_DIRECTORY command line until the source file is found.

For example, type the following command:

```
LSE> SET SOURCE_DIRECTORY [A],[B],[C],[D]
```

Then, type a file manipulation command, such as GOTO FILE. LSE searches through [A], then [B], then [C], then [D], until the source file is found.

You can include CMS$LIB on the SET SOURCE_DIRECTORY directory list so that LSE will fetch files directly from your CMS library into buffers. Note that the CMS SET LIBRARY command must be issued first.

The following commands use the source list to locate files:

- GOTO FILE
- GOTO SOURCE

- INCLUDE
- READ

The GOTO FILE command uses the searchlist to resolve the file specified on the GOTO FILE command line.

The GOTO SOURCE command displays a source file that is specified in the diagnostics file or SCA data that corresponds to your current review or query operation. If LSE cannot find the file within an existing buffer, LSE attempts to find the exact file specified by the current diagnostic or query operation. If that file cannot be found, LSE uses the SET SOURCE_ DIRECTORY list to locate the file.

The INCLUDE and READ commands also use the SET SOURCE_ DIRECTORY list to resolve file specifications.

3.4.3 Setting Directory Defaults

With the SET DIRECTORY command, you can set the default read/write status of files in a specified directory. Initially, all directories are set to write.

The SET DIRECTORY/READ_ONLY command allows you to specify directories that contain files that you do not want to change. If you specify a directory list as /READ_ONLY, LSE brings the files contained in those directories into unmodifiable/read-only buffers by default.

Using Default Settings

If you are working on a software project, you might have the following directories set up:

- [ ], your current directory.
- [MY_DIRECTORY], which contains the files that you are working on, for example, MODULE1.PAS and MODULE3.PAS.
- [PROJECT_DIRECTORY], which contains the files that comprise your project, for example, MODULE1.PAS, MODULE2.PAS, MODULE3.PAS, MODULE4.PAS, and MODULE5.PAS. These files are shared by your project team and should not be modified.

For example, type the following commands to set default settings:

```
LSE> SET SOURCE_DIRECTORY [], [MY_DIRECTORY],[PROJECT_DIRECTORY]
LSE> SET DIRECTORY/READ_ONLY [PROJECT_DIRECTORY]
```

Then, type a GOTO FILE or GOTO SOURCE command. LSE searches through your current directory, MY_DIRECTORY, and the PROJECT_DIRECTORY to locate files. Those files located in your current directory and MY_DIRECTORY can be modified. However, those files located in the PROJECT_DIRECTORY cannot be modified by default because of the /READ_ONLY qualifier on the SET DIRECTORY command. Thus, you can modify files MODULE1.PAS and MODULE3.PAS, and refer to the other project files as necessary without the possibility of modifying the wrong files.

For example:

```
LSE>  GOTO FILE MODULE2.PAS
```

LSE gets the file from the PROJECT_DIRECTORY and puts the file into an unmodifiable/read-only buffer.

```
LSE>  GOTO FILE MODULE1.PAS
```

LSE gets the file from MY_DIRECTORY and puts the file into a modifiable /writeable buffer.

Overriding Default Settings

The GOTO FILE command has the following qualifiers:

- /WRITE
- /READ_ONLY
- /MODIFY

You can use these qualifiers to override any settings established by the SET DIRECTORY command. (See the GOTO FILE command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details.) You can also use the /WRITE and /READ_ONLY qualifiers on the GOTO SOURCE command to override any settings established by the SET DIRECTORY command.

If you want to modify MODULE2.PAS in the [PROJECT_DIRECTORY], you would type the following command:

```
LSE>  GOTO FILE/WRITE MODULE2.PAS
```

LSE gets MODULE2.PAS out of the PROJECT_DIRECTORY and puts the file into a modifiable/writeable buffer.

If you want to look at MODULE1.PAS in MY_DIRECTORY, but you do not want to modify the file, you would type the following command:

```
LSE>  GOTO FILE/READ_ONLY MODULE1.PAS
```

This command overrides the current settings established by the previous SET DIRECTORY command, and it brings MODULE1.PAS into a read-only /unmodifiable buffer.

3.4.4 Getting Files Through VAX DEC/Code Management System

With LSE, you can access files stored in VAX DEC/Code Management System (CMS) directly. You can also issue all CMS commands from within LSE. The LSE commands related to CMS are as follows:

- CMS [cms-command]
- SET CMS
- GOTO FILE
- GOTO SOURCE
- INCLUDE
- READ
- RESERVE
- UNRESERVE
- REPLACE

The CMS [cms-command] Command

With the CMS [cms-command] command, you can execute any CMS command from within LSE. This command operates on your current CMS library.

The SET CMS Command

The SET CMS command sets the defaults for CMS qualifier values for CMS operations performed by the following commands: GOTO FILE, GOTO SOURCE, INCLUDE, READ, REPLACE, RESERVE, and UNRESERVE.

The GOTO FILE, GOTO SOURCE, INCLUDE, and READ Commands

When you issue the GOTO FILE, GOTO SOURCE, INCLUDE, or READ command, if the directory for a file you are looking for is the same as your current CMS library, then LSE fetches the element from CMS and puts it into a read-only/unmodifiable buffer. LSE prompts you for confirmation when performing a fetch operation. This fetch operation does not create a file.

For example, type the following command, and respond with *Y*:

```
LSE>  GOTO FILE CMS$LIB:FILE.TYP
FILE.TYP found in CMS library DISK:[PROJECT_CMS_LIBRARY]
Do you want to fetch it [Y or N]?
```

LSE fetches the element FILE.TYP and reads it into a buffer of the same name. Note that LSE does not create a file in your current directory when doing a fetch operation.

The RESERVE Command

With the RESERVE command, you can reserve an element in your current CMS library. This element is put into an editing buffer. For example:

```
LSE>  RESERVE element-name
```

LSE reserves the element *element-name* in your current CMS library and reads it into a buffer. If you omit the *element-name* parameter, the RESERVE command reserves the element of the same name and type as the input file associated with your current buffer.

The UNRESERVE Command

With the UNRESERVE command, you can unreserve the CMS element of the same name and type as the file associated with your current buffer in your current CMS library. For example:

```
LSE>  UNRESERVE
```

LSE unreserves the element in your current buffer that you reserved in the last example.

The REPLACE Command

The REPLACE command replaces the CMS element of the same name and type as the file associated with your current buffer in your current CMS library. For example:

```
LSE>  REPLACE
```

LSE replaces the element in your current buffer into your current CMS library.

3.5 Recovering from a Failed Editing Session

LSE provides mechanisms to recover edits that exist as changes in LSE buffers when a system or editor failure occurs. LSE can journal your edits in two ways: keystroke journaling and buffer-change journaling. Keystroke journaling records the keys that you press over the course of an editing session. Buffer-change journaling records the changes made to a buffer over the course of an editing session.

Keystroke journal files have an extension of .TJL and buffer-change journal files have an extension of .TPU$JOURNAL. Both types of journaling periodically write information to their respective journal files. When an editing session is terminated abnormally, a journal file may not contain a record of the last few operations that were performed. However, when you recover, using a journal file, your cursor remains at the position that corresponds to the location where the last journaled edit was made.

Using a Keystroke Journal File

Keystroke journal files contain the exact sequence of keys that you pressed over the course of the editing session. When you recover, using a keystroke journal file, you must be sure to restore your environment to the state that it was in when the journaled editing session was started. The following list describes the types of things that must be restored to your environment before you can successfully complete a recovery using a keystroke journal file.

- All files created during the editing session must be either deleted or placed in a directory that will not be referenced during the recovery operation. This will prevent the wrong version of the file from being accessed during the keystroke journal recovery operation.

  You can determine what files were created during an editing session by examining the creation date of the .TJL file. For example, to determine the creation date of a keystroke journal file, type the following command:

  ```
  $   DIRECTORY/DATE=CREATE MEMO.TJL
  ```

  This produces output similar to the following:

  ```
  Directory DUA0:[SMITH]

  MEMO.TJL;1          15-JUN-1989 08:42:20.69

  Total of 1 file.
  ```

To move files from specific directories that were created since that date to a directory that is not referenced during the recovery operation, type the following commands:

```
$  CREATE/DIRECTORY DUA0:[SMITH.TEMP]
$  COPY/SINCE=15-JUN-1989:08:42:20 -
_$  DUA0:[SMITH...],DUA0:[PROJECT...] DUA0:[SMITH.TEMP]
$  DELETE/SINCE=15-JUN-1989:08:42:20 -
_$  DUA0:[SMITH...],DUA0:[PROJECT...]
```

- You must set the terminal characteristics of the terminal to match the terminal characteristics that were in effect at the time that the journaled editing session was started.
- You must invoke LSE with the same command line that you used to start the journaled editing session. Additionally, you must specify the /RECOVER qualifier on the command line.

You may have to take other actions, depending on the events that may have occurred over the course of your editing session.

If you reserve or replace elements in a CMS library during the editing session, you may not be able to perform a recovery using a keystroke journal file because changes in the CMS library cannot be undone. In such cases, you should use a buffer-change journal file.

Using a Buffer-Change Journal File

A buffer-change journal file contains a record of the changes that you made to a buffer over the course of an editing session. There is one buffer-change journal file for each editing buffer.

To recover changes, with a buffer-change journal file, use the following steps:

1. Invoke LSE.
2. Type the RECOVER BUFFER command and specify the name of the file that you want to recover.
3. Examine the information displayed by LSE to verify that the journal file corresponds to the source file you want.
4. Type Y at the prompt if you want to recover the buffer.

After you instruct LSE to start the recovery, using a buffer-change journal file, LSE reads the source file that corresponds to the buffer-change journal file and begins to apply the journaled changes to the file.

When you recover changes, using a buffer-change journal file, you do not have to worry about the files that were created during the editing session or the command line that you used to invoke LSE. Using a buffer-change journal file is much quicker than using a keystroke journal file because LSE recovers only files that it is directed to recover.

3.6 Collapsing and Expanding Program Source

With LSE, you can view programs at various levels of detail. The concept is sometimes called outlining, holophrasting, or code elision.

You can generate overviews of your programs in two ways:

- Interactively in LSE
- In reports

The interactive interface allows you to generate overviews of programs by collapsing lines of code. The report tool lets you present the overviews you select in a structured manner.

With LSE, you can hide the details of your source code. LSE generates an **overview line** that corresponds to one or more real lines in a program. LSE displays overview lines as pseudocode placeholders.

For example, the following fragment contains real lines of code:

```
-- Interchange the numbers.
TEMP := NUMBERS(J);
NUMBERS(J) := NUMBERS(J+1);
NUMBERS(J+1) := TEMP;
```

It can be represented by the following overview line:

```
«-- Interchange the numbers...»
```

3.6.1 Sample Session

Using LSE's code viewing features, you can see more or less detail at a particular point in a program. You can specify the level of detail either uniformly across the buffer or with a focus on a particular line.

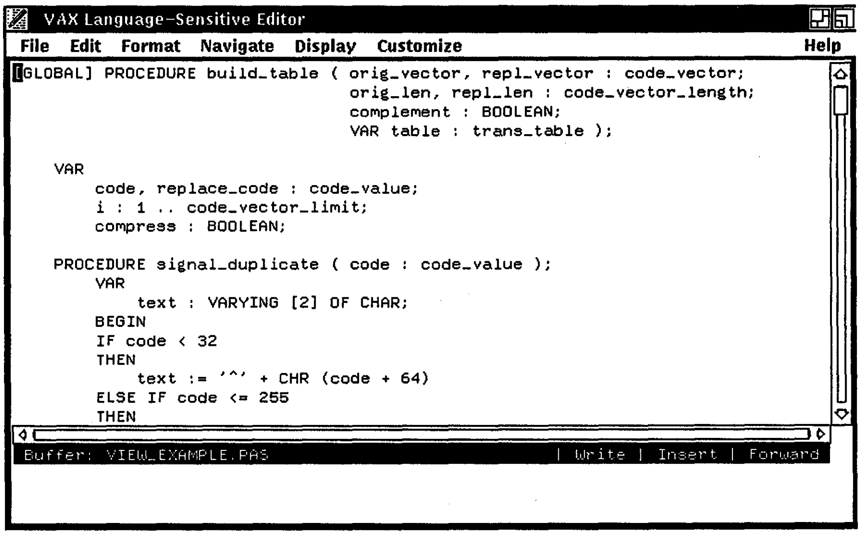
Table 3–3 contains the commands and their default key bindings used for generating overviews.

Table 3–3: Code Viewing Commands

Command	EDT VT200 Keypad	EVE VT100 Keypad	EVE VT200 Keypad
VIEW SOURCE	PF1->	PF1->	PF1->
EXPAND/DEPTH=1	CTRL/E	CTRL//	CTRL//
EXPAND/DEPTH=ALL	PF1-<	PF1-<	PF1-<
FOCUS	PF1-period	PF1-period	PF1-period
COLLAPSE	CTRL/\	CTRL/\	CTRL/\

The following sample session demonstrates how to expand and compress source code. This session uses a sample file from the directory SCA$EXAMPLE, which is supplied with the SCA kit. Figure 3–2 shows a buffer containing a Pascal code fragment from SCA$EXAMPLE:BUILDTABLE.PAS.

Figure 3–2: Buffer Containing Source


VAX Language-Sensitive Editor
File Edit Format Navigate Display Customize Help

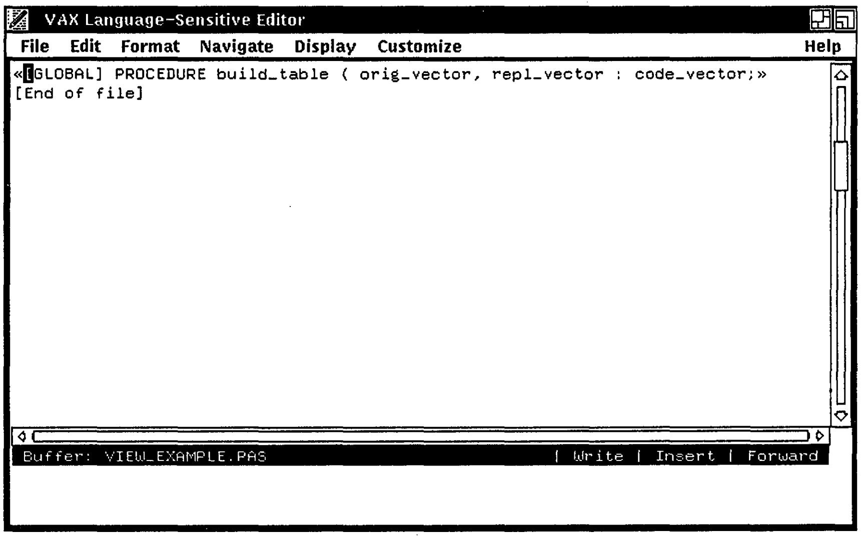
```
[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;
                                 orig_len, repl_len : code_vector_length;
                                 complement : BOOLEAN;
                                 VAR table : trans_table );

    VAR
        code, replace_code : code_value;
        i : 1 .. code_vector_limit;
        compress : BOOLEAN;

    PROCEDURE signal_duplicate ( code : code_value );
        VAR
            text : VARYING [2] OF CHAR;
        BEGIN
        IF code < 32
        THEN
            text := '^' + CHR (code + 64)
        ELSE IF code <= 255
        THEN
```

Buffer: VIEW_EXAMPLE.PAS | Write | Insert | Forward

1. To display a top-level overview of the current file, press the PF1-> key (the VIEW SOURCE command).

   Figure 3–3 shows the resulting screen.

Figure 3–3: Overview of Source


VAX Language-Sensitive Editor
File Edit Format Navigate Display Customize Help

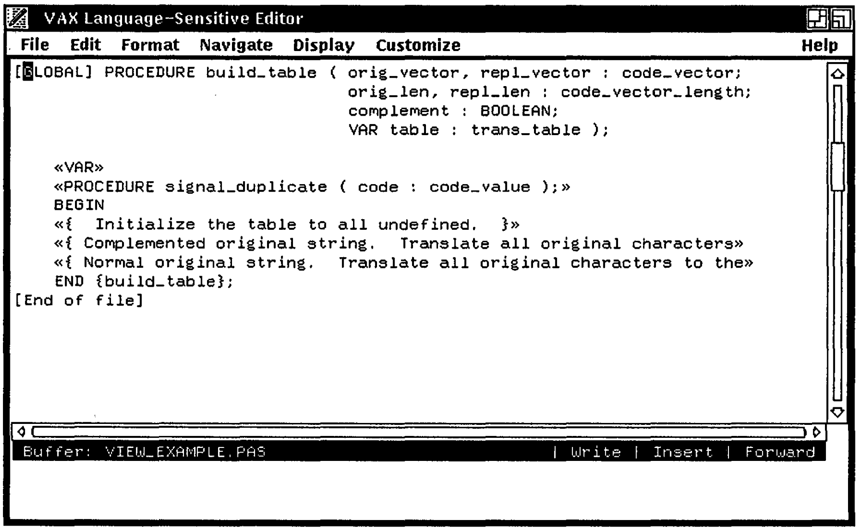
```
«[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;»
[End of file]
```

```
Buffer: VIEW_EXAMPLE.PAS                         | Write | Insert | Forward
```

2. To expand the procedure to the next level of detail, press CTRL/E or CTRL// (the EXPAND key).

   Figure 3–4 shows the resulting screen.

Figure 3–4: Expanding to Lower Detail


VAX Language-Sensitive Editor
File Edit Format Navigate Display Customize Help

```
[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;
                                 orig_len, repl_len : code_vector_length;
                                 complement : BOOLEAN;
                                 VAR table : trans_table );

    «VAR»
    «PROCEDURE signal_duplicate ( code : code_value );»
    BEGIN
    «{  Initialize the table to all undefined.  }»
    «{ Complemented original string.  Translate all original characters»
    «{ Normal original string.  Translate all original characters to the»
    END {build_table};
[End of file]
```

```
Buffer: VIEW_EXAMPLE.PAS                          | Write | Insert | Forward
```

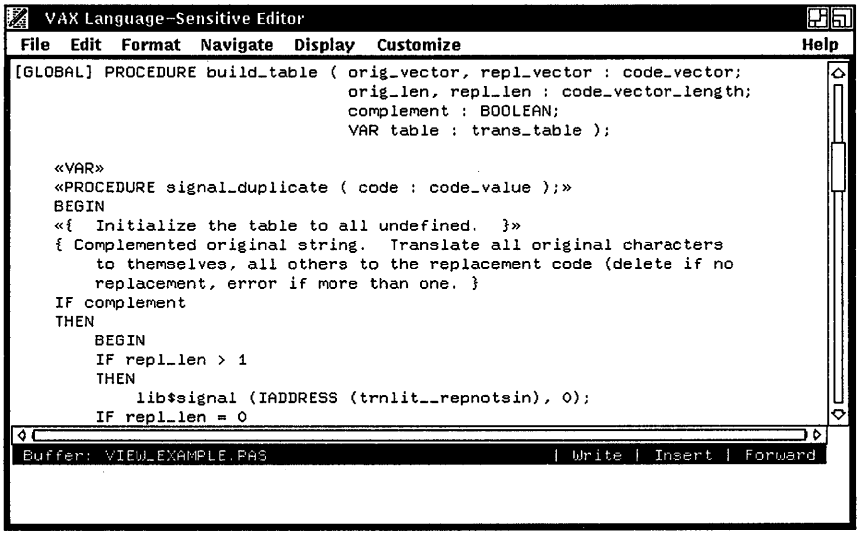
3. To get the lowest level of detail at the current placeholder, do the following:
   a. Position the cursor on the overview line containing the following text:

      «{ Complemented original string. Translate all original characters»

   b. Press the PF1-< key (the EXPAND/DEPTH=ALL command).

   Figure 3–5 shows the resulting screen.

Figure 3–5: Expanding to Lowest Detail


VAX Language-Sensitive Editor
File Edit Format Navigate Display Customize Help

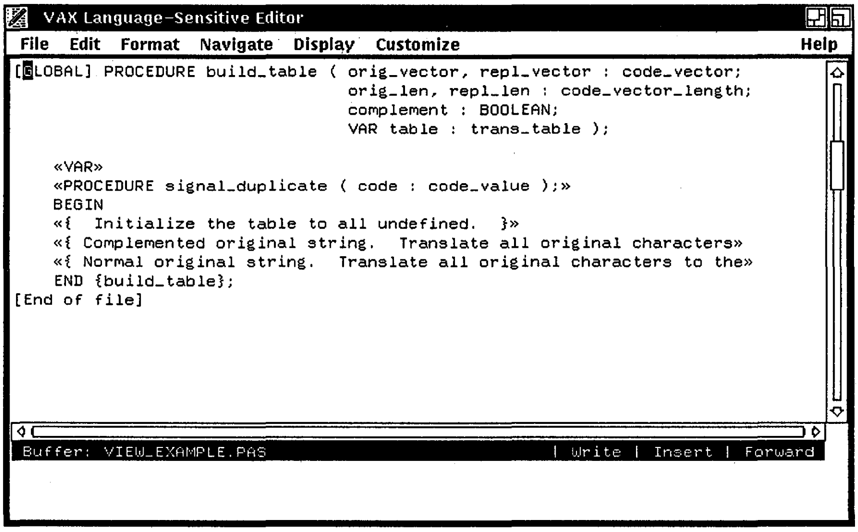
```
[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;
                                 orig_len, repl_len : code_vector_length;
                                 complement : BOOLEAN;
                                 VAR table : trans_table );

    «VAR»
    «PROCEDURE signal_duplicate ( code : code_value );»
    BEGIN
    «{  Initialize the table to all undefined.  }»
    { Complemented original string.  Translate all original characters
        to themselves, all others to the replacement code (delete if no
        replacement, error if more than one. }
    IF complement
    THEN
        BEGIN
        IF repl_len > 1
        THEN
            lib$signal (IADDRESS (trnlit__repnotsin), 0);
        IF repl_len = 0
```

```
Buffer: VIEW_EXAMPLE.PAS                          | Write | Insert | Forward
```

4. To hide the lowest level of detail near the cursor position, press the CTRL/\ key (the COLLAPSE command).

   Figure 3–6 shows the resulting screen.

Figure 3–6: Collapsing Code


VAX Language-Sensitive Editor
File Edit Format Navigate Display Customize Help

```
[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;
                                 orig_len, repl_len : code_vector_length;
                                 complement : BOOLEAN;
                                 VAR table : trans_table );

    «VAR»
    «PROCEDURE signal_duplicate ( code : code_value );»
    BEGIN
    «{  Initialize the table to all undefined.  }»
    «{ Complemented original string.  Translate all original characters»
    «{ Normal original string.  Translate all original characters to the»
    END {build_table};
[End of file]
```

```
Buffer: VIEW_EXAMPLE.PAS                          | Write | Insert | Forward
```

5. To display the original source text, type the VIEW SOURCE /DEPTH=ALL command.

3.6.2 Editing Overviews

The following commands, which remove and insert complete lines, have special behavior on overview lines.

- CUT
- PASTE
- ERASE LINE
- UNERASE LINE

If you use one of these commands to remove or insert an overview line, the corresponding source lines are implicitly removed or inserted. For example, suppose you have a buffer containing the following three functions:

```
FUNCTION max (a,b)
    IF a > b
    THEN
        RETURN a
    ELSE
        RETURN b
END max;

FUNCTION add (a,b)
    RETURN a + b
END add;

FUNCTION subtract (a,b)
    RETURN a + b
END subtract;
```

To alphabetize them, use PF1- > to represent each function with a single line, as follows:

```
«FUNCTION max (a,b)...»
«FUNCTION add (a,b)...»
«FUNCTION subtract(a,b)...»
```

If you use the ERASE LINE command on the overview line for the "add" routine, press the up arrow key, then use the UNERASE LINE command, the resulting display is as follows:

```
«FUNCTION add (a,b)...»
«FUNCTION max (a,b)...»
«FUNCTION subtract(a,b)...»
```

When you expand the overview lines, you see that the routines have been reordered, as follows:

```
FUNCTION add (a,b)
    RETURN a + b
END add;

FUNCTION max (a,b)
    IF a > b
    THEN
        RETURN a
    ELSE
        RETURN b
END max;

FUNCTION subtract (a,b)
    RETURN a + b
END subtract;
```

If you use the CHANGE INDENTATION command to change the indentation of an overview line, the indentation of the corresponding source lines is implicitly altered as well.

If you want to change the case of characters or fill text, you must first expand all the overview lines in the selected range.

Commands that move the cursor to a specified point in the buffer will make hidden source lines visible. The following commands make the target line and its context visible.

- GOTO SOURCE
- SEARCH
- GOTO MARK

Editing operations that concatenate an overview line with another line, such as deleting a line break, are not permitted.

The text on overview lines is not modifiable. If you want to alter the text of an overview, use the following steps:

1. Press the EXPAND key on the overview.
2. Modify the corresponding source line.
3. Press the COLLAPSE key.

When you use the EXIT, WRITE, or COMPILE commands, LSE writes all the source lines and none of the overview lines.

To write out an overview, use the WRITE/VISIBLE command.

If you define or delete any adjustment definition for a language, define or delete the language, or modify the overview options for the language, then all source lines in all buffers associated with the language are automatically made visible.

Chapter 4

Using VAX LSE with DECwindows

This chapter provides an overview of the DECwindows LSE environment. It describes the DECwindows interface for LSE and demonstrates how to use DECwindows LSE to open files, perform basic editing tasks, use multiple windows and buffers, and review and query source code.

You should understand the basic DECwindows concepts, which are described in the *VMS DECwindows User's Guide* and cover the following subjects:

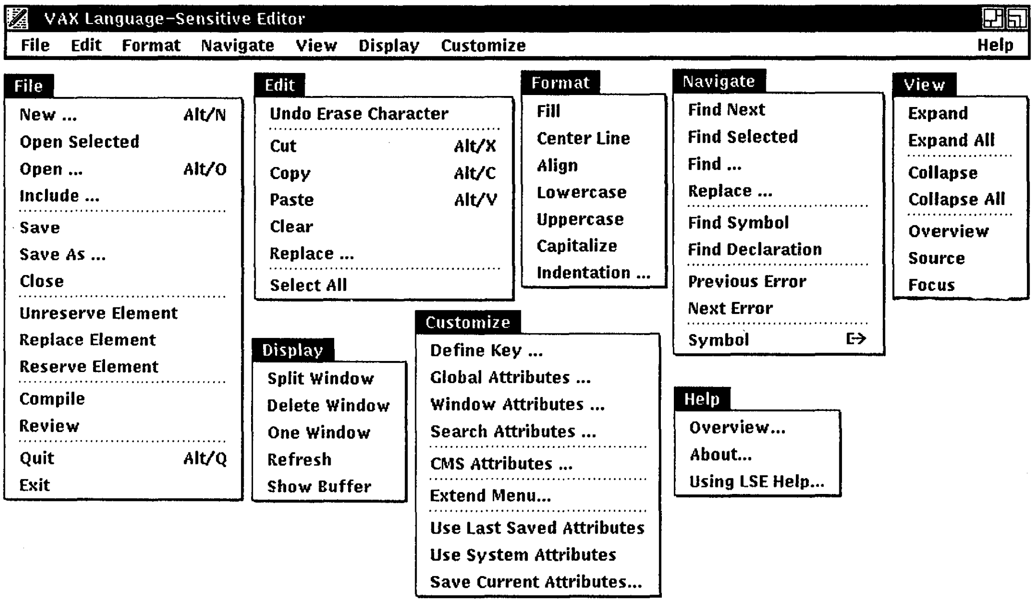
- Using VMS DECwindows user interface
- Beginning a session
- Interacting with the session manager
- Using and managing windows
- Using the mouse to select objects
- Running a DECwindows application

4.1 Overview

This section describes the DECwindows interface to LSE. Figure 4–1 shows the initial LSE menu bar and the menus you can select from the menu bar. To invoke LSE, type the following command in a DECterm window:

```
$   LSEDIT/DISPLAY=DECWINDOWS filename
```

Figure 4–1: LSE DECwindows Title Bar and Menus



4.1.1 The DECwindows LSE Application Window

The DECwindows LSE application window contains the following:

- Menu bar
- Work region
- Command region
- Message region

The **menu bar** contains pull-down menus that allow you to perform the following tasks:

- Manipulate files—Access and replace files and CMS elements, compile and review source code, and exit from LSE.
- Edit Text—Add and delete text in a buffer, use clipboard functions, and restore text.

- Format text—Rearrange text in a buffer, such as fill, center, indent, and case-change.
- Navigate—Move the cursor to a new position in a buffer, including text search, SCA queries, and diagnostic review.
- View Source—Collapse and expand program source, and expand place-holders.
- Manipulate display—Split windows, delete windows, enlarge windows, and shrink windows, as well as change which buffers should be displayed.
- Customize LSE— Select a default keypad, supply new key definitions, change the window width and height, change fonts, and change default settings for use with CMS.
- Access the online HELP Facility—Get help on menu items, keypad layout, control keys, language constructs, tokens and placeholders, and LSE commands.

You use the **work region** to display the contents of editing buffers, SCA query displays, diagnostic review displays, and so on. The work region consists of one or more windows. Each window has a status line at the bottom of the window. You can use the mouse to toggle between options on the status line. For example, you can click on Write to toggle between Write and Read-only, or click on Insert to toggle between Insert and Overstrike. Each window has vertical and horizontal scroll bars. You can use the mouse to establish a position within the work region and select items for commands to work on.

You use the **command region** to issue LSE commands and to respond to prompts. To issue a command, press the Do key to position the cursor on the command prompt, or click the mouse on the prompt. If you press the Do key to move the cursor to the command prompt, the cursor returns to its previous position in the work area after the command is entered. You can also perform command-line editing by using the usual editing keys.

The **message region** is below the work region. It is used to display messages. For example, warnings resulting from executing the contents of a buffer are displayed in the message region.

4.1.2 Getting Help

The DECwindows LSE interface provides context-sensitive online help for all menu and submenu items, commands, and keypad functions.

To get online help on any menu or submenu, use the following steps:

1. Position the pointer to the desired menu item, and hold MB1.
2. Press and hold the Help key.
3. Release MB1; then release the Help key.

To get online help on any dialog box or other object you see on the screen, use the following steps:

1. Position the pointer in the desired dialog box.
2. Press and hold the Help key and press MB1.
3. Release MB1; then release the Help key.

In addition, you can get online help from the Help menu. The Help menu contains the following items:

- Overview—Provides information on getting help and provides additional topics, such as New Users, DECwindows Interface, and the Command Line Interface.
- About—Provides a brief introduction to LSE and includes copyright and version information.
- Using LSE help—Provides general information about context-sensitive help.

You can also get help by typing the HELP command and specifying a topic. For example, typing HELP SEARCH at the command prompt gets you help on the SEARCH command.

Most help topics have a list of related commands or other topics. If you are viewing help on one topic and want to get help on another topic, type the name of that topic and press the Return key.

4.2 LSE DECwindows Sample Session

The following sample session will help you become familiar with some of the basic features of LSE with the DECwindows interface. This session uses the same sample language, called EXAMPLE, that is used in Section 2.3.

To invoke LSE and start the sample session, type the following:

```
$   LSEDIT/DISPLAY=DECWINDOWS
```

4.2.1 Opening a File

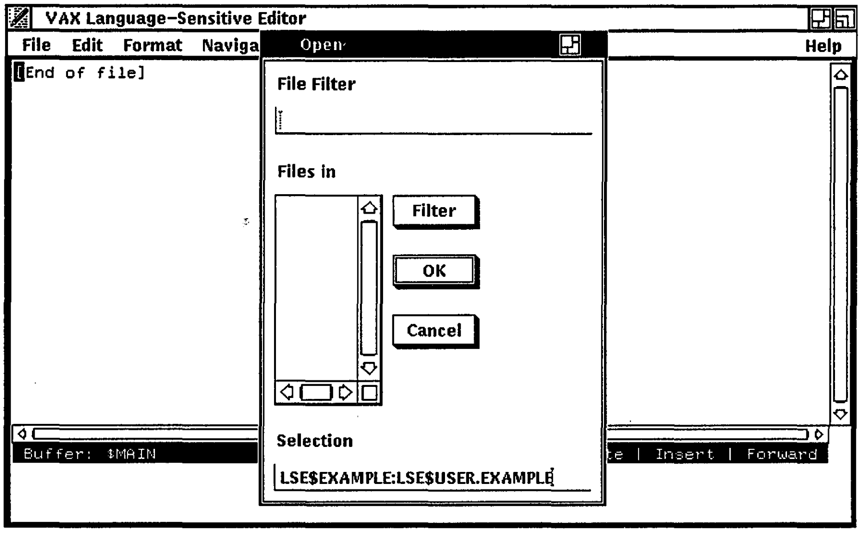
When you begin an editing session, you type the name of the file you want to edit on the LSEDIT command line. If you do not specify a file, LSE creates an empty buffer called $MAIN, as in this example.

To open an existing file, in this case, LSE$USER.EXAMPLE, perform the following steps:

1. Pull down the File menu.
2. Choose the Open... menu item.

   LSE displays the Open dialog box, and asks you for the name of the file you want to edit. Figure 4–2 shows the Open dialog box.

Figure 4–2: Open Dialog Box



```
[End of file]
Buffer: $MAIN
e | Insert | Forward
```

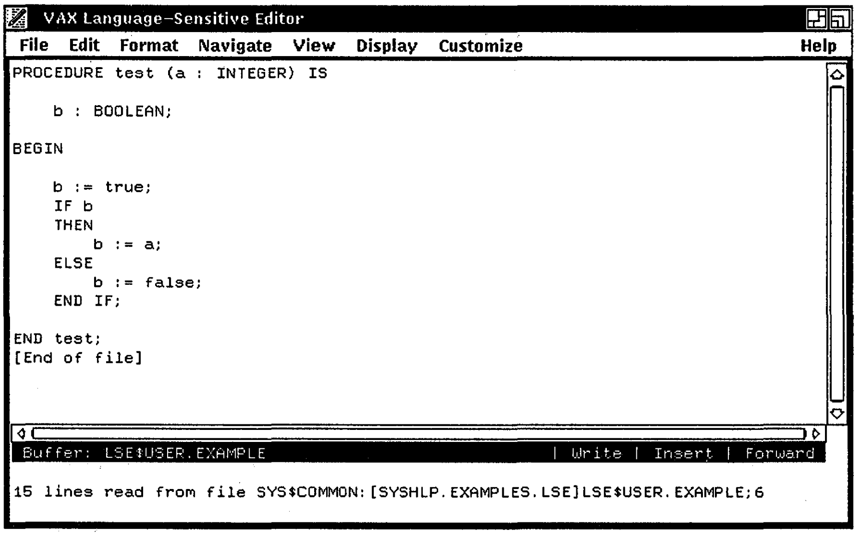
LSE$EXAMPLE:LSE$USER.EXAMPLE

3. Type LSE$EXAMPLE:LSE$USER.EXAMPLE in the Selection field.

4. Click on OK.

   LSE reads the file into the buffer and displays it in the main window. Figure 4–3 shows the screen with LSE$USER.EXAMPLE as the current buffer. Section 4.2.7 describes how to use the filter mechanism.

Figure 4–3: User Buffer


VAX Language–Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
PROCEDURE test (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
        b := a;
    ELSE
        b := false;
    END IF;

END test;
[End of file]
```

```
Buffer: LSE$USER.EXAMPLE                          | Write | Insert | Forward

15 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.LSE]LSE$USER.EXAMPLE;6
```

4.2.2 Positioning the Cursor and Selecting Text

You can position the cursor anywhere in the buffer by moving the pointer to the desired spot and pressing MB1. Each time you do this, you set the current cursor position to that location.

You can select text in a buffer to perform an editing operation, such as changing the case of letters, reformatting a block of text, or copying a block of text to be inserted elsewhere.

To select text, use the Select All menu item (located in the Edit menu), or use the mouse, as follows:

1. Press MB1 in the work area and drag the mouse to select a block of text.

2. Release MB1 when the text you want is highlighted.

An alternative method to dragging MB1 to select text is to press MB1 as follows:

2 clicks selects the current word
3 clicks selects the current line
4 clicks selects the current paragraph
5 clicks selects all of the current buffer

To cancel a selection, choose another selection or press MB1.

4.2.3 Searching for Text

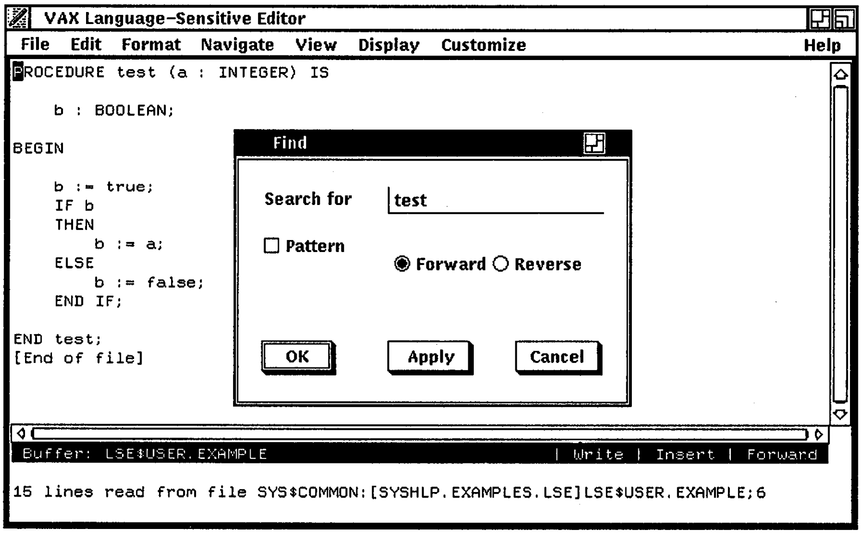
With DECwindows LSE, you can search for a text string. There are two ways to instruct LSE to search for a particular string:

- Choose the FIND... menu item from the Navigate menu
- Select text at the current cursor position

To search for the string *test* by using the FIND... menu item, perform the following steps:

1. Pull down the Navigate menu.
2. Choose the Find... menu item.
   LSE displays the Find dialog box and asks for a string of text. Figure 4–4 shows the Find dialog box.

Figure 4–4: Find Dialog Box


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help
Find
Search for
test
☐ Pattern
● Forward ○ Reverse
OK
Apply
Cancel

```
PROCEDURE test (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
        b := a;
    ELSE
        b := false;
    END IF;

END test;
[End of file]
 Buffer: LSE$USER.EXAMPLE                     | Write | Insert | Forward
15 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.LSE]LSE$USER.EXAMPLE;6
```

3. Type the string *test*.
4. Click on Apply.

   LSE moves the cursor to the first occurrence of *test*.
5. Click on OK.

   LSE moves the cursor to the next occurrence of *test* and removes the Find dialog box.

Alternatively, you can select text at the current cursor position to be the search string by performing the following steps:

1. Move the pointer to the word you want.
2. Double click MB1 on the word to select it.
3. Click MB2 in the work area to get the pop-up menu.
4. Choose the Find Selected menu item.

LSE moves the cursor to the next occurrence of the word.

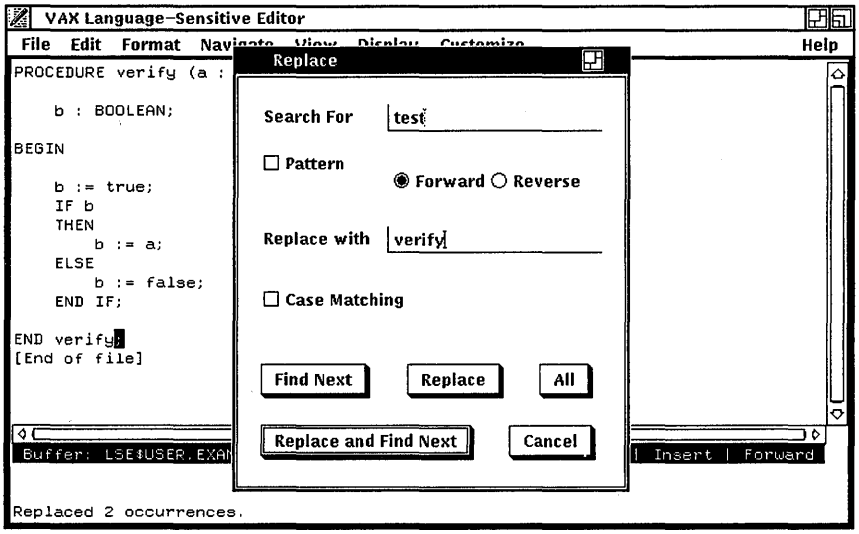
4.2.4 Replacing Text

With DECwindows LSE, you can search for a text string and replace it with another text string. To replace text, perform the following steps:

1. Pull down the Navigate menu.
2. Choose the Replace menu item.

   LSE displays the Replace dialog box, as shown in Figure 4–5.

Figure 4–5: Replace Dialog Box



```
PROCEDURE verify (a :

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
        b := a;
    ELSE
        b := false;
    END IF;

END verify;
[End of file]
```

Search For: test

[ ] Pattern

[✓ Forward] Reverse

Replace with: verify

[ ] Case Matching

```
Replaced 2 occurrences.
```

3. Type the string *test* in the Search For field. In this case, *test* will already be in the Search For field.
4. Type the string *verify* in the Replace with field.
5. Click on All.

   LSE replaces all occurrences of the string *test*.
6. Click on Cancel to remove the Replace dialog box.

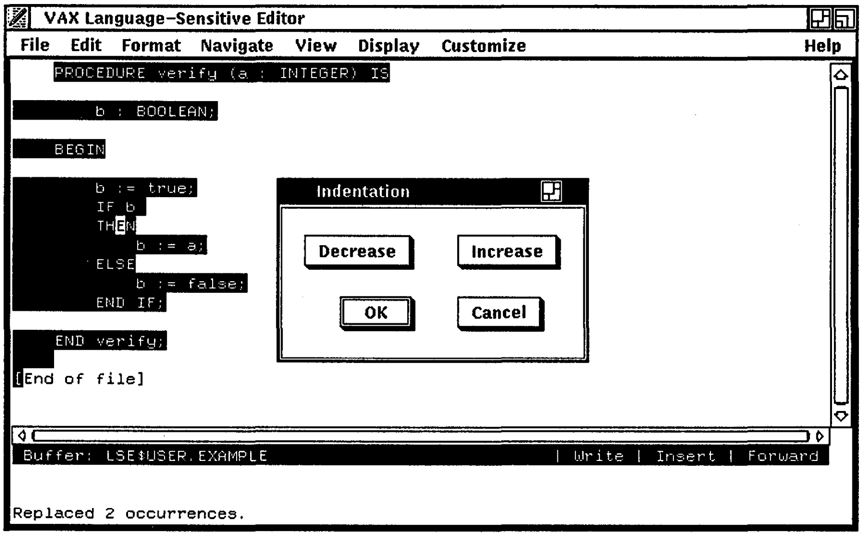
4.2.5 Formatting Text

You can format text in many ways, including filling, aligning, centering, or indenting text. This example shows how to indent text. Perform the following steps:

1. Click MB1 five times to select all of the text.
2. Pull down the Format menu.
3. Choose the Indentation... menu item.

   LSE displays the Indentation dialog box, as shown in Figure 4–6.
4. Click on Increase.

Figure 4–6: Indentation Dialog Box



```
    PROCEDURE verify (a : INTEGER) IS

        b : BOOLEAN;

    BEGIN

        b := true;
        IF b
        THEN
            b := a;
        ELSE
            b := false;
        END IF;

    END verify;

[End of file]
Buffer: LSE$USER.EXAMPLE                     | Write | Insert | Forward

Replaced 2 occurrences.
```

   LSE moves all of the lines of text over several spaces.

5. Click on Decrease.

   LSE moves the text back to its original location.
6. Click on OK to remove the Indentation dialog box and to cancel the selection.

4.2.6 Using Multiple Windows

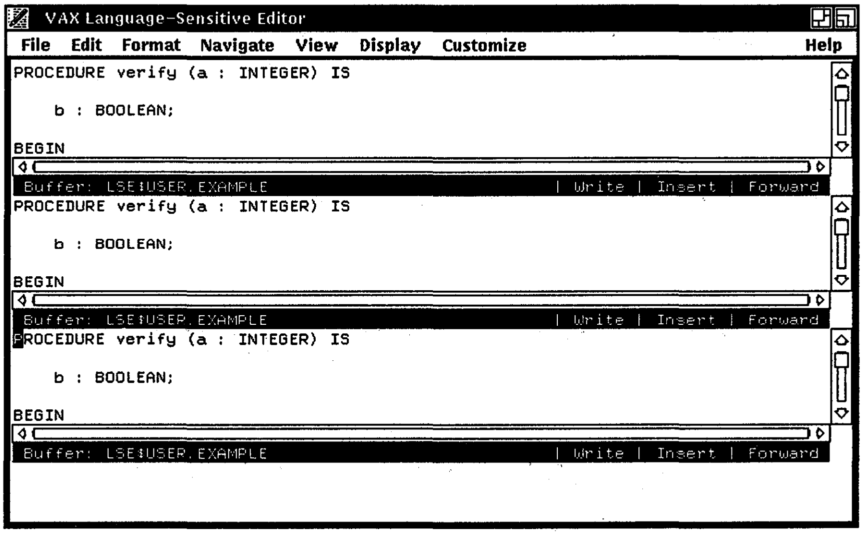
During an editing session, you can create or edit more than one file, and you can view two or more files at the same time. This is useful if you want to cut and paste between files, or refer to a section of a long file while editing another section.

This example shows how to change the number of windows on the screen and then open several files. To change the number of windows, perform the following steps:

1. Pull down the Display menu.
2. Choose the Split Window menu item.

The screen splits into two separate windows. Repeat these steps to display three windows, as shown in Figure 4–7.

Figure 4–7: Using Multiple Windows



```
PROCEDURE verify (a : INTEGER) IS

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPLE                                  | Write | Insert | Forward
PROCEDURE verify (a : INTEGER) IS

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPLE                                  | Write | Insert | Forward
PROCEDURE verify (a : INTEGER) IS

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPLE                                  | Write | Insert | Forward
```

4.2.7 Using a Filter to Open Files

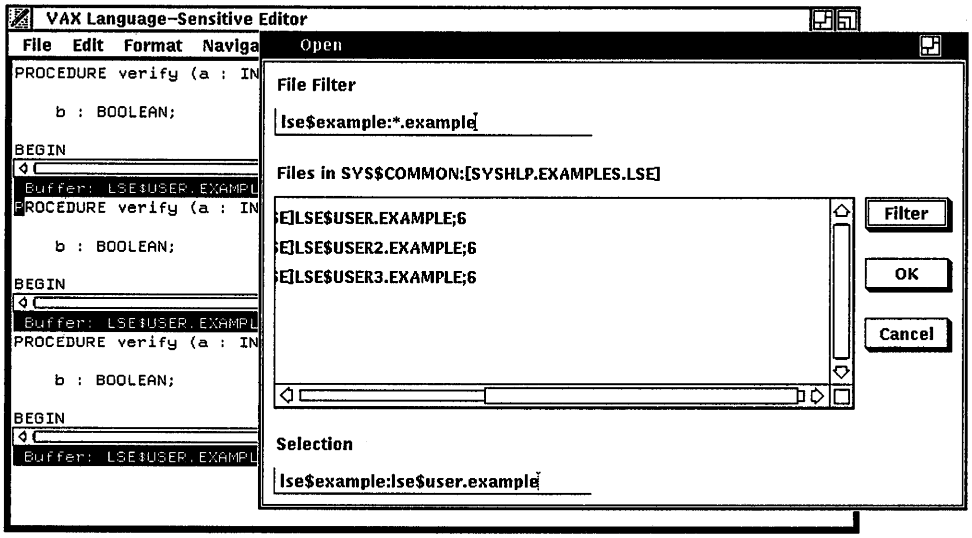
To open additional files while in this editing session, perform the following steps (you should still have three windows on your screen):

1. Position the cursor on the middle window and press MB1 to make that the current window.
2. Pull down the File menu.
3. Choose the Open... menu item.

   The Open... dialog box appears on the screen. Notice that the file LSE$USER.EXAMPLE, which was specified at the beginning of the session, is still in the Selection field.
4. Click on the File Filter field.
5. Type LSE$EXAMPLE:*.EXAMPLE .
6. Click on Filter.

   LSE lists all the files located in LSE$EXAMPLE that have the file type .EXAMPLE, as shown in Figure 4–8. This is useful when you do not remember the exact file that you want to access.

Figure 4–8: Specifying a Filter


VAX Language–Sensitive Editor
File Edit Format Naviga
Open
File Filter
Files in SYS$COMMON:[SYSHLP.EXAMPLES.LSE]
Filter
OK
Cancel
Selection

```
PROCEDURE verify (a : IN

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPL
PROCEDURE verify (a : IN

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPL
PROCEDURE verify (a : IN

    b : BOOLEAN;

BEGIN
 Buffer: LSE$USER.EXAMPL
```

lse$example:*.example

```
E]LSE$USER.EXAMPLE;6
E]LSE$USER2.EXAMPLE;6
E]LSE$USER3.EXAMPLE;6
```

lse$example:lse$user.example

7. Double click on LSE$USER2.EXAMPLE.

   LSE places the file in the current buffer and removes the Open... dialog box.
8. Click MB1 on the third window to make that the current window.
9. Pull down the File menu.
10. Choose the Open... menu item.

    The Open... dialog box appears on the screen. Notice that the file filter and list of files specified earlier are still in the dialog box.
11. Double click on LSE$USER3.EXAMPLE to place the file in the current window and remove the Open... dialog box.

To display only the original buffer, LSE$USER.EXAMPLE, on the screen, perform the following steps:

1. Position the pointer on the buffer LSE$USER.EXAMPLE and press MB1 to make that the current buffer.
2. Pull down the Display menu.
3. Choose the One Window menu item.

LSE displays the buffer LSE$USER.EXAMPLE and removes the two other windows.

4.2.8 Moving Through Buffers

You can move through buffers in two ways:

- Click MB1 on the buffer name status line indicator
- Choose the Show Buffer menu item to display a list of buffers

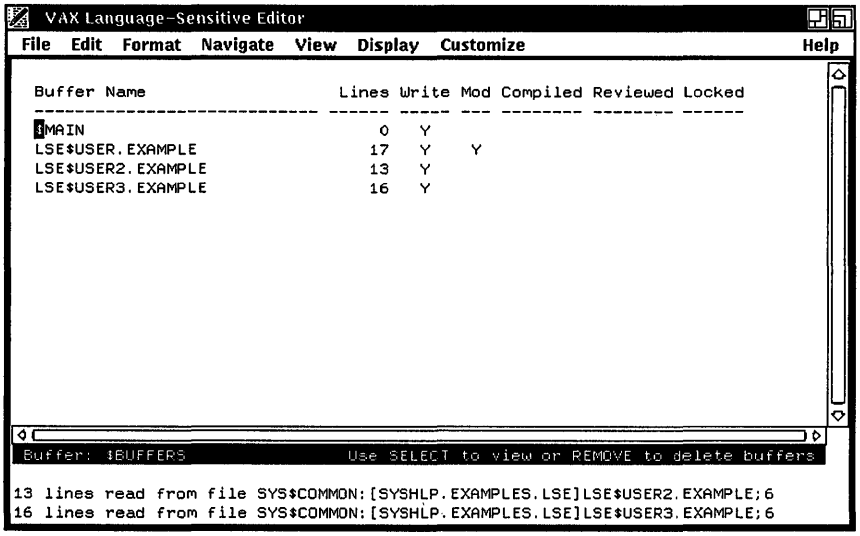
Click MB1 on the buffer name indicator to move through buffers $MAIN, LSE$USER.EXAMPLE, LSE$USER2.EXAMPLE, and LSE$USER3.EXAMPLI

When you have many buffers, or when you know which buffer you want, you can use the Show Buffers menu item in the Display menu to get a list of all the buffers.

To display a list of buffers by using the Show Buffer menu item, perform the following steps:

1. Pull down the Display menu.
2. Choose the Show Buffer menu item.
   LSE displays a list of the buffers in the current window, as shown in Figure 4–9.

Figure 4–9: Displaying a List of Buffers


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Buffer Name                    Lines Write Mod Compiled Reviewed Locked
------------------------------ ----- ----- --- -------- -------- ------
$MAIN                              0   Y
LSE$USER.EXAMPLE                  17   Y    Y
LSE$USER2.EXAMPLE                 13   Y
LSE$USER3.EXAMPLE                 16   Y

Buffer: $BUFFERS                  Use SELECT to view or REMOVE to delete buffers

13 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.LSE]LSE$USER2.EXAMPLE;6
16 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.LSE]LSE$USER3.EXAMPLE;6
```

3. Double click on LSE$USER.EXAMPLE.

   LSE removes the list of buffers and displays LSE$USER.EXAMPLE in the current window.

4.2.9 Reviewing Source Code

The following example shows you how to review source code within LSE. This example assumes that the current buffer, LSE$USER.EXAMPLE, was just compiled (by pulling down the File menu and choosing the Compile menu item).

The REVIEW command instructs LSE to review the compilation errors generated by the COMPILE command. You can invoke the REVIEW command as follows:

- Pull down the File menu
- Choose the Review menu item

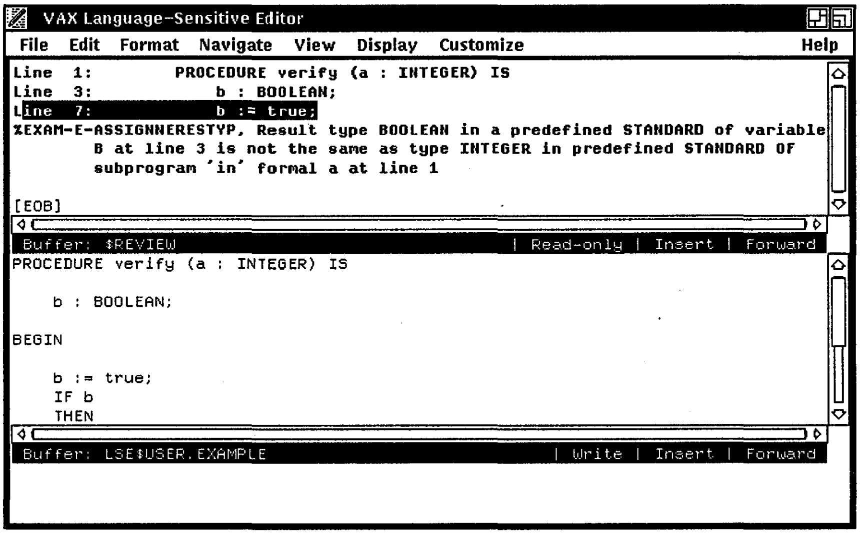
However, for this example, you must use the REVIEW/FILE command to review the errors in LSE$USER.EXAMPLE. This is because you do not actually compile LSE$USER.EXAMPLE, and the corresponding diagnostic file is not in your default directory. LSE provides a command line interface, which is useful in cases like this when you want to issue commands with specific qualifiers or commands that are not found in any menu.

To review the errors in LSE$EXAMPLE:LSE$USER.EXAMPLE, perform the following steps:

1. Press the Do Key to get the LSE Command> prompt.
2. Type the command REVIEW/FILE=LSE$EXAMPLE:LSE$USER.DIA.

   LSE splits the screen into two windows, displaying the compilation errors in the top window and the source code in the bottom window, as shown in Figure 4–10.

Figure 4–10: The REVIEW Buffer


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

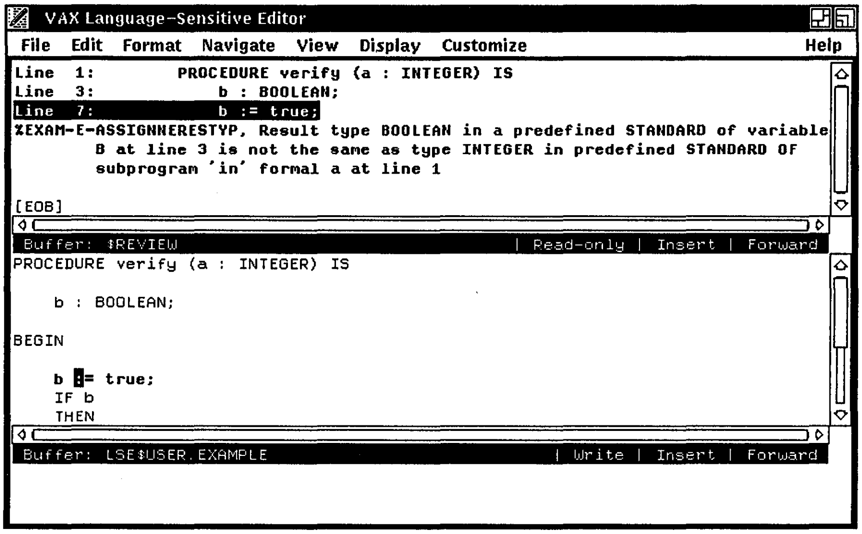
```
Line  1:        PROCEDURE verify (a : INTEGER) IS
Line  3:              b : BOOLEAN;
Line  7:              b := true;
%EXAM-E-ASSIGNMERESTYP, Result type BOOLEAN in a predefined STANDARD of variable
        B at line 3 is not the same as type INTEGER in predefined STANDARD OF
        subprogram 'in' formal a at line 1

[EOB]
 Buffer: $REVIEW                                    | Read-only | Insert | Forward
PROCEDURE verify (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
 Buffer: LSE$USER.EXAMPLE                               | Write | Insert | Forward
```

3. Position the pointer on Line 7 in the $REVIEW buffer and double click MB1.

   Double clicking MB1 makes Line 7 the current error and takes you to the source code associated with that line, as shown in Figure 4–11.

(Clicking MB1 only once makes Line 7 the current error; it does not take you to the source code.)

Figure 4–11: Corresponding Source Code


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  1:        PROCEDURE verify (a : INTEGER) IS
Line  3:            b : BOOLEAN;
Line  7:            b := true;
%EXAM-E-ASSIGNNERESTYP, Result type BOOLEAN in a predefined STANDARD of variable
        B at line 3 is not the same as type INTEGER in predefined STANDARD OF
        subprogram 'in' formal a at line 1

[EOB]
 Buffer: $REVIEW                                   | Read-only | Insert | Forward
PROCEDURE verify (a : INTEGER) IS

    b : BOOLEAN;

BEGIN

    b := true;
    IF b
    THEN
 Buffer: LSE$USER.EXAMPLE                              | Write | Insert | Forward
```

4. Position the pointer on Line 3, and double click MB1.

   Again, LSE takes you to the source code associated with that line. You can use the mouse to go back and forth between errors and the source code until you have corrected all the errors.

To end the review session, perform the following steps:

1. Click MB2 in the $REVIEW buffer.

   LSE displays the $REVIEW buffer pop-up menu.

2. Choose the End Review menu item.

4.3 Querying with SCA

You can use the Source Code Analyzer FIND command and the basic LSE navigational commands to perform queries. The following example uses a sample SCA library (SCA$EXAMPLE) that is supplied with the SCA kit. You can duplicate this example on your screen if you have SCA installed on your system.

To use SCA, you must select an SCA library. This example uses the SCA$EXAMPLE library, as follows:

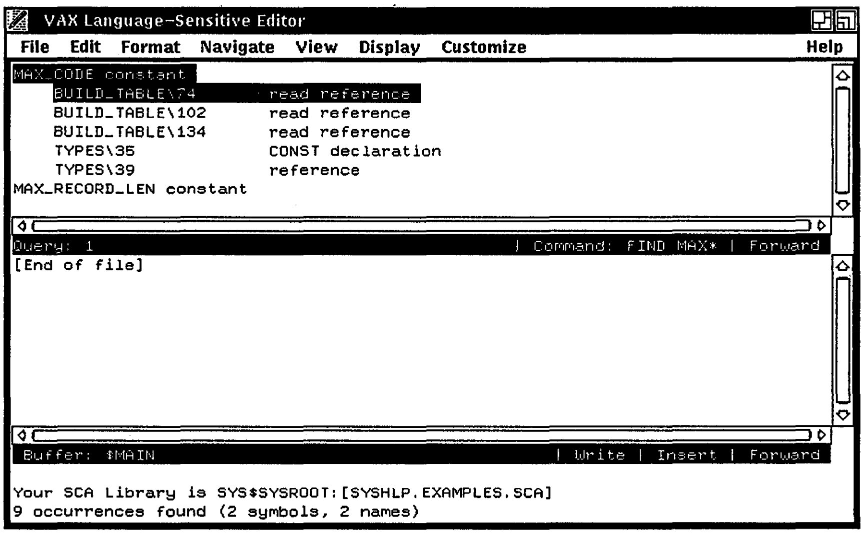
1. Press the Do key to get the LSE Command> prompt.
2. Type SET LIBRARY SCA$EXAMPLE.

To display all the occurrences of symbols that begin with the prefix MAX, perform the following steps:

1. Press the Do key to get the LSE Command> prompt.
2. Type FIND MAX*.

Figure 4–12 shows the resulting query buffer. The first occurrence of the symbol is highlighted.

Figure 4–12: SCA Query Buffer

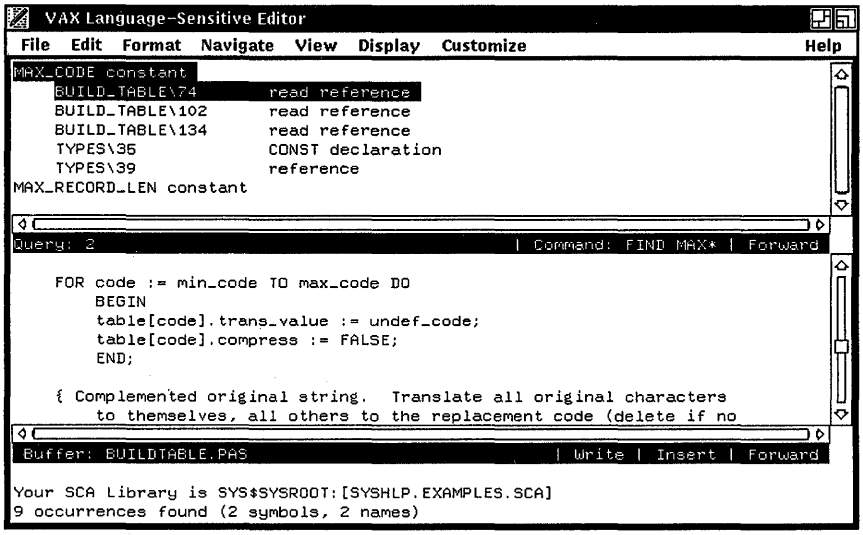

VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
MAX_CODE constant
    BUILD_TABLE\74         read reference
    BUILD_TABLE\102        read reference
    BUILD_TABLE\134        read reference
    TYPES\35               CONST declaration
    TYPES\39               reference
MAX_RECORD_LEN constant
Query: 1                                    | Command: FIND MAX* | Forward
[End of file]
 Buffer: $MAIN                                  | Write | Insert | Forward
Your SCA Library is SYS$SYSROOT:[SYSHLP.EXAMPLES.SCA]
9 occurrences found (2 symbols, 2 names)
```

To go to the corresponding source code for the occurrence of MAX_CODE, click MB1 twice on the first occurrence of MAX_CODE.

Double clicking MB1 makes it the current occurrence and takes you to the corresponding source code. (Clicking MB1 only once makes it the current occurrence; it does not take you to the source code.) The source code is displayed in another window, as shown in Figure 4–13.

Figure 4–13: Source Code Corresponding to First Occurrence


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
MAX_CODE constant
    BUILD_TABLE\74       read reference
    BUILD_TABLE\102      read reference
    BUILD_TABLE\134      read reference
    TYPES\35             CONST declaration
    TYPES\39             reference
MAX_RECORD_LEN constant
Query: 2                                    | Command: FIND MAX* | Forward

    FOR code := min_code TO max_code DO
        BEGIN
        table[code].trans_value := undef_code;
        table[code].compress := FALSE;
        END;

    { Complemented original string.  Translate all original characters
        to themselves, all others to the replacement code (delete if no
 Buffer: BUILDTABLE.PAS                         | Write | Insert | Forward

Your SCA Library is SYS$SYSROOT:[SYSHLP.EXAMPLES.SCA]
9 occurrences found (2 symbols, 2 names)
```

You can also use the query buffer pop-up menu to move between items and source. To end the query session, perform the following steps:

1. Move the pointer to anywhere in the query buffer.
2. Click MB2.

   The query buffer pop-up menu is displayed on the screen.
3. Choose the Delete Query menu item.

4.3.1 Ending the Editing Session

To end this example session, perform the following steps:

1. Pull down the File menu.
2. Choose the Quit menu item.

You can use LSE$USER.EXAMPLE and the SCA$EXAMPLE library at any time to increase your proficiency with LSE DECwindows. The examples in this chapter demonstrate only the most basic uses of the features of LSE and SCA.

Chapter 5

Performing Language-Specific Tasks

LSE's language-specific features simplify the tasks of developing and maintaining software systems. These features include language-specific placeholders and tokens, pseudocode placeholders, aliases, comment and indentation control, and templates for subroutine libraries. This chapter describes these features in detail.

Sections 5.1 and 5.2 describe how to use placeholders and tokens. Section 5.3 describes how to use pseudocode, including typing, processing, and converting pseudocode into comments. Section 5.4 provides information on using aliases, and Section 5.5 provides information on using packages. Section 5.6 provides information on using comments.

5.1 Using Placeholders

You can use LSE as a traditional text editor. In addition, you have the power of using LSE's tokens and placeholders to step through each program construct and to supply text for those constructs needing it.

Placeholders are markers in the source code that indicate locations where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. Generally, you do not need to type placeholders; rather, they are inserted for you by LSE. Placeholders can be recognized by their surrounding brackets or braces, the choice being language-dependent. There are four types of placeholders:

- Terminal placeholders
- Nonterminal placeholders
- Menu placeholders
- Pseudocode placeholders

Terminal placeholders provide text strings that describe valid replacements for the placeholder. **Nonterminal placeholders** expand into additional language constructs. **Menu placeholders** provide a list of options corresponding to the placeholder. **Pseudocode placeholders** are slightly different. They contain free text and are not defined by LSE. Pseudocode placeholders are delimited by different brackets than the other placeholders. The type of a placeholder is a property of the placeholder name.

Placeholders are either optional or required. Required placeholders, indicated by braces, represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets, represent places in the source code where you can either provide additional constructs or erase the placeholder. Pseudocode placeholders are optional placeholders that contain design information that you supply.

The following example shows required and optional placeholders. The declaration

```
INTEGER {identifier}...
```

when expanded becomes

```
INTEGER id,
        [identifier]...
```

The first appearance of the identifier placeholder is surrounded by braces because you need at least one identifier in this declaration. The second appearance is surrounded by brackets because additional identifiers are optional.

Some placeholders are duplicated when expanded. These placeholders are followed by an ellipsis. Generally, these placeholders represent items such as identifiers, statements, expressions, datatypes, or any location where lists of items are expected. A placeholder is duplicated either vertically or horizontally, depending on the context and the placeholder definition. For example, the placeholder [identifier] in the previous example was duplicated vertically.

You may move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed. With LSE, you can specify uppercase, lowercase, or AS_IS in the language definition for all text expanded into the buffer.

You may modify placeholder definitions by means of the EXTRACT command. See the Redefining Language Elements section in Chapter 13 for information on modifying placeholders.

Placeholder definitions may be stored in an environment file. See Chapter 15 for information on defining your own placeholders.

5.2 Using Tokens

Tokens are typically keywords in programming languages. When expanded, tokens provide additional language constructs. Tokens are typed directly into the buffer. Generally, tokens are used in situations when you want to add additional language constructs where there are no placeholders. For example, typing IF and pressing the EXPAND key causes a template for an IF construct to appear on your screen. Tokens are also used to bypass long menus in situations where expanding a placeholder, such as {statement}, would result in a lengthy menu.

You can use tokens to insert text when editing an existing file. Because most languages have tokens for built-in functions and keywords, you type the name for a function or keyword and press the EXPAND key. In addition, most languages provide a token named *statement* or *expression* that expands into a menu of all possible statements or expressions.

The following example demonstrates how to use tokens to edit an existing program. In this case, the buffer TEST.EXAMPLE contains the following code:

```
PROCEDURE test ()

    IF A = B
    THEN
        A = C + 1
    ENDIF
ENDPROCEDURE test
```

If you want to add more statements to this program before the IF construct, do the following:

1. Move the cursor to the beginning of the IF statement line.
2. Press the OPEN LINE key (PF1-KP0).
3. Press the Tab key.

   Note that the cursor is placed at the same level of indentation as the IF statement line.
4. Type *statement* and press the EXPAND key.

A menu of statements now appears on your screen. You use the arrow keys to scroll through the menu. To select a menu item, you press the EXPAND, Enter, or Return key. You can press the spacebar to exit from a menu without selecting an item.

5.3 Using Pseudocode

Pseudocode placeholders are placeholders that contain natural language text that expresses design information.

Pseudocode is easy to write and provides a way to sketch your design ideas. You can convert pseudocode into comments, thus providing a way to preserve design information. You can use a compiler to process pseudocode to detect syntax errors and generate SCA data.

For example, the following code fragment contains pseudocode delimited by the special brackets « and » defined by the DEFINE LANGUAGE command:

```
procedure my_proc (file_name : in out my_type) is
begin
    «read the file»;
    if «the file is empty» then
        «clean up»;
        «stop»;
    else
        «process the file»;
    end if;
    put_message;
end my_proc;
```

Pseudocode placeholders can appear only in well-defined places, such as in place of statements, expressions, and declarations; otherwise, the compiler will issue an error message. The following example shows pseudocode placeholders used in place of statements:

```
procedure do_something is
begin
    «Open the file.»
    «Read and process records.»
    «Write the file.»
    «Close the file.»
end do_something;
```

The following examples show pseudocode used in place of expressions:

```
if «it is a leap year»

while «there is snow» or «it is winter» loop

max := «the largest number in the list»

d := «current temperature» + 50

CASE color FROM «the darkest» to «the lightest» OF

put («the person's full name»);
```

5.3.1 Typing Pseudocode

To enter pseudocode, use the following steps:

1. Press PF1-spacebar (the ENTER PSEUDOCODE command).
2. Type in the text you want.

When you issue the ENTER PSEUDOCODE command, the cursor can be positioned on a regular placeholder if the placeholder is defined with the /PSEUDOCODE qualifier.

For example, position the cursor on the following placeholder:

```
if {condition}
```

Press PF1-spacebar. A pseudocode placeholder replaces the {condition} placeholder, as follows:

```
if «»
```

You can type any text between the pseudocode placeholder delimiters, as follows:

```
if «the order is for hardware»
```

In order to use pseudocode, pseudocode placeholder delimiters must be defined for the target language. See the DEFINE LANGUAGE and MODIFY LANGUAGE commands in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details.

5.3.2 Creating Comment Text from Pseudocode

With LSE, you can move pseudocode to program comments. This makes it easy to turn pseudocode into programs. You can specify the comment format and placement. You can convert pseudocode to block or line comments.

For example, to move text from a pseudocode placeholder into a block comment, use the following steps:

1. Position the cursor on a pseudocode placeholder.
2. Press PF1-B (the ENTER COMMENT/BLOCK command).

For example, use the following code fragment:

```
if «this is leap year» then
    «Add 1 to the number of days in February»
end if;
```

Press PF1-B. LSE replaces the target pseudocode placeholder with the generic placeholder *{tbs}* and moves the free text into comment text on a new line above the current line, as follows:

```
if «this is leap year» then
    -- Add 1 to the number of days in February
    {tbs}
end if;
```

To put pseudocode text into a comment at the end of the line, use the following steps:

1. Position the cursor on a pseudocode placeholder.
2. Press PF1-L (the ENTER COMMENT/LINE command).

Again, use the same code fragment with the cursor positioned on the first pseudocode placeholder:

```
if «this is leap year» then
    «Add 1 to the number of days in February»
end if;
```

Press PF1-L. LSE replaces the pseudocode placeholder with the generic placeholder, {tbs}, and moves the free text into comment text at the end of the current line, as follows:

```
if {tbs} then -- this is leap year
    «Add 1 to the number of days in February»
end if;
```

In both examples, the cursor remains on the generic placeholder, so you can enter program text.

The ENTER COMMENT command also moves sequences of pseudocode into comments. If there is a selected range active when you issue the ENTER COMMENT/BLOCK command, LSE copies the selected text into a comment.

For example, use the following code fragment:

```
if «there is no file»
   «for us to open» then
   «create a file»
end if;
«open the file»
```

If all the lines are selected, and you issue the ENTER COMMENT/BLOCK command, the code fragment becomes as follows:

```
-- if there is no file
--    for us to open then
--    create a file
-- end if;
-- open the file
--
if {tbs} then
   {tbs}
end if;
{tbs}
```

If you place the cursor in an existing comment, and issue the ENTER COMMENT command, LSE converts the next pseudocode placeholder into a comment. For example, the code fragment

```
--
«Average the numbers.»
```

becomes

```
-- Average the numbers.
{tbs}
```

If the cursor is not in a comment or on a placeholder when you issue the ENTER COMMENT command, LSE inserts a new comment and puts the cursor on the first placeholder after the beginning of the comment.

To use the ENTER COMMENT command, the following placeholders must be defined for the target language:

- LSE$BLOCK_COMMENT
- LSE$LINE_COMMENT
- LSE$GENERIC

See the ENTER COMMENT command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details.

5.3.3 Processing Pseudocode

When you compile pseudocode, the compiler checks for syntax errors and generates .ANA files. SCA uses the files to generate reports about pseudocode, including call trees, dependency tables, and other cross-referencing. SCA provides query support for pseudocode just as it does for source code.

To compile your unfinished programs from within LSE, use the appropriate compile command for the language you are using. See the compiler /DESIGN description in the language documentation for more information.

5.4 Using Aliases

An **alias** is an abbreviation for a long text string or identifier that you want to enter repeatedly into your source code.

You can use the DEFINE ALIAS command to define an alias. LSE prompts you for an alias name and value if you do not specify them on the command line. Expanding aliases results in simple string substitutions.

For example, you can define an alias for a long identifier such as PAS_COPY_NNT as follows:

```
LSE>  DEFINE ALIAS PCN PAS_COPY_NNT
```

Now, you can type PCN and press CTRL/E (the EXPAND key) to insert PAS_COPY_NNT into the buffer.

When defining an alias that contains nonalphanumeric characters, you must use quotation marks, as follows:

```
LSE>  DEFINE ALIAS INCR "X = X + 1"
```

You can use the DEFINE ALIAS/INDICATED command (PF1-CTRL/A) to define an alias name for the identifier or text string on which the cursor is currently located. This eliminates the need to provide the alias value while defining an alias.

5.5 Packages

LSE provides templates for subroutine packages. These packages define VMS System Services, Run-Time Library (LIB$, STR$, SMG$), and VAX Record Management System (VAX RMS) routines. In addition, LSE provides a mechanism for defining packages for your own subroutine libraries. (See Chapter 15 for details on defining your own packages.)

The System Services and VAX RMS packages consist of routine definitions and parameter definitions that are available automatically when you use LSE with any of the following languages:

- VAX Ada
- VAX BASIC
- VAX BLISS
- VAX C
- VAX COBOL
- VAX FORTRAN

- VAX Macro
- VAX Pascal
- VAX PL/I

Routines are useful for describing subroutine libraries. Not only are they language-independent but they need to be defined only once. Routine names are used in the same way tokens are used. For example, if you type the routine name SYS$FILESCAN and expand it, the following results:

```
sys$filescan    ( {srcstr},
                  {valuelst},
                  [fldflags] )
```

Most languages access System Services and VAX RMS routines with the prefix *SYS$*. These languages must use the SYSTEM_SERVICES package. Other languages use different prefixes. For example, VAX Ada prohibits the prefix dollar sign ($) and must use the STARLET package. VAX BLISS and VAX Pascal require the prefix dollar sign ($) and must use the KEYWORD_SYSTEM_SERVICES package.

For example, to call the $SNDOPR system service from a VAX PL/I program, you type the following:

```
    status := sys$sndopr
```

Then, you press the EXPAND key with the cursor just after sys$sndopr. This would expand to the following:

```
    status := sys$sndopr (
                  {msgbuf},
                  [chan])
```

This indicates that the $SNDOPR system service has two parameters: MSGBUF, which is required, and CHAN, which is optional. Since CHAN is optional, LSE expands it with an optional placeholder that you can either delete or expand. Languages other than VAX Ada and VAX BLISS have similar features.

In VAX Ada, the dollar sign is not used as part of the system service name. Thus, you could type the following:

```
    starlet.sndopr
```

Then, you would press the EXPAND key. This would expand to the following:

```
    STARLET.SNDOPR (
        STATUS => {status},
        MSGBUF => {msgbuf},
        [CHAN   => {chan}]);
```

In VAX BLISS, the system services start with a dollar sign, without the leading SYS. Thus, you could type the following:

```
    status = $sndopr
```

Then, you would press the EXPAND key. This would expand to the following:

```
    status = $sndopr (
                     msgbuf = {~msgbuf~},
                     [~chan   = {~chan~}~])
```

You can access VMS online help for any of the system services in any language. If you want help on any routine, place the cursor on the routine name and press the HELP/INDICATED key (PF1-PF2). You cannot use HELP/INDICATED on the parameter names; however, the HELP entry for the system service will contain information on the parameters.

If you want to see the contents of a given package, parameter, or routine, you can use the SHOW command. If you want to modify the definitions of a package, you can use the EXTRACT command.

5.6 Using Comments

LSE recognizes many of the comments that occur in code. In many cases, LSE handles comments specially to help you keep comments aligned. You can use the ALIGN command to align all the comments within a region so that they line up in the same column. You can use the FILL command to both align comments and to fill out each comment line by putting as many words on a line as will fit within the margins. In addition, LSE treats comments specially when you erase or duplicate a placeholder.

Special handling of comments applies only to trailing comments. A trailing comment is one that occurs as the last item on a line, excluding blank space. Lines containing only comments are considered to be trailing comments.

Two types of comments are recognized: bracketed comments and line comments. A **bracketed comment** requires both a beginning and ending delimiter. For example, Pascal uses either braces ({}) or a parenthesis and asterisk ((**)) as bracketed comment delimiters. A **line comment** begins with a comment delimiter, but is terminated by the end of the line. For example, Ada uses a double dash (--) to introduce a line comment, while VAX FORTRAN uses an exclamation mark (!).

Note that some languages, such as BLISS, have both bracketed and line comments.

The ALIGN command aligns all trailing comments within the current selected region so that they start in the same column. The default behavior is to align the comments under the first comment within the region. You can also specify an explicit column with the /COMMENT_COLUMN qualifier.

An example of the ALIGN command follows. In the following examples, /COMMENT_COLUMN=CONTEXT_DEPENDENT is in effect.

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                    ! extended end-of-line comment block
   i := i + 1 ;
   j := j + i ;   ! another comment
 !to be filled
```

When aligned, it would look like this:

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                  ! extended end-of-line comment block
   i := i + 1 ;
   j := j + i ;   ! another comment
                  ! to be filled
```

The operation of the FILL command depends on the FILL setting of the current language. If the FILL setting is TEXT, then the FILL command performs an ordinary FILL operation. This is useful for text files or for files written in a markup language, such as DIGITAL Standard Runoff, but is usually inappropriate for most programming languages. If the FILL setting is COMMENTS, then the FILL command affects only the text within the trailing comments of the region. As with the ALIGN command, you can use the /COMMENT_COLUMN qualifier to explicitly specify the column in which to align the comments.

An example of the FILL command follows:

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                    ! extended end-of-line comment block
   i := i + 1 ;
   j := j + i ;   ! another comment
 !to be filled
```

When filled, it would look like this:

```
IF (col >= R_Margin) THEN ! This is the start of an extended
   BEGIN                  ! end-of-line comment block
   i := i + 1 ;
   j := j + i ;           ! another comment to be filled
```

Note that LSE deleted a comment delimiter. LSE inserts or deletes comment delimiters as necessary when you expand or erase placeholders inside comments.

Part 2 Using SCA

This part contains tutorial information on the VAX Source Code Analyzer.

Chapter 6

Introduction to SCA

This chapter provides an overview of the VAX Source Code Analyzer (SCA). Section 6.1 briefly describes the features of SCA and its integration into the VAX Language-Sensitive Editor (LSE) software development environment. Section 6.2 describes the use of compiler analysis data files with SCA. Section 6.3 describes ways you can invoke SCA. Section 6.4 lists the SCA commands.

6.1 Overview

The VAX Source Code Analyzer is an interactive, multilanguage, source code cross-reference and source code analysis tool that aids developers in understanding large-scale software systems. Because SCA deals with an entire software system, instead of individual modules, it is an effective tool during implementation and maintenance phases of a project.

The use of SCA is based on methods commonly used to develop software with VMS systems. For example, the following techniques are assumed:

- A set of sources is conveniently located for development.
- Developers modify, link, and compile sources until an executable image is successfully created.
- When an image is successfully built, the *specific* set of sources associated with the image are captured as a baseline for further development. (If the VAX Code Management System (CMS) were used to store these sources, a CLASS would probably be formed to contain the baseline versions.)
- As development continues, developers must work with the information contained in the build sources.

With these assumptions, SCA stores compiler-generated information about the set of build sources for querying in one unique location (an SCA library). Thus, SCA is a query tool that allows you to reference and query time-stamped source information that directly corresponds to source modules in your system. When these sources are no longer of value, you can modify or delete the SCA library. For more information on using SCA libraries, see Chapter 7 and Chapter 10.

The library data generated by supporting VMS compilers consists of the names of all of the symbols, modules, and files contained in a specific snapshot of the source. Once SCA libraries are created, you can select a library and query its contents from within LSE, at the DCL level, or via the SCA callable interface. This chapter discusses how to use SCA from within LSE and at the DCL level. For more information on using the SCA callable interface, see the appendix on the SCA callable interface in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*

The following sections provide a brief description of SCA features, and demonstrate the use of SCA as an integrated query tool within the LSE environment. For detailed descriptions of data generation, library creation, and querying, see Chapter 7.

6.1.1 SCA Features

The following sections provide brief descriptions of the key features of SCA.

Cross-Referencing and Analysis

The cross-referencing and analysis features of SCA provide information about program symbols and source files. These capabilities are provided by the FIND command, and allow you to do the following:

- Locate symbols and occurrences of symbols
- Limit queries based on selection of attributes, including symbol name, symbol class (routine, variable), and occurrence class (primary declaration, read or write reference)
- Select one symbol based on the relationships between symbols (variables, routines, modules)
- Display relationships between symbols (routine calls, type trees)
- Select symbols based on combinations of attributes and relationships

For more information on SCA cross-referencing and analysis, see Chapter 7 and Chapter 8.

Consistency Checking

The consistency-checking feature of SCA checks occurrences of symbols for consistent use. This capability is provided by the INSPECT command. For more information on SCA consistency checking, see Chapter 11.

LSE Integration

Tightly knit integration with LSE provides for multimodule navigation and the ability to read and modify associated source code.

Navigation within LSE is powerful and flexible. Using SCA, you can create one or more queries. You use cross-reference and analysis commands to query source information, and navigational commands to move through the information and to gain access to related sources.

Navigation through a query buffer is provided by the NEXT and PREVIOUS commands; direct access to source buffers is provided by the GOTO command. For details of query concepts, see Chapter 7.

These features allow you to readily gain access, within a single editing session, to any or all of the modules and files comprising a software system. They not only expand your analysis capabilities but further speed up and simplify your development and maintenance tasks.

6.1.2 Querying with SCA

SCA's inquiry and reporting facilities allow you to query a library for the presence of specific symbol, file, or module information, and to determine such things as declarations of program symbols, references to the symbols, and references to source files. You can also determine the call relationships between routines by displaying call tree information. Within the editing environment, you can navigate through the complexities of an entire system and, as necessary, inspect and edit related source files.

SCA provides the following capabilities:

- Interactive query of symbol, module, and file information
- Display of routine call relationships and type trees
- Inspection of routines, variables, and other symbols
- Maintenance of source code information libraries

LSE provides the following additional capabilities:

- Navigation through one or more SCA queries
- Access and display of source code during an interactive query

With LSE editing features, you can move through an unfamiliar system without regard for module or file boundaries. For example, given the task of modifying the characteristics of a variable, you can locate all of the uses of the variable across the system and make your changes without leaving LSE.

6.2 SCA Analysis Data Files

SCA depends on VMS compilers for the generation of detailed source analysis data. Source analysis data is information about all of the symbols, files, and modules contained in the source. The information is loaded into an SCA library and used as a database for the SCA cross-reference query and source code analysis features.

This section discusses how you generate and use SCA analysis data files. Chapter 7 demonstrates the use of analysis data files and SCA libraries.

You produce analysis data by using the DCL command line of the form:

```
Language/ANALYSIS_DATA[=file-spec] [/...] source-file[,...]
```

The /ANALYSIS_DATA qualifier requests that the specified compiler generate an output file of source information having a default file type of .ANA. Unless otherwise specified, the .ANA files generated appear in your current default directory.

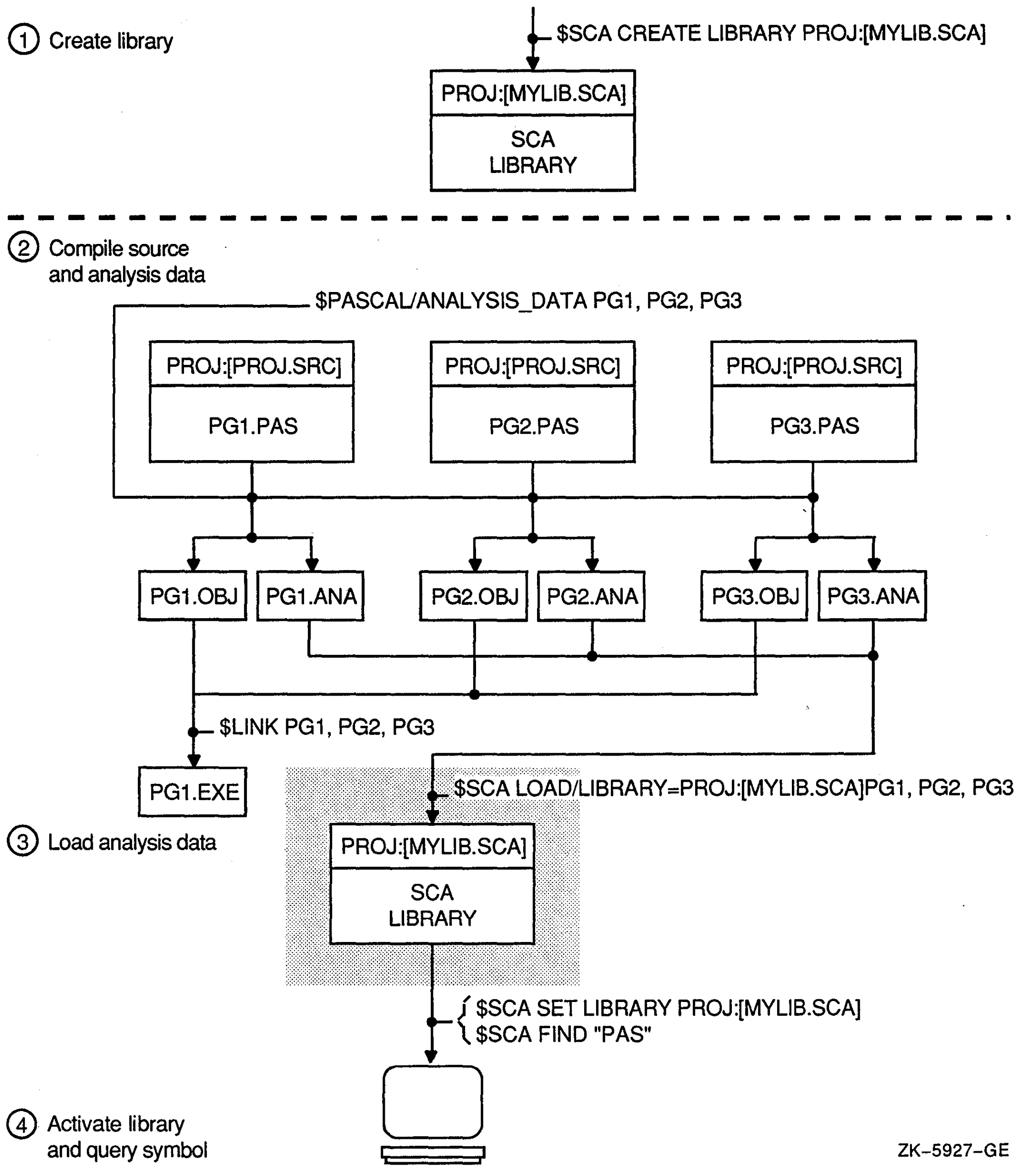
For example, the following Pascal command line compiles the specified input files (.PAS) and creates the requested output files (.OBJ and .ANA).

```
$ PASCAL/ANALYSIS_DATA PG1,PG2,PG3
```

Figure 6–1 shows the steps you can use to set up an SCA environment:

1. Create an SCA library in a local subdirectory (PROJ:[MYLIB.SCA]).
2. Create the data analysis files (PG1.ANA, PG2.ANA, PG3.ANA) with the Pascal compiler.
3. Load the data analysis files into the local SCA library (LOAD).
4. Activate the library (SET LIBRARY) and query the information (FIND).

Figure 6–1: Setting Up an SCA Environment



For a detailed overview of the LSE/SCA integrated environment, see *Using VAXset*. For more information on setting up the SCA environment, see Chapter 10.

6.2.1 Using the VAX Source Code Analyzer ANALYZE Command

SCA provides the ANALYZE command for those languages whose compiler output do not support SCA. The ANALYZE command creates an analysis data file that describes the source file by looking at language-specific rules for forming names (identifiers), comments, quoted strings and placeholders. The analysis data files produced by this command have a minimal description of the source file; they describe the source primarily as a set of references to unbound names.

The ANALYZE command depends on LSE environment files for information about particular languages. It uses the same mechanism to access environment files as LSE does. In particular, it depends on the LSE$ENVIRONMENT and LSE$SYSTEM_ENVIRONMENT logical names.

You must have a language defined in an environment file to use the ANALYZE command with that language. See Chapter 15 for more information on defining a language.

6.3 Invoking SCA

You can invoke SCA in three ways:

- With LSE (as an integrated tool)
- At the DCL level (as a standalone tool)
- With the SCA callable interface

As an integrated tool, LSE supports an expanded command language, which includes all SCA standalone commands and related navigational commands. SCA-related commands are defined in the Command Dictionary section of the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual*. SCA commands are issued in the same manner as LSE commands.

You issue SCA commands within LSE as follows:

```
LSE> command [parameter] [/qualifier...]
```

To invoke standalone SCA, and issue a command at the DCL level, enter the following:

```
$ SCA command [parameter] [/qualifier...]
```

You may also invoke standalone SCA at the DCL level by entering the following:

```
$ SCA
```

The SCA> prompt appears on your screen as follows:

```
SCA>
```

You may enter SCA commands at this prompt in the same way you do at the LSE prompt within LSE: type each command and execute it by pressing the Return or Enter key. An EXIT command ends an SCA session and returns you to the DCL level. You can also press CTRL/Z to end an SCA session.

6.4 SCA Commands

This section lists all SCA commands. You can issue these commands from within LSE, at the DCL level, or at the SCA level. See the Command Dictionary section of the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for definitions, descriptions, and examples of all SCA commands.

Library Commands

- ANALYZE
- CONVERT LIBRARY
- CREATE LIBRARY
- DELETE LIBRARY
- DELETE MODULE
- EXTRACT MODULE
- LOAD
- REORGANIZE
- SET LIBRARY
- SET NOLIBRARY
- SHOW LIBRARY
- SHOW MODULE
- SHOW VERSION
- VERIFY
- VERIFY/RECOVER

Query Commands

- DELETE QUERY
- FIND
- INSPECT
- REPORT
- SHOW QUERY

Navigation Commands

- COLLAPSE
- EXPAND
- GOTO (DECLARATION, QUERY, SOURCE)
- NEXT (OCCURRENCE, QUERY, STEP, SYMBOL)
- PREVIOUS (OCCURRENCE, QUERY, STEP, SYMBOL)

General Commands

- ATTACH
- EXIT
- HELP
- SPAWN

Chapter 7

Performing SCA Tasks

This chapter provides a tutorial that integrates the query and analysis capabilities of SCA with the multiwindow editing capabilities of LSE. For more information on the querying capabilities of SCA, see Chapter 8.

Section 7.1.1 describes how to invoke SCA. Section 7.1.3 through Section 7.1.5 show how an SCA environment is typically configured. The remainder of this chapter demonstrates SCA and SCA/LSE-related commands used for querying, navigating through query buffers, and accessing related sources.

7.1 Getting Started

The following sections describe how to use the basic SCA inquiry commands and related LSE navigational commands. You can duplicate the displays shown in the figures by using the SCA$EXAMPLE library and typing the sample commands. The commands allow you to do the following:

- Verify the successful selection of your SCA library (SHOW LIBRARY)
- Display information about all of the modules contained in your SCA library (SHOW MODULE)
- Locate occurrences of a specified symbol (FIND) and display the results in a query buffer
- Navigate through the information in the query buffer (NEXT STEP, PREVIOUS STEP)
- Access the source related to the specific occurrence of a symbol (GOTO SOURCE)
- Work with SCA queries (GOTO QUERY, NEXT QUERY, PREVIOUS QUERY, SHOW QUERY, DELETE QUERY)

7.1.1 Invoking SCA

To begin, invoke the VAX Language-Sensitive Editor (LSE), as described in Chapter 2.

Once in LSE, you use the SET LIBRARY command to access the example library. Type the following:

```
LSE>  SET LIBRARY SCA$EXAMPLE
```

A message appears in the message buffer at the bottom of your screen to indicate that you have successfully selected an SCA library. For the SCA$EXAMPLE library, the message reads as follows:

```
Your SCA Library is SCA$ROOT:[EXAMPLE]
```

7.1.2 Getting Help

Help is available for SCA within LSE as well as for SCA at the DCL or subsystem level.

To display help information about SCA$EXAMPLE, type the following:

```
LSE>  HELP SCA_TOPICS SCA_EXAMPLE
```

7.1.3 Selecting a Source Library

The SET SOURCE_DIRECTORY command is an LSE command that selects source directories to be used during your current LSE session. The command is not required if you use SCA$EXAMPLE (since related sources are available with the sample library); it is also not required if the sources corresponding to a selected SCA library are still in the directory from which they were originally compiled. The command has the following form:

```
SET SOURCE_DIRECTORY directory-spec [,directory-spec...]
```

As an example, the next command first searches for source files in a default directory, then a project library, and then a CMS library:

```
LSE>  SET SOURCE_DIRECTORY [],PROJ:[USER.BASE1.SRC],CMS$LIB
```

7.1.4 Displaying Library Specifications

The SHOW LIBRARY command displays each of the currently active SCA libraries.

To display the library active during your current session, type the following:

```
LSE> SHOW LIBRARY
```

For SCA$EXAMPLE, the display in the message buffer reads as follows:

```
Your SCA Library is SCA$ROOT:[EXAMPLE]
```

For more information on SCA libraries, see Chapter 10.

7.1.5 Displaying Module Information

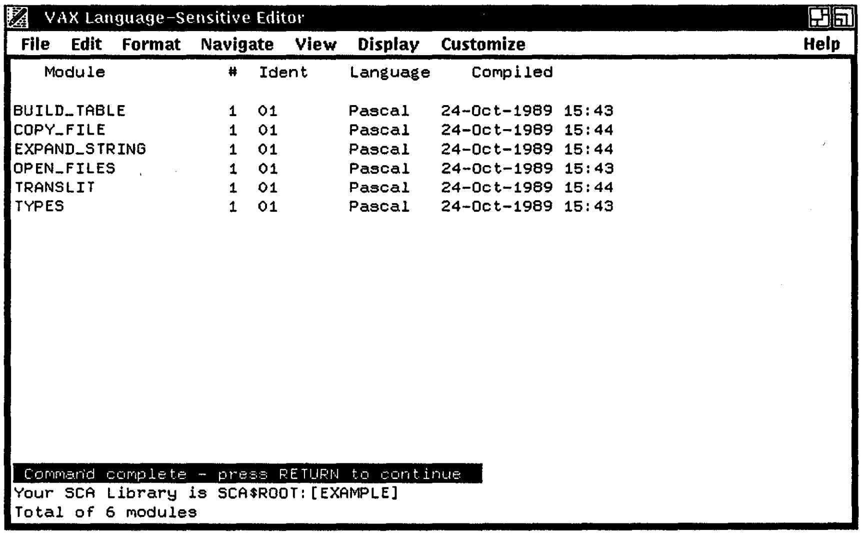
The SHOW MODULE command displays information about the modules contained in the current SCA library.

To display information about the modules contained in the SCA$EXAMPLE library, type the following:

```
LSE> SHOW MODULE
```

The resulting display, Figure 7–1, shows that the library contains six modules of compiled source information. The integers listed under the number sign (#) specify the active library from which the information is derived. Since SCA$EXAMPLE is the first and only active library on the **libraries list**, the modules are identified as being from library #1.

Figure 7–1: The SHOW MODULE Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
   Module             #  Ident     Language    Compiled

BUILD_TABLE           1  01        Pascal    24-Oct-1989 15:43
COPY_FILE             1  01        Pascal    24-Oct-1989 15:44
EXPAND_STRING         1  01        Pascal    24-Oct-1989 15:44
OPEN_FILES            1  01        Pascal    24-Oct-1989 15:43
TRANSLIT              1  01        Pascal    24-Oct-1989 15:44
TYPES                 1  01        Pascal    24-Oct-1989 15:43

 Command complete - press RETURN to continue
Your SCA Library is SCA$ROOT:[EXAMPLE]
Total of 6 modules
```

7.1.6 Using the FIND Command

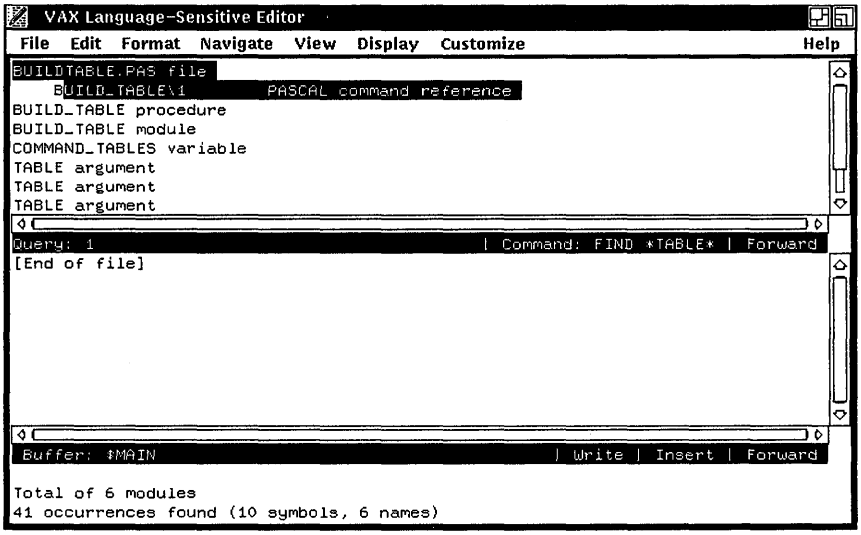
The following examples show how to use the FIND command to query SCA libraries for information about your source program. The examples also demonstrate the basic LSE navigational commands.

The first example demonstrates how to find a routine whose name you cannot remember. All you know is that the word "table" appears in it. To find all the symbols with the word "table," type the following command:

```
LSE> FIND *table*
```

Figure 7–2 shows the resulting display.

Figure 7–2: The FIND *table* Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
BUILDTABLE.PAS file
    BUILD_TABLE\1              PASCAL command reference
BUILD_TABLE procedure
BUILD_TABLE module
COMMAND_TABLES variable
TABLE argument
TABLE argument
TABLE argument
Query: 1                                          | Command: FIND *TABLE* | Forward
[End of file]
Buffer: $MAIN                                          | Write | Insert | Forward

Total of 6 modules
41 occurrences found (10 symbols, 6 names)
```

This query display shows the name and the class of all of the symbols whose name contains "table." The first symbol is a file named BUILDTABLE.PAS. The first symbol has already been expanded. The second one is a procedure named BUILD_TABLE.

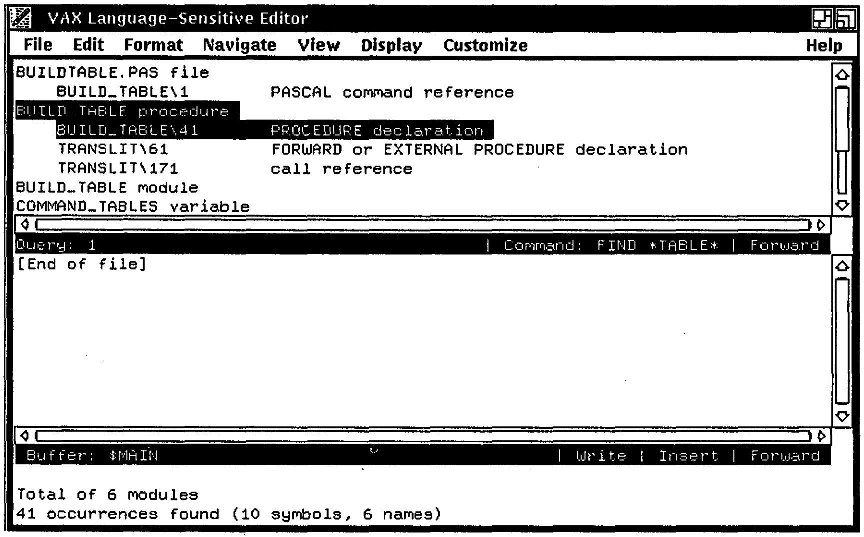
7.1.6.1 Navigating the Query Display

In a query display, you use navigation commands to move forward or backward through the display to select a specific occurrence of a symbol that you want to investigate. When you select an occurrence, the line is highlighted and you can then access the related source. You can step through a query display in increments, using the following commands:

- With the NEXT STEP (CTRL/F) command you can advance through the display from the current symbol to the next.
- With the PREVIOUS STEP (CTRL/B) command you can go back through the display from the current symbol to the previous one.

You can use the NEXT STEP navigation command to select the BUILD_TABLE procedure. You can use the EXPAND command (CTRL/E) to expand this one line and display the set of occurrences corresponding to the symbol. The result of expanding the entry for the BUILD_TABLE procedure is shown in Figure 7–3.

Figure 7–3: The Expanded BUILD_TABLE Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
BUILDTABLE.PAS file
    BUILD_TABLE\1           PASCAL command reference
BUILD_TABLE procedure
    BUILD_TABLE\41          PROCEDURE declaration
    TRANSLIT\61             FORWARD or EXTERNAL PROCEDURE declaration
    TRANSLIT\171            call reference
BUILD_TABLE module
COMMAND_TABLES variable
Query: 1                          | Command: FIND *TABLE* | Forward
[End of file]
 Buffer: $MAIN                          | Write | Insert | Forward

Total of 6 modules
41 occurrences found (10 symbols, 6 names)
```

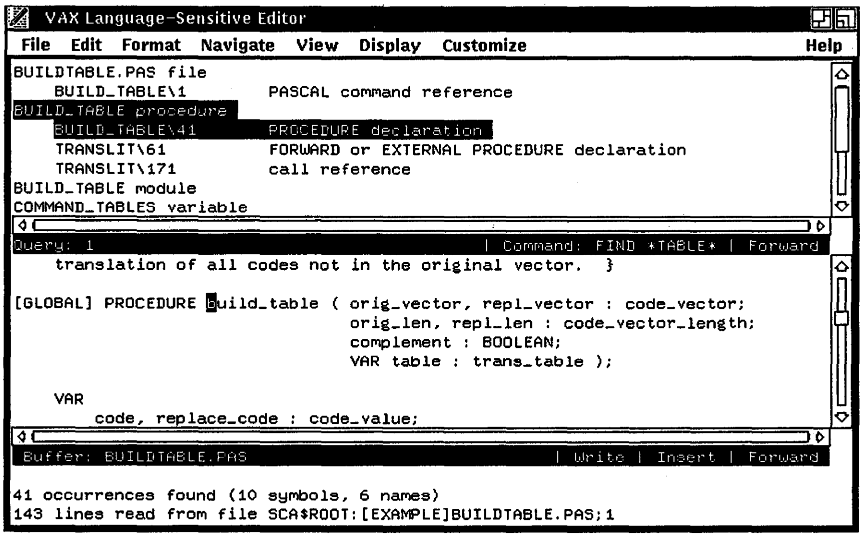
There are three occurrences of BUILD_TABLE. The first is the primary declaration of the procedure in module BUILD_TABLE at line 41. The second is an EXTERNAL declaration in module TRANSLIT. The third is a call to BUILD_TABLE at line 171 in TRANSLIT.

Following the expansion of BUILD_TABLE, the first occurrence is automatically selected. You can use the NEXT STEP and PREVIOUS STEP commands to select other occurrences of BUILD_TABLE.

You can also look at other symbols within this query display. The easiest way to do this is to first collapse the information about BUILD_TABLE by using the COLLAPSE command (CTRL/\). Collapsing the entry for the procedure BUILD_TABLE returns the display to that shown in Figure 7–2. You can then directly select the next symbol in the display, the module named BUILD_TABLE, by using the NEXT STEP command. You can select any of the symbols in the display, expand them, and then select their occurrences.

Returning to Figure 7–3, you determine that the procedure declaration of BUILD_TABLE is your target. Since you have already selected the appropriate entry in the query display, you can use the GOTO SOURCE command (CTRL/G) to access the source. The result of pressing CTRL/G is shown in Figure 7–4.

Figure 7–4: The GOTO Source Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
BUILDTABLE.PAS file
    BUILD_TABLE\1          PASCAL command reference
BUILD_TABLE procedure
    BUILD_TABLE\41         PROCEDURE declaration
    TRANSLIT\61            FORWARD or EXTERNAL PROCEDURE declaration
    TRANSLIT\171           call reference
BUILD_TABLE module
COMMAND_TABLES variable
Query: 1                                       | Command: FIND *TABLE* | Forward
    translation of all codes not in the original vector.  }

[GLOBAL] PROCEDURE build_table ( orig_vector, repl_vector : code_vector;
                                 orig_len, repl_len : code_vector_length;
                                 complement : BOOLEAN;
                                 VAR table : trans_table );

    VAR
        code, replace_code : code_value;
 Buffer: BUILDTABLE.PAS                             | Write | Insert | Forward

41 occurrences found (10 symbols, 6 names)
143 lines read from file SCA$ROOT:[EXAMPLE]BUILDTABLE.PAS;1
```

7.1.6.2 Moving to a Source Declaration

With the GOTO DECLARATION command, you can automatically access the source of a requested declaration. The command has the following form:

GOTO DECLARATION [/qualifier] [symbol_name]

The /INDICATED qualifier specifies that the name at the current cursor position is to be used as the symbol name. The current position in the buffer and the file specification for the current buffer are used to help identify the symbol name occurrence on which the cursor is positioned.

If no information exists in the SCA library for the symbol name at the current cursor position, the name alone is used for the query. If the /INDICATED qualifier is specified, the *symbol_name* parameter must not be specified.

Note that the /INDICATED qualifier is a positional qualifier that must be typed directly following the GOTO DECLARATION command.

The /PRIMARY qualifier displays the source of a symbol where the primary declaration is located. A primary declaration is the declaration determined by SCA to be the most significant. For example, SCA determines that the primary declaration of a routine is the declaration that describes the body of the routine. The default is /PRIMARY.

You type the command as follows:

```
LSE>  GOTO DECLARATION/INDICATED/PRIMARY
```

If you use the /CONTEXT_DEPENDENT qualifier without the /INDICATED qualifier, the primary declaration of the symbol is displayed (/PRIMARY is the default).

If you use the /CONTEXT_DEPENDENT qualifier with the /INDICATED qualifier, the context in which the symbol is found determines which declaration is displayed; however, if the symbol is declared in a file not found in the SCA library, the command ignores context information and displays the primary declaration. Otherwise, the declaration displayed is determined as follows:

- If the indicated symbol occurrence is a reference, the declaration referred to is displayed.
- If the indicated symbol occurrence is an associated declaration, the primary declaration is displayed.
- If the indicated symbol occurrence is a primary declaration, the associated declaration is displayed.

You type the command as follows:

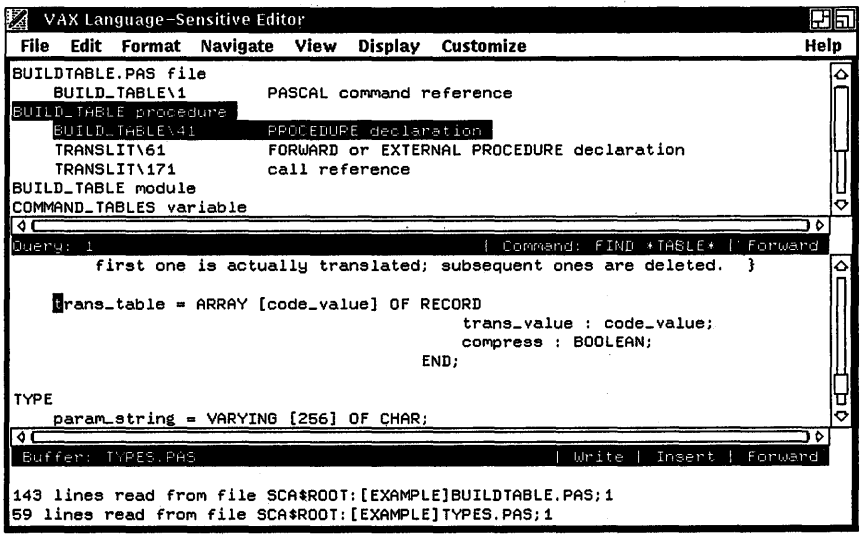
```
LSE>  GOTO DECLARATION/INDICATED/CONTEXT_DEPENDENT
```

The /ASSOCIATED qualifier displays the source of a symbol where the associated declaration is located. An associated declaration is a declaration related to the primary declaration. The default is /PRIMARY.

You can use the GOTO DECLARATION command to investigate the last parameter of the procedure BUILD_TABLE. To determine the type of the parameter TABLE, position the cursor on the occurrence of TRANS_TABLE, and enter the GOTO DECLARATION/INDICATED/PRIMARY command (CTRL/D). This brings you directly to the declaration of TRANS_TABLE.

Figure 7–5 shows the resulting display.

Figure 7–5: The GOTO DECLARATION Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

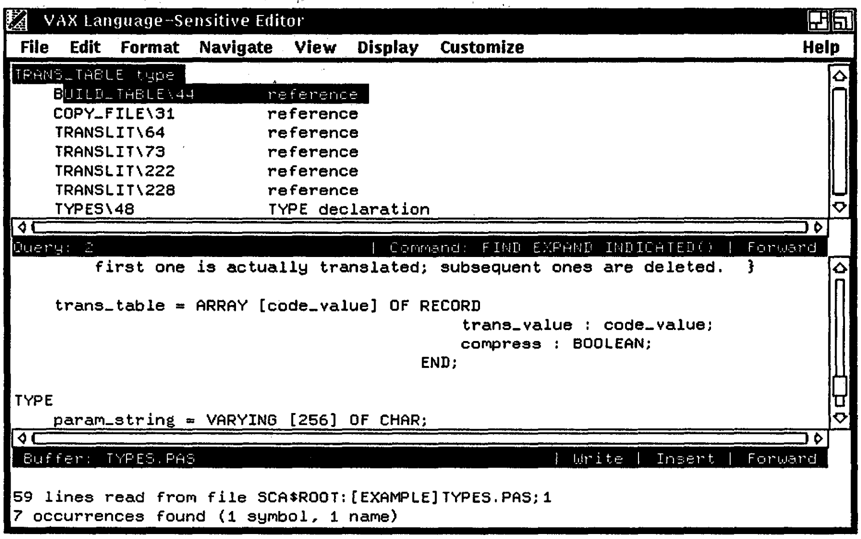
```
BUILDTABLE.PAS file
    BUILD_TABLE\1          PASCAL command reference
BUILD_TABLE procedure
    BUILD_TABLE\41         PROCEDURE declaration
    TRANSLIT\61            FORWARD or EXTERNAL PROCEDURE declaration
    TRANSLIT\171           call reference
BUILD_TABLE module
COMMAND_TABLES variable
Query: 1                              | Command: FIND *TABLE* | Forward
        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                            | Write | Insert | Forward

143 lines read from file SCA$ROOT:[EXAMPLE]BUILDTABLE.PAS;1
59 lines read from file SCA$ROOT:[EXAMPLE]TYPES.PAS;1
```

You can continue investigating parameters while you explore the details of the declaration of TRANS_TABLE. For example, you can find out more about the type named CODE_VALUE by positioning the cursor on that name and pressing CTRL/G.

To see how the definition of TRANS_TABLE is used throughout the program, position the cursor on an occurrence of TRANS_TABLE and enter the FIND EXPAND INDICATED( ) command (GOLD CTRL/F).

Figure 7–6 shows the resulting display.

Figure 7–6: The FIND EXPAND INDICATED Display

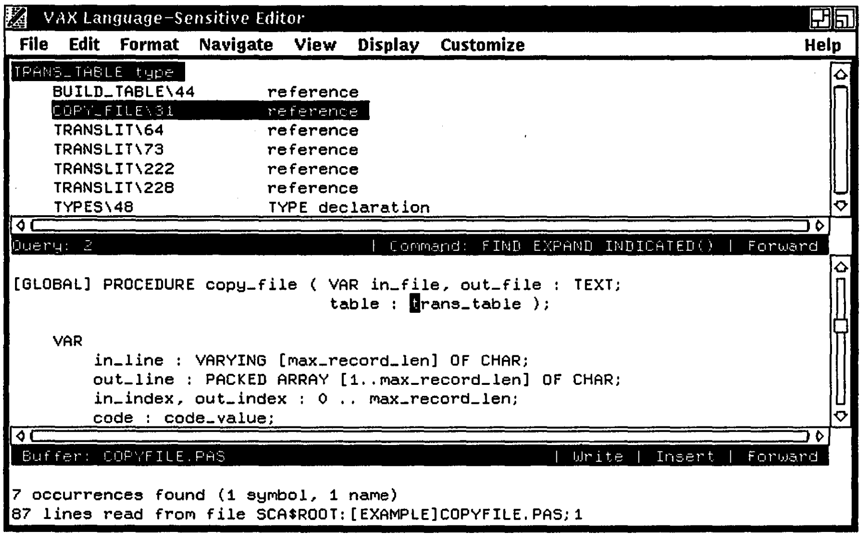

VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
TRANS_TABLE type
    BUILD_TABLE\44          reference
    COPY_FILE\31            reference
    TRANSLIT\64             reference
    TRANSLIT\73             reference
    TRANSLIT\222            reference
    TRANSLIT\228            reference
    TYPES\48                TYPE declaration
Query: 2                             | Command: FIND EXPAND INDICATED() | Forward
        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                                  | Write | Insert | Forward

59 lines read from file SCA$ROOT:[EXAMPLE]TYPES.PAS;1
7 occurrences found (1 symbol, 1 name)
```

The display shows there are seven occurrences of TYPE_TABLE. The first occurrence is the reference with which you began in Figure 7–5. You can move past it by using the NEXT STEP command (CTRL/F) and then use the GOTO SOURCE command to view the reference to TRANS_TABLE that is inside the module COPY_FILE.

Figure 7–7 shows the resulting display.

Figure 7–7: The TRANS_TABLE Source Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

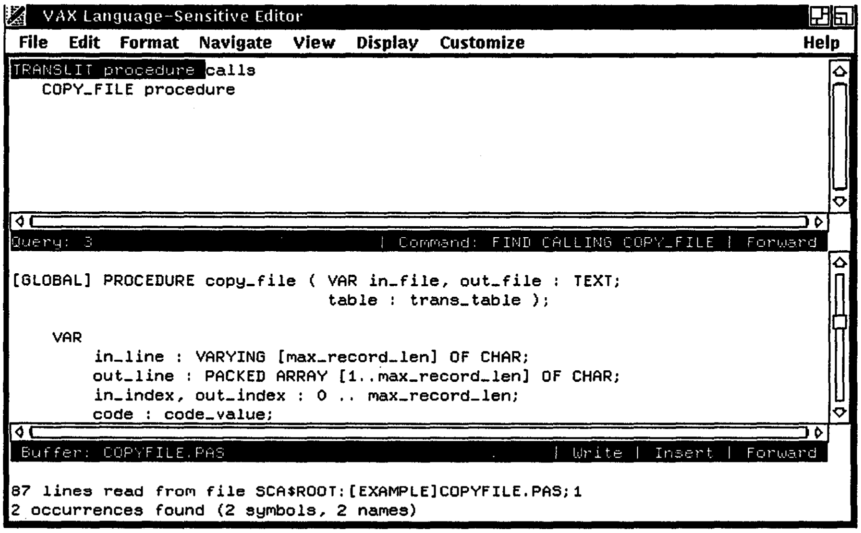
```
TRANS_TABLE type
    BUILD_TABLE\44         reference
    COPY_FILE\31           reference
    TRANSLIT\64            reference
    TRANSLIT\73            reference
    TRANSLIT\222           reference
    TRANSLIT\228           reference
    TYPES\48               TYPE declaration
Query: 2                          | Command: FIND EXPAND INDICATED() | Forward

[GLOBAL] PROCEDURE copy_file ( VAR in_file, out_file : TEXT;
                               table : trans_table );

    VAR
        in_line : VARYING [max_record_len] OF CHAR;
        out_line : PACKED ARRAY [1..max_record_len] OF CHAR;
        in_index, out_index : 0 .. max_record_len;
        code : code_value;
 Buffer: COPYFILE.PAS                              | Write | Insert | Forward

7 occurrences found (1 symbol, 1 name)
87 lines read from file SCA$ROOT:[EXAMPLE]COPYFILE.PAS;1
```

You can use the FIND command to view the relationships between occurrences. For example, you can see which routines call other routines. If you want to determine what calls the procedure COPY_FILE, issue the following command:

```
LSE> FIND calling copy_file
```

Figure 7–8 shows the resulting display.

Figure 7–8: The FIND calling Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
TRANSLIT procedure calls
   COPY_FILE procedure
```

```
Query: 3                    | Command: FIND CALLING COPY_FILE | Forward
```

```
[GLOBAL] PROCEDURE copy_file ( VAR in_file, out_file : TEXT;
                               table : trans_table );

    VAR
        in_line : VARYING [max_record_len] OF CHAR;
        out_line : PACKED ARRAY [1..max_record_len] OF CHAR;
        in_index, out_index : 0 .. max_record_len;
        code : code_value;
```

```
Buffer: COPYFILE.PAS                      | Write | Insert | Forward
```

```
87 lines read from file SCA$ROOT:[EXAMPLE]COPYFILE.PAS;1
2 occurrences found (2 symbols, 2 names)
```

This query display works just like the previous ones. You can use the NEXT STEP and PREVIOUS STEP commands to select entries. You can expand a symbol into its corresponding set of occurrences, and you can go to a source from any of those occurrences.

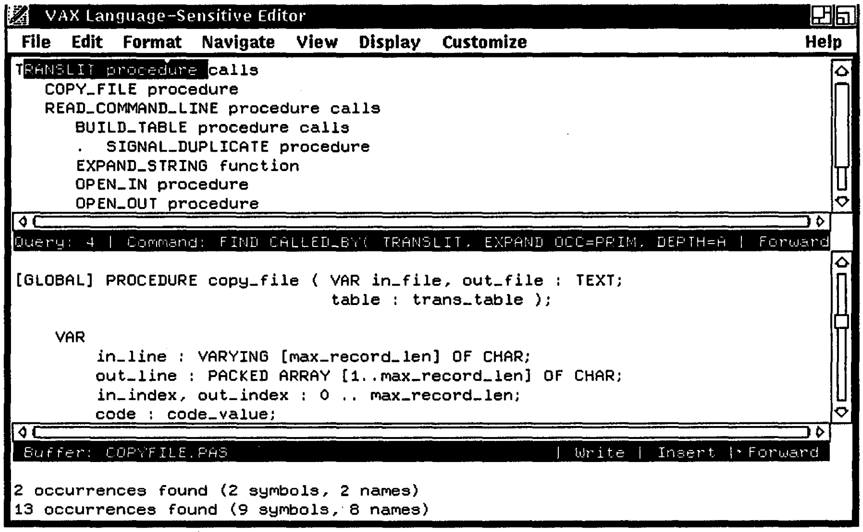
With the FIND command, you can view multiple levels of call relationships. For example, you may want to find all of the routines called by TRANSLIT, and all of the routines called by those routines, and so on, for the entire call tree below TRANSLIT. However, you may want the display to include only those routines whose primary declaration appears in the SCA library (which means, for example, do not include calls to RTL routines).

You initiate the query by entering the following command:

```
LSE>  FIND called_by( translit, expand occ=prim, depth=all )
```

Figure 7–9 shows the resulting display. This is an example of a call tree. Nested routines are indented three spaces from the routine that calls them. Routines preceded by a period (.) indicate that the calling routine calls additional routines. Note also that occurrences are indicated with a fixed level of indentation, four spaces from the left margin of the display.

Figure 7–9: The FIND called_by Display

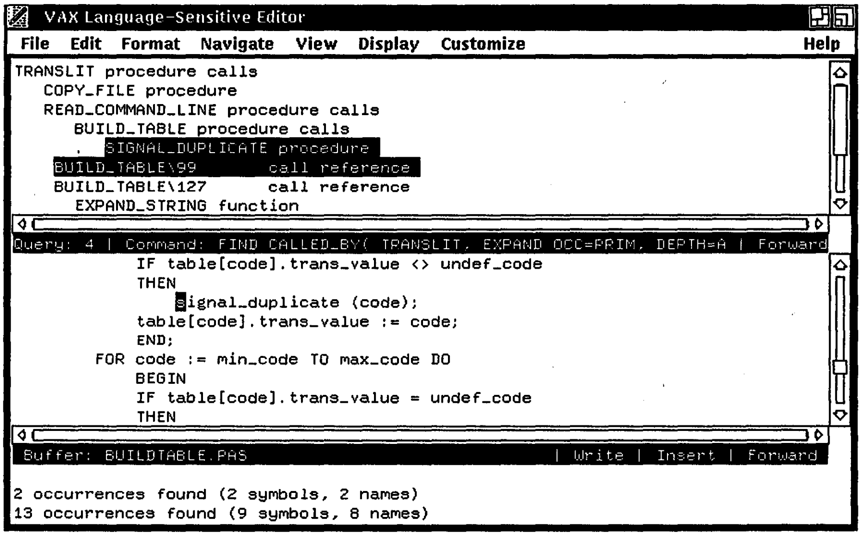

VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
TRANSLIT procedure calls
   COPY_FILE procedure
   READ_COMMAND_LINE procedure calls
      BUILD_TABLE procedure calls
      .  SIGNAL_DUPLICATE procedure
      EXPAND_STRING function
      OPEN_IN procedure
      OPEN_OUT procedure
Query: 4 | Command: FIND CALLED_BY( TRANSLIT, EXPAND OCC=PRIM, DEPTH=A | Forward

[GLOBAL] PROCEDURE copy_file ( VAR in_file, out_file : TEXT;
                              table : trans_table );

    VAR
        in_line : VARYING [max_record_len] OF CHAR;
        out_line : PACKED ARRAY [1..max_record_len] OF CHAR;
        in_index, out_index : 0 .. max_record_len;
        code : code_value;
 Buffer: COPYFILE.PAS                                  | Write | Insert |• Forward

2 occurrences found (2 symbols, 2 names)
13 occurrences found (9 symbols, 8 names)
```

Again, you can select particular symbols, expand them, and go to the source associated with particular occurrences.

Figure 7–10 shows the resulting display when you select and expand the entry for SIGNAL_DUPLICATE.

Figure 7–10: The Expanded SIGNAL_DUPLICATE Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
TRANSLIT procedure calls
   COPY_FILE procedure
   READ_COMMAND_LINE procedure calls
      BUILD_TABLE procedure calls
       .  SIGNAL_DUPLICATE procedure
    BUILD_TABLE\99          call reference
    BUILD_TABLE\127         call reference
      EXPAND_STRING function
Query: 4 | Command: FIND CALLED_BY( TRANSLIT, EXPAND OCC=PRIM, DEPTH=A | Forward
            IF table[code].trans_value <> undef_code
            THEN
                signal_duplicate (code);
            table[code].trans_value := code;
            END;
        FOR code := min_code TO max_code DO
            BEGIN
            IF table[code].trans_value = undef_code
            THEN
 Buffer: BUILDTABLE.PAS                                 | Write | Insert | Forward

2 occurrences found (2 symbols, 2 names)
13 occurrences found (9 symbols, 8 names)
```

The query display shows that SIGNAL_DUPLICATE is called in two different places within the routine BUILD_TABLE. The declaration of SIGNAL_DUPLICATE does not appear in the query result because SIGNAL_DUPLICATE calls only RTL routines. However, this does not pose a problem, because the GOTO DECLARATION/INDICATED command works in a query buffer. You can get to the declaration of SIGNAL_DUPLICATE by using the GOTO DECLARATION/INDICATED command (CTRL/D).

7.1.7 Multiple Queries

SCA provides a multiple query feature that allows you to maintain more than one query session at a time. This feature maximizes the use of SCA by allowing you to perform simultaneous source investigations.

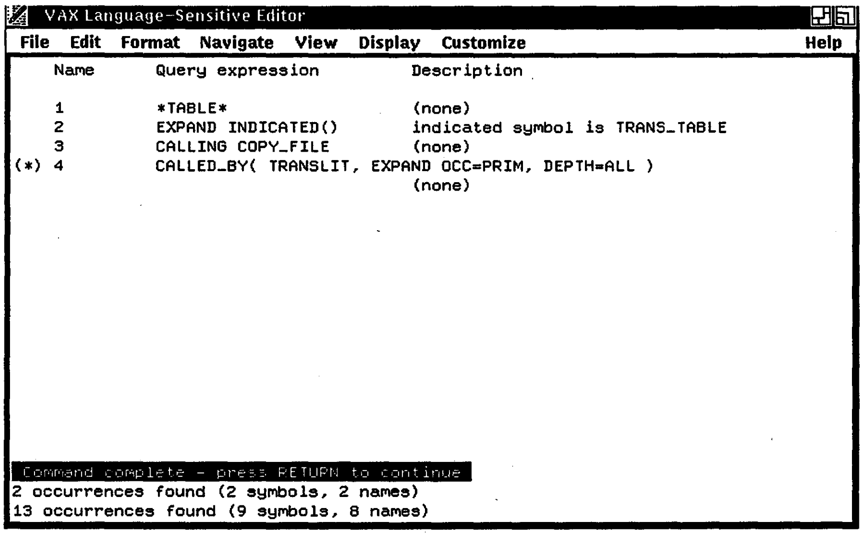
For example, when you issue a query command, a new query session is created. If, during a session, you go to the source of an occurrence and find a symbol that you want to investigate before returning to your last query, you can issue a new query about the symbol. After the inquiries in your new session are completed, you can then go back to your previous session by issuing a PREVIOUS QUERY command.

Current query defines the last query command issued as the target of a GOTO QUERY, NEXT QUERY, or PREVIOUS QUERY command. If no query command has been issued during the current editing session, there is no current query. Using one of the query commands (FIND, INSPECT, GOTO QUERY, NEXT QUERY, PREVIOUS QUERY) reestablishes a query as the current query.

You can also display all the queries you have made during an SCA session by issuing the SHOW QUERY command. The one marked with asterisks (*) indicates the current query.

Figure 7–11 shows the resulting display.

Figure 7–11: The SHOW QUERY Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
    Name        Query expression           Description

    1           *TABLE*                    (none)
    2           EXPAND INDICATED()         indicated symbol is TRANS_TABLE
    3           CALLING COPY_FILE          (none)
(*) 4           CALLED_BY( TRANSLIT, EXPAND OCC=PRIM, DEPTH=ALL )
                                           (none)

Command complete - press RETURN to continue
2 occurrences found (2 symbols, 2 names)
13 occurrences found (9 symbols, 8 names)
```

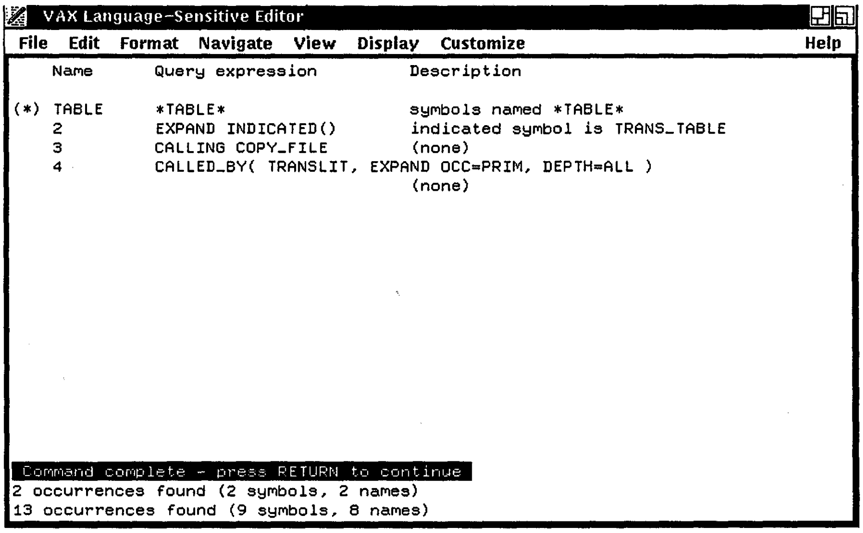
The SHOW QUERY display lists all the queries that you have made during this session of SCA. You can use the GOTO, NEXT, or PREVIOUS QUERY command to return to an existing query. Query names are shown in the first column. You can name a query by using the /NAME qualifier on the FIND command. If you do not specify a query name, SCA generates a unique name for each new query.

By default, each time you use the FIND command a new query is created. However, you can use the /MODIFY qualifier on the FIND command to change some aspect of an existing query. For example, to change the name of the first query to TABLE, and to create a description for that query, you type the following command:

```
LSE> FIND /MODIFY=1 /NAME=table /DESC="symbols named *TABLE*"
```

Figure 7–12 shows the resulting display after typing another SHOW QUERY command.

Figure 7–12: The FIND /MODIFY Display


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
    Name       Query expression          Description

(*) TABLE      *TABLE*                   symbols named *TABLE*
    2          EXPAND INDICATED()        indicated symbol is TRANS_TABLE
    3          CALLING COPY_FILE         (none)
    4          CALLED_BY( TRANSLIT, EXPAND OCC=PRIM, DEPTH=ALL )
                                         (none)

Command complete - press RETURN to continue
2 occurrences found (2 symbols, 2 names)
13 occurrences found (9 symbols, 8 names)
```

7.1.7.1 Moving to a Specified Query

The GOTO QUERY command moves the cursor to the specified query session. The command has the following form:

GOTO QUERY name

You type the command as follows:

```
LSE> GOTO QUERY 1
```

The GOTO QUERY command causes the query number specified by the name parameter to become the current query session and maps the buffer associated with that query session.

7.1.7.2 Moving to the Next Query

The NEXT QUERY command moves the cursor forward through multiple query sessions. You type the command as follows:

```
LSE>  NEXT QUERY
```

The NEXT QUERY command moves forward through the query sessions in their order of creation. The window is remapped to the buffer associated with the next query session. If there is no next query session, the previous query session is used.

7.1.7.3 Moving to the Previous Query

The PREVIOUS QUERY command moves the cursor backward through multiple query sessions. You type the command as follows:

```
LSE>  PREVIOUS QUERY
```

The PREVIOUS QUERY command moves backward through the query sessions in the reverse order of their creation. The window is remapped to the buffer associated with the previous query session. If there is no previous query session, the next query session is used.

7.1.7.4 Terminating a Query

The DELETE QUERY command deletes a query session. The command has the following form:

DELETE QUERY [name]

You type the command as follows:

```
LSE>  DELETE QUERY 1
```

The DELETE QUERY command deletes the specified query session. If no name is specified, the current query session is deleted. If the current query session is deleted, then no current query exists.

7.1.8 Exiting from the SCA Session

To end your SCA session and return to the DCL level, enter the EXIT command or type CTRL/Z.

Chapter 8

Using the SCA Query Language

8.1 Overview

This chapter provides an overview of the SCA Query Language. It contains an overview of the SCA Query Language and its features and demonstrates using the SCA Query Language for simple to advanced operations. See Chapter 9 for more information about the SCA Query Language.

The SCA Query Language is an enhancement to the FIND command. By issuing queries, you can both broaden and refine your use of SCA. With the SCA Query Language, you can make explicit queries of a large system and selectively limit queries to the results of previous query operations. Additional features of the SCA Query Language are also described in this chapter.

8.2 Features of the SCA Query Language

With the SCA Query Language, you can to do the following:

- Analyze source code by using both file and symbol information
- Use names to select symbols
- Use other attributes to select symbols
- Specify precise search parameters
- Use relationship functions to query relationships between symbols

8.3 Basic Concepts

The SCA Query Language is based on the concept of a **query expression**. A query expression is a general algebraic expression in the form of a parameter to the FIND command. It is used to extract specific information from SCA libraries. A query expression can contain subexpressions joined by query operators or functions. A **query operator** is either a logical operator (such as AND or OR) or an SCA-specific relationship function (such as CONTAINING or CALLING).

The SCA Query Language uses two primary elements: **symbols** and **occurrences**. A symbol is an abstract entity in a program. An occurrence is any use of a symbol in source code.

A symbol can be a particular local variable, an RTL routine, a field within a record, or any other clearly distinguishable item in a program. Each symbol has an associated name. Symbols also have attributes called **symbol attributes**. One symbol attribute is class. Symbol classes are, for example, variable, literal, macro, function, or task. Symbols also have **domain attributes**, for example, global or inheritable.

A symbol has associated with it a set of occurrences of that symbol. Also, every occurrence has a corresponding symbol.

An occurrence has attributes called **occurrence attributes**. Occurrence attributes supply additional information about the nature of a symbol. For example, an occurrence can be either a declaration or a reference.

8.4 SCA Query Language Tutorial

With the SCA Query Language, you can perform a wide range of operations, from simple to complex queries. This section contains a set of these operations, based on a Pascal module. It begins with simple queries and gradually introduces more sophisticated ways to use the query language. The FIND commands demonstrated in this section are entered relative to an SCA library describing the following Pascal module.

```
[INHERIT ('SYS$LIBRARY:STARLET', 'TYPES')]
MODULE build_table;

PROCEDURE lib$signal ( %IMMED args : [LIST] UNSIGNED );  EXTERNAL;

VAR
    trnlit__duporig,
    trnlit__reptoolon : [EXTERNAL] INTEGER;

[GLOBAL] PROCEDURE build_table ( orig_vector : code_vector;
                                 orig_len : code_vector_length;
                                 VAR table : trans_table );
    VAR
        code : code_value;
        i : 1 .. code_vector_limit;

    PROCEDURE signal_duplicate ( code : code_value );
        VAR
            text : VARYING [2] OF CHAR;
        BEGIN
        IF code < 32
        THEN
            text := '^' + CHR (code + 64)
        ELSE
            text := CHR (code);
        lib$signal (IADDRESS (trnlit__duporig), 1, %STDESCR (text));
        END;

    BEGIN
    FOR i := 1 TO orig_len DO
        BEGIN
        code := orig_vector[i];
        IF table[code].trans_value <> undef_code
        THEN
            signal_duplicate (code);
        END;
    END;
END.
```

The examples shown in this chapter are based on using SCA in standalone mode. When you use LSE, some results appear differently.

8.4.1 Simple Queries

The simplest query specifies symbols based merely on the name of the symbol. For example, to get information about all of the occurrences of symbols named CODE, type the following command:

```
FIND code
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
    BUILD_TABLE\32       write reference
    BUILD_TABLE\33       read reference
    BUILD_TABLE\35       read reference
CODE argument
    BUILD_TABLE\17       formal parameter declaration
    BUILD_TABLE\21       read reference
    BUILD_TABLE\23       read reference
    BUILD_TABLE\25       read reference
%SCA-S-OCCURS, 8 occurrences found (2 symbols, 1 name)
```

This display shows that SCA found two symbols named CODE. The first symbol is a variable and has four occurrences. The first occurrence is the declaration of the variable and indicates that it is the CODE symbol defined on line 14 of the sample program. The next three occurrences are references to this variable.

The second symbol is the argument defined on line 17 of the sample program. This symbol also has four occurrences.

You can restrict the query in several ways. For example, to get only declarations of symbols named CODE, type the following command:

```
FIND code AND occurrence=declaration
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
CODE argument
    BUILD_TABLE\17       formal parameter declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 1 name)
```

To get only write references of symbols named CODE, type the following command:

```
FIND code AND occurrence=write
```

The result follows.

```
CODE variable
    BUILD_TABLE\32       write reference
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

The preceding two examples show query selection based on occurrence attributes.

With the SCA Query Language, you can use DCL-like wildcards. For example, to display information about procedures and functions without regard to their names, type the following command:

```
FIND * AND symbol=routine
```

Since the name attribute is a wildcard, it can be left out completely. The following command is equivalent to the preceding one.

```
FIND symbol=routine
```

The result follows.

```
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
CHR function
    BUILD_TABLE\23       call reference
    BUILD_TABLE\25       call reference
IADDRESS function
    BUILD_TABLE\26       call reference
LIB$SIGNAL procedure
    BUILD_TABLE\4        FORWARD or EXTERNAL PROCEDURE declaration
    BUILD_TABLE\26       call reference
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17       PROCEDURE declaration
    BUILD_TABLE\35       call reference
%SCA-S-OCCURS, 8 occurrences found (5 symbols, 5 names)
```

The preceding example shows query selection based on symbol attributes. You can combine both symbol and occurrence attributes in one query. For example, to find the primary declarations of routines, type the following command:

```
FIND symbol=routine AND occurrence=primary
```

The result follows.

```
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17       PROCEDURE declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

The preceding examples also show the use of logical operators in forming more complex queries based on subqueries. For more information on logical operators see Section 8.4.3.

You can further restrict the previous query by adding an expression that distinguishes module-specific symbols from those that (potentially) span multiple modules. To display the primary declaration of inter-module routines (only those that have the potential of spanning multiple modules), type the following command:

```
FIND symbol=routine AND occurrence=primary AND domain=multi_module
```

The result follows.

```
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

You can abbreviate attribute-selection expressions like you do with DCL. For example, you can abbreviate the preceding command as follows:

```
FIND symb=rout AND occ=prim AND doma=mult
```

8.4.2 Using the Expand Function to Find Related Occurrences

The process of expanding a set of occurrences to include all of the occurrences of the corresponding set of symbols is called **expansion**. This is not to be confused with the LSE EXPAND command.

To perform an expansion operation, SCA first finds all the occurrences that match the subexpression of the expand function. Once that set of occurrences is found, SCA finds all the symbols that correspond to that set of occurrences. Finally, the result of the query is *all* occurrences of those symbols.

With SCA, you can specify that you want to see all the information available for symbols that have certain types of occurrences. For example, to display all of the occurrences of routines that have primary declarations in the SCA library being queried, type the following command:

```
FIND EXPAND (occurrence=primary AND symbol=routine)
```

The result follows.

```
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17       PROCEDURE declaration
    BUILD_TABLE\35       call reference
%SCA-S-OCCURS, 3 occurrences found (2 symbols, 2 names)
```

The parenthetical expression is the same query used earlier. The addition of the expansion operation causes the result to contain all occurrences of the symbols found, not just those specified by the subexpression.

You can follow an expansion with more restrictions. For example, to display the call references of routines that have primary declarations in the SCA library being queried, type the following command:

```
FIND EXPAND (occurrence=primary AND symbol=routine) AND occ=call
```

This is an example of a nested query expression. The inner query expression, EXPAND (occurrence=primary AND symbol=routine), is evaluated first, resulting in a set of all the occurrences of routines for which there are primary declarations. That set of occurrences is the input to the outer query expression, which has the following form:

```
query-expression AND occ=call
```

The outer query expression removes all occurrences except those that are call-references.

The result follows.

```
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\35       call reference
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

In another example of expansion, to display declarations of symbols that have write references, type the following command:

```
FIND EXPAND (occ=write) AND occ=decl
```

To evaluate this query, SCA begins by finding the set of write reference occurrences. Next, SCA expands this set to include all occurrences of these symbols. Finally, the new set is intersected with the set containing all primary declaration occurrences.

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
I variable
    BUILD_TABLE\15       VAR (variable) declaration
TEXT variable
    BUILD_TABLE\19       VAR (variable) declaration
%SCA-S-OCCURS, 3 occurrences found (3 symbols, 3 names)
```

8.4.3 Using Logical Operators to Select Information

The SCA Query Language can apply logical operations to the results of other query expressions. The logical operations that are supported are union, intersection, negation, and exclusive-or. Some previous examples showed logical operations.

A **union expression** merges two sets, resulting in a set containing all occurrences that exist in either set.

An **intersection expression** identifies occurrences that exist in two different sets, resulting in a set containing each occurrence that exists in both sets; occurrences that appear in only one of the sets are not included.

A **negation expression** identifies occurrences that are not in the set.

An **exclusive-or expression** selects the unique occurrences in two different sets, resulting in a set containing all occurrences that exist in only one of the sets.

For example, if you want to find all the symbols that have TABLE in their name, but you want to exclude those symbols whose name is simply TABLE, type the following command:

```
FIND *table* AND NOT table
```

The result follows.

```
BUILDTABLE.PAS file
    BUILD_TABLE\1        PASCAL command reference
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
BUILD_TABLE module
    BUILD_TABLE\2        MODULE declaration
TRANS_TABLE type
    BUILD_TABLE\12       reference
%SCA-S-OCCURS, 4 occurrences found (4 symbols, 3 names)
```

If you want to find all symbols that begin with CODE, but not those symbols that have read or write references, type the following command:

```
FIND code* AND NOT EXPAND(occ=read OR occ=write)
```

The result follows.

```
CODE_VALUE type
    BUILD_TABLE\14       reference
    BUILD_TABLE\17       reference
CODE_VECTOR type
    BUILD_TABLE\10       reference
CODE_VECTOR_LENGTH type
    BUILD_TABLE\11       reference
CODE_VECTOR_LIMIT constant
    BUILD_TABLE\15       reference
%SCA-S-OCCURS, 5 occurrences found (4 symbols, 4 names)
```

The previous query could also be written as follows:

```
FIND code* AND NOT EXPAND occ=(read,write)
```

To display the declarations of the symbols that are both read and written, type the following command:

```
FIND (EXPAND(occ=read) AND EXPAND(occ=write)) AND occ=decl
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
I variable
    BUILD_TABLE\15       VAR (variable) declaration
TEXT variable
    BUILD_TABLE\19       VAR (variable) declaration
%SCA-S-OCCURS, 3 occurrences found (3 symbols, 3 names)
```

Alternately, to display the declarations of the symbols that are either read or written, but not both, type the following command:

```
FIND (EXPAND(occ=read) XOR EXPAND(occ=write)) AND occ=decl
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
CODE argument
    BUILD_TABLE\17       formal parameter declaration
I variable
    BUILD_TABLE\15       VAR (variable) declaration
ORIG_LEN argument
    BUILD_TABLE\11       formal parameter declaration
ORIG_VECTOR argument
    BUILD_TABLE\10       formal parameter declaration
TABLE argument
    BUILD_TABLE\12       formal parameter declaration
TEXT variable
    BUILD_TABLE\19       VAR (variable) declaration
%SCA-S-OCCURS, 7 occurrences found (7 symbols, 6 names)
```

To find all the symbols that are declared but never referenced, type the following command:

```
FIND NOT EXPAND occ=ref
```

This finds all occurrences of the symbols that are never referenced. The result follows.

```
scalar type
    BUILD_TABLE\15       scalar type declaration
array
    BUILD_TABLE\19       ARRAY declaration
array index
    BUILD_TABLE\19       array index declaration
array component
    BUILD_TABLE\19       array component declaration
ARGS argument
    BUILD_TABLE\4        formal parameter declaration
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
BUILD_TABLE module
    BUILD_TABLE\2        MODULE declaration
REPL_LEN argument
    BUILD_TABLE\11       formal parameter declaration
REPL_VECTOR argument
    BUILD_TABLE\10       formal parameter declaration
TRNLIT__REPTOOLON variable
    BUILD_TABLE\8        EXTERNAL or predefined variable declaration
%SCA-S-OCCURS, 10 occurrences found (10 symbols, 6 names)
```

If you enter this command and realize that your chosen language(s) and code practices tend to give some unimportant cases of declared but not referenced symbols (like modules and formal parameters), you may want to further qualify your request by typing the following command:

```
FIND (NOT EXPAND occ=ref) AND NOT symbol=(module,argument)
```

The result is a display that removes modules and parameters (called ARGUMENTS) from the set of occurrences of the symbols that are never referenced. The result follows.

```
scalar type
    BUILD_TABLE\15       scalar type declaration
array
    BUILD_TABLE\19       ARRAY declaration
array index
    BUILD_TABLE\19       array index declaration
array component
    BUILD_TABLE\19       array component declaration
BUILD_TABLE procedure
    BUILD_TABLE\10       PROCEDURE declaration
TRNLIT__REPTOOLON variable
    BUILD_TABLE\8        EXTERNAL or predefined variable declaration
%SCA-S-OCCURS, 6 occurrences found (6 symbols, 3 names)
```

8.4.4 The Current Query

SCA maintains a current query. A current query is the result of the previously entered query or the one to which you are set using the NEXT QUERY, PREVIOUS QUERY, and GOTO QUERY commands. The current query is specified the same way as any other query. The name of the current query is SCA$CURRENT_QUERY.

The following is an example of a command sequence using the current query:

```
FIND *table* AND NOT table
FIND ( NOT @sca$current_query ) AND symbol=routine AND occurrence=decl
```

The @ function defaults to the current query. Consequently you can also write the previous commands as follows:

```
FIND *table* AND NOT table
FIND ( NOT @() ) AND symbol=routine AND occ=decl
```

The @ function, written without parameters, takes the following form:

```
@()
```

You can assign a name to a query. A query remains available for use throughout a given invocation of SCA, unless it is explicitly deleted using the DELETE QUERY command. If you do not name a query, then SCA automatically assigns a name to it. For example, the preceding command sequence could be rewritten as follows:

```
FIND/NAME=table *table* AND NOT table
FIND/NAME=routine_decls symbol=routine AND occ=decl
FIND (NOT @table) AND @routine_decls
```

You can use the SHOW QUERY command to display all currently available queries. For example, if you follow the preceding set of FIND commands with a SHOW QUERY command, the result follows.

```
    Name      Query expression        Description

    TABLE     *TABLE* AND NOT TABLE    (none)
    ROUTINE_DECLS
              SYMBOL=ROUTINE AND OCC=DECL
                                       (none)
(*) 1         (NOT @TABLE) AND @ROUTINE_DECLS
                                       (none)
```

8.4.5 Structured Relationship Expressions

With the SCA Query Language, you can select occurrences based on their relationship to other occurrences. For example, type the following command:

```
FIND CALLED_BY signal_duplidate
```

The result follows.

```
%SCA-W-NOOCCUR, no symbol occurrence matches your selection criteria
```

You can interpret the previous command as "Find what is called by SIGNAL_DUPLICATE."

You can ask the reverse question, "Find what is calling SIGNAL_DUPLICATE," by typing the following command:

```
   FIND CALLING signal_duplicate
```

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You can use the depth parameter to request that more than one level of structure be displayed. The depth parameter sets the number of levels of structure that you want SCA to trace. By default, one level of structure is traced.

To request a depth level of 2, type the following command:

```
FIND CALLED_BY( build_table, depth=2 )
```

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure calls
      CHR function
      IADDRESS function
      LIB$SIGNAL procedure
%SCA-S-OCCURS, 7 occurrences found (5 symbols, 5 names)
```

You can us the DEPTH=ALL option to specify that all levels of call relationship are to be traced. To trace all call relationships from BUILD_TABLE down, type the following command:

```
FIND CALLED_BY( build_table, depth=all )
```

Since the example program is so simple, this command gives the same result as the previous command.

If you want to know if one routine can be called from within another directly or indirectly (for example, to display all of the paths of the call-graph that lead from BUILD_TABLE to LIB$SIGNAL), type the following command:

```
FIND CALLED_BY( build_table, lib$signal, depth=all )
```

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure calls
      LIB$SIGNAL procedure
%SCA-S-OCCURS, 4 occurrences found (3 symbols, 3 names)
```

You can interpret the preceding command as follows: "Find what is called by BUILD_TABLE, tracing any number of levels of structure, but include only the paths that lead to LIB$SIGNAL."

One common problem with call-tree displays is that they often contain a large percentage of lines describing routines that are not a part of the application under development. These are utility routines that are either RTL routines, system services, RMS routines, or some other set of routines that are viewed by the developer as being a part of the base system. Having these utility routines in a call-tree is often considered a nuisance, for they make it difficult to see the most important structure of the call-tree.

The primary declarations of such utility routines are not described in the SCA library of an application. Hence, you can modify the previous FIND command to remove the unwanted routines, as follows:

```
FIND CALLED_BY( build_table, EXPAND( occ=primary ), depth=all )
```

You can interpret the preceding command as follows: "Find what is called by BUILD_TABLE, tracing any number of levels of structure, but include only the paths that lead to routines that have primary declarations." A more succinct interpretation is "Trace the calls from BUILD_TABLE through the routines that have primary declarations."

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You have even more control over the tracing of relationships by using the TRACE parameter. The TRACE parameter specifies a query expression. As the CALLED_BY function iteratively traces the calls, it continues tracing the called-by relationship only through the occurrences that match the trace-expression, specified as the value of the TRACE parameter.

For example, to display all of the paths of the call-graph from BUILD_TABLE down, except the call-relationships traced through the routine SIGNAL_DUPLICATE, type the following command:

```
FIND CALLED_BY( build_table, depth=all, trace=(NOT signal_duplicate) )
```

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

Note that SIGNAL_DUPLICATE is included in the result, but the tracing of the called-by relationship does not continue through SIGNAL_DUPLICATE.

The TRACE parameter does not affect the first iteration. That first iteration is controlled by the BEGIN parameter.

Note that you can terminate tracing the called-by relationship through SIGNAL_DUPLICATE and you can exclude calls to SIGNAL_DUPLICATE, by typing the following command:

```
FIND CALLED_BY( build_table, -
                NOT signal_duplicate, -
                depth=all, -
                trace=(NOT signal_duplicate) )
```

This command displays all of the paths of the call-graph from BUILD_TABLE down, except that it will not match calls to the routine SIGNAL_DUPLICATE. As a result, SIGNAL_DUPLICATE is not included in the display. The result follows.

```
%SCA-W-NOOCCUR, no symbol occurrence matches your selection criteria
```

8.4.6 Nonstructured Relationship Expressions

There is a simple but important difference between the commands about to be described and those described in the previous section. Both sets of commands use information about the relationships between occurrences. However, the commands described in the previous sections use that relationship information to create a collection of occurrences and the relationships between those occurrences. The commands described in the following sections discard the relationship information from the query result. Consequently, these relationship query expressions are considered nonstructured because the result is solely a flat set of occurrences.

Nonstructured relationship expressions are realized by using the result-parameter of the relationship functions. For example, you saw in the previous section the results of the following command:

```
FIND CALLED_BY signal_duplicate
```

The result follows.

```
BUILD_TABLE procedure calls
   SIGNAL_DUPLICATE procedure calls
      CHR function
      IADDRESS function
      LIB$SIGNAL procedure
%SCA-S-OCCURS, 7 occurrences found (5 symbols, 5 names)
```

A nonstructured version of the same command follows:

```
FIND CALLED_BY( signal_duplicate, result=nostructure )
```

The result follows.

```
CHR function
    BUILD_TABLE\23       call reference
    BUILD_TABLE\25       call reference
IADDRESS function
    BUILD_TABLE\26       call reference
LIB$SIGNAL procedure
    BUILD_TABLE\26       call reference
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17       PROCEDURE declaration
%SCA-S-OCCURS, 5 occurrences found (4 symbols, 4 names)
```

You can also use the result-parameter to restrict the result to just the beginning or just the end of the relationship expression. For example, if you want to identify the routines that call RTL routines, type the following command:

```
FIND CALLING( lib$*, result=begin )
```

The result follows.

```
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17       PROCEDURE declaration
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

You can interpret this command as follows: "Find what is calling LIB$*, and report only the caller, SIGNAL_DUPLICATE, not the callee, LIB$*."

If you want to find all the occurrences of routines that call RTL routines, expand the result of the previous query expression by typing the following command:

```
FIND EXPAND CALLING( lib$*, result=begin )
```

The result follows.

```
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17      PROCEDURE declaration
    BUILD_TABLE\35      call reference
%SCA-S-OCCURS, 2 occurrences found (1 symbol, 1 name)
```

You can also find routines that call LIB$* indirectly by typing the following command:

```
FIND CALLING( lib$*, depth=all, result=begin )
```

This command asks for the routines that call an RTL routine either directly or indirectly. The result follows.

```
BUILD_TABLE procedure
    BUILD_TABLE\10      PROCEDURE declaration
SIGNAL_DUPLICATE procedure
    BUILD_TABLE\17      PROCEDURE declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

8.4.7 Other Relationships

TYPING and CONTAINING also allow you to query a software system. For example, you can determine the type of the routine parameter TABLE by typing the following command:

```
FIND TYPING table
```

The result follows.

```
TABLE argument is typed by
   TRANS_TABLE type
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You can trace these type relationships through multiple levels. Suppose TRANS_TABLE is defined as follows:

```
    TYPE
        trans_table = ARRAY [1..10] OF RECORD
                                       trans_value : INTEGER;
                                       compress : BOOLEAN;
                                       END;
```

You can trace multiple levels by typing the following command:

```
FIND TYPING( table, depth=all )
```

The result follows.

```
TABLE argument is typed by
   TRANS_TABLE type is typed by
      array is typed by
         array component is typed by
         .  record is typed by
         .     COMPRESS component is typed by
         .     .  BOOLEAN scalar type
         .     TRANS_VALUE component is typed by
         .        CODE_VALUE type is typed by
         .           scalar type is typed by
         .              INTEGER scalar type
         CODE_VALUE type  (See above)
%SCA-S-OCCURS, 14 occurrences found (11 symbols, 8 names)
```

You can also find all variables of type INTEGER by typing the following command:

```
FIND TYPED_BY( integer, symbol=variable, result=begin )
```

The result follows.

```
TRNLIT__DUPORIG variable
    BUILD_TABLE\7       EXTERNAL or predefined variable declaration
TRNLIT__REPTOOLON variable
    BUILD_TABLE\8       EXTERNAL or predefined variable declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You can interpret the preceding command as follows: "Find the symbols that are typed-by INTEGER and that match the query expression, SYMBOL=VARIABLE, and return just the beginning of the typed-by relationship."

There is a general "containment" relationship. You can use it to get a "declaration-tree." To show the (primary) declaration structure of the module BUILD_TABLE, type the following command:

```
FIND CONTAINED_BY( build_table, occurrence=primary, depth=all )
```

The first parameter says "begin at BUILD_TABLE." The second parameter says "End with primary declarations." The third parameter says "Repeat any number of levels." The result follows.

```
BUILD_TABLE module contains
   BUILD_TABLE procedure contains
   .  CODE variable
   .  I variable contains
   .  .  scalar type
   .  ORIG_LEN argument
   .  ORIG_VECTOR argument
   .  REPL_LEN argument
   .  REPL_VECTOR argument
   .  SIGNAL_DUPLICATE procedure contains
   .  .  CODE argument
   .  .  TEXT variable contains
   .  .     array contains
   .  .        array index contains
   .  .           array component
   .  TABLE argument
   LIB$SIGNAL procedure contains
      ARGS argument
%SCA-S-OCCURS, 18 occurrences found (18 symbols, 13 names)
```

You can use the containment relationship to specify more precisely which symbol you want. To request all of the occurrences of symbols named CODE that are directly contained by the routine BUILD_TABLE, type the following command:

```
FIND CONTAINED_BY( build_table, code, result=begin )
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
    BUILD_TABLE\32       write reference
    BUILD_TABLE\33       read reference
    BUILD_TABLE\35       read reference
%SCA-S-OCCURS, 4 occurrences found (1 symbol, 1 name)
```

Alternately, to request all of the occurrences of symbols named CODE that are directly or indirectly contained by the routine BUILD_TABLE, type the following command:

```
FIND CONTAINED_BY( build_table, code, depth=all, result=begin )
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
    BUILD_TABLE\32       write reference
    BUILD_TABLE\33       read reference
    BUILD_TABLE\35       read reference
CODE argument
    BUILD_TABLE\17       formal parameter declaration
    BUILD_TABLE\21       read reference
    BUILD_TABLE\23       read reference
    BUILD_TABLE\25       read reference
%SCA-S-OCCURS, 8 occurrences found (2 symbols, 1 name)
```

.4.8 The IN Function

The CONTAINED_BY function is so general that even the most common queries involve the specification of several parameters. Therefore, the IN function has been defined as a special case of the CONTAINED_BY function.

The IN function returns all of the specified occurrences that are contained directly or indirectly (DEPTH=ALL) by a specified (set of) occurrences. See Chapter 9 for more information.

As an example, the FIND CONTAINED_BY command in the previous section could be more simply written as follows:

```
FIND IN( build_table, code )
```

You can interpret this command as "Find all occurrences in BUILD_TABLE named CODE."

To find all occurrences in BUILD_TABLE, including those nested within declarations of BUILD_TABLE, you omit the second parameter, as shown in the following command:

```
FIND IN build_table
```

As another example, imagine that you are working on a compiler project and that you have defined a query named PARSER to contain the list of all of the modules that make up the parser. To find all of the occurrences of symbols named LIB$GET_VM contained directly or indirectly within the parser modules, type the following command:

```
FIND IN( @parser, lib$get_vm )
```

.4.9 Pathnames

The SCA query language accepts pathname notation. For example, to find only symbols named CODE that are declared directly within BUILD_TABLE, type the following command:

```
FIND build_table\code
```

The result follows.

```
CODE variable
    BUILD_TABLE\14       VAR (variable) declaration
    BUILD_TABLE\32       write reference
    BUILD_TABLE\33       read reference
    BUILD_TABLE\35       read reference
%SCA-S-OCCURS, 4 occurrences found (1 symbol, 1 name)
```

Alternately, you can type the following command:

```
FIND signal_duplicate\code
```

The result follows.

```
CODE argument
    BUILD_TABLE\17      formal parameter declaration
    BUILD_TABLE\21      read reference
    BUILD_TABLE\23      read reference
    BUILD_TABLE\25      read reference
%SCA-S-OCCURS, 4 occurrences found (1 symbol, 1 name)
```

You can build up pathnames repeatedly to increase precision. For example, the previous display could have been produced by typing the following command:

```
FIND build_table\signal_duplicate\code
```

You can use wildcards within pathnames. For example, the previous display could also have been produced by typing the following command:

```
FIND build_table\*\code
```

You can interpret this command as "Find symbols named CODE that are declared within any primary declaration that is contained within the primary declaration of BUILD_TABLE."

You can specify that any number of containment levels are acceptable. This is done by leaving out a pathname, as in the following command:

```
FIND build_table\\code
```

You can interpret this command as "Find symbols named CODE that are declared directly or indirectly by the primary declaration of BUILD_TABLE." The result follows.

```
CODE variable
    BUILD_TABLE\14      VAR (variable) declaration
    BUILD_TABLE\32      write reference
    BUILD_TABLE\33      read reference
    BUILD_TABLE\35      read reference
CODE argument
    BUILD_TABLE\17      formal parameter declaration
    BUILD_TABLE\21      read reference
    BUILD_TABLE\23      read reference
    BUILD_TABLE\25      read reference
%SCA-S-OCCURS, 8 occurrences found (2 symbols, 1 name)
```

You can include a general query expression as a pathname by typing the following command:

```
FIND (mod1 OR mod2)\\code
```

You can interpret this command as "Find symbols named CODE that are declared directly or indirectly by the primary declaration of either MOD1 or MOD2."

Similarly, if you are working on a compiler project, and you have defined a query named PARSER to contain the list of all of the modules that make up the parser, then to find all of the symbols named CODE that are declared in the parser, type the following command:

```
FIND @parser\\code
```

8.4.10 Combined Relationship Examples

You can combine more than one relationship function into one query. If you want to know whether or not it is possible for a call to the routine BUILD_TABLE to modify a global variable, you need to consider not only BUILD_TABLE itself, but also the whole call-tree from BUILD_TABLE down. You can find out by typing the following command:

```
FIND IN( CALLED_BY( build_table, depth=all ), -
         sym=var AND occ=write AND domain=multi )
```

The result follows.

```
%SCA-W-NOOCCUR, no symbol occurrence matches your selection criteria
```

In a still more complicated query, you may want to find all of the occurrences of symbols of type TRANS_TABLE that are contained within the call-tree from BUILD_TABLE down. To make this query you type the following command:

```
FIND IN( CALLED_BY( build_table, depth=all ), -
         EXPAND TYPED_BY( trans_table, result=begin ) )
```

The result follows.

```
TABLE argument
    BUILD_TABLE\12       formal parameter declaration
    BUILD_TABLE\33       read reference
%SCA-S-OCCURS, 2 occurrences found (1 symbol, 1 name)
```

You can use the TRACE parameter to restrict a query to a particular subset of a program.

If you were a developer working on a LOAD command, and the module LOAD contained one entry point, LOAD_FILE, you could show all of the call relationships within the module LOAD beginning with the routine LOAD_FILE by typing the following command:

```
FIND CALLED_BY( load_file, depth=all, trace=IN(load) )
```

This command shows all calls that occur within module LOAD, but traces through only the routines whose primary declaration is within LOAD. For example, if LOAD_FILE calls to a routine READ_EVENT whose primary declaration is outside of LOAD, then the call to READ_EVENT shows up in the display, but no calls within READ_EVENT are included. This is because tracing is turned off outside of the module LOAD.

Alternately, to trace call-relationships from LOAD_FILE down, but display calls only to routines whose primary declaration occurs within LOAD, you type the following command:

```
FIND CALLED_BY( load_file, IN(load), depth=all, trace=IN(load) )
```

You can interpret this command (via the TRACE parameter) as "Only continue tracing the called-by relationship through routines whose primary declaration occurs within LOAD;" and (via the BEGIN parameter) "Only include the paths that end with routines whose primary declaration occurs within LOAD." Consequently, this command does not include calls to READ_EVENT.

You can trace all call-relationships from LOAD_FILE down, subject to only one limitation: each path must end with a routine whose primary declaration occurs within LOAD. Since the TRACE expression has been defaulted (to *), this command traces the called-by expression through any routine as long as the path eventually leads back to a routine declared in LOAD. You can perform this operation by typing the following command:

```
FIND CALLED_BY( load_file, IN(load), depth=all )
```

Chapter 9

Evaluating SCA Query Expresssions

This chapter describes the rules governing the use of the SCA Query Language. The following tables provide an overview of the components of the SCA Query Language.

Table 9–1: Attribute Selection Expressions

Attribute	Syntax	Example/Usage
name	name	foo
symbol class	**symbol=**symbol_class	symbol=argument
symbol domain	**domain=**symbol_domain	domain=global
occurrence class	**occurrence=**occ_class	occur=primary
file specification	**file=**file_spec	file="foo.b32"

See Section 9.5 for a list of attribute selection values and their meanings.

Table 9–2: Binary Operators

Type	Syntax	Example/Usage
pathname	exp1\exp2	subrx\y
	exp1\\exp2	routa\\y
intersection	exp1 **AND** exp2	a AND occ=decl
union	exp1 **OR** exp2	symb=arg OR symb=var
exclusive or	exp1 **XOR** exp2	occ=read XOR occ=write

Table 9–3: Nonrelationship Function Expressions

Function	Syntax	Example/Usage
negation	**NOT** query expression	l* AND NOT lib*
expansion	**EXPAND** query expression	EXPAND (occ=primary)
indicated	**INDICATED( )**	EXPAND INDICATED( )
query usage	@query name	@any_query and domain=module

Relationship function expressions have the following general syntax, which is described in more detail in Table 9–4 and Table 9–5:

```
        rel_function_name(end=query_expression,
                          begin=query_expression,
                          depth=n,
                          result=result_keyword,
                          trace=query_expression)
```

Table 9–4: Function Names

Function Name/Example	Description
CALLING x	displays who is calling X
CALLED_BY (a,b,depth=all)	displays the call tree from A to B
TYPING (y,depth=all)	displays the type information of Y
TYPED_BY real	displays all symbols of type REAL
CONTAINED (x,symbol=rout)	displays all routines in X
CONTAINING (domain=global, symbol=module, result=begin, depth=all)	displays modules that contain globally defined symbols

Table 9–5: Function Parameters

Parameter	Type	Default
END	query expression	*
BEGIN	query expression	*
DEPTH	integer	1
RESULT	keyword value	STRUCTURE
TRACE	query expression	*

See Section 9.7.6.2 for a list of RESULT keyword values and their meanings.

9.1 Query Expression Syntax

This section defines the syntax of a query expression. The following example is a high-level description of the syntax. It defines, for example, the form of a function call, but it does not describe which functions are available. The low-level details are described in later sections.

```
query-expression ::= attribute-selection-expression |
                     binary-op-expression |
                     function-call-expression |
                     (query-expression )

attribute-selection-expression ::= actual-parameter

binary-op-expression ::= query-expression binary-operator query-expression

binary-operator ::= AND | OR | XOR | \ | \\

function-call-expression ::= function-name actual-parameter |
                             function-name ([actual-parameter],... )

function-name ::= nonwildcard-string

actual-parameter ::= named-actual-parameter | positional_actual_parameter

named-actual-parameter ::= formal-parameter-name = actual-parameter-value

positional_actual_parameter ::= actual-parameter-value

formal-parameter-name ::= nonwildcard-string

actual-parameter-value ::= query-expression | name-expression |

                           keyword-list | range-list | number

keyword-list ::= keyword | (keyword,... )

keyword ::= nonwildcard-string

number ::= digit... | ALL

range-list ::= range | ( range,... )

range ::= number | number:number

name-expression ::= simple-string | "complex-string"

nonwildcard-string ::= {letter | digit | graphic-character}...

simple-string ::= {letter | digit | graphic-character |
                   wildcard-character | escape-character}...

complex-string ::= any-character...

letter ::= any-alphabetic-character

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

graphic-character ::= - | _ | $

wildcard-character  ::= * | %

escape-character ::= &
```

9.2 Operator Precedence and Associativity

Table 9–6 is a syntax diagram that gives the forms of query expressions. The forms are grouped into priority levels, and an associativity is given for each priority level. In the following table,

```
    exp1 = query-expression
    exp2 = query-expression
```

Table 9–6: Query Expression Forms

Priority	Operator Expression	Associates from
highest	function-name actual-parameter	right to left
	exp1 \ exp2, exp1 \\ exp2	left to right
	exp1 AND exp2	left to right
	exp1 OR exp2	left to right
lowest	exp1 XOR exp2	left to right

9.3 Default Parenthesizing

Default parenthesizing for query expressions is determined by the operator priorities and associativity given in the previous diagram. The following rules apply:

- Parenthesize the functions and operators of a given expression in order of descending priority. That is, first parenthesize all function calls (highest priority), then parenthesize pathname expressions (\ and \\) (next highest priority), and so on.
- If an expression contains several occurrences of the same operator, then parenthesize those operators in the order indicated by their associativity.

When an operator is parenthesized, the parentheses surround the operator and the one or two operands required by the operator.

As an example of the application of these rules, consider the following query expression:

```
CALLING CALLED_BY x OR y AND NOT z
```

This expression contains two binary operators and three function calls (NOT is a function). There are many ways in which it could be explicitly parenthesized. The following steps illustrate adding default parenthesizing:

```
CALLING CALLED_BY(x) OR y AND NOT (z)

CALLING( CALLED_BY(x) ) OR y AND NOT(z)

CALLING( CALLED_BY(x) ) OR (y AND NOT(z))

(CALLING( CALLED_BY(x) ) OR (y AND NOT(z)))
```

9.4 Semantics

SCA evaluates a query expression as follows:

1. Evaluates the operand(s) of the expression.
2. Calculates a value according to the specific rules for the given operator. The value obtained from this step is the value of the expression.

The order in which SCA evaluates the operands of a query expression is not defined. Since query expressions have no side effect, the order of evaluation does not matter. Furthermore, expressions are not necessarily evaluated from the innermost to the outermost, but they are evaluated in a semantically equivalent way.

The value of a query expression is a collection of symbol occurrences and, possibly, relationships between occurrences. A query result that has information about relationships between occurrences is called a **structured query result**. A nonstructured collection of occurrences has no interoccurrence relationship information.

A structured query result is a directed graph (possibly cyclic and possibly disjoint) in which the nodes are occurrences and the arcs are relationships between occurrences. A nonstructured query result is simply a special instance of a structured set: a graph in which every node is disjoint from every other node.

9.5 Attribute Selection Expressions

An *attribute-selection* expression selects occurrences based on the setting of occurrence and symbol attributes. An *attribute-selection* expression has the following form:

```
attribute-selection-exp ::= [ attribute-name = ] actual-parameter
```

If no attribute name is specified, then the *name-expression* attribute is assumed.

SCA supports the following types of attribute selection:

- Name
- Symbol class
- Symbol domain
- Occurrence class
- File specification

The rest of this section describes the types of attribute selection in more detail.

9.5.1 Name Selection

A *name-selection* expression selects occurrences that have names that match a specified name expression.

A *name-selection* expression has the following form, where *name* is a formal parameter name and a *name-expression* is a string of characters, possibly including wildcards.

```
name-selection-exp ::= name-expression |
                       name=name-expression |
                       name=( name-expression,... )
```

An *attribute-selection* expression with no formal parameter name is a *name-selection* expression. A name expression that includes a wildcard character is equivalent to a union of all the names that match the *name-selection* expression. A list of name expressions is equivalent to a union of *name-selection* expressions, each having a single name expression. Given these rules, the following three examples are equivalent:

```
name=( namexp1, namexp2 )

name=namexp1 OR name=namexp2

namexp1 OR namexp2
```

When a complex string is enclosed in quotation marks, the string can contain any ASCII character except a quotation mark. If you want a quotation mark in such a string, it must be represented by two successive quotation marks. For example, the following quoted complex string contains a single quotation mark enclosed in parentheses:

```
"one quotation mark ("")"
```

You can override the wildcard characters (% and *) using the escape character (&). For example, you can find the name consisting of a single asterisk by means of the name expression &*. If you want an ampersand in a string, it must be represented by two successive ampersands. For example, the name consisting of a single ampersand can be found by means of the name expression &&.

The enclosing of a complex string in quotation marks does not affect the case-sensitivity of the matching. String matching is not sensitive to the case of the string specified in the name expression.

Note that while a hyphen (-) is allowed in a simple name, a command line that ends in a hyphen is a continued command, just as in DCL.

9.5.2 Symbol Class Selection

A *symbol-class-selection* expression selects occurrences whose symbol class is one of those specified in the *symbol-class-selection* expression. A *symbol-class-selection* expression has the following form:

```
symbol-class-selection-exp ::= symbol=symbol-class |
                               symbol=( symbol-class,... )
```

Symbol is a formal parameter name and *symbol-class* is one of the following keywords:

- ARGUMENT — Formal argument (such as a routine argument or macro argument)
- COMPONENT, FIELD — Component of a record
- CONSTANT, LITERAL — Named compile-time constant value
- EXCEPTION — Exception
- FILE — File
- FUNCTION, PROCEDURE, PROGRAM, ROUTINE, SUBROUTINE — Callable program function
- GENERIC — Generic unit
- KEYWORD — Keyword
- LABEL — User-specified label
- MACRO — Macro
- MODULE, PACKAGE — Collection of logically related elements
- PLACEHOLDER — Marker where program text is needed
- PSECT — Program section

- TAG — Comment heading
- TASK — Task
- TYPE — User-defined type
- UNBOUND — Unbound name
- VARIABLE — Program variable
- OTHER — Any other class of symbol

You use one or more of the generic (multilanguage) keywords to request specific classes of symbols. Since different languages use different terminology, several alternatives are provided for some classes of symbols.

A list of symbol classes is equivalent to a union of *symbol-class-selection* expressions, each having a single symbol class.

9.5.3 Symbol Domain Selection

A *symbol-domain-selection* expression selects occurrences whose symbol domain is one of those specified in the *symbol-domain-selection* expression.

Symbol domain is the range of source code in which a symbol has the potential of being used. For example, a BLISS OWN declaration creates a symbol that has a module-specific symbol domain; it cannot be used outside of that module. On the other hand, a BLISS GLOBAL declaration creates a symbol that has a multimodule symbol domain; it has the potential of being used in more than one module. The symbol domain of a GLOBAL is multimodule regardless of how many modules there are in which the symbol is used.

A *symbol-domain-selection* expression has the following form:

```
symbol-domain-selection-exp ::= domain=symbol-domain |
                                domain=( symbol-domain,... )
```

Domain is a formal parameter name and *symbol-domain* is one of the following keywords:

- INHERITABLE — Able to be inherited into other modules (for example, by means of BLISS library, Pascal environment, or Ada compilation system mechanisms)
- GLOBAL — Known to multiple modules via linker global symbol definitions
- PREDEFINED — Defined by the language (examples: BLISS ap, FORTRAN sin, Pascal writeln)

- MULTI_MODULE — Domain spans more than one module (domain=multi_module is equivalent to domain=(inheritable,global, predefined)
- MODULE_SPECIFIC — Domain is limited to one module

A list of symbol domains is equivalent to a union of *symbol-domain-selection* expressions, each having a single symbol domain.

9.5.4 Occurrence Selection

An *occurrence-selection* expression selects occurrences whose occurrence class is one of those specified in the *occurrence-selection* expression. An *occurrence-selection* expression has the following form:

```
occurrence-selection-exp ::= occurrence=occurrence-class |
                             occurrence=( occurrence-class,...)
```

Occurrence is a formal parameter name and *occurrence-class* is one of the following keywords:

Declarations

- PRIMARY — Most significant declaration (such as FUNCTION)
- ASSOCIATED — Associated declaration (such as EXTERNAL)

References

- READ, FETCH — Fetch of a symbol value
- WRITE, STORE — Assignment of a symbol value
- ADDRESS, POINTER — Reference to the location of a symbol
- CALL — Call to a routine or macro
- COMMAND_LINE — Command line file reference
- INCLUDE — Source file include reference
- PRECOMPILED — Precompiled file include reference
- OTHER — Any other kind of reference (such as a macro expansion or use of a constant)

Other Occurrence Classes

- EXPLICIT — Explicitly declared
- IMPLICIT — Implicitly declared
- VISIBLE — Occurrence appears in the source

- HIDDEN — Occurrence does not appear in the source
- COMPILATION_UNIT — Occurrence is compilation-unit

9.5.5 File Specification Selection

A *file-specification-selection* expression selects occurrences whose source position is in one of the files specified in the *file-specification-selection* expression. A *file-specification-selection* expression has the following form;

```
file-spec-selection-exp ::= file_spec=name-expression |
                            file_spec=( name-expression,... )
```

File_specification is a formal parameter name and *name-expression* is a name expression that is interpreted as a file specification.

9.6 Operator Expressions

This section describes the operators you use with the SCA Query Language. Operators are a mechanism for querying SCA libraries. The value of an operator expression is the set of occurrences and relationships that result from applying the operator to the operands. SCA query expression operators are similar to functions in high-level languages, such as Pascal and Ada. Operator expressions have an operator name, enclosed by two operands, which are query expressions. The result of an operator expression is a query-expression result.

9.6.1 Path-Name Expressions

A *path-name* expression identifies symbols based on the nesting of primary declarations. The expression has the following form:

```
pathname-expression ::= exp1 \ exp2 |
                        exp1 \\ exp2
```

The pathname operators (\ and \\) are actually special cases of the general CONTAINED_BY function. The expression *exp1* \ *exp2* is equivalent to the following expression:

```
EXPAND CONTAINED_BY( exp1 AND occ=primary,
                     exp2 AND occ=primary,
                     result=begin,
                     depth=all,
                     trace="" )
```

The expression *exp1* \\ *exp2* is equivalent to the following expression:

```
EXPAND CONTAINED_BY( exp1 AND occ=primary,
                     exp2 AND occ=primary,
                     result=begin,
                     depth=all )
```

9.6.2 Intersection Expressions

An *intersection* expression identifies occurrences that exist in two different sets. The expression has the following form:

```
intersection-expression ::= exp1 AND exp2
```

The value of this expression is a set containing each occurrence that exists in both sets (exp1 and exp2); occurrences that appear in only one of the sets are not included.

9.6.3 Union Expressions

A *union* expression merges two sets. The expression has the following form:

```
union-expression ::= exp1 OR exp2
```

The value of this expression is a set containing all occurrences that exist in either set (exp1 or exp2).

9.6.4 Exclusive-Or Expressions

An *exclusive-or* expression selects the unique occurrences in two different sets. The expression has the following form:

```
exclusive-or-expression ::= exp1 XOR exp2
```

The value of this expression is a set containing all occurrences that exist in only one of the sets (exp1 and exp2); occurrences that appear in both sets are not included.

9.7 Function-Call Expressions

Function-call expressions are a general mechanism for querying SCA libraries. The value of a *function-call* expression is the set of occurrences and relationships that result from the evaluation of the function body.

SCA query expression functions are similar to functions in high-level languages, such as Pascal and Ada. Functions have a name, a parameter list, and a result. The result of a function is a query-expression result.

A parameter list consists of zero, one, or more parameters. Every parameter has a data type and, optionally, a default actual parameter value. All predefined functions described in this section have default actual parameter values for all parameters. A data type is either a query expression, a name expression, a keyword list, a range list, or a number.

The typical form of a *function-call* expression is as follows:

```
function-call-expression ::= function-name( [actual-parameter],... )
```

A *function-call* expression is a function name followed by an actual parameter list enclosed in parentheses. An actual parameter list can be empty if the function has no parameters or all of the parameters have default actual parameter values. In this case, the *function-call* expression consists of a function name followed by empty parentheses, or function-name( ).

If the actual parameter list consists of exactly one parameter, then the parentheses can be dropped. This form of the *function-call* expression is as follows:

```
function-call-expression ::= function-name actual-parameter
```

9.7.1 Parameter Association

A function call must pass exactly one actual parameter for each formal parameter. The actual parameter is either listed explicitly in the function call or is supplied by means of a default value in the function declaration. When a function call supplies no actual parameter for a formal parameter that was declared with a default value, you use the default. An error occurs if you fail to supply an actual parameter for a formal parameter that does not have a default value. All formal parameters for predefined functions have default values.

One way of establishing the correspondence between actual and formal parameters is to give the parameter in each list the same position. That is, the association of the actual and formal parameters proceeds from left to right, item by item, through both lists. This form of association is called *positional*.

Another way of establishing correspondence is to specify the formal parameter name and the actual parameter being passed to it. You can associate an actual parameter with a formal parameter by using the assignment (=) operator. The actual parameters in the call do not have to appear in the same order that the formal parameters appeared in the declaration. This form of association is called *named.*

You may use both positional and named actual parameters in the same call. However, you must still supply at most one actual parameter for any formal parameter, and you must list the positional parameters first.

9.7.2 Negation Function

The negation function finds occurrences that do not match a query expression. The function has the following form:

```
FUNCTION NOT( query_expression : query-expression = * )
```

The result of a call to this function is a set containing all occurrences that are not contained in *query_expression.* Note that the expression NOT( ) evaluates to the empty set.

9.7.3 Expansion Function

The expansion function expands a set of occurrences to include all the occurrences of the symbols that correspond to the original occurrence set.

The function has the following form:

```
FUNCTION EXPAND( query_expression : query-expression = * )
```

See Section 8.4.2 in Chapter 8 for an example of the expansion function.

9.7.4 Indicated Function

The indicated function is available only from within LSE. The indicated function matches the occurrence that the cursor is pointing at. Section 7.1.6.2 in Chapter 7 shows an example of the indicated function. The function has the following form:

```
FUNCTION INDICATED
```

The indicated function has no parameters. Thus, a call to the indicated function must have the following form:

```
INDICATED()
```

An indicated function can be nested within other query expressions.

9.7.5 Query Usage Function

A query usage function incorporates the results of previous queries into query expressions. The function has the following form:

```
FUNCTION @( query_name : query-name = sca$current_query )
```

The value of this expression is that of the expression that is specified as *query_name*. The default query is the current query, SCA$CURRENT_QUERY.

9.7.5.1 The Current Query

The current query specifies the result of the previous query or the one to which you are set using the NEXT QUERY, PREVIOUS QUERY, and GOTO QUERY commands. The name of the current query is SCA$CURRENT_QUERY. The current query is used as follows:

```
@sca$current_query
```

Since the @ function defaults to the current query, this expression may also be written as follows:

```
@()
```

9.7.6 Relationship Functions

A relationship function selects occurrences based on relationships between occurrences. There are two kinds of relationship expressions: *structured-relationship* expressions and *nonstructured-relationship* expressions.

A *structured-relationship* expression selects both occurrences and relationships between them. A *structured-relationship* expression preserves relationship information in the value of the expression. The result of such an expression is a structured query result.

A *nonstructured-relationship* expression selects occurrences based on relationships between occurrences. A *nonstructured-relationship* expression uses information about the relationships between occurrences, but does not preserve relationship information in the result of the expression. The result of such an expression is a nonstructured query result.

All of the relationship functions have the same set of parameters. The different relationship functions differ in their name and, of course, in their semantics. A relationship function has the following form:

```
FUNCTION function-name( end : query-expression = *,
                        begin : query-expression = *,
                        depth : number = 1,
                        result : keyword-list = structure,
                        trace : query-expression = * )
```

9.7.6.1 Individual Relationship Functions

This section describes the individual relationship functions. There are two kinds of relationship functions: basic functions and inverse functions. You can transform every basic function into its corresponding inverse function by removing the ING at the end of the function and adding ED_BY.

For example, you can transform the basic function CALLING to the inverse-function, CALLED_BY. So the two commands, CALLING(y,x) and CALLED_BY(x,y), produce the same result: a graph of call relationships from X to Y.

Relationship Functions

CALLING

CALLED_BY

The CALLING relationship finds those occurrences in *begin-exp* that are calling occurrences in *end-exp*. Typically, if not exclusively, declarations in *begin-exp* call references in *end-exp*. The CALLED_BY relationship finds those occurrences in *begin-exp* that are called by occurrences in *end-exp*.

CONTAINING

CONTAINED_BY

The CONTAINING relationship finds those occurrences in *begin-exp* that contain occurrences in *end-exp*. The CONTAINED_BY relationship finds those occurrences in *begin-exp* that contain occurrences in *end-exp*.

TYPING

TYPED_BY

The TYPING relationship finds those occurrences in *begin-exp* that determine the type of occurrences in, or are typing, *end-exp*. The TYPED_BY relationship finds those occurrences in *begin-exp* whose type is one of the occurrences in, or are typed by, *end-exp*.

9.7.6.2 Relationship Parameters

Relationship parameters determine the precise semantics of a relationship expression. The following is a list of relationship parameters and how you use them.

Relationship Parameters

END=end-expression
Specifies those occurrences at which the tracing of relationships can end. Only paths that end on one of these occurrences are included in the result. The default is *end=**.

BEGIN=begin-expression
Specifies those occurrences at which the tracing of relationships can begin. Only paths that begin on one of these occurrences are included in the result. The default is *begin=**.

DEPTH=depth-level
Specifies the number of levels of structure that are to be traced. The default depth-level is 1. *Depth=all* indicates that there is no limit to the number of levels of structure that are to be traced.

RESULT=result-keyword-list
Where *result-keyword-list* is one or more of the following keywords:

- [NO]STRUCTURE — Indicates whether relationship information is to be preserved in the query result.
- ANY_PATH — Indicates that any path that traces from the begin-expression to the end-expression will satisfy the query. SCA will return the first complete path(s) it finds. By default, all such paths are returned.

- BEGIN — Indicates that only those occurrences that begin the relationship graph are to be included in the result. This keyword implies RESULT=NOSTRUCTURE. It is incompatible with RESULT=STRUCTURE.
- END — Indicates that only those occurrences that end the relationship graph are to be included in the result. This keyword implies RESULT=NOSTRUCTURE. It is incompatible with RESULT=STRUCTURE.

The default is *result=structure*.

TRACE=trace-expression
A query expression whose result specifies those occurrences through which relationship tracing is to be continued. The *trace* parameter does not affect the first iteration. That first iteration is controlled by the *begin* parameter. The default is *trace=**.

9.7.7 The IN Function

The IN function restricts a set of occurrences to those occurrences that are directly or indirectly contained by another set of occurrences.

The function has the following form:

```
FUNCTION IN( end : query-expression = *,
             begin : query-expression = * ) =

CONTAINED_BY( end AND decl=primary,
              begin,
              result=begin,
              depth=all )
```

The IN function is a special case of the CONTAINED_BY function. It has been included in the set of predefined functions because it provides a particularly useful subset of the capabilities of the CONTAINED_BY function.

9.8 Abbreviation Rules

You may abbreviate attribute-selection formal parameter names and attribute-selection actual parameter keywords to their first four characters. You may truncate these names and keywords to fewer characters as long as the truncation is unique. For example, the symbol-class-selection attribute name is the only such name that begins with S. Therefore, you can abbreviate the *symbol* attribute name to just one character.

Attribute-selection actual parameter keywords work the same way.

Special considerations apply when you type these names and keywords in command procedures. To ensure readability, you should not abbreviate at all. If you do abbreviate, never abbreviate to fewer than four characters or you risk the possibility that your command procedure may not be compatible with future releases of SCA.

Only attribute-selection formal parameter names and attribute-selection actual parameter keywords can be abbreviated.

Chapter 10

Using SCA Libraries

This chapter describes SCA libraries, their capabilities, and associated commands.

10.1 Overview

An SCA library is a collection of source information that is generated by supporting VMS compilers in the form of .ANA files. Unless otherwise specified, an .ANA file is created in your current default directory from which the source information can be loaded (with a LOAD command) into a specified SCA library. Each .ANA file contains one or more modules of data with each module being a logically complete set of source information or, in compiler terms, a **compilation unit**.

Library commands allow you to create and manipulate libraries and their contents. The commands allow you to produce and maintain a **library list**, which establishes a library for selection and manipulation (such as loading, querying, or deletion of library modules) or for access to multiple physical libraries as a single **virtual** library.

10.1.1 Using Remote Libraries

With SCA, you can access SCA libraries over the network. To use an SCA library over the network, use either the SET LIBRARY or the CREATE LIBRARY command with a directory specification that includes a node name. Once the SET LIBRARY or CREATE LIBRARY command has completed, you can issue queries and update the library as if it were local.

There are two requirements for using a remote SCA library.

1. The target node you specify must have the SCA server installed. If you use the SET LIBRARY command to connect a node that does not have an SCA server, you will get an error message.
2. The target SCA library also must be accessible to the SCA server. SCA automatically grants the SCA server access to your SCA library through an Access Control List when you create the library.

The following is an example of how to specify a remote SCA library directory on the node MYNODE using the SET LIBRARY command.

```
$ SCA SET LIBRARY MYNODE::MYDEV$:[USER.PROJ.SCALIB]
```

You can also give an access control string or a proxy account name as part of the node specification for the library. The SCA server ensures that only authorized remote users access your library.

10.2 Library Manipulation

With the following commands you can create, select, load, and delete SCA libraries. You can use these commands within LSE or at the DCL level.

- CREATE LIBRARY
- SET LIBRARY
- SET NOLIBRARY
- LOAD
- DELETE LIBRARY

10.2.1 Creating a Library Directory

The first step in creating an SCA library is to create a directory for it at the DCL level. The following command creates a subdirectory for a local SCA library:

```
$  CREATE/DIRECTORY PROJ:[USER.LIB1]
```

10.2.2 Creating a Library

With the CREATE LIBRARY command, you can initialize a new SCA library by specifying its directory. The command has the following form:

```
CREATE LIBRARY [/qualifier...] directory-spec[,...]
```

In the following example, the CREATE LIBRARY command initializes and activates two libraries (LIB1,LIB3) in the local subdirectories specified.

```
$  SCA CREATE LIBRARY [.LIB1],[.LIB3]
%SCA-S-NEWLIB, SCA Library created in PROJ:[USER.LIB1]
%SCA-S-LIB, your SCA Library is PROJ:[USER.LIB1]
%SCA-S-NEWLIB, SCA Library created in PROJ:[USER.LIB3]
%SCA-S-LIB, your SCA Libraries are
-SCA-S-LIB,      PROJ:[USER.LIB1]
-SCA-S-LIB,      PROJ:[USER.LIB3]
```

Adding Libraries to the Front of a List

SCA searches for libraries in the order they are listed on library lists. The /BEFORE qualifier adds the libraries specified on the command line to the beginning of the current library list. The /BEFORE=library-spec qualifier inserts the libraries specified on the command line before the library specified by the qualifier.

In the following example, the CREATE LIBRARY command creates a library (LIB2) and inserts its directory specification in the current library list before the library (LIB3) specified by the /BEFORE qualifier.

```
$  SCA CREATE LIBRARY/BEFORE=[.LIB3] [.LIB2]
%SCA-S-NEWLIB, SCA Library created in PROJ:[USER.LIB2]
%SCA-S-LIB, your SCA Libraries are
-SCA-S-LIB,      PROJ:[USER.LIB1]
-SCA-S-LIB,      PROJ:[USER.LIB2]
-SCA-S-LIB,      PROJ:[USER.LIB3]
```

Replacing an Existing Library

The /[NO]REPLACE qualifier replaces an existing library in the specified directory with a new empty library. The default is /NOREPLACE. In the following example, the CREATE LIBRARY command reinitializes LIB3.

```
$  SCA CREATE LIBRARY/REPLACE [.LIB3]
%SCA-S-NEWLIB, SCA Library created in PROJ:[USER.LIB3]
%SCA-W-NEWLIB, your SCA Library is PROJ:[USER.LIB3]
```

Adding Libraries to the End of a List

The /AFTER qualifier adds the libraries specified on the command line to the end of the current library list. The /AFTER=library-spec qualifier inserts libraries specified on the command line after the library specified by the qualifier.

In the following example, the CREATE LIBRARY command inserts the library specified on the command line (LIB4) in the list following the library (LIB3) specified by the /AFTER qualifier.

```
$  SCA CREATE LIBRARY/AFTER=[.LIB3] [.LIB4]
%SCA-S-NEWLIB, SCA Library created in PROJ:[USER.LIB4]
%SCA-S-LIB, your SCA Libraries are
-SCA-S-LIB,      PROJ:[USER.LIB1]
-SCA-S-LIB,      PROJ:[USER.LIB2]
-SCA-S-LIB,      PROJ:[USER.LIB3]
-SCA-S-LIB,      PROJ:[USER.LIB4]
```

Estimating Numbers of Modules

The /MODULES=module-count qualifier specifies an estimated size for a library, expressed in terms of the numbers of modules that will appear. The default is /MODULES=25. The number serves only as an estimate of initial library size for performance purposes. It does not restrict a library from expanding beyong the module number specified.

Specifying Library Size

The /SIZE=block-count qualifier specifies an estimated size for a library, expressed in terms of numbers of blocks of compiler-generated source analysis data. The default is /SIZE=250 (blocks).

Network Access

SCA allows network access as part of the CREATE LIBRARY command. SCA adds the following access control list to the library directory:

```
(IDENTIFIER=[SCA$SERVER],ACCESS=READ+WRITE+EXECUTE)
(IDENTIFIER=[SCA$SERVER],OPTIONS=DEFAULT,ACCESS=READ+WRITE+_
EXECUTE+DELETE+CONTROL)
```

This ensures that the SCA server will have access to your library for remote commands. The SCA server verifies that any remote user has the proper access to your library before allowing any remote commands.

10.2.3 Specifying a Library

You must specify an SCA library for use during an SCA session. To do this, you use the SET LIBRARY command. If a library list exists, it is replaced by default. The SET LIBRARY command has the following form:

```
SET LIBRARY [/qualifier...] directory-spec[,...]
```

In the following example, the SET LIBRARY command replaces the entries on the library list with those specified (LIB1,LIB2) and selects them for access as a single virtual library.

```
$  SCA SET LIBRARY [.LIB1],[.LIB2]
%SCA-S-LIB, your SCA libraries are
-SCA-S-LIB,      PROJ:[USER.LIB1]
-SCA-S-LIB,      PROJ:[USER.LIB2]
```

Adding Libraries to a List

The /BEFORE and /AFTER qualifiers add libraries specified on the command line to the current library list as described for the CREATE LIBRARY command.

10.2.4 Removing a Library

You use the SET NOLIBRARY command to remove SCA libraries from the list of current libraries. The SET NOLIBRARY command has the following form:

```
SET NOLIBRARY [/qualifier] [library-spec[,...]]
```

In the following example, the SET NOLIBRARY command removes the specified libraries from the current library list. The parameter specifies the libraries to be removed. If you omit the parameter, all the libraries will be removed from the list.

```
$  SCA SET NOLIBRARY [.LIB1],[.LIB2]
%SCA-S-LIBREM, SCA LIBRARY PROJ:[USER.LIB1] deactivated
%SCA-S-LIBREM, SCA LIBRARY PROJ:[USER.LIB2] deactivated
```

The only qualifiers available with this command are the message control qualifiers (/LOG (the default) and /NOLOG).

10.2.5 Loading Library Information

The LOAD command loads one or more files of compiler-generated source analysis data (.ANA) into an SCA library. If you want to load more than one .ANA file, you may use wildcard file specifications to identify the files. The LOAD command has the following form:

```
LOAD [/qualifier...] file-spec[,...]
```

When you issue a LOAD command, the first library on the current list is loaded by default unless you specify another library. For example, if the first library on the list is located at [.LIB1], loading occurs as follows:

```
$  SCA LOAD PG1,PG2,PG3
$  SCA LOAD/LIBRARY=[.LIB2] PG4,PG5
$  SCA LOAD/LIBRARY=[.LIB3] PG6,PG7
```

By default, the first command loads the first library listed (LIB1) with the modules contained in the specified data analysis files (PG1-PG3); the next commands then load the libraries (LIB2,LIB3) specified by the /LIBRARY qualifier. You must use the /LIBRARY qualifier to specify libraries on your library list.

Replacing and Adding Analysis Data

The /REPLACE qualifier replaces modules in the specified library, if they exist, and adds any newly specified modules. The /NOREPLACE qualifier adds new modules to the library. The default is /REPLACE.

In the following example, the /NOREPLACE qualifier adds a new file of source analysis data to the current primary library (/LIBRARY=primary-library by default).

```
$  SCA LOAD/NOREPLACE  PG1,PG4
%SCA-W-LOADED, module PG1 has already been loaded
%SCA-S-LOADED, module PG4 loaded
%SCA-S-COUNT, 1 module loaded, (1 new, 0 replaced)
```

Specifying the Update Library

The /LIBRARY=library-spec qualifier specifies the SCA library to be updated. The update library must be one of the libraries on the current list. The default is /LIBRARY=primary-library.

In the following example, the LOAD command replaces (/REPLACE by default) the specified module (PG1) if it exists in the specified library (LIB2). If the module does not exist, it is added to the library.

```
$  SCA LOAD/LIBRARY=LIB2 PG1
%SCA-S-LOADED, module PG1 loaded
```

Deleting Analysis Data Files

The /DELETE qualifier deletes an .ANA file from its present location when it is successfully loaded into an SCA library. You can recover deleted .ANA files from SCA libraries by using the EXTRACT MODULE command.

10.2.6 Deleting a Library

The DELETE LIBRARY command deletes the library in the local subdirectory specified. The file containing the library is deleted and the library returns to its state before a CREATE LIBRARY command was issued. The DELETE LIBRARY command has the following form:

```
DELETE LIBRARY [/qualifier...] directory-spec[,...]
```

10.2.7 Multiple Libraries

A virtual library is formed by specifying multiple physical libraries in a search list. Multiple libraries give all users access to common physical libraries while allowing individuals to modify separate physical libraries. Multiple libraries also have the following benefits:

- Make more effective use of disk space
- Relieve concurrent usage problems
- Perform LOAD and REORGANIZE operations faster than large single libraries
- Allow REORGANIZE operations to proceed when limited disk space impedes the same operation on a large single library

A virtual library appears to the user as one large physical library. The precise location of the modules in the virtual library is unimportant.

You set up a virtual library with a library search list, which includes the names of the physical libraries to comprise the virtual library. A search list is a comma-separated list on a CREATE LIBRARY or SET LIBRARY command line. Identical modules are never duplicated; modules contained in libraries in a search list supersede modules of the same name contained in libraries later on in the search list.

For example, you might create three (empty) libraries in three separate directories. Each library is then separately loaded with the following modules:

```
               LIB1            LIB2            LIB3
               ----            ----            ----
Modules:     A, B, C         B, C, D         D, E, F
```

Next, with the following command, Modules A, B, and C are selected from LIB1; Module D is selected from LIB2; and Modules E and F are selected from LIB3:

```
$  SCA SET LIBRARY LIB1,LIB2,LIB3
```

The multiple physical libraries (LIB1, LIB2, LIB3) are thus defined by SCA for the current session as a single virtual library containing Modules A, B, C, D, E, and F.

For access purposes, the selected modules are defined as **visible modules.** Those modules passed over because they have the same name as a module already selected are defined as **hidden modules**.

Virtual libraries can satisfy the separate and collective library needs of the members of a project team. For instance, LIB1, in the virtual library shown, could be a local library dedicated to a task assigned to an individual team member. LIB2 could be a group library, used for a task shared with other members of the team, while LIB3 could be a project-wide library that is used by all team members. Thus, each team member is capable of "seeing" through their local work area to other work areas. If a module is not found in a local area, the search continues automatically through each area listed until the module is found.

The Library List

A library list provides you with a modifiable search list of library directories. Once you define a library list, it remains available for the current process; you can then use SCA commands during subsequent sessions to access the listed libraries for manipulation or querying.

When you initialize a new library by issuing a CREATE LIBRARY command, a library is created in each library directory specified on the command line. If a library list does not exist when the command is executed, the specified directories will form the list; if a library list does exist, it will be replaced by the specified directories.

The list remains defined for the current process or until it is either replaced or modified by another CREATE LIBRARY command, or replaced or modified by a SET LIBRARY command.

As an example of the production and modification of a library list, assume that the following commands are executed during a single process.

The following command initializes a library in the directory specified.

```
$  SCA CREATE LIBRARY [.LIB1]
```

This command initializes the new library (LIB1) for subsequent access, and establishes it as the new virtual library.

You can now position a library (or group of libraries) anywhere within the current list by using a /BEFORE or /AFTER qualifier on the command line. For example:

```
$  SCA CREATE LIBRARY/AFTER=[.LIB1] [.LIB2]
$  SCA CREATE LIBRARY/AFTER=[.LIB2] [.LIB3]
```

These commands create the specified libraries (LIB2,LIB3) and insert them in the current library list following the libraries (LIB1,LIB2) specified by the qualifier. At this point, the libraries listed are LIB1,LIB2,LIB3.

Once the libraries are loaded, you can use commands to query the contents of the libraries currently listed (LIB1, LIB2, LIB3). Note that all three libraries will be viewed as a single virtual library.

10.2.8 Library Planning

There are many ways to incorporate SCA libraries into software development projects. This section briefly describes one method of planning and maintaining your libraries during the development of a large-scale software system.

You can arrange an SCA library system to provide an index of detailed source information for your project that includes the following:

- A single comprehensive project-wide SCA library that reflects the development of an entire software system
- Individual SCA libraries that reflect local sources required for individual development tasks

Basic Library Structures

Prior to implementing development work, if an earlier version of your application software exists, you create a project-wide SCA library to reflect the preliminary version of the system. Such a library grows with the development process, and allows current information and sources to be made interactively available to all members of your project team.

During development, SCA libraries are individually created by team members to reflect local sources and provide small temporary working libraries. Such libraries are created and updated to serve a specific task, and their use is enhanced by the ability of SCA to view several physical libraries as a single virtual library. Thus, each team member sets up a virtual library structure consisting of their individual libraries followed (in the library list) by the project-wide library.

As interim tasks are completed, and source files are updated in your source library, you also update your project-wide SCA library by replacing existing modules of source analysis data with new information. This allows the library to reflect current modifications and to keep the project team informed of interim developments. Finally, when predefined milestones are met and major system builds are initiated, you may choose to re-create your entire project-wide library to allow current build sources and development status to be tied to specific milestones.

Library Maintenance

When the final development phases of a project are completed, a project-wide SCA library is created or modified to ensure library information is in place that reflects the current version of the software. Repeating this procedure for each subsequent version of a system provides comprehensive SCA libraries that satisfy the need for both debugging information during the maintenance phase of development and general information for future development.

Note, however, that project-wide SCA libraries can absorb a significant amount of disk space, and can be reproduced. Therefore, except for specific applications, multiple versions should not be kept.

Creating and modifying local personal libraries is the responsibility of the individual user. Because such libraries generally serve a temporary need, updates will reflect an individual's need for current information.

In all cases, build procedures can be created or modified to update SCA libraries at appropriate times (such as during the compilation of local source files or during major system builds).

Structures and Concurrent Access

A virtual library structure can contain one or more physical libraries. When you set up a multiple library system, the size and volatility of each physical library composing the system should be considered in the following context:

- A physical library can be concurrently queried by multiple users.
- Exclusive access is imposed on a physical library during updating.

- Small physical libraries are more efficiently updated than large physical libraries.

These facts recommend a library structure for a team environment that arranges small volatile libraries, independently, followed by a large (and essentially static) project-wide library. Such arrangements allow most of the physical libraries within a structure to remain concurrently available to the members of your team while exclusive updates to other libraries are being performed.

10.3 Library Maintenance

With the following commands, you can display and reorganize information about your libraries and their contents. You can use these commands within LSE or at the DCL level.

- SHOW LIBRARY
- SHOW MODULE
- DELETE MODULE
- VERIFY
- REORGANIZE

10.3.1 Displaying Library Specifications

You can display the directory specifications for the current SCA library with the following command:

```
$  SCA SHOW LIBRARY
%SCA-S-LIB, your current SCA library is PROJ:[USER.SCA]
$
```

10.3.2 Displaying Module Information

The SHOW MODULE command selectively displays information about modules in SCA libraries. The SHOW LIBRARY command has the following form:

```
SHOW MODULE [/qualifier...] [module-name-expr[,...]]
```

Complete or partial information about all modules, or selected modules, can be displayed. The terms "visible" and "hidden" refer to the results of the module selection process that occur when multiple libraries are accessed.

Selecting a Library

The /LIBRARY=library-spec qualifier specifies the directory specification of the library from which module information is displayed.

Displaying Abbreviated Information

If you use a general module query, abbreviated information is displayed. The default qualifier is /BRIEF. For example:

```
$  SCA SHOW MODULE
   Module              Ident     Language    Compiled

BUILD_TABLE          1  01        Pascal   24-Oct-1989 15:43
COPY_FILE            1  01        Pascal   24-Oct-1989 15:44
EXPAND_STRING        1  01        Pascal   24-Oct-1989 15:44
OPEN_FILES           1  01        Pascal   24-Oct-1989 15:43
TRANSLIT             1  01        Pascal   24-Oct-1989 15:44
TYPES                1  01        Pascal   24-Oct-1989 15:43
%SCA-S-MODULES, total of 6 modules
```

Displaying Detailed Information

If you use a general module query with the /FULL qualifier, details of all the module information in the library are displayed.

If you type a specific module name, detailed information on the specified module is displayed.

Displaying Visible Modules

The /VISIBLE qualifier displays only visible modules. The default is /VISIBLE.

Displaying Hidden Modules

The /HIDDEN qualifier displays hidden modules.

Displaying All Modules

The /ALL qualifier displays all modules (both visible and hidden).

10.3.3 Deleting Module Information

The DELETE MODULE command deletes source information from the current library for one or more source modules. The DELETE MODULE command has the following form:

```
DELETE MODULE [/qualifier...] module-name-exp[,...]
```

In the following example, Module PG1 is deleted from the specified library.

```
$  SCA DELETE MODULE/LIBRARY=[.LIB1] PG1
```

Specifying a Library

The /LIBRARY[=library-spec] qualifier specifies the current library (established by a SET LIBRARY command) containing the modules to be deleted. The default is /LIBRARY=primary-library.

Confirming Deletions

The /CONFIRM qualifier provides a confirmation prompt for each module specified for deletion. The default is /NOCONFIRM.

10.3.4 Verifying and Recovering a Library

The VERIFY command makes the following validity checks:

- Checks for corrupted libraries resulting from abnormal termination of a LOAD, DELETE MODULE, or REORGANIZE command
- Optionally, recovers corrupted libraries

The VERIFY command has the following form:

```
VERIFY [/qualifier...] [library-spec[,...]]
```

The parameter specifies the SCA libraries to be verified. If no library is specified, the first library on the library list is assumed.

Recovering a Corrupted Library

The /RECOVER qualifier indicates whether a corrupted library should be recovered. The default is /NORECOVER.

The library is repaired by deleting the module that was being processed at the time of the LOAD or DELETE MODULE failure.

10.3.5 Optimizing a Library

The REORGANIZE command sorts, compresses, and reorders the data structures in an SCA library, freeing up memory space and improving library performance. The result is a smaller, more efficient SCA library.

The REORGANIZE command has the following form:

```
REORGANIZE [/qualifier...] [library-spec[,...]]
```

You can use this command after a library has been substantially updated, such as after creation and loading, or after a series of LOAD or DELETE MODULE commands. For example, the following sequence is recommended to create an SCA library. You can change the qualifiers on the CREATE LIBRARY command to suit your needs.

```
$  SCA
SCA>  CREATE LIBRARY/SIZE=8000/MOD=200 library-directory
SCA>  LOAD data-file-directory:*.ANA
SCA>  REORGANIZE
```

The library-directory parameter specifies the location of the library to be reorganized. The default is the current library directory.

The REORGANIZE commands creates scratch files in SYS$SCRATCH approximately equal in size to the files in the library being reorganized.

Chapter 11

Using the VAX Source Code Analyzer INSPECT Command

This chapter describes the functions and use of the INSPECT command. Section 11.1 is an overview. Section 11.2 describes INSPECT command concepts. Section 11.3 discusses the qualifiers for the INSPECT command. Section 11.4 describes INSPECT command diagnostic error messages. Section 11.5 tells you how you can tailor SCA for diverse programming styles.

11.1 Overview

SCA provides the INSPECT command to allow you to check consistency in a software system both within modules and across module boundaries. As with the FIND command, the results are reported in the form of a query with diagnostic error messages.

The INSPECT command has the following form:

```
INSPECT [/qualifier...] query_expression
```

11.2 INSPECT Command Concepts

This section covers INSPECT command concepts, including the following:

- General checking philosophy, query expressions, and results
- Routines and common blocks
- Master declarations and checking

11.2.1 General Checking Philosophy

The INSPECT command checks elements within a software system that have a high probability of causing genuine problems. With the INSPECT command, you can tailor checks and eliminate elements from a check that you have established are correct. This allows you to focus on more questionable areas. The INSPECT command tries to avoid producing voluminous output, making it easy for you to search for problem areas.

The format of the INSPECT command is like that of the FIND command. You use query expressions to specify what to check just as you would with FIND. Used with the /CHARACTERISTICS qualifier, query expressions form part of your input to direct which checks are applied to your software system.

The output, or result, of each INSPECT check is reported as a query with diagnostic error messages. The query display reveals the structure of routines and common blocks.

11.2.2 Routines and Common Blocks

As the INSPECT command performs checks, it takes into account the relationships between routines and their arguments, and the relationships between common blocks and the variables in common blocks. When you check a routine or common block, the INSPECT command also checks the arguments of the routine or the variables contained in the common block; an exception is the case of unused symbol checks, where only the routine is checked.

You should note that the INSPECT command treats routines somewhat differently than you might expect. Because INSPECT checks both the routine and its arguments, its arguments determine the outcome of the INSPECT operation. The same holds true for common blocks and their variables.

Checking an argument or common block variable by itself checks only for that occurrence of the routine or common block.

11.2.3 Master Declarations and Checking

The INSPECT command performs three different kinds of checking: **consistency checks**, **symbol checks**, and **occurrence checks**.

To perform checking, INSPECT chooses a master declaration within your system against which other elements are compared. The master declaration is displayed in the result, allowing you to confirm what the INSPECT command has used as the basis for a check. Sometimes (typically when type-checking), the INSPECT command is unable to locate a master declaration because of more stringent requirements. For example, when doing type checking, the master declaration must have type information. In this case, the INSPECT command produces a message saying it was unable to locate the master declaration.

The TYPE and UNIQUE_NAMES options of the /CHARACTERISTICS qualifier perform consistency checking.

The consistency check examples in this section are based on the following program:

```
PROGRAM prog1
COMMON /common1/ a, b, i, j
COMMON /common2/ e, f, g, h
a = 1.0
b = 1.0
i = 1
j = 0
e = 1.0
f = 2.0
g = 3.0
h = 4.0

CALL sub1

TYPE *, i,j
END

SUBROUTINE sub1
COMMON /common1/ a, i, j, b
COMMON /common2/ e, f, g, h

i = j * h
j = i * f

RETURN
END
```

The following INSPECT command checks the common blocks in the example program for consistency. Specifically, it looks for type mismatches and name mismatches. Notice that although the common block COMMON1 has the same variable names in both PROG1 and SUB1, the order is different, so B, I, or J in SUB1 are not the same as in PROG1. Also, notice that the variable

J in PROG1, which corresponds to variable I in SUB1, has the same type, so checking for type mismatches would not find it.

```
INSPECT symbol=psect/CHARACTERISTICS=(TYPE,UNIQUE_NAMES)
```

The result follows.

```
COMMON1 common block has fatal errors
    PROG1\2                 COMMON declaration
    SUB1\3                  COMMON declaration
   B variable has fatal errors
    PROG1\2                 variable declaration
    SUB1\3                  declaration is mistyped
                            declaration has the wrong name
   I variable has the wrong name
    PROG1\2                 variable declaration
    SUB1\3                  declaration has the wrong name
   J variable has fatal errors
    PROG1\2                 variable declaration
    SUB1\3                  declaration is mistyped
                            declaration has the wrong name
%SCA-S-OCCURS, 5 problems found (2 names, 4 symbols, 4 occurrences)
```

The result of checking for type mismatches follows.

```
COMMON1 common block has fatal errors
    PROG1\2                 COMMON declaration
    SUB1\3                  COMMON declaration
   B variable has fatal errors
    PROG1\2                 variable declaration
    SUB1\3                  declaration is mistyped
   J variable has fatal errors
    PROG1\2                 variable declaration
    SUB1\3                  declaration is mistyped
%SCA-S-OCCURS, 2 problems found (2 names, 4 symbols, 4 occurrences)
```

The INSPECT command also checks for problems with a particular symbol. Symbol check errors indicate errors with the whole symbol rather than with a particular occurrence. The resulting display shows occurrences that indicate which symbol has an error to help you determine exactly where the error is. Symbol checks do not require master declarations.

The USAGE and UNUSED_SYMBOLS options perform symbol checks.

The symbol check example in this section is based on the following program:

```
     PROGRAM fortran_test2
     COMMON /common_block1/ a, b, c, d
     COMMON e, f, g, h
```

```
        a = 1.0
        b = 2.0
        c = 3.0
        l = 4
        CALL fortran_sub2
        IF (a .EQ. e) THEN
            WRITE (UNIT=*, FMT=*) 'A = E'
        END IF

        END
        SUBROUTINE fortran_sub2
        COMMON /common_block1/ a, b, c, d
        COMMON e, f, g, h

        e = 1
        f = 2
        g = 3
        h = 4
        RETURN
        END
```

The following INSPECT command checks all common blocks for any unused symbols, or symbols with usage problems. These are symbol-level errors; there is no particular occurrence that is in error, although the INSPECT command places all declarations it finds into the result to help you find the real problem.

```
INSPECT symbol=psect/CHARACTERISTICS=(UNUSED,USAGE)
```

The result follows.

```
$BLANK common block is never read
    FORTRAN_SUB2\3       COMMON declaration (hidden)
    FORTRAN_TEST2\3      COMMON declaration (hidden)
   F variable is never read
    FORTRAN_SUB2\3       declaration is never read
    FORTRAN_TEST2\3      declaration is never read
   G variable is never read
    FORTRAN_SUB2\3       declaration is never read
    FORTRAN_TEST2\3      declaration is never read
   H variable is never read
    FORTRAN_SUB2\3       declaration is never read
    FORTRAN_TEST2\3      declaration is never read
COMMON_BLOCK1 common block has warnings
    FORTRAN_SUB2\2       COMMON declaration
    FORTRAN_TEST2\2      COMMON declaration
   B variable is never read
    FORTRAN_SUB2\2       declaration is never read
    FORTRAN_TEST2\2      declaration is never read
   C variable is never read
    FORTRAN_SUB2\2       declaration is never read
    FORTRAN_TEST2\2      declaration is never read
   D variable is never used
    FORTRAN_SUB2\2       declaration is never used
    FORTRAN_TEST2\2      declaration is never used
%SCA-S-OCCURS, 6 problems found (2 names, 4 symbols, 4 occurrences)
```

The INSPECT command checks occurrences to determine whether a particular occurrence is correct on its own. Like symbol checks, occurrence checks do not require master declarations.

The IMPLICIT DECLARATIONS option checks occurrences.

The occurrence check example in this section is based on the following program:

```
#module main

int b();

main()
{
    char p1 [10];
    int p2;

    b(p1,p2);
}

#module b

int b(p1,p2)
    char *p1;
    int p2;
{
    float x;
    int i;

    x = atof(p1);
    for (i = 0;  i < p2;  i++)
    {
        printf("X = %f\n", x );
    }
}
```

The following INSPECT command checks all routines in the example program for implicit declarations. Since the ATOF function is not explicitly declared, the C compiler gives it an implicit declaration, returning a type of *int*. ATOF is returning a double, so the value returned is going to be wrong most of the time. However, implicit declarations are usually an indication of poor coding practices rather than genuine problems.

```
INSPECT symbol=routine/CHARACTERISTICS=IMPLICIT
```

The result follows.

```
ATOF function is implicit
    B\10                 declaration is implicit
PRINTF function is implicit
    B\13                 declaration is implicit
%SCA-S-OCCURS, 2 problems found (4 names, 4 symbols, 8 occurrences)
```

11.3 INSPECT Command Qualifiers

You can use the following command qualifiers with the INSPECT command:

- All FIND command qualifiers
- /CHARACTERISTICS
- /SEVERITY_LEVEL
- /[NO]ERROR_LIMIT

The following three sections describe specific uses of the INSPECT command qualifiers. For additional information on command qualifiers, see Chapters 7, 10, and the Command Dictionary section of the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*

11.3.1 Performing Various Types of Checking

With the /CHARACTERISTICS qualifier, you can perform different types of checks. The type of check to be done is indicated with an option keyword. Adding a query expression parameter specifies the set of occurrences to be inspected.

The /CHARACTERISTICS qualifier has the following form:

```
INSPECT/CHARACTERISTICS=option query_expression
```

Table 11–1 shows the options available for use with the /CHARACTERISTICS qualifier.

Table 11–1: /CHARACTERISTICS Type Options

Option	Type of Check
TYPE	Compares occurrences of symbols to see that they have the same type, especially within routines and common blocks where the symbol's position is the key to its meaning.
UNIQUE_NAME	Checks whether multiple declarations of the same symbol have the same name.

(continued on next page)

Table 11–1 (Cont.): /CHARACTERISTICS Type Options

Option	Type of Check
USAGE	Finds variables read and not written or variables written and not read.
UNUSED_SYMBOL	Checks to see if each symbol is referenced; reports symbols declared but never used.
IMPLICIT_DECLARATION	Finds symbols used without previously declaring them; for languages such as FORTRAN.
ALL	Performs all characteristics checks.
NONE	Performs no characteristics checks. This is the default option.

You can negate any option keyword except ALL or NONE, for example, NOTYPE.

With the UNIQUE_NAMES option, symbols are assumed to be the same based on symbol information, such as the module where the symbol is declared, domain, and so on. The name of the symbol is compared to other symbol names as a consistency check. You use this option with routine parameters or common block variables.

The following is an example of using INSPECT type-checking to find a problem with type mismatches in a cross-language routine call. The type-checking example in this section is based on the following program:

```
        INTEGER FUNCTION do_something( argument1, argument2 )
        INTEGER*4 argument1
        REAL*8 argument2

        IF (argument1 .GT. 0) THEN
            do_something = 1
            argument2 = argument1 * argument1 * 3.14159
        ELSE
            do_something = 0
        END IF
        RETURN
        END

PROGRAM using_characteristics ( input, output ) ;

VAR
    input_value, output_value : INTEGER;
    output_result : REAL;
    count : INTEGER;

FUNCTION do_something( param1 : INTEGER;
                       VAR param2 : REAL )
                     : INTEGER ; EXTERNAL;

BEGIN
```

```
    count := 0;
    REPEAT
        WRITELN( 'Enter an integer, or 0 to end' );
        READLN( input_value );
        IF input_value <> 0
        THEN
            BEGIN
            output_value := do_something( input_value, output_result );
            IF output_value <> 1
            THEN
                WRITELN( input_value, ' is an invalid value' )
            ELSE
                WRITELN( 'The result is ', output_result );
            count := count + 1
            END
    UNTIL input_value = 0;
    WRITELN( 'A total of ', count, ' values were processed' )
END.
```

If you apply the INSPECT command as follows:

```
INSPECT symbol=routine/CHARACTERISTICS=TYPE
```

The result follows.

```
DO_SOMETHING function has type mismatches
    DO_SOMETHING\1       FUNCTION declaration
    USING_CHARACTERISTICS\8
                         FORWARD or EXTERNAL FUNCTION declaration
   ARGUMENT2 argument is mistyped
    DO_SOMETHING\1       formal argument declaration
    USING_CHARACTERISTICS\9
                         declaration is mistyped
%SCA-S-OCCURS, 1 problem found (4 names, 4 symbols, 9 occurrences)
```

The problem in this example is that Pascal passes an F_FLOAT as the second parameter, while FORTRAN expects a D_FLOAT. When FORTRAN writes the new parameter out, it writes over an extra 4 bytes.

11.3.2 Severity Levels

The INSPECT command results contain diagnostic error messages of four severity levels: fatal, error, warning, and informational. Section 11.4 gives more information about the INSPECT command diagnostic error messages.

The /SEVERITY_LEVEL qualifier limits the depth of the INSPECT command checks to the minimum severity level you assign. Thus, you can effectively turn off the reporting of unwanted messages. The /SEVERITY_LEVEL qualifier has the following form:

```
INSPECT/SEVERITY_LEVEL=severity-level
```

The default value is /SEVERITY_LEVEL=INFORMATIONAL.

11.3.3 Error Limits

To avoid generating excessive numbers of diagnostic error messages when a single error affects many aspects of the software system, you can use the INSPECT command with the /ERROR_LIMIT qualifier to set global and local error limits. The /ERROR_LIMIT qualifier has the following form:

```
INSPECT/[NO]ERROR_LIMIT=(global-limit[,symbol-limit])
```

Global error limits set the total number of errors to be reported by a single INSPECT check. Once INSPECT reaches this limit, the check ends.

Local error limits set the number of errors pertaining to a single symbol. Once INSPECT reaches this limit, it stops reporting errors for this symbol. However, it will continue to check other elements of the software system.

The *symbol-limit* parameter is optional. The default is /NOERROR_LIMIT.

11.4 Diagnostic Error Messages

The results of the INSPECT command checks contain diagnostic error messages. Error messages are assigned a severity-level status based on the four DCL error severity categories, as follows:

- FATAL
- ERROR
- WARNING
- INFORMATIONAL

The diagnostic message examples in this section are based on the following program:

```
        PROGRAM prog3
        COMMON /common3/ a, i, b, j
        COMMON /common4/ e, f, g, h
        COMMON x, y, z
        INTEGER n
        INTEGER m
        a = 1.0
        b = 1.0
        i = 1
        j = 0
        e = 1.0
        f = 2.0
        g = 3.0
        h = 4.0
        n = 10
        m = 20
        CALL sub3( n, m )
        CALL pascal_proc

        TYPE *, i, j
        END

        SUBROUTINE sub3( n, m)
        INTEGER*2   n
        INTEGER*2   m
        COMMON /common3/ a, b, j, i
        COMMON /common4/ e, f, g, h
        COMMON x, y, z

        i = j * h + n
        j = i * f + m
        n = n + 1
        m = m + 1
        RETURN
        END
        SUBROUTINE sub4( n, m)
        INTEGER n
        INTEGER M

        n = m + 1
        m = n + 1
        RETURN
        END
MODULE severity_levels2( input, output );

PROCEDURE sub4( VAR n, m : [READONLY]integer ) ; EXTERNAL;
FUNCTION func1( VAR n, m : integer ) : INTEGER; EXTERNAL;

PROCEDURE pascal_proc;
    VAR
        n, m, result : integer;
    BEGIN
    n := 1;
    m := 2;
    sub4( n, m );
    writeln( 'N and M are ', n, ' and ', m );
    result := func1( n, m );
    writeln( 'Result is ', result );
    END;
END.

MODULE severity_levels3;

[ GLOBAL ]
FUNCTION func1( n, m :[ READONLY ]  integer ) : integer ;

    BEGIN
    func1 := n * m
    END;
END.
```

11.4.1 Fatal-Level Error Messages

Fatal-level error messages indicate an error that will probably cause the software system to fail when run. Fatal-level error messages are reported only when the TYPE option of the /CHARACTERISTICS qualifier is used.

The INSPECT command follows.

```
INSPECT common*/CHARACTERISTICS=TYPE
```

The result follows.

```
COMMON3 common block is mistyped
    PROG3\2                COMMON declaration
    SUB3\5                 COMMON declaration
   B variable is mistyped
    PROG3\2                variable declaration
    SUB3\5                 declaration is mistyped
   I variable is mistyped
    PROG3\2                variable declaration
    SUB3\5                 declaration is mistyped
%SCA-S-OCCURS, 2 problems found (2 names, 4 symbols, 4 occurrences)
```

11.4.2 Error-Level Error Messages

Error-level error messages indicate a condition that may cause serious problems within the software system when it is run. Error-level error messages are reported only when the TYPE option of the /CHARACTERISTICS qualifier is used.

The following INSPECT command checks PROC1 for any type mismatches and related errors. In this case, there are no real type mismatches, but there are a number of attribute mismatches on the parameters of the routine PROC1.

```
INSPECT func1/CHARACTERISTICS=TYPE
```

The result follows.

```
FUNC1 function has attribute mismatches
    SEVERITY_LEVELS2\4   FORWARD or EXTERNAL FUNCTION declaration
    SEVERITY_LEVELS3\4   FUNCTION declaration
   M argument has attribute mismatches
    SEVERITY_LEVELS2\4   declaration should not be an output parameter
                         declaration should be read-only
    SEVERITY_LEVELS3\4   formal parameter declaration
   N argument has attribute mismatches
    SEVERITY_LEVELS2\4   declaration should not be an output parameter
                         declaration should be read-only
    SEVERITY_LEVELS3\4   formal parameter declaration
%SCA-S-OCCURS, 5 problems found (1 name, 1 symbol, 3 occurrences)
```

11.4.3 Warning-Level Error Messages

Warning-level error messages indicate a condition that may cause problems within the software system when it is run. Warning-level error messages are reported when the USAGE or /TYPE options of the /CHARACTERISTICS qualifier are used.

The INSPECT command follows.

```
INSPECT sub3/CHARACTERISTICS=TYPE
```

The result follows.

```
SUB3 procedure is possibly mistyped
    PROG3\18             call reference
    SUB3\2               SUBROUTINE or PROGRAM declaration
   actual argument is possibly mistyped
    PROG3\18             declaration is possibly mistyped
    SUB3\2               formal argument declaration
   actual argument is possibly mistyped
    PROG3\18             declaration is possibly mistyped
    SUB3\2               formal argument declaration
%SCA-S-OCCURS, 2 problems found (1 name, 1 symbol, 3 occurrences)
```

11.4.4 Informational-Level Error Messages

Informational-level error messages indicate a condition that may cause maintenance problems within the software system. Error messages of this level show where system cleanup and streamlining may be done. Informational-level error messages are reported when the USAGE, IMPLICIT_DECLARATION, UNUSED_SYMBOL, and UNIQUE_NAME options of the /CHARACTERISTICS qualifier are used.

The INSPECT command follows.

```
INSPECT symbol=psect/CHARACTERISTICS=UNUSED
```

The result follows.

```
$BLANK common block has unused variables
    PROG3\4                COMMON declaration (hidden)
    SUB3\7                 COMMON declaration (hidden)
   X variable is never used
    PROG3\4                declaration is never used
    SUB3\7                 declaration is never used
   Y variable is never used
    PROG3\4                declaration is never used
    SUB3\7                 declaration is never used
   Z variable is never used
    PROG3\4                declaration is never used
    SUB3\7                 declaration is never used
%SCA-S-OCCURS, 3 problems found (3 names, 6 symbols, 6 occurrences)
```

11.5 Tailoring the INSPECT Command for Diverse Programming Styles

With the INSPECT command, you can tailor searches according to the programming language that you use and your special needs or situation. This section provides examples of tailoring in four categories as follows:

- Using severity levels to eliminate unwanted messages
- Using error limits to eliminate excessive messages
- Using the /CHARACTERISTICS qualifier to eliminate unwanted checks
- Using other techniques to eliminate unwanted diagnostics

The tailoring examples in this section are based on the following program:

```
PROGRAM prog5
COMMON /common4/ a, b, i, j
COMMON /common5/ e, f, g, h

a = 3.0
e = 1.0
f = 4.0
h = 0.0
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
CALL sub5( a, i )
END
SUBROUTINE sub5
COMMON /common4/ a, j, b, i
COMMON /common5/ e, f, g, h
```

```
j = j + 1
b = b * i
f = f + 1.0
g = 3.14159
h = h + 1.0
i = i - 1
RETURN
END
```

11.5.1 Using Severity Levels to Eliminate Unwanted Messages

By limiting the severity of diagnostics, you can avoid looking at relatively unimportant problems until after the serious problems are fixed. Also, some minor problems may be a result of the same errors that cause the type mismatches.

This section presents two ways of using the INSPECT command. The first is an example of using INSPECT without limiting the results by setting a minimum severity level. It finds all the problems with the common blocks in an SCA library. There are some type mismatches and several lesser errors that should be examined, but the large number of these lesser error messages obscure the more serious problems.

The first example of using the INSPECT command follows.

```
INSPECT symbol=psect
```

The result follows.

```
COMMON4 common block has problems
    PROG5\2              COMMON declaration
    SUB5\2               COMMON declaration
   A variable has usage problems
    PROG5\2              declaration is implicit
                         declaration may never be read
    SUB5\2               declaration is implicit
                         declaration may never be read
   B variable has fatal errors
    PROG5\2              declaration is implicit
    SUB5\2               declaration is mistyped
                         declaration is implicit
                         declaration has the wrong name
    SUB5\5               reference has the wrong name
    SUB5\5               reference has the wrong name
   I variable has fatal errors
    PROG5\2              declaration is implicit
    SUB5\2               declaration is mistyped
                         declaration is implicit
                         declaration has the wrong name
    SUB5\6               reference has the wrong name
    SUB5\6               reference has the wrong name
   J variable has warnings
    PROG5\2              declaration is implicit
    SUB5\2               declaration is implicit
                         declaration has the wrong name
    SUB5\6               reference has the wrong name
    SUB5\10              reference has the wrong name
    SUB5\10              reference has the wrong name
COMMON5 common block has usage problems
    PROG5\3              COMMON declaration
    SUB5\3               COMMON declaration
   E variable has usage problems
    PROG5\3              declaration is implicit
                         declaration is never read
    SUB5\3               declaration is implicit
                         declaration is never read
   F variable is implicit
    PROG5\3              declaration is implicit
    SUB5\3               declaration is implicit
   G variable has usage problems
    PROG5\3              declaration is implicit
                         declaration is never read
    SUB5\3               declaration is implicit
                         declaration is never read
   H variable is implicit
    PROG5\3              declaration is implicit
    SUB5\3               declaration is implicit
%SCA-S-OCCURS, 31 problems found (2 names, 4 symbols, 4 occurrences)
```

The second example shows the same INSPECT command operating on the same program, but with severity levels limited to tailor the result so that the most important information is reported. The command follows.

```
INSPECT symbol=psect/SEVERITY=ERROR
```

The result follows.

```
COMMON4 common block is mistyped
    PROG5\2               COMMON declaration
    SUB5\2                COMMON declaration
   B variable is mistyped
    PROG5\2               variable declaration
    SUB5\2                declaration is mistyped
   I variable is mistyped
    PROG5\2               variable declaration
    SUB5\2                declaration is mistyped
%SCA-S-OCCURS, 2 problems found (2 names, 4 symbols, 4 occurrences)
```

11.5.2 Using Error Limits to Eliminate Excessive Messages

Another way to reduce the size of the output is to set an error limit. The INSPECT command stops checking after a certain number of errors are found. You can set either global error limits to specify the maximum number of errors overall, or local error limits to specify the number of errors allowed for each symbol.

The following command sets the global limit at 10 and the local limit at 1.

```
INSPECT symbol=psect/ERROR_LIMIT=(10,1)
```

The result follows.

```
COMMON4 common block has exceeded an error limit
    PROG5\2               COMMON declaration
   A variable is implicit
    PROG5\2               declaration is implicit
   B variable is implicit
    PROG5\2               declaration is implicit
   I variable is implicit
    PROG5\2               declaration is implicit
   J variable is implicit
    PROG5\2               declaration is implicit
COMMON5 common block has exceeded an error limit
    PROG5\3               COMMON declaration
   E variable is implicit
    PROG5\3               declaration is implicit
   F variable is implicit
    PROG5\3               declaration is implicit
   G variable is implicit
    PROG5\3               declaration is implicit
   H variable is implicit
    PROG5\3               declaration is implicit
%SCA-S-OCCURS, 8 problems found (2 names, 4 symbols, 4 occurrences)
```

11.5.3 Using the /CHARACTERISTICS Qualifier to Eliminate Unwanted Checks

With an undifferentiated INPSECT command, the results may contain several diagnostic error messages about elements of your software system that result from individual coding practices, but which may not be errors, such as implicit declarations. These results may contain useless information, as shown in the example at the beginning of Section 11.5.

You can use the /CHARACTERISTICS qualifier to limit the INSPECT check, as follows:

```
INSPECT symbol=psect/CHARACTERISTICS=(ALL,NOIMPLICIT)
```

The result follows.

```
COMMON4 common block has fatal errors
    PROG5\2                COMMON declaration
    SUB5\2                 COMMON declaration
   A variable may never be read
    PROG5\2                declaration may never be read
    SUB5\2                 declaration may never be read
   B variable has fatal errors
    PROG5\2                variable declaration
    SUB5\2                 declaration is mistyped
                           declaration has the wrong name
    SUB5\5                 reference has the wrong name
    SUB5\5                 reference has the wrong name
   I variable has fatal errors
    PROG5\2                variable declaration
    SUB5\2                 declaration is mistyped
                           declaration has the wrong name
    SUB5\6                 reference has the wrong name
    SUB5\6                 reference has the wrong name
   J variable has the wrong name
    PROG5\2                variable declaration
    SUB5\2                 declaration has the wrong name
    SUB5\6                 reference has the wrong name
    SUB5\10                reference has the wrong name
    SUB5\10                reference has the wrong name
COMMON5 common block is never read
    PROG5\3                COMMON declaration
    SUB5\3                 COMMON declaration
   E variable is never read
    PROG5\3                declaration is never read
    SUB5\3                 declaration is never read
   G variable is never read
    PROG5\3                declaration is never read
    SUB5\3                 declaration is never read
%SCA-S-OCCURS, 15 problems found (2 names, 4 symbols, 4 occurrences)
```

11.5.4 Using Other Techniques to Eliminate Unwanted Diagnostics

You can use the INSPECT command with both the /CHARACTERISTICS and /ERROR_LIMIT qualifiers to reduce the number of diagnostic error messages reported.

To begin, consider the example at the beginning of Section 11.5.

This INSPECT command produces several diagnostics, making it likely that you may miss the type mismatches. You cannot control this by limiting the severity levels reported because you would then not get the diagnostic messages relating to using the incorrect name that indicate a serious problem in this case. You could eliminate the implicit declarations by using the /CHARACTERISTICS qualifier, but you would still get several messages about incorrect names, and it would be preferable to have only a few.

You can use the /CHARACTERISTICS qualifier and /ERROR_LIMIT qualifiers, as follows, to reduce the number of diagnostic messages reported, without compromising your ability to find real problems.

```
INSPECT symbol=psect/CHARACTERISTICS=(ALL,NOIMPLICIT)/ERROR_LIMTI=(20,3)
```

The result follows.

```
COMMON4 common block has fatal errors
    PROG5\2               COMMON declaration
    SUB5\2                declaration has exceeded an error limit
   B variable has fatal errors
    PROG5\2               variable declaration
    SUB5\2                declaration is mistyped
                          declaration has the wrong name
    SUB5\5                reference has the wrong name
    SUB5\5                reference has the wrong name
   I variable has fatal errors
    PROG5\2               variable declaration
    SUB5\2                declaration is mistyped
                          declaration has the wrong name
    SUB5\6                reference has the wrong name
    SUB5\6                reference has the wrong name
   J variable has the wrong name
    PROG5\2               variable declaration
    SUB5\2                declaration has the wrong name
    SUB5\6                reference has the wrong name
    SUB5\10               reference has the wrong name
    SUB5\10               reference has the wrong name
COMMON5 common block is never read
    PROG5\3               COMMON declaration
    SUB5\3                COMMON declaration
   E variable is never read
    PROG5\3               declaration is never read
    SUB5\3                declaration is never read
   G variable is never read
    PROG5\3               declaration is never read
    SUB5\3                declaration is never read
```

```
%SCA-S-OCCURS, 14 problems found (2 names, 4 symbols, 4 occurrences)
```

Many more combinations of the INSPECT command qualifiers are possible. The use and results of these checks depend on your software system and your own programming style.

Part 3 Designing Programs

This part contains tutorial information on using LSE, SCA, and VMS compilers to design programs.

Chapter 12

Using LSE and SCA to Design Programs

This chapter provides a scenario of how you might generate a detailed program design. This is only a guideline, which takes you through the stages of generating a detailed design. In addition, this chapter describes how to create and process the design. It also shows how to evolve an implementation from this design, and shows how to reverse-engineer the implementation to retrieve a design corresponding to the original.

Section 12.1 is an introduction. Section 12.2 describes how to use LSE to create the design. Section 12.3 describes how to use the VMS compilers and SCA to process the design. Section 12.4 describes how to analyze designs. Section 12.5 shows how to store design information in tagged comments and how to define new tags and keyword lists. Section 12.6 describes how to generate design reports, and Section 12.7 describes how to reverse-engineer designs.

12.1 Introduction

In many software engineering environments, the last step before actual coding is generating a detailed design. Frequently, a Program Design Language (*PDL*) is used for that purpose. In the VAX/VMS environment, you create detailed designs as follows:

- Use traditional programming languages
- Embed design information in comments
- Write algorithms with pseudocode placeholders

The language you use for your implementation can be your Program Design Language.

Once written, you can process and analyze designs to produce a variety of design reports. You can reverse-engineer existing code to create a design report that describes the design of the code as actually implemented.

Definitions for a detailed design vary, but detailed designs usually include the following:

- A specification of module organization
- Global interfaces
- Global data and data types
- Outlines of crucial algorithms

Since design is an ongoing, iterative process, there are no rules for determining when a design is complete, or for which pieces of a design must be specified. In the VMS environment, designs consist of one or more modules, in one or more of the available languages. Within each module, there is great flexibility concerning how much must be fully specified and how much can be left as pseudocode.

12.2 Creating Designs

When you create a design for a single module that contains two routines, such a design is likely to identify some general information about the module, such as name, purpose, global data, design issues, and so on. Similarly, it will identify the routines, their purposes, basic algorithms, and possibly parameters, return values, and other design information.

The following example shows how to generate a design, using Ada as the base language. Use the following steps:

1. Invoke LSE to create a new Ada file.

   LSE creates a file containing the single placeholder *{compilation_unit}*.
2. Expand the placeholder and choose the package body.
3. Expand the header comment and start filling it in.

The following example shows how it might appear:

```
-- ++
-- FACILITY:
--
--          Sample facility 1
--
-- ABSTRACT:
--
--      This package is a sample package used to illustrate the way you
--      use LSE and SCA to create a detailed design.
--
-- AUTHORS:
--
--      Dave Ang
--
-- CREATION DATE: 4 July 1989
--
-- DESIGN ISSUES:
--
--      This is a sample design.  There is one module, which contains two
--      routines and one global data declaration.
--
--      To illustrate the various levels of design that are possible, you
--      can expand one of the routines in some detail, while leaving the other
--      routine at a very abstract level.
--
-- KEYWORDS:
--
--      Examples, sample design
--
-- MODIFICATION HISTORY:
--
-- --
[context_clause]...
{package_body}
```

This example shows how you can use comment tags for design information. Most tags contain ordinary text that describes specific pieces of the design. Here, keyword tags are used to express relationships and associations. The keyword tag FACILITY indicates that this module is part of *Sample facility 1*. In addition, the keyword tag KEYWORDS is used to associate the terms *examples* and *sample design* with this module.

If you establish appropriate conventions for such tags, you can use SCA for queries such as “find all packages that belong to a particular facility” or “find all packages that have to do with examples.” For any given project, there will probably be tags that are specific to that project. Section 12.5.2 discusses how to add new tags.

To write the outline of the package body, expand the *{package_body}* placeholders to provide skeletons for the following:

- Type declaration
- Function body declaration
- Procedure body declaration

The following results:

```
package body first_module is
    type {identifier} ([discriminant_part]) is {type_definition};

    function function_1 ([formal_part]) return {type_mark} is
    -- [function_header_comment]
        [declarative_part]
    begin
        [statement]...
        return {expression};
    [exception_part]
    end function_1;

    procedure procedure_2 ([formal_part]) is
    -- [procedure_header_comment]
        [declarative_part]
    begin
        {statement}...
    [exception_part]
    end procedure_2;
begin
    {statement}...
end first_module;
```

The next section shows how to expand these placeholders to obtain useful routine designs.

12.2.1 Designing Routine Declarations

To create designs for individual routines, you expand LSE placeholders as necessary. Use tags and pseudocode placeholders to contain design information that is still at an abstract level, and use actual language constructs for those portions of the algorithm that are known.

If, for example, the design has only a few details, it might appear as follows:

```
function function_1 (
    P1 : in P1_type;
    P2 : in P2_type := null_P2)
        return integer is
-- ++
-- FUNCTIONAL DESCRIPTION:
--
--      This function computes the integer function of the P1, with
--  or without P2s.
--
-- FORMAL PARAMETERS:
--
--      P1:
--
--          The P1 whose function we want.
--
--      P2:
--
--          The P2 to involve with the P1.
--
-- RETURN VALUE:
--
--      The computed function.
--
-- ALGORITHM:
--
--      Use the regular function algorithm if the P2 is
--      present, and use Murphy's function algorithm if
--      it isn't.
--
-- [logical properties]
--
-- [optional subprogram tags]
-- --
    [declarative_part]
begin
    [statement]...
    return {expression};
[exception_part]
end function_1;
```

Much of the calling sequence has been specified. This information is useful to people who use this function. Thus far, this example shows how you can use the ALGORITHM tag to describe the top layer of the algorithm in ordinary English. Later, you can use pseudocode to describe the algorithm. Other placeholders are left in place, as they will also be expanded as work progresses.

To complete the algorithm design, use the ENTER PSEUDOCODE command to write the algorithm design.

The following example shows only the routine body:

```
    partial_function,              -- Used to store the partial
                                   -- results from Murphy's algorithm
    final_function : integer;      -- Used to store the final result
begin
    if «the P2 is present» then ❶
       «Use the standard algorithm» ❷
    else
       «Use Murphy's algorithm»
       final_function :=
       fix_partial_function(partial_function); ❸
    end if;
    [statement]... ❹
    return final_function;
end function_1;
```

Line ❶ uses pseudocode as the conditional expression in the *if* statement.

Line ❷ uses pseudocode to represent the entire body of the *then* clause of the statement.

Line ❸ shows a procedure call. The procedure specification (not shown) must also be present for Ada to recognize this procedure call. Subsequently, you will be able to use SCA to get information about calls to this routine, including this call.

Line ❹ contains an ordinary LSE placeholder. You can use placeholders as part of a design in any context in which they normally appear. In this example, the *[statement]...* placeholder remains as a convenience because the algorithm is not yet complete.

12.2.2 Refining the Design

As the design is refined, more details may be filled in. To preserve the original design information, use the ENTER COMMENT command. This applies both during the low-level design phase and during the implementation phase.

For example, the *if* statement in the previous example might be refined as follows:

```
if P2 /= null_P2 then       -- the P2 is present ❶
    -- Use the standard algorithm  ❷
    [loop_identifier]: loop
        «Calculate function iteratively from P2»
    end loop;
    {tbs}
else
    «Use Murphy's algorithm»
    final_function := fix_partial_function(partial_function);
end if;
```

On line ❶ the ENTER COMMENT/LINE command is used to move the pseudocode for the *if* statement over to the right, before writing the actual condition.

On line ❷ the ENTER COMMENT/BLOCK comment is used to turn the pseudocode placeholder into a block comment before writing the first statement, which is a *loop* statement. The ENTER COMMENT/BLOCK command produces the *{tbs}* placeholder.

12.2.3 Designing Data Declarations

The design of data structures is an important part of a detailed design. If the design calls for an array of records to be shared between the two routines, but not visible outside the package, such an array would be declared in the package body, before the declaration of the two subprograms. Furthermore, if the design has specified only a few of the fields of the record and has not specified the length of the array, the design would appear as follows:

```
type record_type is
    record
        count : integer := 0;
        record_name : string({discrete_range}...);❶
        subfield_1 : «A type suitable for subfield 1»;❷
        «subfield 2, which has property x»❸
        [component_declaration]...
        [variant_part]
    end record;
shared_array : array ({discrete_range}...) of record_type;
```

In this example, LSE placeholders are used several ways. On line ❶ LSE generates the placeholder *discrete_range*.

Line ❷ shows a pseudocode placeholder. This is created using the ENTER PSEUDOCODE command, and then typing the contents.

Line ❸ shows a pseudocode placeholder, which describes the next field of the record in general terms.

Two important points concerning pseudocode placeholders are illustrated by this example.

- The contents of a pseudocode placeholder typically has whatever information is available to describe the object. The level of detail and the nature of the information varies from design to design.
- Whenever possible, it is preferable to fill in as much detail of the design as possible in the native language.

The previous example could have been declared as follows:

```
type record_type is
    «a complicated record definition»;
```

While this format may seem to have the same information, it actually suppresses information that could be parsed by the compiler and entered in your SCA database. For instance, in this case, the compiler does not recognize the type definition as a record definition and will not be able to do as much design checking later. This would make subsequent review of your design more difficult. Of course, if the nature of the high-level design makes it improper to make decisions, such as the names of the field, then it may be appropriate to use the natural language description in pseudocode. The choice depends upon the particular goals of the low-level design.

12.3 Processing Designs

Once there is a partial or complete design, you can process the design by using a VMS compiler and the VAX Source Code Analyzer.

With all of the VMS compilers that support SCA Version 2.0, you use the /DESIGN qualifier to tell the compiler to process design information. This qualifier takes two keyword values, as follows:

- [NO]COMMENT

  This tells the compiler to search inside comments for program design information.
- [NO]PLACEHOLDERS

  This tells the compiler to recognize placeholders as valid program syntax.

To process the previous design, you could type the following command:

```
$   ADA first_module/DESIGN=(COMMENT,PLACEHOLDERS)/ANALYSIS_DATA
```

Because the default keyword values for the /DESIGN qualifier are (COMMENT,PLACEHOLDERS), you could also type the following:

```
$   ADA first_module/DESIGN/ANALYSIS_DATA
```

12.3.1 Loading Design Information into an SCA Library

To load analysis data files created with the /DESIGN qualifier into an SCA library, you use the VAX Source Code Analyzer LOAD command. From the point of view of SCA, there is no difference between an analysis data file containing design information and one containing pure code. If a design evolves directly into an implementation, you can use the same arrangement of libraries during design as during implementation.

To preserve your design as a fixed reference point while continuing implementation, you can set up your SCA libraries to keep design information in one file and the implementation in another file. With SCA, you can use a list of individual SCA libraries as your current virtual library. If a module appears in more than one library in the list, the first instance of the module occludes subsequent instances. Thus, you can set up your SCA libraries so that modules being implemented occlude their designs. For those modules that have not been converted to code, the designs are still available. For example:

```
$   SCA SET LIBRARY [user.code.sca_library],[project.code.sca_library], -
    [user.design.sca_library],[project.design.sca_library]
```

To refer to both the code and the designs from SCA at the same time, you have two options.

- You can choose a naming convention at the module level to distinguish between the design of a module and its code. This is necessary because SCA allows only one module of a given name in any virtual library; any other modules are occluded.
- You can switch back and forth, using the VAX Source Code Analyzer SET LIBRARY command.

12.4 Analyzing Designs

Once the analysis data files from a design are loaded into an SCA library, you can use SCA queries to retrieve information, as with any other SCA library. There are a number of symbol classes defined by SCA specifically for design information, such as keyword, placeholder, and tag. To get the indicated design information, you use these classes with the SYMBOL= construct.

For example, if you want to find all routines that are marked with the keyword *interface*, you could use the following SCA query:

```
$  SCA FIND CONTAINED_BY(SYMBOL=routine, 'interface' AND -
SYMBOL=KEYWORD, DEPTH=1)
```

12.5 Expressing Design Information in Comments

You can capture much of a detailed design by using pseudocode placeholders; however, a significant amount of information is expressed using **tagged comments**. With LSE, you can easily enter tagged comments into your programs. The templates for LSE include a standard set of comment tags. In addition, you can change these tags or add new tags.

When programs are compiled with the /DESIGN=COMMENTS and /ANALYSIS_DATA qualifiers, the compiler performs the following:

- Scans the contents of comments
- Parses tags and their values
- Inserts relevant data about those comments into the SCA analysis file

This information can then be retrieved by SCA and matched with corresponding identifiers, such as routine names that appear in the code, and used to generate design reports.

12.5.1 Using Tagged Comments

Tagged comments are based upon a simple structure. Each comment is treated as a sequence of (tag, tag value) pairs. You define tags in LSE and save the definitions in an LSE environment file, which is read by the compiler. Default tags are in the LSE$SYSTEM_ENVIRONMENT file, where they are also available to compilers. There are several types of tags, and the value of the tag is parsed differently depending on the tag type.

There are also a number of special case tags, each of which begins with a dollar sign ($).

As the compiler scans comments, it groups the comments into **comment blocks**. Comment blocks are separated either by code (any visible text that is not contained in a comment) or by a totally blank line (any blank line that is not contained in a comment). Within each comment block, the compiler scans the text of the comment line by line, looking for tags. In order to be recognized, the tag must be the first text on the line of the comment, not counting the comment delimiters. Furthermore, the tag must either be the only text on the line, or the tag must be terminated by a colon or hyphen. Anything after the tag, either on the same line or on subsequent lines, forms the value of the tag, up to but not including the next tag found.

There are three types of tags: **text**, **keyword**, and **structured**.

Text tags contain ordinary text and are the most common type of tag. No further processing or special scanning is done on the value of a text tag.

Keyword tags contain a list of zero or more keywords that are used to flag sections of code. Any given keyword tag may be defined so as to accept keywords from a predefined list, which in turn is defined with the DEFINE KEYWORD command, or may take arbitrary keywords. In either case, keywords are separated by commas, and may contain space characters. The keywords are scanned by the compiler, and each keyword is stored in the SCA analysis file, making subsequent retrieval easy.

Structured tags add a second level of structure to the tag. The value of a structured tag consists of a sequence of one or more subtags. For example, the FORMAL PARAMETERS tag consists of a sequence in which each parameter name is a subtag, and the description of the parameter is the value of the subtag. Unlike ordinary tags, subtags need not be predefined. To be recognized, subtags must conform to the following rules:

- Each subtag must be preceded by a blank comment line.
- The subtag must be indented at least as much as the structured tag to which it belongs.
- The subtag *must* be terminated by a colon or hyphen.

To make sure that the next tag after the structured tag is properly recognized as a new tag, and not as a subtag or the value of a subtag, it too must conform to the following special rules:

- It must be preceded by a blank comment line.

- One of the following must hold:
  - Indented less than the previous subtag and less than or equal to the indentation of the previous tag, or
  - Terminated with a punctuation character different from the one used to terminate the last subtag

Two implicit tags are defined for all languages. These are the $UNTAGGED tag and the $REMARK tag. The $UNTAGGED tag refers to any comment text that occurs at the beginning of a comment block, before the first tag of the comment block is found.

For example:

```
function function_1 (...)
--
-- This function computes the integer function of the P1,
-- with or without P2s.
--
-- FORMAL PARAMETERS:
...
```

The text *This function computes the integer ...* would be the value of the $UNTAGGED tag, because no tag name precedes it in the comment block.

The $REMARK tag is the first line of text in the comment block, not counting any tag names. In the previous example, the $REMARK string would be *This function computes the integer function of the P1,*. You use the $REMARK tag for cases where only a single line of text is required. They are especially useful in sequences of variable declarations, which frequently look like the following:

```
v1,               -- remark for v1
v2,               -- remark for v2
...
vN :INTEGER;      -- remark for vN
```

12.5.2 Adding New Tags and Keyword Lists

You can use user-defined tags to represent various kinds of design information. To define new tags, use the DEFINE TAG and DEFINE KEYWORDS commands. To save tag definitions in an environment file, use the SAVE ENVIRONMENT command. To tell the compiler about the tag definitions, define the logical name LSE$ENVIRONMENT to include the environment file (LSE$ENVIRONMENT can be a search list). Then these tags are available when compiling programs with the /DESIGN qualifier.

For example, to label each module with a list of requirements that are satisfied by this module, type the following commands:

```
DEFINE TAG requirements/TYPE=KEYWORD/KEYWORDS=requirement_list/LANGUAGE=ADA

DEFINE KEYWORDS requirement_list
    "Requirement 1"
    "Requirement 2"
    "Requirement 3"
END DEFINE
```

Now you can save these definitions in an environment file by typing the following command:

```
LSE>  SAVE ENVIRONMENT/NEW MYDISK:[MYDIRECTORY]MYTAGS
```

The /NEW qualifier tells LSE to save only the new definitions that you have added during the current editing session. This creates a file called MYDISK:[MYDIRECTORY]MYTAGS.ENV. To have the compilers and LSE use this file, type the following DCL command:

```
$   DEFINE LSE$ENVIRONMENT MYDISK:[MYDIRECTORY]MYTAGS.ENV
```

You can now use the /DESIGN qualifier to compile your program.

12.5.3 Associating Tags with Objects

You can use tagged comments to associate design information with objects in your program. They are meaningful only when used in conjunction with declarations. Tagged comments that occur in executable portions of your code, where there are no adjacent declarations, are not used for design reports.

To find the association between tags and objects, use the SCA containment functions, CONTAINING and CONTAINED_BY. See the appendix that describes SCA query expressions in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details. To find the FUNCTIONAL DESCRIPTION of routine R1, for example, you could specify the following SCA command:

```
        FIND CONTAINED_BY ( -
                R1 AND SYMBOL=ROUTINE, -
                "FUNCTIONAL DESCRIPTION" AND SYMBOL=TAG, -
                1)
```

Since, in some languages, routines can contain other routines, it is important to specify the DEPTH parameter in the CONTAINED_BY function as 1. There are two important exceptions to this.

- The $REMARK tag is always contained inside another tag, typically inside the $UNTAGGED tag. Hence it will be at DEPTH=2.
- Subtags of structured tags are contained inside the structured tag. Therefore, subtags will be at DEPTH=2 with respect to the associated object, and DEPTH=1 with respect to the structured tag that contains the subtags.

Because the SCA containment functions can be slow for depths greater than 1, you should use only DEPTH=2 when necessary, that is, when you know that you are in one of these situations.

Sometimes tagged comments are not strictly nested inside a declaration. For example, a common formatting style for the C programming language is to put the comment block for a function in front of the function declaration. A strict interpretation of containment would imply that the function declaration does not contain the comment block. To solve this problem, SCA implicitly extends the lexical range of definitions so that they include comment blocks that are adjacent to those definitions.

SCA generally looks for the closest declaration that is immediately adjacent to the comment block. If the code fragments on both sides of the comment block are not part of declarations, then no comment association is done by SCA. In that case, the comment is simply contained in whatever outer-level declaration contains the comment, if any.

This comment association can sometimes be ambiguous. For example, suppose you had the following C fragment:

```
int x;
/* This comment describes a variable */
int y;
```

Then the declarations of x and y would both be adjacent to the comment. You can control this explicitly by putting in blank lines to create the association you want. For example:

```
int x;
/* This comment describes a variable */

int y;
```

This results in the comment being associated with x.

```
int x;

/* This comment describes a variable */
int y;
```

This results in the comment being associated with y.

If you leave an ambiguous situation in your code, SCA uses the setting of the /COMMENT=(ASSOCIATED_IDENTIFIER=keyword) qualifier on the LSE command DEFINE LANGUAGE. (See the entry for DEFINE LANGUAGE in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details on the syntax of this qualifier.)

The ASSOCIATED_IDENTIFIER=keyword qualifier is subtle. SCA does not use the current value of the qualifier when you run SCA from within LSE. Rather, when you use the /DESIGN=COMMENTS qualifier to compile your source program, the compiler uses your LSE$ENVIRONMENT file and the LSE$SYSTEM_ENVIRONMENT file to determine the setting of the /COMMENT=(ASSOCIATED_IDENTIFIER=keyword) qualifier. That setting is stored in your .ANA file. SCA performs the comment association, using that setting, at the time you load the file into the SCA library. If you wish to change the setting of that qualifier, you must change the setting in LSE, save a new LSE environment file, recompile your program, and load the new .ANA file into SCA.

12.6 Generating Design Reports

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

You generate reports with the SCA command REPORT. The REPORT command requires both SCA and LSE because the reports are actually implemented in TPU code. REPORT uses the SCA callable interface that is available in TPU code running under LSE. The REPORT command cannot use the TPU that is supplied with VMS because that version does not have the SCA callable interface.

You can customize reports by modifying the TPU code or writing new code. See Chapter 18 for information on customizing reports.

12.6.1 Using Design Report Formats

You use the REPORT command both for reports provided by Digital and for customized reports that you have created. The REPORT command takes the following form:

```
LSE>   REPORT report_name
```

The reports provided by Digital are as follows:

- HELP—A VMS Help file generated from your design or code.
- PACKAGE—An LSE package definition.
- INTERNALS—A general report that describes your entire design in an organized manner.
- 2167A_DESIGN–A report that produces a document that meets the requirements of the U.S. Defense Department's DOD-STD-2167A Software Design Document.

The output of the REPORT command is typically not in its final state. For example, HELP reports must be loaded by the VMS librarian into a help library, and PACKAGE reports must be executed by LSE to produce package definitions. With INTERNALS and 2167A reports, you can produce reports that can be read in three different ways: directly, with VAX Document, or with Digital Standard Runoff. You get more power by using either VAX Document or Digital Standard Runoff.

Since reports typically perform many SCA queries over your SCA library, they can be time-consuming. For this reason, Digital recommends that you use the REPORT command from batch jobs. However, when customizing reports, you should use a small SCA library for testing purposes. You should debug these reports by executing them from within LSE, and by using TPU features to help with your debugging.

You make reports work by building an SCA query that represents the files in your system. To extract the data for the report, it steps through the files one at a time and steps through the routines within each file one at a time. Most of the data in reports is taken directly from the appropriate comment tags in your program. Certain significant data is based on properties of your code, such as the parameters to a routine. Reports are designed to accept a variety of synonymous tags for specific sections of reports. For example, the FORMAL PARAMETERS and FORMAL ARGUMENTS tags are treated as synonyms.

The reports provided by Digital use tags that are included in the system environment file supplied with LSE. You can use the SHOW TAGS command to show the tags for a particular language.

An important convention obeyed by these tags is that the tags that are applicable for an entire file or module are distinct from the tags that are applicable for a single subroutine. For example, the ABSTRACT tag describes a module, while the FUNCTIONAL DESCRIPTION tag describes a subroutine or function. This convention makes it easier for the report tool to distinguish between the two levels of tag information.

Because there are so many tags, not all of them are actually used by reports, so that the reports do not become unwieldy. You can customize reports to include tags of interest to you, and you can add new tags in addition to the tags supplied by Digital. See Chapter 18 for details on customizing reports.

The default domain for reports is the set of all files that have command line references in your SCA library, as follows:

```
FIND SYMBOL=FILE AND OCCURRENCE=COMMAND_LINE
```

To limit reports to specific files in your system, use the following steps:

1. Determine an SCA query that represents the specific files.
2. Perform the query, and give it a name by using the FIND/NAME command.
3. Use the query name as the domain for the report.

The following example limits the report to just those files that contain the string "matrix" as part of the file name.

```
FIND/NAME=myquery -
    *matrix* AND SYMBOL=FILE AND OCCURRENCE=COMMAND_LINE

REPORT/DOMAIN=myquery report_name
```

Reports are driven by the source files, and this limits the ability of the reports to present information that is not explicit in your source files. For example, if a routine declaration or comment block crosses a file boundary by including another file, then the report may behave unpredictably. In addition, declarations that are generated by macros or preprocessors are not processed by the reports provided by Digital. Declarations that occur in precompiled files, such as BLISS library files or Pascal environment files, will show up in the report for the precompiled file—not for the source files that use the precompiled file.

An additional restriction is that your SCA library must reflect the current state of your source files. Otherwise, the report tool will be unable to locate the tags in your source files. In many cases, you can customize reports to solve particular problems of this nature.

12.6.2 Creating Online HELP

The HELP report produces an .HLP file, suitable for loading by the VMS Librarian into a standard VMS help library. See the *VMS Librarian Utility Manual* for information on help libraries. The default output file name for the HELP report is HELP.HLP. You can change this by using the /OUTPUT qualifier on the REPORT command to specify a different file name.

The HELP report recognizes the HELP and HLP target types, both of which result in .HLP files. Since this is the default, there is no need to specify the /TARGET qualifier when using the HELP report supplied by Digital, unless you have added customizations for different targets.

For each file in the domain, the HELP report creates a top-level entry for the file. The help information for that entry is taken verbatim from the module description tag for the file. (The tags MODULE DESCRIPTION, PROGRAM DESCRIPTION, PACKAGE DESCRIPTION, and ABSTRACT are considered synonyms for this purpose.) Then, for each routine in the file, a level 2 entry is created. Again, the help text is taken from a tag (in this case, the FUNCTIONAL DESCRIPTION tag) for the routine. Finally, level 3 entries are created for the parameters of the routine, with the text from the comment associated with the parameter or the text from the appropriate subtag of the FORMAL PARAMETERS tag used as the help text. (The tags FORMAL PARAMETERS, FORMAL ARGUMENTS, PARAMETERS, and ARGUMENTS are considered synonyms.)

The /FILL qualifier is not meaningful for the HELP report. All output text in the help report is copied verbatim from tags in your program, with no filling or justification.

12.6.3 Creating LSE Package Definitions

The PACKAGE report produces an .LSE file, suitable for execution by LSE to define LSE packages for your program. (See Section 5.5 for information on packages.) The default output file name for the PACKAGE report is PACKAGE.LSE. You can change this by using the /OUTPUT qualifier on the REPORT command to specify a different file name.

The PACKAGE report recognizes only the LSE target type. Since this is the default, there is no need to specify the /TARGET qualifier when using the PACKAGE report.

For each file in the domain, the PACKAGE report creates an LSE DEFINE PACKAGE command. It then generates a DEFINE ROUTINE command for each routine in the file and DEFINE PARAMETER commands, as appropriate. The description string on the DEFINE ROUTINE command is the $REMARK string associated with the routine. The /TOPIC string for the DEFINE PACKAGE command is the name of the package, while the /TOPIC string for each DEFINE ROUTINE is the name of the routine.

The PACKAGE report uses two additional qualifiers. The /HELP_LIBRARY qualifier specifies the name of the help library to use for the DEFINE PACKAGE commands created by the report. The /LANGUAGES qualifier specifies the languages to use for the DEFINE PACKAGE command.

The /FILL qualifier is not meaningful for the package report.

12.6.4 Creating INTERNALS Reports

The INTERNALS report is a comprehensive report on the design of your system, on a module-by-module, routine-by-routine basis. The INTERNALS report extracts information from tags contained in comments to describe the various aspects of your program. For example, information under the FUNCTIONAL DESCRIPTION tag is used to describe each routine, while information under the RETURN VALUE tag is used to describe the return value of each routine. The INTERNALS report also uses the LSE overview mechanism to present the code of each routine in a structured, top-down way.

Three targets are recognized by the INTERNALS report. These targets are as follows:

- DOCUMENT—This is the default target. This outputs an .SDML file suitable for processing by VAX DOCUMENT.
- RUNOFF—This outputs an .RNO file suitable for processing by Digital Standard Runoff (DSR).
- TEXT—This outputs a .TXT file that you can read directly.

The default file name in all three cases is INTERNALS, with the default file type being determined from the target type. For example, if you want to produce an INTERNALS report that can be processed by VAX DOCUMENT, type the following command:

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

When you process the resulting file with VAX DOCUMENT, you must use the SOFTWARE.REFERENCE doctype, as follows:

```
$   DOCUMENT INTERNALS.SDML SOFTWARE.REFERENCE destination
```

The /FILL qualifier is important for INTERNALS reports. In cases where text tags are copied into the report, the /FILL qualifier determines whether or not the text will be filled. Use /NOFILL if your comments typically contain tables or other formatted output that should not be filled.

For each file in the domain, the INTERNALS report creates a chapter in the output file. The chapter contains the following:

- Description of the file or module, taken from the ABSTRACT or MODULE DESCRIPTION tags
- Sections that describe the global objects of the module, such as imported variables and exported variables
- A section on each routine

  The format of each routine section is similar to the format of routines in the *VMS Run-Time Library Routines Volume*. That is, each routine section has a title, a brief description of the routine (taken from the $REMARK tag for the routine), a sample invocation, a more complete description (taken from the FUNCTIONAL DESCRIPTION tag), sections for the other tags in the comment block for the routine, and the body of the routine.

The body of the routine is presented in a top-down, hierarchical fashion, using overviews to hide details at the upper layers, and proceeding until the entire body has been produced. For each overview placeholder that appears in the body, there is a cross-reference number (white-on-black callout for DOCUMENT output; boldface for RUNOFF output) to the expansion corresponding to that placeholder. An example of the output for a routine in the INTERNALS report is presented in Section 12.7.

12.6.5 Creating 2167A Software Design Reports

You can use the REPORT command to automatically create the body of a report that conforms to the requirements of the Software Design Document that is specified by MIL-STD-2167A. The report tool creates the design section, which is Section 4 of the 2167A Software Design report. You can include these output files in your complete Software Design Report, as follows:

- Use the VAX DOCUMENT <INCLUDE> or <ELEMENT> tags for VAX DOCUMENT reports.
- Use the .REQUIRE directive for Digital Standard Runoff (DSR) reports.
- Manually merge the output of the report tool with other text, for text reports.

Sample template files for the top levels of these reports are included in the SCA$2167A directory, as follows:

```
2167A_PROFILE.SDML
2167A_PROFILE.RNO
```

The PROFILE files use the appropriate commands to include the lower-level files in the report. This examples directory also contains stub files for each of those lower-level files. Typically, you create the chapters, other than the requirements chapter, manually or you use some other design tool.

To create one file with the default file name 2167A_DESIGN and a default file type appropriate for the target, type the following SCA command:

```
$   SCA REPORT 2167A_DESIGN
```

The profile files use 2167A_DESIGN as the name of the file to include as Section 4 of the report. If you change the output file name by specifying the /OUTPUT qualifier on the REPORT command, you must also change the profile file to correspond to the new file name.

12.6.5.1 Describing 2167A Structure in your Code

The specifications for the DOD-STD-2167A Software Design Report call for a hierarchy of program elements. A design is separated into COMPONENTS, which may be further separated into sub-level COMPONENTS, or into UNITS. UNITS are the lowest level of entity described in the design. The design facility allows you to use tagged comments to represent this structure in your code.

The mapping implemented by the 2167A_DESIGN report treats the individual files in your system as the UNITS of the 2167A design. You specify design information relevant to each unit by including the information in a comment block in the source file corresponding to that unit. Because 2167A COMPONENTS are collections of units and other components, the 2167A_DESIGN report maps sets of files into components. However, it would be redundant to duplicate all of the design information at the component level in each file of the component. Instead, you select one file as the main design file of the component and put the design information there. The other files in the component contain a single tag that names the component to which they belong.

The special tags used to designate 2167A relationships are as follows:

- UNIT OF
- COMPONENT
- COMPONENT OF

The UNIT OF tag is used in each unit (each file of your system) and names the component to which the file belongs.

The COMPONENT tag is used only in those files that you have designated as the design file for specific components; the tag names the component that the file specifies.

The COMPONENT OF tag is used to establish the relationships between components. It, too, is used only in designated design files, but it names the parent of the component being specified in the file. For example:

```
File: TOP_LEVEL_COMPONENT_.ADA
    --  COMPONENT: Top level component
    --  ABSTRACT: This is the top level component in a system.
    --  [additional tags that describe the design of the component]
    package top_level_component is
    -- This can be an empty package, or it might contain data that is used
    -- throughout the component, or perhaps data exported by the component, or
    -- even an entire unit.
    end top_level_component

File: SUB_LEVEL_COMPONENT_.ADA
    --  COMPONENT: Sub-level component
    --  COMPONENT OF: Top level component
    --  ABSTRACT: This is a lower-level component in a system.  Note that the
    --  value of the COMPONENT OF tag in this file must be spelled exactly the
    --  same as the value of the COMPONENT tag in the parent component.
    --  [additional tags that describe the design of the component]
    --
    --  UNIT OF: Sub-level component
    --  UNIT DESCRIPTION: For the purposes of this example, we assume that
    --  this file contains a complete unit.  Therefore, it must also have the
    --  UNIT OF tag, even though the component has already been named in the
    --  COMPONENT tag.
    --  [additional tags that describe the design of the unit]
    --  package sub_level_component
    package sub_level_component is
    procedure ...
    function ...
    [other declarations]
    end sub_level_component

File: UNIT_1_.ADA
    --  UNIT OF: top_level_component
    --  UNIT DESCRIPTION:  This is a simple unit that belongs to the
    --  top_level_component.
    --  [additional tags that describe the design of the unit]
    package unit_1 is
    function ...
    end unit_1
```

```
--  UNIT OF: sub_level_component
--  UNIT DESCRIPTION:  This is another simple unit that belongs to the
--  sub_level_component.
--  [additional tags that describe the design of the unit]
package unit_2 is
function ...
end unit_2
```

You can find a more complete example in the SCA$2167A directory, assuming this option was chosen when SCA was installed. For this example, use the following steps:

1. To set your SCA library to be SCA$2167A, type the following command:

   ```
   $   SCA SET LIBRARY SCA$2167A
   ```

2. To create a report, type the following command:

   ```
   $   SCA REPORT 2167A_DESIGN/OUTPUT=mydir:2167a_design
   ```

To process the report with VAX DOCUMENT, use the following steps:

1. Copy the SDML files from SCA$2167A into your directory, as follows:

   ```
   $   COPY SCA$2167A:*.SDML mydir:
   ```

2. Define 2167A_DESIGN to point at the version you just generated, as follows:

   ```
   $   DEFINE 2167A_DESIGN mydir:2167a_design.sdml
   ```

3. Invoke VAX DOCUMENT with a destination_type recognized by VAX DOCUMENT, such as POSTSCRIPT, or LINE. as follows:

   ```
   $   DOCUMENT 2167A_PROFILE MILSPEC destination_type
   ```

12.6.5.2 Retrieving 2167A Structure Information

You can use SCA to get information about the structure of your system. For example, if you want to find all the components in your system, type the following query:

```
SCA>   FIND COMPONENT AND SYMBOL=TAG
```

Because the three primary 2167A tags are all keyword tags, you can use them in keyword queries. For example, if you want to find all the units of a component named Component 1, use the following query expression:

```
CONTAINED_BY ( -
    END = "UNIT OF" AND SYMBOL=TAG, -
    BEGIN = "Component 1" AND SYMBOL=KEYWORD, -
    DEPTH = 1)
```

Similarly, you can use queries on the COMPONENT OF tag to find sub-level components of a given component.

The 2167A_DESIGN report uses these mappings to create the report. It starts with the following SCA query expression:

```
CONTAINED_BY( -
    END = "COMPONENT", -
    BEGIN = SYMBOL=KEYWORD, -
    DEPTH = 1, -
    RESULT = BEGIN)
```

This returns the occurrences of the names of each component of the system. The report then goes through the components one at a time, and writes the component section for each. For each component, it then constructs a query of the following form:

```
CONTAINED_BY( -
    END = "UNIT OF", -
    BEGIN = component_name AND SYMBOL=KEYWORD, -
    DEPTH = 1, -
    RESULT = BEGIN),
```

This returns the UNITS that belong to this component. For each such unit, the corresponding unit subsection is written.

The actual data in the 2167A Software Design Report is obtained from the various tags in your program. The general description information for components is taken from the COMPONENT DESCRIPTION tag. The general description information for units is taken from the UNIT DESCRIPTION tag. The following tables show the tags corresponding to other paragraphs in the report:

TAGS FOR COMPONENT INFORMATION:

Tag:	Description of corresponding section:
INPUT/OUTPUT DATA	Input and output data for the component
ALGORITHMS	Algorithms used by the component
ERROR HANDLING	Error detection/recovery features
DATA CONVERSION	Data conversions done by the component
LOGIC FLOW	Logic flow of the component
REQUIREMENTS ALLOCATION	Requirements satisfied by this component

TAGS FOR UNIT INFORMATION:

Tag:	Description of corresponding section:

INPUT/OUTPUT DATA ELEMENTS	Input and output data for the unit
LOCAL DATA ELEMENTS	Data used only in this unit
INTERRUPTS AND SIGNALS	Interrupts/signals handled by this unit
UNIT ALGORITHMS	Algorithms used by this unit
UNIT ERROR HANDLING	Error detection/recovery for the unit
UNIT DATA CONVERSION	Data conversions done by unit
USE OF OTHER ELEMENTS	Other elements used by this unit
UNIT LOGIC FLOW	Logic flow of the unit
DATA STRUCTURES	Data structures implemented by unit
LOCAL DATA FILES	Data files or databases used by unit
LOCAL DATABASES	Same as LOCAL DATA FILES
LIMITATIONS	Limitations of the unit
REQUIREMENTS ALLOCATED TO THIS UNIT	Requirements satisfied by this unit

For Ada programs, these tags can be put into your comment headers automatically by expanding the 2167A placeholder in the header comment for the file.

Because the exact mapping between elements of your program and 2167A items is highly dependent on your particular application and policies, the 2167A report as supplied by Digital makes no attempt to use program elements (packages, routines, etc.). All information in the report is obtained from tags. It is, however, possible to customize reports to use information from your program elements. It is also possible to change the mapping of UNITS to files and COMPONENTS to sets of files. (See Chapter 18 for more information on customizing 2167A reports.) It is expected that you will want to use TPU to do at least some customization of the 2167A report.

12.7 Reverse-Engineering a Design

A powerful feature of the VMS design environment is the ability to reverse-engineer existing code into layers at various levels, thus retrieving much of the actual algorithm design. The INTERNALS report produces this decomposition. You can fine-tune the report with the DEFINE ADJUSTMENT command. The following example is taken out of context from an Ada package.

```
function matrix_multiply (left, right : in integer_matrix)
    return integer_matrix is
-- ++
-- FUNCTIONAL DESCRIPTION:
--
--      This function computes the matrix product of two integer matrices.
--
--      It uses a simple, triple-nested loop, and does not do any checking t
--      see if the matrices conform.
--
-- FORMAL PARAMETERS:
--
--        left:
--           The left operand.
--
--        right:
--           The right operand.
--
-- FUNCTION VALUE:
--
--           The result of multiplying the two matrices.
-- --
    result_matrix :
        integer_matrix(left'range,right'range(2));
            := (others => (others => 0));
begin
    -- Loop over the rows of the left matrix
    outer_loop: for i in left'range loop
        -- loop over the columns of the right matrix
        middle_loop: for j in right'range(2) loop
            -- compute the inner product of the current row and column
            inner_loop: for k in left'range(2) loop
                result_matrix(i,j)
                    := result_matrix (i,j) + left(i,k) * right(k,j);
            end loop inner_loop;
        end loop middle_loop;
    end loop outer_loop;
    return result_matrix;
end matrix_multiply;
```

12.7.1 Sample Report

You could compile the previous function design, load it into an SCA library, and then type the following command to produce an INTERNALS report:

```
LSE>   REPORT INTERNALS
```

The report might include a routine section similar to that on the following pages.

matrix_multiply

This function computes the matrix product of two integer matrices.

Format

result := matrix_multiply *left, right*

Returns

The result of multiplying the two matrices.

Arguments

left
The left operand.

right
The right operand.

Description

This function computes the matrix product of two integer matrices.

It uses a simple, triple-nested loop, and does not do any checking to see if the matrices conform.

Body

❶
```
-- Loop over the rows of the left matrix
outer_loop: for i in left'range loop
    -- loop over the columns of the right matrix
    middle_loop: for j in right'range(2) loop
        «compute the inner product of the current row and column» ❷
    end loop middle_loop;
end loop outer_loop;
return result_matrix;
```

❷
```
-- compute the inner product of the current row and column
inner_loop: for k in left'range(2) loop
    result_matrix(i,j)
        := result_matrix (i,j) + left(i,k) * right(k,j);
end loop inner_loop;
```

Part 4 Customizing Functions

This part contains tutorial information on modifying your programming development environment, including the following:

- Customizing editing functions
- Customizing LSE/DECwindows Menus
- Defining LSE templates
- Providing diagnostic file support
- Customizing overviews
- Customizing reports

Chapter 13

Customizing Editing Functions

With the VAX Language-Sensitive Editor, you can customize the development environment according to your programming style. This chapter describes the following customizing options:

- Defining your own keys and commands
- Defining aliases
- Redefining tokens and placeholders
- Using VAXTPU
- Using initialization and command files
- Using environment and section files

Section 13.1 describes how to define keys, commands, and aliases; how to redefine language elements; and how to execute VAXTPU statements. Section 13.2 describes modifying LSE/DECwindows attributes. Section 13.3 describes how to store modifications, and Section 13.4 describes how to speed up LSE initialization once you have modified LSE.

13.1 Modifying LSE

With LSE, you can bind commands to keys and define your own set of commands. You can also redefine tokens and placeholders. As you use the language-supplied tokens and placeholders, you may find that some are not suited to your needs. For example, a language template may contain an optional phrase or clause that you never use and therefore always delete. In this situation, it would be more convenient to permanently eliminate that clause from the template.

You can modify LSE, as follows:

- Make modifications interactively while in an editing session

  These changes are kept only as long as you are in the current editing session.
- Create a text file containing your modifications that you execute on LSE startup

  These changes are available automatically each time you use LSE.
- Save your modifications in binary files if you want to speed up LSE startup

  LSE reads in the modifications each time LSE is invoked.

Modifications that are made interactively remain in effect for only the current editing session. For information on saving modifications, see Section 13.3 and Section 13.4.

13.1.1 Defining Keys

With LSE, you can bind any command to a key. Keys that have already been defined can be redefined.

You can define a key interactively by using the DEFINE KEY command. You must include the name of the key you want defined and the command string you want bound to that key. (For key names, refer to the DEFINE KEY command description in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*)

For example, if you want to define a key to exit from LSE, you would type the following command:

```
LSE> DEFINE KEY CTRL_B_KEY "EXIT"
```

Now you can press CTRL/B to exit from this editing session.

In addition, you can define a key by pressing that key at the _Key: prompt instead of typing the name of the key. For example, type the following command:

```
LSE> DEFINE KEY
```

LSE prompts for a key. Press CTRL/B.

```
_key:  CTRL/B
```

LSE responds, as follows:

```
_Key:  "CTRL/B"
```

You must press the RETURN key to confirm. (You can erase the key name and press another key if you want.) LSE prompts you for a string. Then, you type the EXIT command at the prompt, as follows:

```
_String:  EXIT
```

With LSE, you can also bind a user-defined command to a key. For example, you can bind a previously defined MAIL command (see Section 13.1.2) to a key by typing the following command:

```
LSE>  DEFINE KEY F17 "MAIL"
```

Now you can press the F17 key to enter the Mail Utility.

In addition, you can bind several commands to a key. To do this, you must use the DO command and follow the DIGITAL Command Language (DCL) quoting rules. The following example shows you how to bind the ENTER LINE and TAB commands to one key.

```
SE>  DEF KEY CTRL_M_KEY "DO ""SET INDENT CURRENT"", ""ENTER LINE"",""ENTER TAB"""
```

Now you can position the cursor at the end of the line above where you want to insert text, and press the Return key. A blank line is inserted and the cursor is placed at the proper depth for your code.

As an alternative to binding commands to a key, you can bind a sequence of keystrokes to a key. This is called a **learn sequence**. To begin recording a learn sequence, type the following command:

```
LSE Command>  DEFINE KEY/LEARN
```

At the key prompt, press the key that you want to define, for example:

```
_Key:  F17
```

LSE echoes the key name.

Next, press the RETURN key to start recording keystrokes. Every key that you press now will be recorded. This includes all keys that enter text into the editing buffer. All commands typed at the LSE Command> prompt and all responses to prompts such as "_Search for:" are also recorded.

To end the sequence, press the key that you are defining, in this case the F17 key.

Press F17 again to replay the learn sequence.

13.1.2 Defining Commands

You can define your own commands or abbreviate an existing command by using the DEFINE COMMAND command. For example, if you use the SET SELECT_MARK command frequently, you can abbreviate the command by typing the following command:

```
LSE>  DEFINE COMMAND MARK "SET SELECT_MARK"
```

Now you can type MARK at the command prompt to issue the SET SELECT_MARK command.

You can also define your own commands. For example, if you want to define a command to suspend the current editing session and automatically enter the Mail Utility (MAIL), you type the following command:

```
LSE>  DEFINE COMMAND MAIL "SPAWN MAIL"
```

Now you can type MAIL at the LSE> prompt to suspend the editing session and enter the Mail Utility. When you exit from the Mail Utility, the editing session resumes.

In addition, you can define a command that is composed of multiple commands. For example, you may find that you change the case of words frequently while editing. To simplify that task, type the following command:

```
LSE>  DEF COMMAND CHANGE_WORD "DO ""SET SELECT"",""GOTO WORD/FOR"",""CHANGE CASE"""
```

With the DO command, you can execute multiple commands. Note the use of double quotation marks in the example. LSE follows the DCL quoting rules. (See the appendix that describes VAXTPU Builtins in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for an example of defining a command that invokes VAXTPU.)

13.1.3 Defining Aliases

With LSE, you can substitute a short name (an alias) for long identifier names in your source code. Once you define an alias, you can type the alias name in your current buffer and press the EXPAND key. The value of the alias appears in the buffer. To define an alias interactively, you type the DEFINE ALIAS command with an abbreviation name for the text string you want inserted into the buffer, and the value of the text string.

For example, if you use a routine frequently, you can define a short, easy-to-remember name for that routine. By typing the following DEFINE ALIAS command, you can reduce the amount of typing.

```
LSE>  DEFINE ALIAS EXE PPL_EXECUTE_COMMAND
```

Then, you can type EXE and expand it to get the text PPL_EXECUTE_COMMAND inserted into your buffer.

If you want to define an alias for a string of text including nonidentifier characters, you must enclose the text in quotes.

In addition, you can use the DEFINE ALIAS/INDICATED command to associate a specified alias name with the value of a contiguous string of identifier characters at the current buffer position. Thus, you do not have to specify the value of the alias name when defining an alias.

For example, if you define a variable named PPL_GLS_COMMAND_TABLE that you will be referencing frequently, you can define an alias for it by using the DEFINE ALIAS/INDICATED command (PF1-CTRL/A). You put the cursor on the variable name and press PF1-CTRL/A. LSE prompts you for an alias name, as follows:

```
_Alias:
```

You can type the alias name CMDTAB at the prompt and press the Return key. Now you can type the alias name and press the EXPAND key to get PPL_GLS_COMMAND_TABLE inserted into the buffer.

Once an alias is defined, it is associated with the language of your current buffer. If your current buffer is not associated with a language, you must specify the /LANGUAGE qualifier on the DEFINE ALIAS command line.

13.1.4 Defining Buffer Attributes

LSE sets the buffer attributes and properties for each newly created buffer. However, you can override these settings by using an initialization file. This can be accomplished, as follows:

1. Create a file in which to put your commands.

   This file is known as an initialization file. (See Section 13.3.2 for details on initialization files.)

2. When you want these buffer attributes, invoke LSE with the /INITIALIZATION qualifier, as follows:

   ```
   $  LSEDIT/INITIALIZATION=init_filename filename
   ```

Now each buffer you create while in that editing session will have these new buffer attributes. The following commands set buffer attributes:

- SET AUTO_ERASE
- SET FORWARD
- SET INDENTATION
- SET INSERT
- SET LANGUAGE
- SET LEFT_MARGIN
- SET MODIFY
- SET NOAUTO_ERASE
- SET NOMODIFY
- SET NOWRAP
- SET OUTPUT_FILE
- SET OVERSTRIKE
- SET READ_ONLY
- SET REVERSE
- SET RIGHT_MARGIN
- SET TAB_INCREMENT
- SET WRAP
- SET WRITE

There are several cases where these attributes are overridden. They are as follows:

- If a language can be determined from the file type, then the attributes for that language are in effect.
- If a newly created buffer is associated with a defined language, then the left margin, right margin, tab increment, and wrap attributes defined for that language are used.
- When a buffer is created by the READ or CUT commands, the buffer is always modifiable.

13.1.5 Customizing Windows

You can set the characteristics of the screen to meet your needs. You can save these setting in an initialization file (see Section 13.1.4 for details on saving the settings).

For example, you can specify the number of windows LSE displays on the screen, as follows:

```
LSE>   SET SCREEN WINDOW=2
```

If you put this command in your initialization file, LSE displays that number of windows on the screen each time you invoke LSE.

In addition, you might want to alter the message window. The message window is located at the bottom of the screen and displays broadcast messages and messages issued by LSE and SCA. These messages are flashed in reverse video. You can disable the flashing by using the following command:

```
LSE>   DO/TPU "SET(MESSAGE_ACTION_TYPE,NONE)"
```

You can alter the size of the message window by issuing the VAXTPU statement, as follows:

- Press PF1-CTRL/Z to get the TPU> prompt.
- Issue the EVE$SET_MESSAGE_WINDOW_SIZE(2) command.

13.1.6 Redefining Language Elements

You can redefine token, placeholder, package, and language definitions interactively. Thus, you can add or delete constructs, reformat menus, or edit descriptions within existing definitions. The EXTRACT command places the current definition of a token, placeholder, package, or language in the current buffer. You can then make the appropriate modifications and execute the new definitions.

To redefine a token, placeholder, package, or language, use the following steps:

1. Issue the GOTO BUFFER/CREATE command and a new buffer name to enter an empty buffer.
2. Issue the EXTRACT TOKEN, EXTRACT PLACEHOLDER, EXTRACT PACKAGE, or EXTRACT LANGUAGE command followed by the name of the token, placeholder, or language.

3. Edit the definition of the selected token, placeholder, package, or language.
4. Issue the DO command to execute the new definition.

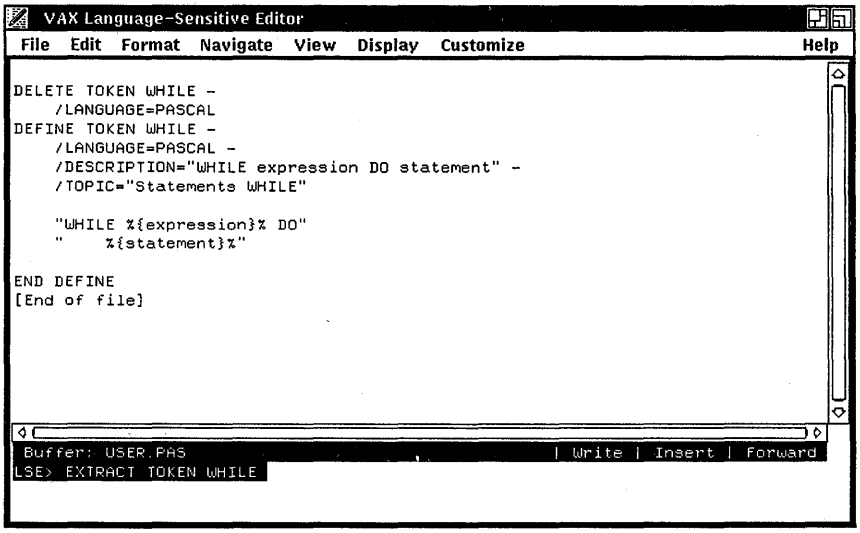
When redefining tokens or placeholders, the previous definitions must be deleted. As shown in Figure 13–1, the EXTRACT command automatically puts a DELETE command before the DEFINE command when it places the definition in the buffer. The DELETE TOKEN command deletes the previous definition of a token, and the DEFINE TOKEN command provides a new definition.

For example, if you want to add a BEGIN/END construct to the definition of a Pascal *WHILE* statement, use the following steps:

1. Issue the following command while in an empty buffer:

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

Figure 13–1: Extracting a Token


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

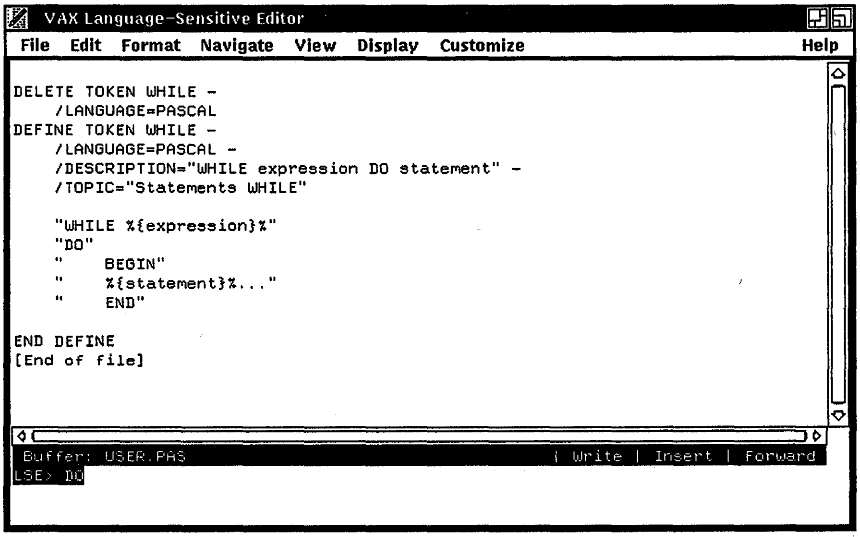
```
DELETE TOKEN WHILE -
    /LANGUAGE=PASCAL
DEFINE TOKEN WHILE -
    /LANGUAGE=PASCAL -
    /DESCRIPTION="WHILE expression DO statement" -
    /TOPIC="Statements WHILE"

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

END DEFINE
[End of file]
 Buffer: USER.PAS                              | Write | Insert | Forward
LSE> EXTRACT TOKEN WHILE
```

2. Press CTRL/Z to remove the LSE> prompt and edit the token by adding BEGIN and END statements.

3. Press CTRL/Z to get the LSE> prompt once you have the definition the way you want it.
4. Issue the DO command to execute the new definition, as shown in Figure 13–2.

Now each time you use the WHILE token in Pascal in the current editing session, LSE will provide you with the new definition.

Figure 13–2: Executing a New Definition


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

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

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

END DEFINE
[End of file]
```

```
 Buffer: USER.PAS                                  | Write | Insert | Forward
LSE> DO
```

You can use the MODIFY LANGUAGE command as an alternative method for redefining language definitions. This method works best when you want to modify just one or two qualifiers in your language definition or the attributes of all the languages. (See the MODIFY LANGUAGE command in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details.)

For example, if you want to change the setting of the /EXPAND_CASE qualifier from AS_IS to UPPER, you type the following command:

```
LSE> MODIFY LANGUAGE PASCAL /EXPAND_CASE=UPPER
```

The MODIFY LANGUAGE command lets you modify your language definition without having to execute the language definition again.

13.1.7 Using the VAX Text Processing Utility (VAXTPU)

VAXTPU is a high-performance programmable editing tool that is part of LSE. LSE provides two methods of executing VAXTPU statements.

- The CALL command, which invokes a VAXTPU procedure
- The DO/TPU command, which executes a VAXTPU statement

You can use the DO/TPU command in several different ways, as follows:

- Type the DO/TPU/PROMPT="TPU>" command (or press PF1-CTRL/Z) to get the TPU> prompt, and issue your VAXTPU statement.
- Type the DO/TPU command with your VAXTPU statement at the LSE> prompt.
- Type the DO/TPU command to execute a buffer of VAXTPU statements.

For example, if you want to access the VAXTPU built-in JOURNAL_CLOSE, press CTRL/Z to get the LSE> prompt, then type the following command:

```
LSE> DO/TPU JOURNAL_CLOSE
```

The DO/TPU command passes one command to the VAXTPU command interpreter and returns to the LSE> prompt.

An alternative way of accessing a VAXTPU built-in is to issue the built-in at the TPU> prompt. This avoids having to follow the DCL quoting rules. For example, if you want to access the VAXTPU built-in ERASE with the predefined buffer, *message_buffer*, press PF1-CTRL/Z to get the TPU> prompt, then type the following command:

```
TPU> ERASE (MESSAGE_BUFFER)
```

The VAXTPU command interpreter executes the command and resumes the editing session.

In addition, with VAXTPU, you can write functions not provided by LSE. You can define VAXTPU statements or functions interactively during an editing session or put them into a file for later use. See the *VAX Text Processing Utility Reference Manual* for information on the sophisticated text processing capabilities available through VAXTPU.

There are some restrictions on altering the LSE interface to VAXTPU. See the appendix that describes how to write and execute VAXTPU procedures in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details on these restrictions. Also see the CALL and DO/TPU commands in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*

13.2 Modifying LSE/DECwindows Attributes

You can modify the LSE/DECwindows attributes by using the Customize menu. You can set the number of windows, the width and height of screens, and change the menus and menus items. See Chapter 14 for more details on customizing menus.

The prompts are in either insert or overstrike mode. The setting defaults to the current setting of the terminal. In LSE/DECwindows, the default is overstrike. You can change this setting by including the following commands in an initialization file:

```
SET INSERT/BUFFER=$COMMANDS
SET INSERT/BUFFER=$PROMPTS
```

13.3 Storing Modifications

The previous sections explained how to make modifications interactively. Such modifications remain in effect only for the current editing session. If you want to keep your modifications, you can put them into a file and then access that file whenever you want to use the modifications. Alternatively, you can create an initialization file or a command file that allows you to automatically access the modifications at LSE startup. The following sections describe these methods of storing and using your modifications.

13.3.1 Storing Modifications in Text Files

To store your modifications in a file, you can create a text file with a file extension of .LSE. Similarly, if you want to keep your VAXTPU statements, you can create a text file with a file extension of .TPU.

With LSE, you can use these files interactively when you need them. The following examples show how you execute your commands. The DO command directs LSE to execute the commands found in the specified buffer. In both these examples, the current buffer is used.

Press CTRL/Z to get the LSE> prompt, and type the following commands:

For LSE commands:

```
LSE>  GOTO FILE filename.LSE
LSE>  DO
```

For VAXTPU statements:

```
LSE>  GOTO FILE filename.TPU
LSE>  DO/TPU
```

Now the commands contained in the text file you just executed with the DO command will affect the current editing session.

13.3.2 Using Initialization and Command Files

You may want your modifications automatically executed when you invoke LSE. You can do this by specifying the file when you invoke LSE. An initialization file contains LSE commands that are executed when LSE is invoked. A command file contains VAXTPU statements that are executed when LSE is invoked.

To use your LSE modifications, type the following command:

```
$  LSEDIT/INITIALIZATION=device:[directory]filename.LSE ...
```

To use your VAXTPU file, type the following command:

```
$  LSEDIT/COMMAND=device:[directory]filename.TPU ...
```

You can use both the /INITIALIZATION and /COMMAND qualifiers on the LSEDIT command line to use both your files.

You can also use your initialization and command files without specifying the files each time you invoke LSE. To do this, add the following commands to your LOGIN.COM file.

```
$ DEFINE LSE$INITIALIZATION device:[directory]filename.LSE
$ DEFINE LSE$COMMAND device:[directory]filename.TPU
```

LSE will execute the initialization and command files each time you type the LSEDIT command.

Example 13–1 shows the types of modifications you can put into an initialization file.

Example 13–1: Sample Initialization File

```
!Sample initialization file
!
! Command to spawn a subprocess, run MAIL, and clear the message
! buffer when the editing session is resumed.
!
DEFINE COMMAND MAIL "DO ""SPAWN MAIL"",""CLEAR_MESSAGE"""
!
! Command to clear the message window by writing three blank lines
!
DEFINE COMMAND CLEAR_MESSAGE-
  "DO/TPU ""MESSAGE("""""""")"",""MESSAGE("""""""")"",""MESSAGE("""""""")"""
!
! Command to bind the MAIL command to a key.
!
DEFINE KEY F20 "MAIL"
!
! DEFINE ALIAS command for abbreviating the name of an include file in Pascal.
!
DEFINE ALIAS DEFS/LANGUAGE=PASCAL "USER1:[PROJECT]COMMON_DEFINITIONS.PAS"
!
! DEFINE TOKEN command to redefine the DO token in C, so that it always contains a
! compound statement
!
!DELETE TOKEN DO -
  /LANGUAGE=C
DEFINE TOKEN DO -
  /LANGUAGE=C -
  /DESCRIPTION="executes a statement as long as a particular condition is satisfied" -
  /TOPIC="Language_topics Statements do"

  "do"
  "{"
  "    {@statement@}..."
  "}"
  "while    ({@expression@});"

END DEFINE
!
! VAXTPU statements to change the scrolling of LSE windows so that the
! screen only scrolls when the cursor is at the bottom or top of the screen.
!
DO/TPU "SET (SCROLLING, LSE$MAIN_WINDOW, ON,    0, 0, 0)"
!
DO/TPU "SET (SCROLLING, LSE$TOP_WINDOW, ON,     0, 0, 0)"
!
DO/TPU "SET (SCROLLING, LSE$BOTTOM_WINDOW, ON,  0, 0, 0)"
```

Example 13–2 shows the types of VAXTPU procedures you can put into a command file. Note that procedure definitions must precede statements in a command file.

Example 13–2: Sample Command File

```
! Sample command file
!
! VAXTPU procedure to replace tabs with spaces.  In this example, the tabs
! are set at 8.
PROCEDURE ELIMINATE_TABS

    local       target,
                n,
                saved_mode;
    position(beginning_of(current_buffer));
    loop
        target := search(ascii(9), FORWARD);
        exitif (target = 0);
        position(beginning_of(target));
        erase_character(1);
        n := current_offset;
        n := n - (8 * (n / 8));
        saved_mode := get_info (current_buffer, "mode");
        set (insert, current_buffer);
        copy_text(substr("        ", 1, 8 - n));
        set (saved_mode, current_buffer);
    endloop;
ENDPROCEDURE

! LSE command to invoke the ELIMINATE_TABS procedure.
LSE$DO_COMMAND("DEFINE COMMAND NOTABS ""CALL ELIMINATE_TABS""");
! Command to change the text of the "Working..." message to "BUSY..."
SET (TIMER, ON, "BUSY...");

! VAXTPU procedure to change the scrolling of LSE windows so that the screen
! only scrolls when the cursor is at the bottom or top of the screen.
set (scrolling, lse$main_window, ON,    0, 0, 0);
set (scrolling, lse$top_window, ON,     0, 0, 0);
set (scrolling, lse$bottom_window, ON,  0, 0, 0);
```

13.4 Speeding Up LSE Initialization

LSE provides two mechanisms for speeding up LSE initialization.

- An environment file, which contains all language-specific definitions
- A section file, which contains all key definitions and VAXTPU procedures

Since both environment and section files are binary files, LSE does not have to execute them each time LSE is invoked. This saves time at LSE startup.

In addition, environment files and section files have a mechanism for a system manager to provide users with a tailored environment while still allowing you to do some customization of your own with the use of command files and initialization files.

If your initialization file or command file is large, you may want to put your changes into an environment file or a section file. If you do, you should save your source files for future LSE updates because LSE sometimes requires you to rebuild your environment and section files when you install a new version of LSE.

13.4.1 Creating Environment and Section Files

To create an environment or section file, establish the language definitions, placeholder definitions, command and key definitions, and mode settings that you want to save. Then type the SAVE ENVIRONMENT or SAVE SECTION commands while in an editing session.

The SAVE ENVIRONMENT command saves the following:

- Languages
- Packages
- Placeholders
- Tokens
- Aliases
- Routines
- Parameters
- Adjustments
- Keywords
- Tags

The SAVE SECTION command saves the following:

- All current key definitions
- VAXTPU procedures and variables
- Learn sequences
- User-defined commands
- Mode settings

You must include the name of the environment or section file, including device and directory names, when you type the SAVE ENVIRONMENT or SAVE SECTION commands.

The following examples show you how to create environment and section files. Press CTRL/Z to get the LSE> prompt, and type the following commands:

For creating environment files:

```
LSE>  DO
LSE>  SAVE ENVIRONMENT filename
```

For creating section files:

```
LSE>  DO/TPU
LSE>  SAVE SECTION filename
```

13.4.2 Using Environment and Section Files

To use your definitions, you must type the /ENVIRONMENT or /SECTION qualifiers on the LSE command line.

To use your environment file, type the following command:

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

You can include a list of file specifications with the /ENVIRONMENT qualifier.

To use your section file, type the following command:

```
$  LSEDIT/SECTION=device:[directory]filename.TPU$SECTION
```

You can automatically access all language-specific definitions from one editing session to another without specifying the /ENVIRONMENT qualifier each time you invoke LSE. To do this, add the following command to your LOGIN.COM file:

```
$ DEFINE LSE$ENVIRONMENT device:[directory]filename.ENV
```

You can automatically access all your key definitions and VAXTPU procedures from one editing session to another without specifying the /SECTION qualifier each time you invoke LSE. To do this, add the following command to your LOGIN.COM file:

```
$ DEFINE LSE$SECTION device:[directory]filename.TPU$SECTION
```

13.4.3 Using Multiple Files

With LSE, you can specify any combination of the /INITIALIZATION, /COMMAND, /ENVIRONMENT, and /SECTION qualifiers on the command line. LSE processes these files in the following order:

1. /SECTION
2. /COMMAND
3. /ENVIRONMENT
4. /INITIALIZATION

If you specify more than one environment file with the /ENVIRONMENT qualifier, the definitions in the first file on the list take precedence over definitions in subsequent environment files. All environment file definitions take precedence over /SYSTEM_ENVIRONMENT file definitions.

Table 13–1 lists where your modifications to LSE are stored.

Table 13–1: Where LSE Stores Modifications

Text Files		**Binary Files**	
Initialization	**Command**	**Environment**	**Section**
LSE commands	VAXTPU statements	Language-specific definitions	Key definitions
			VAXTPU procedures
			VAXTPU symbol names
			DEFINE COMMAND definitions
			Learn sequences
			LSE mode settings

Chapter 14

Customizing LSE/DECwindows Menus

LSE provides a Menu Extension Service that allows you to add, modify, or delete menu entries from LSE pop-up and pull-down menus. This chapter describes how to use this Menu Extension Service.

Menu entries are limited to pushbutton widgets and separator widgets. You can add widgets only to the bottom of a menu.

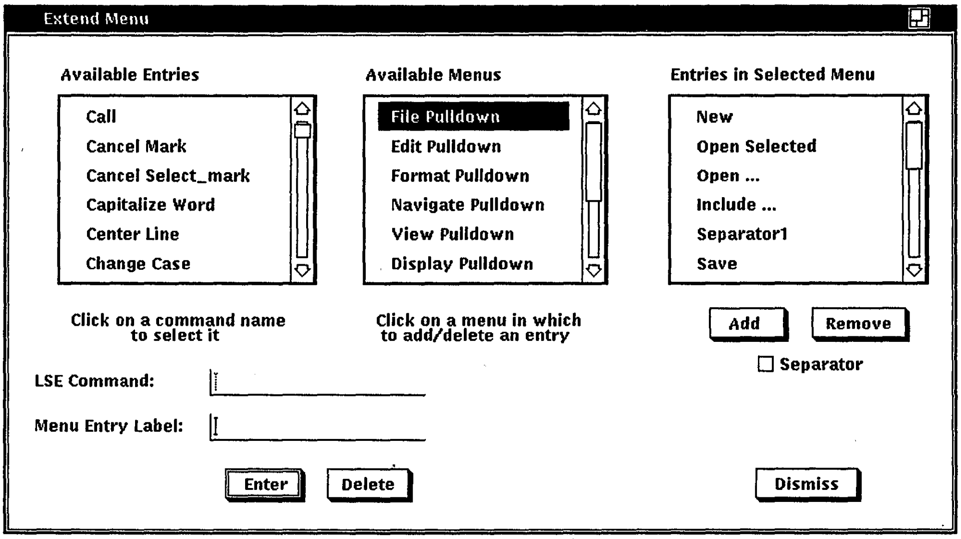
14.1 Using the Extend Menu Dialog Box

You use the Extend Menu dialog box to customize menus. To access the menu, perform the following steps:

1. Pull down the Customize menu.
2. Choose the Extend Menu... menu item.

LSE displays the Extend Menu dialog box, as shown in Figure 14–1.

Figure 14–1: Extend Menu Dialog Box



The Extend Menu dialog box has three list boxes as follows:

- Available Entries—This list contains all the currently available commands.
- Available Menus—This list contains the currently available pull-down and pop-up menus.
- Entries in Selected Menu—This list contains the menu items found within the selected menu in the Available Menus list.

There are two text widgets: LSE Command: and Menu Entry Label:. When you click on a command name in the Available Entries list, the LSE Command: text widget displays the selected command, and the Menu Entry Label: text widget displays the selected command name, which is the default label. This is the default program executed by the menu entry.

14.2 Adding a New LSE Command Entry to a Menu

To add an available LSE command entry to a menu, perform the following steps:

1. Click on the command entry you want in the Available Entries list box.
2. Choose the menu to which you want to add a menu entry.
3. Click on Add, which is located directly below the Entries in Selected Menu box.
4. Click on Dismiss to remove the Extend Menu dialog box.

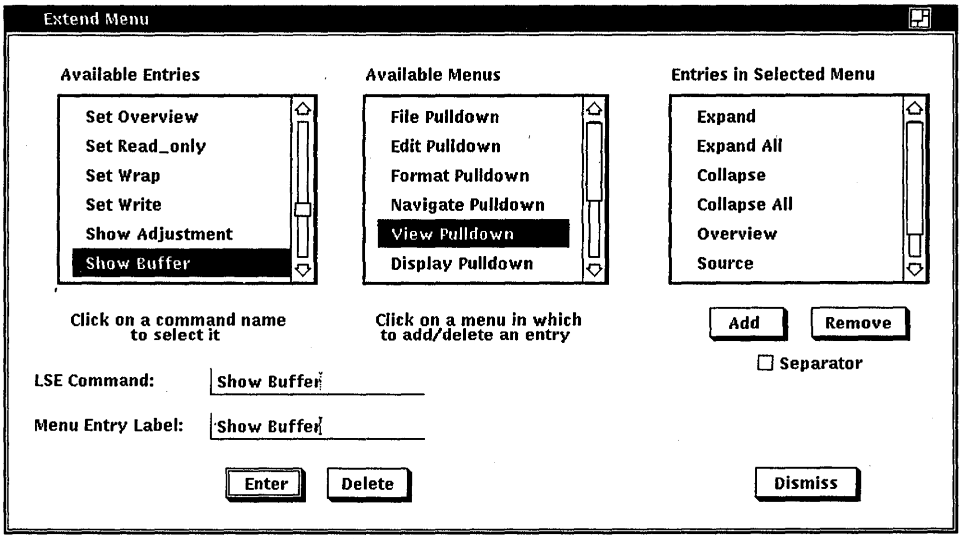
For example, if you want to add the Show Buffer menu entry to the View menu, use the following steps:

1. Click on the Show Buffer menu entry in the Available Entries list. The menu entry is highlighted.

   Note that the text widget Menu Entry: now contains the Show Buffer command name. The Menu Entry Label: is the name of the LSE command as it appears in the Available Entry: text widget. The program bound to that menu entry is what appears in the LSE Command: text widget.
2. Click on the View Pulldown menu entry in the Available Menus list box.

   The View Pulldown entry is highlighted. In addition, the Entries in Selected Menu box lists the menu entries currently contained in the View Pulldown menu, as shown in Figure 14–2.

Figure 14–2: Adding a Command to a Menu



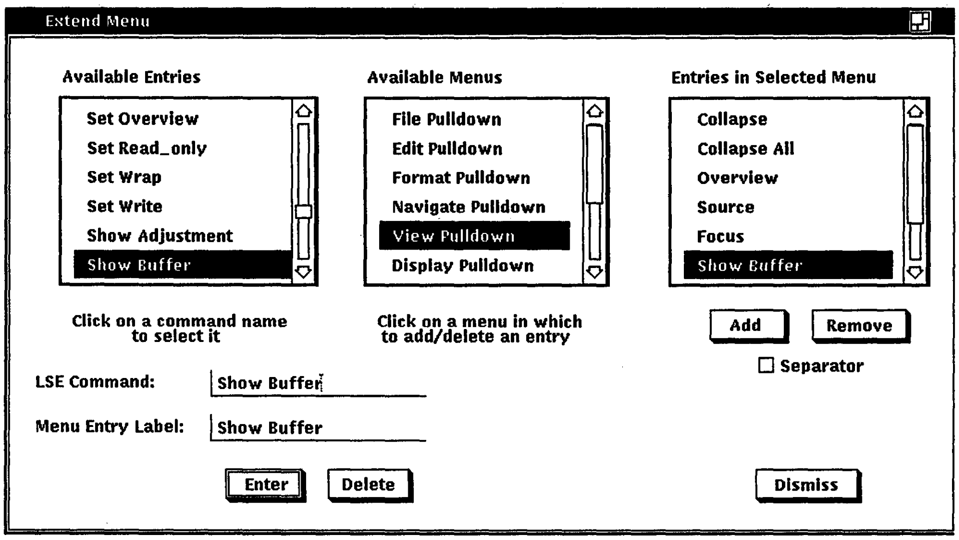
LSE Command: Show Buffer

Menu Entry Label: Show Buffer

3. Click on Add, which is directly below the Entries in Selected Menu box.

   The Entries in Selected Menu list box is updated to show that Show Buffer has been added as a menu entry at the bottom of the menu, as shown in Figure 14–3.

Figure 14–3: Menu Item Added



LSE Command: Show Buffer

Menu Entry Label: Show Buffer

If the separator toggle is On, a separator widget is added before the menu entry.

4. Click on Dismiss to remove the Extend Menu dialog box.

Now when you click on the View pulldown menu, you can choose the Show Buffer menu item.

14.3 Saving Menu Modifications

If you want to keep your modifications, use the following steps while you are still in the editing session:

- Pull down the Customize menu.
- Choose the Save Current Attributes... menu item.
  The Save Current Attributes: dialog box appears on the screen.
- Click on OK.

Chapter 15

Defining LSE Templates

With LSE, you can define your own languages. In addition to programming languages, you can define languages for memos, letters, or other written material. Once you have defined your own language, you can save your language in an environment file, recall it for later editing sessions, and update it with new definitions. A single environment file can contain sets of templates for several languages.

This chapter describes how to define your own templates and how to create and use environment files. It also includes two examples of environment files created with LSE. Section 15.1 provides an example of a text template for a memo. Section 15.3 describes how to save language definitions. Section 15.2 describes how to define a template for a programming language. Section 15.4 provides information on how LSE controls indentation of tokens and placeholders. Section 15.5 describes how to define packages.

LSE provides a mechanism for interfacing non-Digital processors to the diagnostic review facility. See the appendix on user diagnostic file format in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details.

15.1 Defining a Text Template

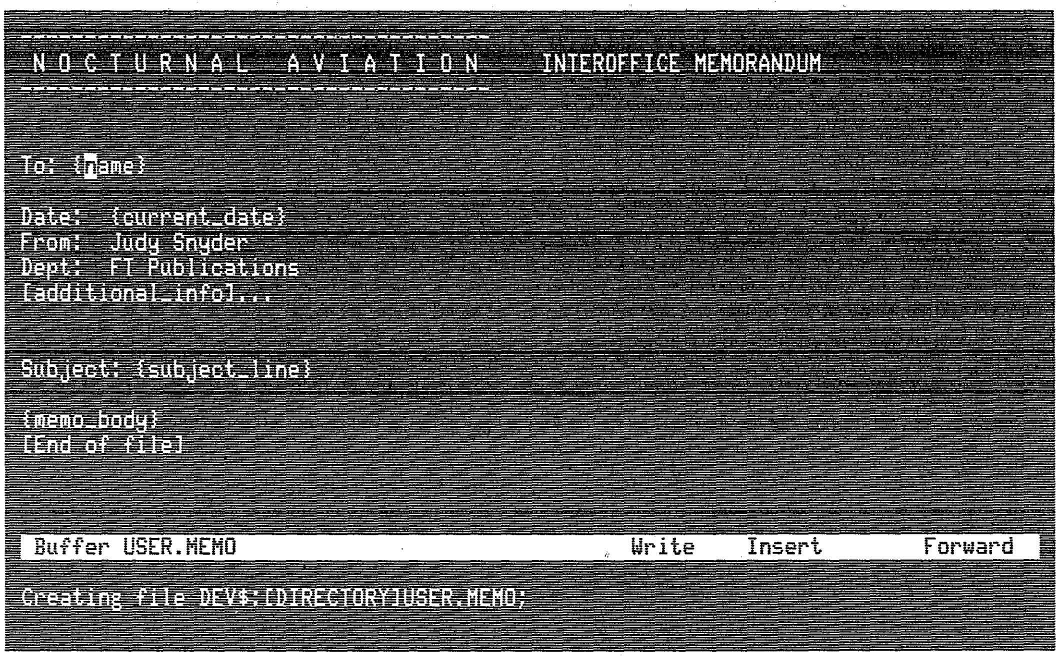
When defining a language, you create a file in which to put LSE commands that will be used to define elements of your language. Once the file is complete, you execute the commands in the source file and create an environment file for use with LSE. This section describes how to create a language for writing memos. This memo language is defined using LSE language, placeholder, and token definitions.

Before starting, you should be aware of the following coding rules:

- If a command and its qualifiers extend beyond one line, a hyphen must be the last character on each continuation line except the last line of the command.
- Comment lines begin with an exclamation point (!), and ignore leading blank spaces.
- Lines containing only comments do not terminate continued commands.

Figure 15–1 shows a screen display of the memo template as it appears after you expand the initial string, *{memo_template}*.

Figure 15–1: Memo Template



```
 ---------------------------------------
  N O C T U R N A L   A V I A T I O N      INTERNOFFICE MEMORANDUM
 ---------------------------------------


 To: {name}

 Date:  {current_date}
 From:  Judy Snyder
 Dept:  FT Publications
 [additional_info]...


 Subject: {subject_line}

 {memo_body}
 [End of file]


  Buffer USER.MEMO                               Write    Insert         Forward

 Creating file DEV$:[DIRECTORY]USER.MEMO;
```

15.1.1 Language Definition

All template definitions, whether text- or language-oriented, must begin with a language definition. To define a language, you use the DEFINE LANGUAGE command. This command takes a series of qualifiers that set the characteristics of the language.

The following example shows the language definition for a memo.

```
DEFINE LANGUAGE MEMO -
  /IDENTIFIER_CHARACTERS = -
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890" -
  /INITIAL_STRING = "{memo_template}" -
  /FILE_TYPES = (.MEMO) -
  /TAB_INCREMENT = 4 -
  /PLACEHOLDER_DELIMITERS = ( -
   REQUIRED = ("{","}"), -
   REQUIRED_LIST = ("{","}..."), -
   OPTIONAL = ("[","]"), -
   OPTIONAL_LIST = ("[","]...") ) -
  /PUNCTUATION_CHARACTERS=",;()"
```

The /IDENTIFIER_CHARACTERS qualifier determines what sequences of characters LSE considers to be a word. A word is any sequence of identifier characters delimited by either white space or a nonidentifier character. The /IDENTIFIER_CHARACTERS qualifier also specifies what characters may appear in alias names. Typically, identifier characters include the alphabet (both uppercase and lowercase), digits, and a few special characters.

The /INITIAL_STRING qualifier specifies the initial text that will appear in a newly created file. LSE automatically inserts this text whenever you use either the LSEDIT command from the DCL command line or the GOTO FILE/CREATE command from within LSE to create a new file.

The /FILE_TYPES qualifier specifies what file types correspond to this language. In this case, when you edit a file with the file type .MEMO, LSE automatically sets the language MEMO for the buffer.

The /TAB_INCREMENT qualifier sets the tab stops at every four columns.

The /PLACEHOLDER_DELIMITERS qualifier specifies the starting and ending strings that delimit placeholders. Placeholders can be required or optional, and they may specify single constructs or lists of constructs. There are four types of placeholder delimiters: required, optional, required list, and optional list. The delimiters for each type are specified as a pair of quoted strings separated by commas and enclosed in parentheses. By convention, list placeholders are indicated by appending ellipses (. . .) to the corresponding nonlist placeholders.

The /PUNCTUATION_CHARACTERS qualifier specifies the characters that are considered punctuation marks or delimiters. Punctuation characters are not important to the MEMO language. The value specified in the language definition is simply the default value. For a more detailed discussion of both punctuation characters and identifier characters, see Section 15.2 on defining a programming language.

15.1.2 Placeholder Definitions

There are three types of placeholders:

- Nonterminal placeholders, which expand into text that is inserted into the buffer
- Terminal placeholders, which expand into descriptive text that is displayed in a temporary window
- Menu placeholders, which provide a list of options for expanding the placeholder

To define a placeholder, you use the DEFINE PLACEHOLDER command. For the MEMO language example, define the initial string *memo_template* as a nonterminal placeholder. In this case, the definition will contain a **placeholder body**. A placeholder body contains the text that is inserted in the buffer when you expand the placeholder. Each line of the placeholder body must be enclosed in quotation marks. Quotation marks with no text produce a blank line when expanded.

The following example shows how the *memo_template* placeholder is defined.

```
DEFINE PLACEHOLDER memo_template -
  /LANGUAGE = MEMO -
  /TYPE = NONTERMINAL
  /DESCRIPTION = "Interoffice Memorandum"
  "-------------------------------------"
  " N O C T U R N A L   A V I A T I O N     INTEROFFICE MEMORANDUM"
  "-------------------------------------"
  ""
  ""
  "To: {name}"
  ""
  "Date:  {current_date}"
  "From:  Judy Snyder"
  "Dept:  FT Publications"
  "[additional_info]..."
  ""
  ""
  "Subject: {subject_line}"
  ""
  "{memo_body}"
END DEFINE
```

The /LANGUAGE qualifier specifies that this placeholder is associated with the MEMO language.

The /DESCRIPTION qualifier specifies a single line of text that is displayed along with the placeholder name whenever the placeholder appears in a menu.

The text following the /DESCRIPTION qualifier in the *memo_template* placeholder definition is the placeholder body. Placeholder bodies represent the text that is inserted into the buffer when the placeholder is expanded. Text within a placeholder body can include additional placeholders. These placeholders must include the appropriate delimiters so that LSE will know how to expand them.

All placeholders found within a placeholder body must be defined. These placeholders do not have to be defined in any particular order. When defining placeholders, you do not include delimiters.

In this case, the *name*, *current_date*, *subject_line*, and *memo_body* placeholders are defined as terminal placeholders, indicated by the /TYPE=TERMINAL qualifier. Terminal placeholders, when expanded, provide a description of the appropriate values you must type over the text to replace the placeholder. The text for a terminal placeholder can be as many lines as are needed to explain the correct text insertion for the placeholder. You must use quotation marks at the beginning and end of each line of text.

Note that the *additional_info* placeholder appears within the delimiters for an optional list. See the *additional_info* placeholder definition at the end of this section for more information on list placeholders.

The following example shows how the terminal placeholders are defined.

```
DEFINE PLACEHOLDER name -
  /LANGUAGE = MEMO -
  /TYPE = TERMINAL
  "Name of the person receiving this memo."
END DEFINE

DEFINE PLACEHOLDER current_date -
  /LANGUAGE = MEMO -
  /TYPE = TERMINAL
  "Date the memo is written."
  "Date can be of the format day/month/year."
END DEFINE

DEFINE PLACEHOLDER subject_line -
  /LANGUAGE = MEMO -
  /TYPE = TERMINAL
  "Subject of the memo."
END DEFINE

DEFINE PLACEHOLDER memo_body-
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Body of the memo." -
  /TYPE = TERMINAL
  "Supply the text of the memo here."
END DEFINE
```

The *additional_info* placeholder is defined as a menu placeholder.

```
DEFINE PLACEHOLDER additional_info -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Additional identification information" -
  /DUPLICATION = VERTICAL -
  /TYPE = MENU
  "Phone" /TOKEN
  "Location" /TOKEN
  "Network" /TOKEN
  "CC" /TOKEN
  "All" /TOKEN
END DEFINE
```

The *additional_info* placeholder is also used as an optional list placeholder, indicated by the delimiters ([ ]) . . . . Since *additional_info* is used as a list placeholder, you need to specify what type of duplication should be performed when the placeholder is expanded by using the /DUPLICATION qualifier. Duplication options are VERTICAL, HORIZONTAL, and CONTEXT_DEPENDENT. In this case, the VERTICAL option is used, which places the duplicate placeholder on the next line, immediately under the original placeholder.

Next, use the /TYPE qualifier to specify that this is a menu placeholder. The placeholder body for a menu placeholder is defined the same way as a nonterminal placeholder and contains the elements of the menu. In this case, each element in the menu is defined as a token, as is indicated by the /TOKEN qualifier. The next section describes how to define tokens.

15.1.3 Token Definitions

Tokens are keywords that you type directly into the buffer and expand into additional templates. This example uses tokens for additional fields of the memo header. By using tokens instead of placeholders, you simplify the entry of frequently used options.

To define a token, use the DEFINE TOKEN command. The /LANGUAGE qualifier in the following token definitions tells LSE that this token is defined for the MEMO language.

With the /DESCRIPTION qualifier, you can supply a text string that is displayed with the token name in a menu. In this case, each token definition includes the /DESCRIPTION qualifier. Thus, when the *additional_info* placeholder is expanded into a menu, each token and its description is displayed.

The body of a token is defined in the same way as nonterminal and terminal placeholders.

By defining these elements as tokens, you can type the token name you want directly into the buffer and expand it without going through a menu.

The following example shows how tokens are defined.

```
DEFINE TOKEN phone -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Office phone number"
  "Phone:  523-440-3287"
END DEFINE

DEFINE TOKEN location -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Office location"
  "LOC:  URE-0096"
END DEFINE

DEFINE TOKEN network -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Network address"
  "NET:  EMLEN::SNYDER"
END DEFINE
```

```
DEFINE TOKEN cc -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Name of person to receive a copy of this memo."
  "CC:  {name}"
END DEFINE

DEFINE TOKEN all -
  /LANGUAGE = MEMO -
  /DESCRIPTION = "Choose all options in menu."
  "Phone:  523-440-3287"
  "LOC:  URE-0096"
  "NET:  EMLEN::SNYDER"
  "CC:  {name}"
END DEFINE
```

The token definitions for *cc* and *all* include the *name* placeholder. All placeholders found within a token body must be defined. In this case, you already defined *name* in the previous section, so you do not have to define it again.

Now that the template definition is complete, you must execute your source file. The following section describes how to execute your source file and create an environment file for later use.

15.2 Defining a Programming Language

This section describes how to define templates for a programming language. The concepts and coding rules for defining templates for a programming language are similar to those introduced in the MEMO language. However, some advanced features necessary for handling more complex templates are introduced.

The following sections explain how the EXAMPLE language, used in the sample editing session in Chapter 2, was defined. The EXAMPLE language is available on line with the VAX Language-Sensitive Editor.

You should follow a grammar, or other syntax summary, as a guide if you are defining a programming language. A grammar provides a summary from which to work when defining the equivalent LSE definitions. It also helps you avoid mistakes. Example 15–1 contains a syntax summary for the EXAMPLE language.

Those expressions on the left side of the syntax summary are known as nonterminals. Typically, nonterminals with multiple options, such as *statement*, are defined as menu placeholders. Nonterminals that expand into a single option, such as *if_stmnt*, are defined as nonterminal placeholders or tokens. Nonterminals that expand into text you must supply, such as *identifier*, are defined as terminal placeholders. Optional syntactic elements, such as *[else statement_list]*, require placeholders that do not explicitly appear as nonterminals in the syntax summary. Major keywords, such as IF, are defined as tokens.

Example 15–1: Syntax Summary for the Example Language

```
program_unit         ::= PROCEDURE procedure_name [(parameter_list)] IS
                         [variable_declarations]
                         BEGIN
                         statement_list
                         END
                         ENDPROCEDURE procedure_name
procedure_name       ::= identifier
identifier           ::= alphabetic | alphabetic identifier_chars
identifier_chars     ::= alphanumeric | alphanumeric identifier_chars
alphabetic           ::= a | b | ... | z
numeric              ::= 0 | 1 | ... | 9
alphanumeric         ::= alphabetic | numeric
parameter_list       ::= parameter:type | parameter:type, parameter_list
parameter            ::= identifier
var_decl_list        ::= var_decl; | var_decl; var_decl_list
var_decl             ::= identifier_list : type [init_value_list]
identifier_list      ::= identifier | identifier, identifier_list
init_value_list      ::= initial_value | initial_value, init_value_list
initial_value        ::= boolean_exp | arithmetic_exp
statement_list       ::= statement; | statement; statement_list
statement            ::= if_stmnt | assignment_stmt | loop_construct |exit_stmt
if_stmnt             ::= IF expression THEN statement_list [ELSE statement_list] ENDIF
assignment_stmnt     ::= identifier := expression
expression           ::= boolean_exp | arithmetic_exp
boolean_exp          ::= identifier | unary identifier | identifier relop identifier
relop                ::= = | <> | < | > | <= | >=
arithmetic_exp       ::= identifier oper identifier
oper                 ::= + | - | * | /
unary                ::= NOT
loop_construct       ::= [label:] LOOP statement_list END LOOP
label                ::= identifier
exit_stmt            ::= EXIT [label] WHEN expression
type                 ::= BOOLEAN | INTEGER
```

15.2.1 Language Definition

To define a language (in this case, EXAMPLE) use the DEFINE LANGUAGE command and its qualifiers (see the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details on the LANGUAGE command). The following example shows the language definition for the EXAMPLE language.

```
DEFINE LANGUAGE EXAMPLE -
  /CAPABILITIES = NODIAGNOSTICS -
  /COMMENT = (BEGIN = "/*", END = "*/", TRAILING = "--") -
  /COMPILE_COMMAND = "" -
  /EXPAND_CASE = AS_IS -
  /FILE_TYPES = (.EXAMPLE) -
  /HELP_LIBRARY = "" -
  /IDENTIFIER_CHARACTERS = -
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890$_" -
  /INITIAL_STRING = "{program_unit}" -
  /LEFT_MARGIN = CONTEXT_DEPENDENT -
  /RIGHT_MARGIN = 80 -
  /PUNCTUATION_CHARACTERS = ",;().':" -
  /PLACEHOLDER_DELIMITERS = ( -
    REQUIRED = ("{","}"), -
    REQUIRED_LIST = ("{","}..."), -
    OPTIONAL = ("[","]"),
    PSEUDOCODE = ("<",">")) -
  /NOQUOTED_ITEM -
  /TAB_INCREMENT = 4 -
  /TOPIC_STRING = "" -
  /VERSION = "2.3" -
  /NOWRAP
```

The /CAPABILITIES qualifier specifies whether or not the compiler for the language creates a diagnostic file when it is compiled from within LSE. If /CAPABILITIES=DIAGNOSTICS is specified, LSE automatically appends the /DIAGNOSTICS qualifier to the compile command. There is no compiler for the EXAMPLE language. This is consistent with the /COMPILE_COMMAND qualifier having a null value. If a compiler existed, it could optionally produce a diagnostic file upon compilation. Then the LSE command REVIEW would use this file.

The /COMMENT qualifier specifies the character sequences of comments in the language. The BEGIN="/*" and END="*/" qualifiers specify that bracketed comments begin with the sequence "/*" and terminate with the sequence "*/". Bracketed comments can extend over several lines. The TRAILING="—" qualifier specifies that trailing comments begin with a pair of dashes. Trailing comments terminate at the end of the line.

The /COMPILE_COMMAND qualifier specifies the default command string to be used to invoke the language processor when you issue the LSE command COMPILE. If there were an actual compiler for the EXAMPLE language, you would specify the command that invokes that compiler here. In the definition of the VAX C language, for example, the qualifier is /COMPILE_COMMAND="cc" .

The /EXPAND_CASE qualifier specifies the case that the text of inserted templates is to have. The AS_IS parameter specifies that the expanded text is to retain the same case as in the placeholder or token definition.

The /FILE_TYPES qualifier specifies what file type corresponds to the EXAMPLE language. When you specify the file type .EXAMPLE, LSE automatically sets the language EXAMPLE for the buffer. You must not specify a file type that is already associated with another language.

The /HELP_LIBRARY qualifier specifies the name of a help library from which LSE will get help for the placeholders and tokens defined in the language. The default library is SYS$HELP:HELPLIB.HLB, but you can also create your own help library for the language.

The /TOPIC_STRING qualifier also looks up help on placeholders and tokens. This qualifier specifies a common prefix string to which the *topic_string* parameter specified in a placeholder or token definition is concatenated. For example, in the definition of VAX Pascal, the language topic string is "Pascal", and the help library qualifier is null. The definition of the token IF contains the qualifier /TOPIC_STRING="statement IF_THEN_ELSE". If a user gives the HELP/INDICATED command while positioned on the token IF in a Pascal buffer, LSE forms the string "PASCAL STATEMENT IF_THEN_ELSE", and uses that as an index into the default help library SYS$HELP:SYSHELP.HLB.

If there is a common prefix for all the help topics for a language, the /TOPIC_STRING qualifier should not be specified. If you have created your own help library for the language, and help for this language is the only topic in this help library, then the /TOPIC_STRING qualifier should not be specified. This is because the single topic in the library is looked up by default.

The /IDENTIFIER_CHARACTERS qualifier specifies what characters are considered as part of a word. A word in LSE is defined as either any nonwhite space, nonidentifier character, or a sequence of identifier characters delimited by white space or a nonidentifier character. In addition to all alphanumeric characters, you may want to have other characters, such as those that often appear in variable names (eg "$" or "_"), be identifier characters.

The /PUNCTUATION_CHARACTERS qualifier specifies those characters that LSE considers as punctuation characters. LSE uses punctuation characters in two ways, as follows (both affect how LSE deletes placeholders):

- When a placeholder is erased and only punctuation characters are left on the line, the line is deleted and the punctuation characters are moved to the end of the preceding line. For example:

  ```
          INTEGER ( x,
                    [var]);
  ```

  When the placeholder *[var]* is erased, only "");) is left on the line. Assuming both "") and ";" are specified as punctuation characters for the language, "");) is moved to the end of the preceding line. (The comma is also erased if specified as the separator for *var*.

- When a placeholder is erased, LSE examines the characters immediately before and after the placeholder. If neither of these are punctuation characters, LSE inserts a blank between them. For example:

  ```
  IF [NOT]{boolean_expression} THEN
  ```

  If the optional placeholder is deleted, and the brace is a punctuation character, the following results:

  ```
  IF{boolean_expression} THEN
  ```

  Thus, in this case it would be better to have the brace ({) not be included as a punctuation character.

The /INITIAL_STRING qualifier specifies the initial text that appears in a newly created buffer. For example, when you create a new file with an extension of .EXAMPLE, the buffer contains just the initial string *{program_unit}*.

The /LEFT_MARGIN and /RIGHT_MARGIN qualifiers specify the left and right margin values, respectively, to be associated with the language. /LEFT_MARGIN=CONTEXT_DEPENDENT means that the left margin varies depending on the surrounding context. For example, the FILL command uses the indentation of the first line of the text being filled as the left margin for the fill operation. When wrap mode is enabled, the right margin value is used to determine where to break the line. It is also used by the FILL command. The FILL command packs text up to, but not beyond, the right margin. When filling comments, the close comment delimiters are placed at the right margin.

The /PLACEHOLDER_DELIMITERS qualifier specifies the starting and ending strings that delimit placeholders. Placeholders can be required or optional, and they may specify single constructs or lists of constructs.

By convention, braces are used for required placeholder delimiters, and square brackets are used for optional delimiters in all Digital languages. If these characters are part of the syntax of the language, then some other character is used with them.

The /TAB_INCREMENT qualifier indicates that tab stops are set at every four columns.

The /QUOTED_ITEM qualifier specifies the characters to be used in the language as delimiters for strings and escape characters. For example, /QUOTED_ITEM=(QUOTES="""'"", ESCAPE="\") (which is the specification for some VAX languages) specifies that strings in the language may be delimited by either quotes (") or apostrophes ('), and that the escape character for the language is the backslash (\ ). LSE always assumes that the strings begin and end with the same character, although different strings can use different delimiters. Note that DCL quoting conventions are used here, so that the value being provided for the /QUOTED_ITEM qualifier must be surrounded by quotes; if you want to include the quote character within this value, you must use double quotes.

You can optionally use the /VERSION qualifier to help you keep track of your product. LSE does not actually use this value, except to display it in the SHOW LANGUAGE command. You may want to use it to show what revision number the language definition is, for example, or what version of the associated compiler the templates support.

The /WRAP qualifier specifies whether or not wrap mode is in effect by default in the language. The /WRAP qualifier affects the left margin value if the /LEFT_MARGIN qualifier was specified with a value of CONTEXT_DEPENDENT.

15.2.2 Defining Language Elements

Now you can define the placeholders that will be used in this language. To define a placeholder, use the DEFINE PLACEHOLDER command (see the command descriptions in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details). Referring to the syntax summary, start with the top-level nonterminal of the language, *program_unit*. This is a nonterminal placeholder.

```
DEFINE PLACEHOLDER program_unit -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Program unit" -
  /DUPLICATION = CONTEXT_DEPENDENT -
  /TYPE = NONTERMINAL
  "-- [procedure level comments]"
  ""
  "PROCEDURE {procedure_name} ([parameter list]...) IS"
  ""
  "    [variable_declaration]...;"
  ""
  "BEGIN"
  ""
  "    [statement]...;"
  ""
  "END {procedure_name};"
  ""
END DEFINE
```

The /LANGUAGE qualifier on the DEFINE PLACEHOLDER command tells LSE that this placeholder is defined for the language EXAMPLE.

The /DESCRIPTION qualifier supplies the description *"Program unit"* for the *program_unit* placeholder.

The /TYPE qualifier shows that this is a nonterminal placeholder. This means that the body for the placeholder follows. The placeholder body for *program_unit* is specified after the qualifiers. A placeholder body is the template that is inserted into the buffer when you expand *program_unit*. Each line of the placeholder body is enclosed in quotation marks. The quotation marks with no text between them produce blank lines at expansion.

The placeholder body for *program_unit* references other placeholders that have not yet been defined, specifically, *procedure level comments*, *procedure_name*, *parameter list*, *variable_declaration*, and *statement*.

The *procedure level comments* placeholder is enclosed by optional placeholder delimiters. Therefore, if you do not want to have comments at the top of your EXAMPLE program, you can erase the placeholder. LSE recognizes the fact that the placeholder is the only item in a trailing comment. After erasing the placeholder, LSE also erases the comment string and the line. This way, unwanted blank comment lines are not left behind.

```
DEFINE PLACEHOLDER "procedure level comments" -
  /DESCRIPTION = "PROCEDURE LEVEL COMMENT TEMPLATE" -
  /LANGUAGE = EXAMPLE -
  /TYPE = NONTERMINAL
  "PROCEDURE:"
  " "
  "    {tbs}"
  " "
  "AUTHOR:"
  " "
  "    {tbs}"
  " "
  "DESCRIPTION:"
  " "
  "    {tbs}..."
  " "
END DEFINE
```

The placeholder name may contain spaces, but if it does, it must be put in quotation marks.

The *procedure level comments* placeholder is another nonterminal placeholder. It expands into a template for comments at the head of the procedure. Templates like this are useful for projects where you want to be sure certain information appears consistently throughout the project. In this case, there are three pieces of information: PROCEDURE NAME, AUTHOR, and DESCRIPTION. For each category there exists the single placeholder *tbs*. The placeholder body contains no comment characters. When you expand a multiline or vertically duplicating placeholder within a comment, the lines into which it expands have comment characters inserted automatically. For example:

```
-- [procedure level comments]
```

When you expand it, LSE recognizes that "—" is a comment string and repeats the comment string at the beginning of each line of the expansion, thus producing the following:

```
-- PROCEDURE:
--
--     {tbs}
--
-- AUTHOR:
--
--     {tbs}
--
-- DESCRIPTION:
--
--     {tbs}...
--
```

The lines in the placeholder body that are simply a quoted space character correspond to lines in the expansion that contain only the comment delimiter. If the lines in the placeholder body were null strings, the corresponding line in the expansion would be completely empty.

Note that the *tbs* placeholder under DESCRIPTION is a list placeholder, while the other two are not. List placeholders are automatically duplicated when expanded or typed over. The duplicate placeholder is always optional.

You should use required list delimiters to indicate a list that must be replaced by at least one element, for example, *{tbs}...*. You should use optional list delimiters to indicate a list that is optional, for example, *[parameter list]...*.

The *tbs* placeholder is a terminal placeholder. Its definition is as follows:

```
DEFINE PLACEHOLDER tbs -
  /DESCRIPTION = "tbs" -
  /DUPLICATION = VERTICAL -
  /LANGUAGE = EXAMPLE -
  /TYPE = TERMINAL
  "to be specified"
END DEFINE
```

Since you use the placeholder as a list placeholder, you should specify the /DUPLICATION qualifier. It tells LSE how to duplicate the placeholder. In this case, VERTICAL is specified, telling LSE to duplicate the placeholder on the next line. As with the previous example, when you type on the list *tbs* placeholder, it gets duplicated, and LSE automatically inserts the comment string at the beginning of the line.

The /TYPE qualifier indicates that *tbs* is a terminal placeholder. When defining a terminal placeholder, the text can be as many lines as are needed to explain the correct text insertion for the placeholder. You must use quotation marks at the beginning and end of each line of text. In this case, the text *"to be specified"* is displayed when you expand the *tbs* placeholder. When you press any key, the text is removed from the screen.

The next placeholder to be defined is the *procedure_name* placeholder.

```
DEFINE PLACEHOLDER procedure_name -
  /LANGUAGE = EXAMPLE -
  /AUTO_SUBSTITUTE -
  /DESCRIPTION = "Procedure name" -
  /TYPE = TERMINAL
  "A string of letters and digits starting with a letter."
END DEFINE
```

The /AUTO_SUBSTITUTE qualifier specifies that during source code entry, the next occurrence of *procedure_name* is automatically replaced with the same text that you typed over the current *procedure_name* placeholder. This is useful for languages that require matching names.

The /TYPE qualifier indicates that *procedure_name* is also a terminal placeholder. In this case, the text, *"A string of letters and digits starting with a letter"* is displayed when you expand *procedure_name*.

The *parameter list* placeholder is defined next. It expands to a template corresponding to the form for a parameter list given in the syntax summary. Since it is a list placeholder, the /DUPLICATION qualifier is specified. Also the /SEPARATOR qualifier is given.

```
DEFINE PLACEHOLDER "parameter list" -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Parameter list" -
  /DUPLICATION = CONTEXT_DEPENDENT -
  /LEADING = "(" -
  /SEPARATOR = "; " -
  /TRAILING = ")" -
  /TYPE = TERMINAL
  "{param_name}... : {type}"
END DEFINE
```

The definition of *parameter list* is described later in this section.

The /DUPLICATION qualifier tells LSE that the duplicated *parameter list* placeholder should be duplicated according to context, the context being whether or not the placeholder is the only item within its segment.

The /SEPARATOR qualifier indicates that a semicolon (;) is inserted in the text to separate the duplicated placeholders. Note also the trailing space on the /SEPARATOR qualifier. This makes a horizontal expansion more readable.

The expansion contains two placeholders: *param_name* and *type*.

```
DEFINE PLACEHOLDER param_name -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Parameter name" -
  /DUPLICATION = HORIZONTAL -
  /SEPARATOR = ", " -
  /TYPE = TERMINAL
  "A string of letters and digits starting with a letter."
END DEFINE
```

The /DUPLICATION qualifier tells LSE that the *param_name* placeholder should be duplicated horizontally. That is, the duplicated placeholder appears on the same line, following the original placeholder.

The /SEPARATOR qualifier indicates that a comma (,) is inserted in the text to separate the duplicated placeholders. Note, also, the trailing space on the /SEPARATOR qualifier. This makes a horizontal expansion more readable.

The *type* placeholder is defined as a menu placeholder.

```
DEFINE PLACEHOLDER type -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Type" -
  /TYPE = MENU -
  "INTEGER"/DESCRIPTION="Integer data type"
  "BOOLEAN"/DESCRIPTION="Boolean data type"
END DEFINE
```

The /TYPE qualifier indicates that *type* is a menu placeholder.

When *type* is expanded, a menu is displayed in the buffer. The placeholder body for a menu placeholder is defined in a similar way as a nonterminal placeholder and contains the elements of the menu.

The expansion of *variable_declaration* looks much like the expansion of *parameter list*. The difference is the addition of an optional *initial_value* placeholder. In the EXAMPLE language, the placeholder allows you to optionally initialize the variables you are declaring.

```
DEFINE PLACEHOLDER variable_declaration -
    /LANGUAGE=EXAMPLE -
    /DESCRIPTION="Declare a variable" -
    /DUPLICATION=VERTICAL -
    /SEPARATOR=";" -
    /TRAILING=";" -
    /TYPE=NONTERMINAL
    "{identifier}... : {type} := [initial_value]..."
END DEFINE
```

If you do not have any variables to declare in your program, you can erase the optional *variable_declaration* placeholder. But you do not want to leave the semicolon (;) behind. The /TRAILING qualifier in the definition of *variable_declaration* /TRAILING=";" specifies that the string ";" be associated with the *variable_declaration* placeholder. If the placeholder is deleted, LSE looks immediately to the right (not including white space). If it sees the trailing string, it deletes it.

The *initial_value* placeholder is defined as follows:

```
DEFINE PLACEHOLDER "initial_value" -
    /DESCRIPTION = "Variable declaration initial value" -
    /DUPLICATION = HORIZONTAL -
    /LANGUAGE = EXAMPLE -
    /LEADING = ":=" -
    /SEPARATOR = ", " -
    /TYPE = TERMINAL
    "The initial value of identifier - either Integer or Boolean"
END DEFINE
```

If you do not want to initialize the variables, you can erase the optional placeholder. But you do not want to leave the ":=" behind. This is analogous to the ";" after the *variable_declaration* placeholder. In this case, use the /LEADING qualifier in the definition of *initial_value*. /LEADING= ":=" specifies that the string ":=" be associated with the *initial_value* placeholder. If the placeholder is deleted, LSE looks immediately to the left (not including white space). If it sees the leading string, it deletes it. A placeholder may have both leading and trailing characters associated with it.

The preceding descriptions of the /LEADING and /TRAILING qualifiers are not quite complete. Based on the descriptions given thus far, the definition of *parameter_list*, repeated here for convenience, would appear to be wrong.

```
DEFINE PLACEHOLDER "parameter list" -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Parameter list" -
  /DUPLICATION = CONTEXT_DEPENDENT -
  /LEADING = "(" -
  /SEPARATOR = "; " -
  /TRAILING = ")" -
  /TYPE = TERMINAL
  "{param_name}... : {type}"
END DEFINE
```

For example, after a few expansions of *program_unit*, the following code fragment may result:

```
PROCEDURE x (a : INTEGER; [parameter list]...) IS
```

Deleting optional placeholders based on the previous description incorrectly leaves the following:

```
PROCEDURE X (a : INTEGER IS
```

That is, both the separator string ";", and the trailing string "") have been removed. But if you go through these steps, using the EXAMPLE language provided with LSE, you find that this does not happen. What actually results is correct, as follows:

```
PROCEDURE X (a : INTEGER) IS
```

The full procedure that LSE goes through when a placeholder is deleted follows:

1. Is the placeholder in a comment? If so, delete the comment characters as well (leading, trailing, and separator characters are not used in this situation).
2. Is a separator defined for the placeholder? Is the placeholder preceded by the separator? If so, delete it and skip the next step.
3. Have leading or trailing strings been defined for the placeholder? If so, look for the leading and trailing strings, and if they are found, delete them.
4. If, after these deletions, nothing or only punctuation characters remain on the line, then delete the white space, except one blank.
5. Finally, if one or both characters immediately to the left and right of the deletion are punctuation characters, delete all white space.

In the example, step 2 causes the *parameter_list* placeholder to work as you want it to. Because the placeholder was expanded, a separator character was present. After LSE found and deleted it, LSE did not go on to look for leading or trailing strings. Thus, the parentheses correctly remained around the parameter.

The *statement* placeholder is defined as a menu placeholder. It is also used as a list placeholder, as specified in the *program_unit* placeholder definition.

```
DEFINE PLACEHOLDER statement -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "EXAMPLE statements" -
  /DUPLICATION = VERTICAL -
  /SEPARATOR = ";" -
  /TYPE = MENU
  "ASSIGNMENT" /TOKEN
  "IF" /TOKEN
  "LOOP" /TOKEN
  "EXIT" /TOKEN
END DEFINE
```

The /DUPLICATION qualifier tells LSE that the *statement* list placeholder should be duplicated vertically. The /SEPARATOR qualifier indicates that a semicolon is used to separate the duplicated placeholders.

The /TOKEN qualifier on the menu options indicates that ASSIGNMENT, IF, LOOP, and EXIT are tokens. They are defined as tokens so that you can type ASSIGNMENT, IF, LOOP, or EXIT directly into the buffer.

When *statement* is expanded, a menu is displayed and another copy of *statement* with optional list delimiters is inserted on the next line in the buffer. When this menu is displayed, the text strings associated with the /DESCRIPTION qualifier on the token definitions is displayed as well. These text strings also appear when you issue the SHOW TOKEN command.

Token Definitions

Tokens are defined for keywords or punctuation characters that you want to type directly into the buffer. When expanded, they provide templates for corresponding language constructs.

ASSIGNMENT, IF, LOOP, and EXIT are defined as tokens. To define a token, use the DEFINE TOKEN command, as follows.

```
DEFINE TOKEN ASSIGNMENT -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Assignment statement"
  "{identifier} := {expression}"
END DEFINE
```

The /LANGUAGE qualifier tells LSE that this token is defined for the EXAMPLE language.

The /DESCRIPTION qualifier supplies the description *"Assignment statement"* for the token ASSIGNMENT. This text string appears in the menu when you expand *{statement}* . . . as well as when you issue the SHOW TOKEN command.

Token names may consist of any combination of characters. However, token names may not have leading or trailing white space. This feature allows you to define tokens with names that are suited for the context in which they may be used. It would be useful to define the assignment operator, colon equal sign (:=), as a token so that if you wanted to insert an assignment operation into an editing buffer, you would type the assignment operator (:=) followed by the EXPAND key and have *{identifier} := {expression}* placed in the editing buffer.

In addition to the ASSIGNMENT token, the assignment operator (:=) is defined as a token, as follows:

```
DEFINE TOKEN ":=" -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Assignment statement"
  "{identifier} := {expression}"
END DEFINE
```

The IF statement contains an optional *ELSE statement_list* construct, as indicated in the syntax summary. Therefore, you must include the construct in the IF token definition and define an additional placeholder for *ELSE statement_list*.

```
DEFINE TOKEN IF -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "IF {expression} THEN..."
  "IF {boolean_exp}"
  "THEN"
  "     {statement}...;"
  "[ELSE {statement}...]"
  "ENDIF"
END DEFINE

DEFINE PLACEHOLDER "ELSE {statement}..." -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "ELSE {statement}"
  "ELSE"
  "     {statement}...;"
END DEFINE
```

The definition for the LOOP token is as follows:

```
DEFINE TOKEN "LOOP" -
  /DESCRIPTION = "[loop_id]: LOOP ... END LOOP;" -
  /LANGUAGE = EXAMPLE
  "[loop_id]: LOOP"
  "     {statement}...;"
  "END LOOP"
END DEFINE
```

The colon (:) following the optional *loop_id* placeholder presents a problem analogous to the ";" after the *variable_declaration* placeholder. That is, if the optional placeholder is erased, the colon is left behind.

```
DEFINE PLACEHOLDER "loop_id" -
  /DESCRIPTION = "LOOP LABEL" -
  /LANGUAGE = EXAMPLE -
  /TRAILING = ":" -
  /TYPE = TERMINAL
  "A string of letters and digits starting with a letter."
END DEFINE
```

Once again, use the /TRAILING qualifier. If the *loop_id* placeholder is erased, LSE looks immediately to the right for the colon (:). If it is found, it is erased.

Finally, the definition for the token EXIT follows:

```
DEFINE TOKEN EXIT -
  /DESCRIPTION = "EXIT [loop_id] WHEN ... " -
  /LANGUAGE = EXAMPLE -
  "EXIT [loop_id] WHEN {boolean_exp}"
END DEFINE
```

All placeholders referenced in token definitions must be defined. The *identifier* placeholder is a terminal placeholder and is defined as follows:

```
DEFINE PLACEHOLDER identifier -
  /LANGUAGE = EXAMPLE -
  /DESCRIPTION = "Identifier" -
  /TYPE = TERMINAL
  "A string of letters and digits starting with a letter."
END DEFINE
```

In the syntax summary, *expression* is specified as a menu placeholder and is defined as follows:

```
DEFINE PLACEHOLDER expression -
    /LANGUAGE=EXAMPLE -
    /DESCRIPTION="Boolean or arithmetic expression" -
    /TYPE=MENU
    "boolean_exp" /PLACEHOLDER
    "arithmetic_exp" /PLACEHOLDER
END DEFINE
```

You use the /PLACEHOLDER qualifier on the menu items to specify that *boolean_exp* and *arithmetic_exp* are placeholders. As such, they need to be defined. When you expand *{expression}* and select one of the menu options, the /PLACEHOLDER qualifier causes the expansion of the menu item to be placed into the buffer, not the menu item itself.

```
DEFINE PLACEHOLDER boolean_exp -
    /LANGUAGE = EXAMPLE -
    /DESCRIPTION = "Boolean expression" -
    /TYPE = NONTERMINAL
    "[NOT] {identifier} [relop identifier]"
END DEFINE

DEFINE PLACEHOLDER "arithmetic_exp" -
    /LANGUAGE = EXAMPLE -
    /DESCRIPTION = "Arithmetic expression" -
    /TYPE = TERMINAL
    "An arithmetic expression"
END DEFINE
```

For example, if you select *boolean_exp* from the menu, the following is inserted into the buffer:

```
    "[NOT] {identifier} [relop identifier]"
```

You would have to expand *{boolean_exp}* again to have it inserted into the buffer. *arithmetic_exp* is a terminal placeholder, so selecting it causes *{arithmetic_exp}* to appear in the buffer along with its expanded description. The same thing occurred earlier, in the definition of *statement*. ASSIGNMENT (for example) was a menu item, with the /TOKEN qualifier appended. Choosing ASSIGNMENT from the menu causes the expansion of the ASSIGNMENT token to appear in the buffer.

If you do not want automatic expansion, the definition of *expression* would appear as follows:

```
DEFINE PLACEHOLDER expression -
    /LANGUAGE=EXAMPLE -
    /DESCRIPTION="Boolean or arithmetic expression" -
    /TYPE=MENU
    "{boolean_exp}"
    "{arithmetic_exp}"
END DEFINE
```

NOT is a keyword, so it should be defined as a token. But it is also used as a placeholder in the expansion of *boolean_exp*. This is an example of a situation in which you could use the alternate form of the DEFINE TOKEN command.

```
DEFINE TOKEN NOT -
    /PLACEHOLDER = NOT

DEFINE PLACEHOLDER NOT -
    /DESCRIPTION = "KEYWORD NOT" -
    /LANGUAGE = EXAMPLE -
    /TYPE = NONTERMINAL
    "NOT"
END DEFINE
```

The token *NOT* takes its parameters from the placeholder *NOT*. That is, even though it is not explicitly specified, the token is associated with the EXAMPLE language and has the description “Keyword NOT.”

The final placeholders are defined as follows:

```
DEFINE PLACEHOLDER "relop identifier" -
    /LANGUAGE = EXAMPLE -
    /DESCRIPTION = "Optional part of a boolean expression" -
    /TYPE = NONTERMINAL
    "{relop} {identifier}"
END DEFINE

DEFINE PLACEHOLDER relop -
    /LANGUAGE = EXAMPLE -
    /DESCRIPTION = "rel op" -
    /TYPE = MENU
    "="
    "<>"
    "<"
    ">"
    "<="
    ">="
END DEFINE

DEFINE PLACEHOLDER "arithmetic_exp" -
    /LANGUAGE = EXAMPLE -
    /DESCRIPTION = "arithmetic expression" -
    /TYPE = TERMINAL
    "An arithmetic expression"
END DEFINE
```

Additional placeholders for the arithmetic expressions are not defined. Rather, the *arithmetic_exp* placeholder is defined as a terminal placeholder to notify you that an arithmetic expression is required. A choice such as this is strictly up to the author of the templates. It all depends upon the level of detail you want. A relevant criterion might be, for example, the knowledge of the people using the templates.

Now that the template definition is complete, you must execute the commands in your source file. Refer to Section 15.3 for details on executing your source file and creating an environment file for later use.

15.3 Saving Language Definitions

To create and save an environment file for a new language, use the following steps:

1. Create your source file in an empty buffer. The source file should have an extension of .LSE.
2. Put all the necessary LSE language, token, and placeholder definitions into that source file.
3. Execute the commands in the source file with the DO command while in LSE.

   This loads the definitions into the current editing session but does not save them.
4. Issue the SAVE ENVIRONMENT command while in LSE.

   The SAVE ENVIRONMENT command saves your definitions and produces a binary file with a file extension of .ENV.

The following sequence of commands must be issued to create this binary file (press CTRL/Z to get the LSE> prompt):

```
LSE>  DO
LSE>  SAVE ENVIRONMENT filename
LSE>  EXIT
```

The SAVE ENVIRONMENT command writes out all user-defined languages, placeholders, tokens and aliases to an editing environment file. This command executes immediately to save the environment active at the moment you issue the command.

To use your environment file, you must specify the following command when invoking LSE:

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

To automatically access your environment file, add the following command to your LOGIN.COM file:

```
$  DEFINE LSE$ENVIRONMENT device:[directory]filename.ENV
```

15.4 Indentation Control

LSE controls the indentation of tokens and placeholders that are placed in an editing buffer by the EXPAND command. Tabulation is determined by the format of the placeholder or token template. In addition, LSE provides the SET TAB_INCREMENT command, which enables you to adjust indentation to your needs.

When LSE expands a template, it inspects each line of the template for leading tab characters. Each tab character is evaluated using the current TAB_INCREMENT setting. Thus, if the TAB_INCREMENT is three, then a leading tab is expanded into three spaces. The initial TAB_INCREMENT for a buffer is taken from the language definition. You can change the TAB_INCREMENT setting with the SET TAB_INCREMENT command.

The starting column for the expansion is determined by the first line of the template. If the first line is null (an empty string), then the expansion starts under the first nonblank character in the line containing the item being expanded. If the line is not empty, then the expansion starts in the first character position of the item being expanded.

For example, suppose you have a placeholder with <TAB> indicating a single ASCII tab character defined as follows:

```
DEFINE PLACEHOLDER RECORD_TYPE_DEFINITION -
   . . .
    /TYPE=NONTERMINAL
    ""
    "TABrecord"
    "TABTAB[component declaration]..."
    "TABTAB[variant_part]"
    "TABend record"
END DEFINE
```

You have the following text:

```
type MYTYPE is {record_type_definition};
```

If you expand the placeholder with the TAB_INCREMENT set to 4, the result is as follows:

```
type MYTYPE is
    record
        [component declaration]...
        [variant_part]
    end record
```

The third and fourth lines each have a single leading tab and no leading spaces, while the second and fifth lines each have four leading spaces. This is because LSE compresses spaces into tabs after positioning the expanded text.

Using the same example text, remove the first empty line in the placeholder definition and the first tab from the remaining lines, as follows:

```
DEFINE PLACEHOLDER RECORD_TYPE_DEFINITION -
    . . .
    /TYPE=NONTERMINAL
    "record"
    " TAB [component_declaration]..."
    " TAB [variant_part]"
    "end record"
END DEFINE
```

Then, if you expand the placeholder with the TAB_INCREMENT set to 4, the result is as follows:

```
type MYTYPE is record
                   [component_declaration]...
                   [variant_part]
               end record
```

Now, insert a line containing a single blank at the front of the definition, as follows:

```
DEFINE PLACEHOLDER RECORD_TYPE_DEFINITION -
    . . .
    /TYPE=NONTERMINAL
    " "
    "record"
    " TAB [component_declaration]..."
    " TAB [variant_part]"
    "end record"
END DEFINE
```

Then, if you expand the placeholder with the TAB_INCREMENT set to 4, the result is as follows:

```
type MYTYPE is
               record
                   [component_declaration]...
                   [variant_part]
               end record
```

For more information about indentation control, see the DEFINE PLACEHOLDER, DEFINE TOKEN, and EXPAND commands in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual.*

15.5 Defining a Package

The following section provides instructions on how to define your own packages.

You can use the following commands to create package definitions:

- DEFINE PACKAGE
- DEFINE ROUTINE
- DEFINE PARAMETER

The DEFINE PACKAGE command defines a subroutine package for which subroutine-call templates are automatically generated. Packages can contain routine definitions that describe calls to subroutines, and parameter definitions that describe parameters for subroutine calls. The DEFINE ROUTINE command defines templates for a routine within a subroutine package. This command makes the routine an element of a package. The DEFINE PARAMETER command defines a parameter within a package.

The following is an example of a package definition:

```
DEFINE PACKAGE system_services -
   /LANGUAGES =(BASIC,C,COBOL,FORTRAN,PLI) -
   /HELP_LIBRARY = HELPLIB -
   /TOPIC_STRING = "system_services" -
   /ROUTINE_EXPAND = "LSE$PKG_EXPAND_ROUT_" -   ! Special routines for
   /PARAMETER_EXPAND = "LSE$PKG_EXPAND_PARM_"   ! system services
```

LSE provides two sets of predefined VAXTPU routines to help you write your own packages. The sets of routines are useful in most cases. However, you may want to create your own VAXTPU procedures to help you write more complex packages. For more details, see the appendix in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* that describes writing VAXTPU procedures for the package facility.

15.5.1 Routine Definitions

When you define a routine, LSE automatically generates the necessary DEFINE TOKEN or DEFINE PLACEHOLDER command to produce the templates for the given procedure call. Thus, you can expand and unexpand routines in the same manner as tokens. The following is an example of a routine definition and the corresponding LSE command DEFINE TOKEN:

```
DEFINE ROUTINE sys$add_holder -
   id/BY_VALUE, -
   holder/BY_REFERENCE, -
   attrib/BY_VALUE/OPTIONAL -
   /PACKAGE = system_services -
   /DESCRIPTION = "Add Holder Record To The Rights Database"
```

LSE generates a DEFINE TOKEN sys$add_holder command with the appropriate body for the current language.

15.5.2 Parameter Definitions

You can associate a parameter with more than one routine within a package. You use the DEFINE ROUTINE command to associate parameters with routines. The following are examples of parameter definitions:

```
DEFINE PARAMETER id -
   /PACKAGE = system_services -
DEFINE PARAMETER holder -
   /PACKAGE = system_services -
DEFINE PARAMETER attrib -
   /PACKAGE = system_services -
```

Chapter 16

Providing Diagnostic File Support

Diagnostic files communicate diagnostic messages to LSE from various tools. A tool, such as a compiler, generates a diagnostic file that LSE uses to display the diagnostics. Once you have displayed a diagnostic file in LSE, you can navigate through the file from one diagnostic to the next. You can use the GOTO SOURCE command to display the source that corresponds to a diagnostic in another window.

There are two formats for diagnostic files:

- User-file format
- Digital internal-file format

The user-file format provides a simple format for customer tools to communicate diagnostic information to LSE. You can list this format without a special dump utility.

The Digital internal-file format is a binary format that is used by Digital products to communicate diagnostic messages to LSE.

You can concatenate user-file and Digital internal-file diagnostic modules into one file and review them together.

Typically, a tool generates a module of zero or more diagnostics each time it processes a source file. For example, a compiler generates a diagnostic module for each compilation. Diagnostics typically are errors. Each diagnostic consists of the following:

- Regions
- Messages

Regions define the location of the source that is associated with the diagnostic. There can be more than one region.

Messages are textual descriptions that explain the diagnostic. There can be more than one message.

The rules that apply to DCL apply to the user-file format. For example, nonquoted strings are converted to uppercase.

Section 16.1 shows an example diagnostic module in the user-file format and explains how the module is used. Section 16.2 describes each of the commands that are used in the user-file format.

16.1 User-File Format Example

Example 16–1 shows a diagnostic module in the user-file format. Comments are introduced by an exclamation mark (!).

Key to Example 16–1

❶ The first diagnostic shows how regions and messages work together.

❷ The file regions refer to lines in the source that cause the error described in the text message.

❸ The nested regions in each of the file regions refer to the location in each line that contributes to the error.

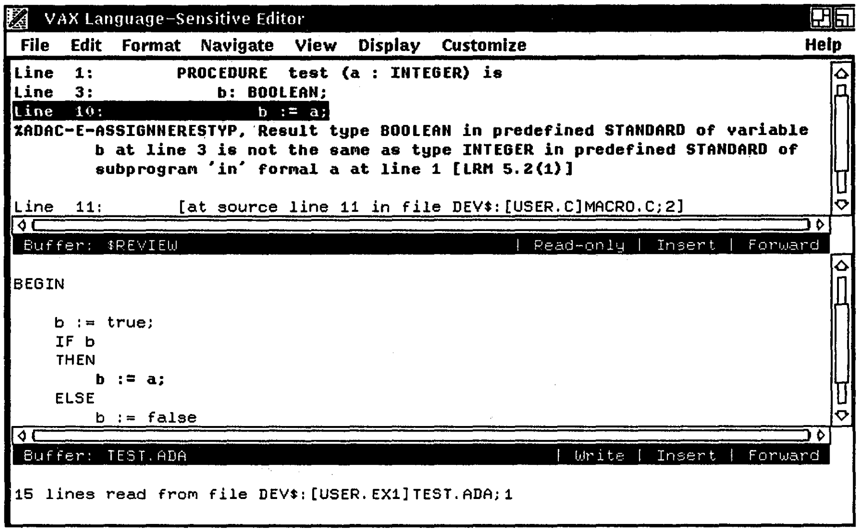
❹ The second diagnostic shows how a text region can be used to display macro text for error messages.

Figure 16–1 shows the screen after the GOTO SOURCE command is executed with the cursor positioned on the line number of the first diagnostic. The source file is displayed in the lower window, and the cursor is positioned at the beginning of the innermost nested region of the primary region.

Example 16–1: User-File Format Diagnostic

```
start module ! This command signals the start of a module.
❶ start diagnostic    ! This region marks line 1 in the file, and
! it is not a primary region.
❷ region/file DEV$:[user.ex1]test.ada;1/line=1/column_range=(1,65535)
❸ region/nested/column_range=(18)! Marks the 18th column in the above region.
! 2nd region
region/file DEV$:[user.ex1]test.ada;1/line=3/column_range=(1,65535)
! The following nested region marks column 4 of line 3 for the file specified above.
region/nested/column_range=(4)! Marks the 4th column in the above region.
! This is the primary region that LSE will highlight when positioned on this
! diagnostic.
region/file DEV$:[user.ex1]test.ada;1/line=10/column_range=(1,65535)-
  /primary    !  This region marks all of line 10 in the file.
region/nested/column_range=(4,4) ! Specifies a subregion at the above region.
! Messages
message/text=quoted "%ADAC-E-ASSIGNNERESTYP, Result type BOOLEAN in pre ..."
message/text=quoted "        b at line 3 is not the same as type INTEGER ..."
message/text=quoted "        subprogram 'in' formal a at line 1 [LSM 5.2(1)]"
end diagnostic
! The next example is taken from a C diagnostic.  The file region refers to a line
! in the text that contains a macro call and the text supplied by the text
! region is the macro expansion.
❹ start diagnostic
region/file DEV$:[user.c]macro.c;2/line=11/column_range=(5,25)-
        /primary
region/text "    if (i>0) j=k else l=m;"-
        /line=1/column_range=(1,26)
message/text=quoted "%CC-W-INSBEFORE, Insert "";"" before reserved word ..."
end diagnostic
end module
```

Figure 16–1: First Diagnostic and Corresponding Source


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
Line  1:        PROCEDURE  test (a : INTEGER) is
Line  3:            b: BOOLEAN;
Line  10:               b := a;
%ADAC-E-ASSIGNMERESTYP, Result type BOOLEAN in predefined STANDARD of variable
        b at line 3 is not the same as type INTEGER in predefined STANDARD of
        subprogram 'in' formal a at line 1 [LRM 5.2(1)]

Line  11:       [at source line 11 in file DEV$:[USER.C]MACRO.C;2]
```

Buffer: $REVIEW | Read-only | Insert | Forward

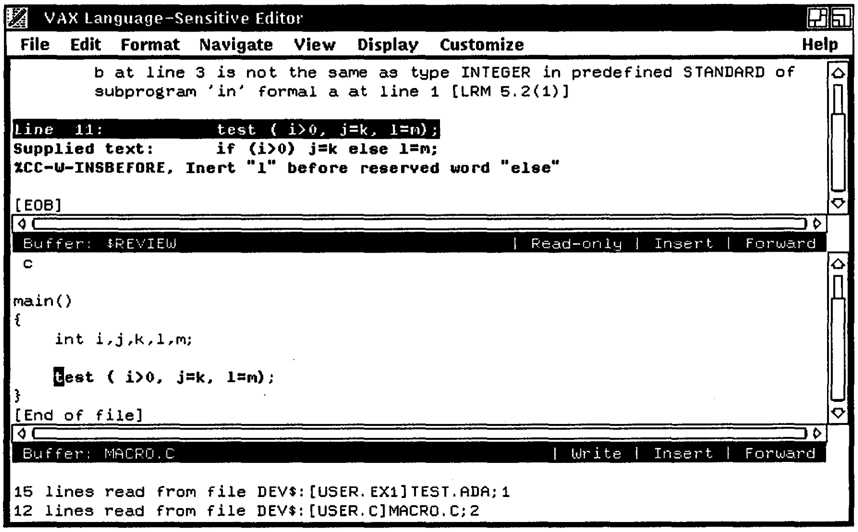
```
BEGIN

    b := true;
    IF b
    THEN
        b := a;
    ELSE
        b := false
```

Buffer: TEST.ADA | Write | Insert | Forward

```
15 lines read from file DEV$:[USER.EX1]TEST.ADA;1
```

Figure 16–2 shows the screen after the GOTO SOURCE command is executed with the cursor positioned on the line number of the second diagnostic. The supplied text from the text region refers to the expanded macro “test” in the source.

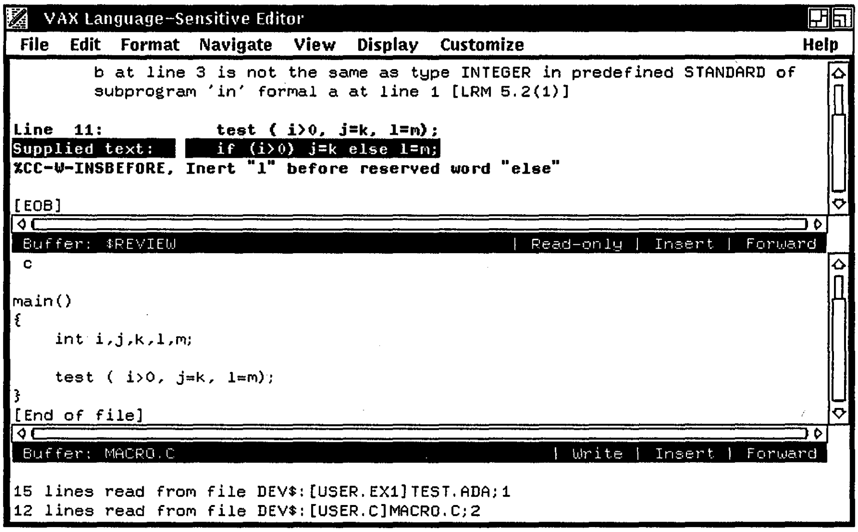
Figure 16–2: Second Diagnostic and Corresponding Source


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
        b at line 3 is not the same as type INTEGER in predefined STANDARD of
        subprogram 'in' formal a at line 1 [LRM 5.2(1)]

Line  11:          test ( i>0, j=k, l=m);
Supplied text:     if (i>0) j=k else l=m;
%CC-W-INSBEFORE, Inert "l" before reserved word "else"

[EOB]
 Buffer: $REVIEW                                | Read-only | Insert | Forward
 c

main()
{
    int i,j,k,l,m;

    test ( i>0, j=k, l=m);
}
[End of file]
 Buffer: MACRO.C                                    | Write | Insert | Forward

15 lines read from file DEV$:[USER.EX1]TEST.ADA;1
12 lines read from file DEV$:[USER.C]MACRO.C;2
```

Figure 16–3 shows the screen after the GOTO SOURCE command is executed with the cursor positioned on the supplied text of the second diagnostic. Since the NEXT and PREVIOUS ERROR commands always position the cursor on the line of the diagnostic that is the primary region, you must use the arrow keys to position the cursor on the supplied text. The area defined for the text region is underlined after GOTO SOURCE is executed.

Figure 16–3: Supplied Text of the Second Diagnostic and Corresponding Source


VAX Language-Sensitive Editor
File Edit Format Navigate View Display Customize Help

```
        b at line 3 is not the same as type INTEGER in predefined STANDARD of
        subprogram 'in' formal a at line 1 [LRM 5.2(1)]

Line  11:            test ( i>0, j=k, l=m);
Supplied text:         if (i>0) j=k else l=m;
%CC-W-INSBEFORE, Inert "l" before reserved word "else"

[EOB]
Buffer: $REVIEW                                    | Read-only | Insert | Forward
 c

main()
{
    int i,j,k,l,m;

    test ( i>0, j=k, l=m);
}
[End of file]
Buffer: MACRO.C                                        | Write | Insert | Forward

15 lines read from file DEV$:[USER.EX1]TEST.ADA;1
12 lines read from file DEV$:[USER.C]MACRO.C;2
```

16.2 User-File Format Command Descriptions

The following section describes the commands that define the user-file format.

END DIAGNOSTIC

Ends a diagnostic that begins with a START DIAGNOSTIC command.

Format

END DIAGNOSTIC

Description

The END DIAGNOSTIC command ends a sequence of commands that make up a diagnostic.

Example

See Example 16–1 for a sample of the END DIAGNOSTIC command.

END MODULE

Ends a module in the user-file format that begins with the START MODULE command.

Format

END MODULE

Description

The END MODULE command ends a sequence of commands that make up a user-file format diagnostic module.

Example

See Example 16–1 for a sample of the END MODULE command.

MESSAGE/FILE

Defines a message in a file for a diagnostic that appears in the REVIEW buffer during a review session.

Format

MESSAGE/FILE *file-spec*

Command Parameter

file-spec
Specifies the file containing the message.

Description

The MESSAGE/FILE command specifies a file that contains the message to be displayed in the REVIEW buffer for a diagnostic. The entire file is displayed in the REVIEW buffer. The message is usually an error message.

Example

```
MESSAGE/FILE DEV$:[USER]MESSAGE.TXT
```

The contents of the file specified are displayed as the message in the REVIEW buffer.

MESSAGE/TEXT

Defines a quoted or unquoted message for a diagnostic that appears in the REVIEW buffer during a review session.

Format

MESSAGE/TEXT=[UN]QUOTED *message-definition*

Command Parameter

message-definition
Specifies the message.

Description

The MESSAGE/TEXT=QUOTED command specifies that the message for the diagnostic is a quoted string. A quoted message is enclosed in quotes ("") with double quotes ("""") used to embed quotes in the string.

The MESSAGE/TEXT=UNQUOTED command specifies that the message is the remaining text in the line. It does not have to be quoted. Nonquoted text is converted to uppercase, and leading and trailing white space is removed.

The message is usually an error message. If no qualifier is specified for the MESSAGE command, /TEXT_QUOTED is the default.

Examples

1. `MESSAGE/TEXT=UNQUOTED  Here is another message.`

   This message is displayed in the REVIEW buffer. Leading and trailing white space is truncated and the lowercase letters are converted to uppercase, as follows:

   ```
   HERE IS ANOTHER MESSAGE.
   ```

2. `MESSAGE "Inserted ";" at end of line"`

   If no qualifier is specified or /TEXT alone is specified, the default becomes /TEXT=QUOTED. This message is included in the REVIEW buffer without the beginning and trailing quotes, as follows:

   ```
   Inserted ";" at end of line.
   ```

REGION/FILE

Specifies that the source location associated with a diagnostic is in a file.

Format

REGION/FILE *file-spec*

Command Qualifiers	**Defaults**
/LINE=number	/LINE=1
/COLUMN_RANGE=(number,number)	/COLUMN_RANGE=(1,1)
/LABEL=string	See text
/PRIMARY	

Command Qualifiers

/LINE=number
/LINE=1 (D)
Specifies the line number in the file for the region. The first line in a file is 1. The valid range for the /LINE qualifier is –1 and higher. The –1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or –1, any column range values specified are ignored.

/COLUMN_RANGE=(number,number)
/COLUMN_RANGE=(1,1) (D)
Specifies a range of columns in the file that defines the region. If only the first number is specified, then the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string

Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is Line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY

Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

file-spec

Specifies the file that contains the region. The full file specification for the file region, which includes device, directory, and version, should be used to help ensure that LSE accesses the correct file when the GOTO SOURCE command is executed.

Description

The REGION/FILE command defines an area in a file that is associated with a diagnostic. This area cannot span more than one line. If /FILE, /LIBRARY, /NESTED, or /TEXT is not specified with the REGION command, /FILE is the default and therefore need not be typed.

Example

```
REGION/FILE DEV$:[user]program.src;23 -
            /LINE=10 -
            /COLUMN_RANGE=1 -
            /Label="Src Line 10:" -
            /PRIMARY
```

This region points to the first column of the tenth line in file DEV$:[user]program.src;23. The region has a user-specified label and is a primary region.

REGION/LIBRARY

Specifies that the source location associated with a diagnostic is in a module within a text library.

Format

REGION/LIBRARY *file-spec*

Command Qualifiers	**Defaults**
/MODULE=module-name	
/LINE=number	/LINE=1
/COLUMN_RANGE=(number,number)	/COLUMN_RANGE=(1,1)
/LABEL=string	See text
/PRIMARY	

Command Qualifiers

/MODULE=module-name
Specifies the module in the library that contains the region.

/LINE=number
/LINE=1 (D)
Specifies the line number in the library module for the region. The first line in the module is 1. The valid range for the /LINE qualifier is −1 and higher. The −1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or −1, any column range values specified are ignored.

/COLUMN_RANGE=(number,number)
/COLUMN_RANGE=(1,1) (D)
Specifies a range of columns in a module that defines the region. If only the first number is specified, then the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating

the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string
Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is Line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY
Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

file-spec
Specifies the library that contains the region. The full file specification for the library region, which includes device, directory, and version, should be used to help ensure that LSE accesses the correct file when the GOTO SOURCE command is executed.

Description

The REGION/LIBRARY command defines an area in a library module for a diagnostic. This area cannot span more than one line.

Example

```
REGION/LIBRARY  DEV$:[user]textlib.tlb;3 -
             /MODULE=textmod -
             /LINE=1 -
             /COLUMN_RANGE=(1,65535)
```

This region defines the entire first line in module textmod of library DEV$:[user]textlib.tlb;3. No label is specified, so the default of line 1 is used. This is not a primary region.

REGION/NESTED

Specifies that the source location associated with a diagnostic is a subregion of the previous region.

Format

REGION/NESTED

Command Qualifier	**Default**
/COLUMN_RANGE=(number,number)	/COLUMN_RANGE=(1,1)

Command Qualifier

/COLUMN_RANGE=(number,number)
/COLUMN_RANGE=(1,1) (D)
Specifies a range of columns that define a subregion of the previous region. If only the first number is specified, then the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

Description

The REGION/NESTED command defines an area that is a subregion of the /FILE, /TEXT, /LIBRARY, or /NESTED region. This area cannot span more than one line.

Each type of sequential region (file, text, or library) can have a nested region inside it. Nested regions can have nested regions. However, each subsequent nested region must fit inside the previous region. Regions of the same size are considered to fit inside each other. A nested region cannot appear by itself; it must be a subregion of a sequential region. If more than four nested regions follow a sequential region, the rest are ignored.

If the GOTO SOURCE command is executed when reviewing a diagnostic file, LSE moves to the beginning of the innermost region of the region it is positioned on in the REVIEW buffer.

Example

```
REGION/FILE DEV$:[user]program.src;1 -
            /LINE=10 -
            /COLUMN_RANGE=(1,65535)

REGION/NESTED/COLUMN_RANGE=(2,10)

REGION/NESTED/COLUMN_RANGE=10
```

The nested regions define subregions of the file region. The first nested region defines the area from column 2 to column 10, inclusive, on line 10 in file DEV$:[user]program.src;1. The second nested region defines the last column in that region.

REGION/TEXT

Specifies that the source location associated with a diagnostic is in the text that is included in this command as arguments.

Format

REGION/TEXT *string [,string...]*

Command Qualifiers	**Defaults**
/LINE=number	/LINE=1
/COLUMN_RANGE=(number,number)	/COLUMN_RANGE=(1,1)
/LABEL=string	See text
/PRIMARY	

Command Qualifiers

/LINE=number
/LINE=1 (D)
Specifies the line number of the strings included in the region. The first string included is line 1. The valid range for the /LINE qualifier is –1 and higher. The –1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or –1, any column range values specified are ignored.

/COLUMN_RANGE=(number,number)
/COLUMN_RANGE=(1,1) (D)
Specifies a range of columns in the specified string that defines the region. If only the first number is specified, then the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string
Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is Line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY
Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

string[,string...]
A quoted string (or strings separated by commas) that is the supplied text for this command. This text appears in the REVIEW buffer for the region.

Description

The REGION/TEXT command defines an area in the text supplied with the command for a diagnostic. This area cannot span more than one line.

Example

```
REGION/TEXT "A := B;", -
            "C := D," -
            /LINE=1 -
            /COLUMN_RANGE=(7,7)
            /PRIMARY
```

This region points to the last column of the first supplied string. No label is specified, so the default of supplied text is used. This is a primary region. See Example 16–1 for more samples.

START DIAGNOSTIC

Specifies the start of a diagnostic.

Format

START DIAGNOSTIC

Diagnostic Body

END DIAGNOSTIC

Command Parameter

Diagnostic Body
A diagnostic consists of a START DIAGNOSTIC command, one or more regions, one or more messages, and an END DIAGNOSTIC command.

Description

This command marks the start of a diagnostic module in the user format.

Example

See Example 16–1 for samples of the START DIAGNOSTIC command.

START MODULE

Specifies the start of a diagnostic module in the user format.

Format

START MODULE

Module Body

END MODULE

Command Parameter

Module Body
A module consists of a START MODULE command, zero or more diagnostics, and an END MODULE command.

Description

This command marks the start of a diagnostic module in the user format.

Example

See Example 16–1 for a sample of the START MODULE command.

Chapter 17

Customizing Overviews

This chapter provides guidance for those who need to customize overviews for programming languages, including languages that Digital does not directly support.

Section 17.1 is an introduction. Section 17.2 describes how to customize overviews by using the DEFINE ADJUSTMENT command. Section 17.3 provides information on tab increments, and Section 17.4 provides information on debugging DEFINE ADJUSTMENT definitions.

17.1 Introduction

Overviews hide the details of your source code. An **overview line** is generated by LSE and represents a sequence of lines in a buffer. Each overview line is displayed as a pseudocode placeholder. For example:

```
«--Put each pair of numbers in order...»
```

A **source** or **real line** is a line of code that occurs in a program. For example:

```
if NUMBERS(J) > NUMBERS(J+1) then
```

A **detailed line** is a line that is hidden by an overview line. It can be either a source line or a lower-level overview line.

A **group** is a sequence of lines where the first line is indented less than any other line in the sequence and the same as or more than the first line of text after the sequence.

The grouping of lines depends upon indentation conventions, combined with simple language-specific syntax processing. In the following example, the first five lines constitute a group:

```
WHILE x
    BEGIN
    a = b
    c = d
    END
q = r
```

When the group is compressed, a single line represents the five lines, as follows:

```
«WHILE x...»
q = r
```

You can nest groups of lines, so it is possible to present a single program at various levels of detail. The following example shows an indented, nested group of lines associated with an *if* statement:

```
--Put each pair of numbers in order
for J in 1 .. HOW_MANY - 1 loop
    if NUMBERS(J) > NUMBERS(J+1) then
        --Interchange the numbers
        TEMP := NUMBERS(J);
        NUMBERS(J) := NUMBERS(J+1);
        NUMBERS(J+1) := TEMP;

        --We are not finished sorting
        SORTED := FALSE;
    end if;
end loop;
```

The indented lines can be represented by pseudocode, as follows:

```
--Put each pair of numbers in order
for J in 1 .. HOW_MANY - 1 loop
    if NUMBERS(J) > NUMBERS(J+1) then
        « --Interchange the numbers...»
        « --We are not finished sorting...»
    end if;
end loop;
```

The entire code fragment can be represented by a single overview line, as follows:

```
« --Put each pair of numbers in order...»
```

There are two kinds of indentation to consider: **visible indentation** and **adjusted indentation**. The visible indentation of a line is the column number of the first visible character on the line. The visible indentation of a blank line is 0.

The adjusted indentation of a line is how LSE treats the indentation when it has been "adjusted." You cannot see such adjusted indentation because LSE does not actually move any text; it just treats lines as though the indentation has changed.

Details of the overviews are specific to the language you are using. Sometimes visible indentation information alone is not enough to generate good overviews. To adjust the indentation of your code, use the DEFINE ADJUSTMENT command. With the DEFINE ADJUSTMENT command, you can modify the behavior of overviews to match your formatting conventions. In addition, you can develop overview support for languages that Digital does not provide. You can save the DEFINE ADJUSTMENT commands in LSE environment files.

You can use the VIEW SOURCE/DEBUG command to generate a copy of the source buffer showing the indentation as LSE views the indentation (all source lines visible with numeric values for the indentation).

17.2 Making Adjustments

The following sections describe how to use the DEFINE ADJUSTMENT command to customize overviews. Adjustment definitions are stored in LSE environment files. Each adjustment definition supplies a pattern and a set of adjustment actions. If a source line matches the pattern, LSE applies the corresponding adjustment actions.

The DEFINE ADJUSTMENT command has the following format:

```
DEFINE ADJUSTMENT name [pattern] [action-qualifiers] [/LANGUAGE=language-name]
```

17.2.1 Testing Overviews

If you are going to write definitions for a new language, look at the adjustments for a similar language to learn how to make the adjustments.

If you are going to modify the behavior of overviews to match your coding conventions, you can start with the adjustments shipped with a language. You can see the adjustments defined for a language by typing the following command:

```
LSE>   SHOW ADJUSTMENT *
```

To test overviews, choose or create some typical program text. Your test files need not be complete programs. Then try the VIEW SOURCE command on your test programs. Without adjustment definitions, LSE relies exclusively on visible indentation. You may want to try your definitions on larger, more realistic files later.

This exercise shows you which language constructs need attention. Overviews should make sense. For those language constructs that produce good overviews, you do not need to write definitions. However, you should write DEFINE ADJUSTMENT commands for those portions of the program that do not have good overviews.

The following are some general criteria for judging the quality of the overviews you generate:

- Each major language construct should be compressible to a single line.
- Comments should hide the constructs they describe.
- Low-level constructs should never hide high-level constructs.

17.2.2 Using Adjustment Qualifiers

In the following examples, if you do not specify the /LANGUAGE qualifier, LSE defaults to the language of the current buffer.

17.2.2.1 Adjusting Single Lines

You use the /CURRENT qualifier with most indentation adjustment commands. It instructs LSE to treat the current line as though the indentation were changed. Use it to adjust the indentation of a single line. For example, the following definitions are suitable for use with the parts of *if* statements:

```
DEFINE ADJUSTMENT then /CURRENT=1
DEFINE ADJUSTMENT else /CURRENT=1
```

Consider the following lines of COBOL code:

```
IF P
THEN
    ADD 5 TO K
    END-ADD
ELSE
    ADD 6 TO K
    END-ADD
```

The adjustment definitions cause these lines to be treated as though they were indented as follows:

```
IF P
 THEN
    ADD 5 TO K
    END-ADD
 ELSE
    ADD 6 TO K
    END-ADD
```

The highest-level overview for this fragment of code is as follows:

```
«IF P»
```

This technique works well for any construct that is similarly well-structured and indented and that has multiple keywords at the same indentation level. The idea is to adjust the lines so that the first line appears to be indented less than the lines that follow. Then the first line can serve as an overview for the entire construct.

The following definitions are suitable for use with Ada BEGIN blocks:

```
DEFINE ADJUSTMENT exception /CURRENT=1
DEFINE ADJUSTMENT end       /CURRENT=1
```

When implemented, such definitions cause the following text to be compressed to a single line.

```
begin
    {statement}...
exception
    {exception_handler}...
end;
```

Situations may arise when you want to adjust the indentation of a line to the left instead of to the right. For example, comments should frequently be adjusted to the left. To do this, use negative values.

The following definition for BASIC comments uses a negative value:

```
DEFINE ADJUSTMENT "!" /CURRENT=-1
```

With this adjustment, LSE treats comment lines as though they were shifted one column to the left. For example:

```
! Store away the info needed to update the history.
Update_info::Edit_no = Edit_no
Update_info::Version_record = Rec_num
```

LSE treats the construct as follows:

```
! Store away the info needed to update the history.
 Update_info::Edit_no = Edit_no
 Update_info::Version_record = Rec_num
```

Thus, these lines resolve to the following overview:

```
« --Store away the info needed to update the history...»
```

17.2.2.2 Adjusting Multiple Lines

If a program construct does not have an indented body, then use the DEFINE ADJUSTMENT command to indent the whole construct. For example, if the lines between BLISS BEGIN and END are not indented, you can adjust for that with the following definitions:

```
DEFINE ADJUSTMENT begin /SUBSEQUENT=1
DEFINE ADJUSTMENT end   /SUBSEQUENT=-1
```

These definitions mean "If the word BEGIN occurs, adjust all subsequent lines in the buffer one column to the right. If the word END occurs, adjust subsequent lines one column to the left." These adjustments are cumulative. See Section 17.2.2.3 for more complicated definitions for BEGIN and END.

Consider the following program lines:

```
BEGIN
a
b
END
x
y
```

The result of applying the preceding definitions is as follows:

```
BEGIN
 a
 b
 END
x
y
```

This compresses the BEGIN block to one line.

When you use the /SUBSEQUENT qualifier, be sure the total of the values for a single construct add up to zero. For example, in Ada and Pascal the PROCEDURE keyword sometimes has a matching END and sometimes does not. Therefore, the /SUBSEQUENT qualifier should not be applied to PROCEDURE.

Definitions with the /SUBSEQUENT qualifier need not be in pairs, with values of N and -N. Groups of three or more definitions are acceptable as long as the combined values add up to zero.

17.2.2.3 Interactions of Definitions

Choose adjustment numbers and signs for definitions carefully so that they interact well together. For example, the following set of definitions needs more work:

```
DEFINE ADJUSTMENT begin /SUBSEQUENT=2
DEFINE ADJUSTMENT end   /SUBSEQUENT=-2
DEFINE ADJUSTMENT "!"   /CURRENT=-1
```

Consider these definitions operating on the following text:

```
BEGIN
! Do something.
a
b
END
```

The resulting adjustment is as follows:

```
BEGIN
 ! Do something.
  a
  b
  END
```

When the code after the comment is compressed, the END line is hidden but the BEGIN line is not. When a construct has two "endpoints" it is inappropriate to display one and not the other. To remedy the situation, add a /CURRENT value to the END definition, as follows:

```
DEFINE ADJUSTMENT begin /SUBSEQUENT=2
DEFINE ADJUSTMENT end   /SUBSEQUENT=-2  /CURRENT=-1
DEFINE ADJUSTMENT "!"   /CURRENT=-1
```

Then the preceding sample program code is adjusted as follows:

```
BEGIN
 ! Do something.
  a
  b
 END
```

The comment hides only the two lines immediately following it, not the END statement.

Sometimes it is difficult to decide whether to move text to the left or to the right. For example, if your coding convention has IF and THEN keywords at the same indentation level, and you want the adjusted indentation to have the THEN indented more than the IF, you can adjust the IF to the left or you can adjust the THEN to the right; either way, the THEN gets indented more than the IF. Your choice should be influenced by the interaction of this definition with other definitions, in particular those involving comments.

Moving comments to the left and code to the right works well for most situations.

17.2.2.4 Languages Without Indentation

Languages without indentation can still take advantage of overviews. For example, tools such as VAX DOCUMENT use files containing markup language "tags." Many of these tags are constructs with well-marked beginnings and ends that can be treated as follows:

```
DEFINE ADJUSTMENT "<NOTE>"     /SUBSEQUENT=2
DEFINE ADJUSTMENT "<ENDNOTE>"  /SUBSEQUENT=-2  /CURRENT=-1
```

VAX DOCUMENT markup language constructs that do not have visible endings can be treated as follows:

```
DEFINE ADJUSTMENT "<CHAPTER>"  /CURRENT= -1
DEFINE ADJUSTMENT "<HEAD1>"    /CURRENT= -1
DEFINE ADJUSTMENT "<HEAD2>"    /CURRENT= -1
DEFINE ADJUSTMENT "<HEAD3>"    /CURRENT= -1
```

17.2.2.5 Preventing Text Compression

The /NOCOMPRESS qualifier means "If a group starts on the current line, do not compress it." For example, you might decide that LSE should never compress BEGIN blocks like the following:

```
«BEGIN»
```

The following definition instructs LSE to display the detailed text instead of the overview line:

```
DEFINE ADJUSTMENT begin /NOCOMPRESS
```

In very high-level overviews, the whole BEGIN block could be hidden as part of a higher-level structure.

17.2.2.6 Finding Appropriate Overview Text

Usually, LSE takes the overview text for a group from the first line of that group. This text may not always be appropriate, for example, if it is a blank line, an empty comment, or a row of stars. LSE looks for identifier characters in the text, and if there are none, it looks at subsequent lines for better text. If no suitable overview text is found in a group, the group is not formed.

You can explicitly control what text appears in overview lines by using the /[NO]OVERVIEW qualifier. If a string is inappropriate as overview text, you can define it using the /NOOVERVIEW qualifier. For example, suppose your C programs are formatted as follows:

```
char *
getline(row)
int row;
{
    ...
}
```

If each C function is compressed to a single line, then *char ** becomes the overview text. However, *char ** is not informative text. *getline* is the function name and should be the overview text. To prevent *char ** from appearing as the overview text, use the following definition:

```
DEFINE ADJUSTMENT "char *"  /NOOVERVIEW
```

17.2.2.7 Inheriting Indentation

In some situations, the visible indentation of a line is not a good indication of what its adjusted indentation should be. The visible indentation of a blank line is always 0. The visible indentation of a FORTRAN or COBOL comment line is usually 1. In these situations, you can use the /INHERIT qualifier to cause a line to inherit its visible indentation from the adjusted indentation of a neighboring line.

The /INHERIT qualifier takes one of the following values:

- PREVIOUS—The visible indentation for the current line is taken from the adjusted indentation of the previous line.
- NEXT—The visible indentation for the current line is taken from the adjusted indentation of the next line.
- MINIMUM—The visible indentation for the current line is taken from the adjusted indentation of either the previous line or the next line, whichever is smaller.

- MAXIMUM—The visible indentation for the current line is taken from the adjusted indentation of either the previous line or the next line, whichever is larger.

A possible adjustment for COBOL comment lines is /INHERIT=NEXT /CURRENT=-1 /UNIT. For example:

```
*
*       RESET WORK-CTR FOR ALPHA FIELDS
*
    MOVE 15 TO WORK-CTR.
```

When you collapse the previous example, it becomes as follows:

```
«*      RESET WORK-CTR FOR ALPHA FIELDS...»
```

With C conditional compilation, a common coding convention is to type the conditional compilation lines at the left margin, although the corresponding program code is indented. For example:

```
DEFINE ADJUSTMENT "$(COLUMN=1)#if" /INHERIT=NEXT
```

When you use this definition, LSE takes the indentation of a conditional compilation line from the indentation of the line below it.

The /INHERIT qualifier is typically used in adjustments for blank lines, form feeds, conditional compilations, and labeled lines.

17.2.2.8 Blank Lines

The default adjustment for blank lines is /INHERIT=MAXIMUM /UNIT /NOCOUNT. This usually causes a blank line to be absorbed into a neighboring group. For example:

```
BEGIN

  a = b;
```

In this example, the blank line following BEGIN inherits the adjusted indentation of the a = b line because that line is indented more than the BEGIN line. Therefore, the blank line is part of the group that starts with the BEGIN line.

Consider the following code fragment:

```
IF something
 THEN
    a = b;

 ELSE
    c = d;
```

The blank line preceding ELSE inherits the adjusted indentation of the a = b line. Therefore, the blank line is part of the group that starts with the THEN line.

To redefine the behavior of blank lines, you must define an adjustment for the LINE_END predefined pattern. (See Section 17.2.3 for details on pattern matching.) For example:

```
DEFINE ADJUSTMENT "$(LINE_END)" /UNIT/NOCOUNT -
/INHERIT=NEXT/CURRENT=-1/NOCOMPRESS
```

You may use this adjustment if you want a blank line to terminate a group started by a preceding comment line. For example, suppose you use this definition with the following code fragment:

```
! comment text
 IF something
  THEN
    a = b;

 c = d;
```

The group started by the comment line will include only the lines through the a = b line.

In languages where there is little syntax to distinguish program constructs, such as VAX C, you can define empty space to be significant. For example:

```
DEFINE ADJUSTMENT "$(LINE_END)" /CURRENT = -1
DEFINE ADJUSTMENT "$(FORMFEED)" /CURRENT = -2
```

This definition causes every blank line to have an adjusted indentation of -1 since its visible indentation is always 0, and every form-feed line to have an adjusted indentation of -1 (if each form feed is in column 1).

17.2.2.9 Prefixes

In some situations you want to avoid having the first text on the line determine indentation or influence adjustment. Labels are a common example of such prefix text. To instruct LSE to treat leading text on a line as prefix text, use the PREFIX pattern element and the /PREFIX qualifier. The PREFIX pattern element defines a specified part of a pattern as a prefix. The /PREFIX qualifier indicates what action to perform when LSE encounters a source line that matches a pattern that includes the PREFIX element.

The following example instructs LSE to treat an identifier followed by a colon as a prefix if a BEGIN construct follows the identifier and colon on a source line:

```
DEFINE ADJUSTMENT "$(IDENTIFIER):$(PREFIX)BEGIN"
```

BEGIN is the first text after the prefix.

If you use PREFIX in an adjustment pattern, then you must specify the /PREFIX qualifier on the definition. Conversely, if you specify the /PREFIX qualifier on the definition, then PREFIX must appear in the pattern string.

The following is an example of a prefix definition with (FOLLOWING,FOLLOWING):

```
DEFINE ADJUSTMENT "$(IDENTIFIER):$(OPTIONAL_SPACE)$(PREFIX)" -
  /PREFIX=(FOLLOWING,FOLLOWING)
```

This definition matches the labeled line in the following example:

```
        a = b;
loop:   WHILE c DO
            d = e;
        f = g;
```

In this example, the indentation is that of WHILE, adjusted by the qualifiers for the definition for WHILE (or defaults if there is no definition for WHILE).

The following is an example of a prefix definition with (CURRENT,FOLLOWING):

```
DEFINE ADJUSTMENT "$(IDENTIFIER):$(OPTIONAL_SPACE)$(PREFIX)" -
  /PREFIX=(CURRENT,FOLLOWING)
```

This definition matches the labeled line in the following example:

```
a = b;
label1:BEGIN
    d = e;
    f = g
END;
```

In this example, the indentation is that of the label adjusted by the qualifiers for the definition for BEGIN.

The following is an example of a definition that you could use for COBOL comments where the indentation of the comment indicator is insignificant but the fact that the line is a comment is significant:

```
DEFINE ADJUSTMENT "$(COLUMN=1)*$(PREFIX)" -
    /PREFIX=(FOLLOWING,CURRENT)   /CURRENT=-1
```

In this example, the indentation of a line with an asterisk in column 1 is taken from the first visible text after the asterisk, adjusted by -1 because of the /CURRENT=-1 qualifier on the current definition. The text after the asterisk is not compared against any patterns.

17.2.2.10 Grouping Comment Lines

With the /UNIT qualifier, you can treat contiguous comment lines as a single unit if they have the same adjusted indentation. For example:

```
! Store away the info needed to update
! the history.
Update_info::Edit_no = Edit_no
Update_info::Version_record = Rec_num
```

Suppose you used the following definition, which does not include a /UNIT qualifier:

```
DEFINE ADJUSTMENT "!"  /CURRENT=-1
```

When LSE applies this adjustment, the following overview appears:

```
! Store away the info needed to update
« !the history...»
```

The result is undesirable because the second comment becomes the overview line for the assignments.

To treat both comment lines as one unit, and to have the first comment serve as the overview for both comment lines and the associated statements, combine the comment lines into one group by using the following definition:

```
DEFINE ADJUSTMENT "!"   /CURRENT=-1 /UNIT
```

When LSE applies this adjustment, the comment lines are treated as a unit, and the following overview appears:

```
« Store away the info needed to update»
```

17.2.2.11 Bracketed Comments

Language compilers and LSE do not necessarily have the same definitions for comments. LSE treats a line as a comment only if it includes comment delimiting text. The second line in the following example is not a comment in LSE:

```
/*
   Open the file.
*/
```

Consider the following adjustment definition:

```
DEFINE ADJUSTMENT "/*"  /CURRENT=-1
```

This particular style does not present a problem for the overview facility. Since the inner lines of a comment are indented further than the comment delimiter text, LSE treats the whole comment as one group.

You can write definitions for "/*" and "*/" with the /SUBSEQUENT qualifier to cause intervening lines to indent. Use this approach with caution, however, because if "*/" does not appear at the beginning of a line, LSE cannot detect the "*/."

17.2.2.12 Fixed Comments

In some languages, such as FORTRAN and COBOL, the indentation of a comment is of no interest because it is in a fixed column. One way to handle this situation is to adjust the comment to inherit the indentation from the line after the comment line. To do this, use the /INHERIT qualifier, as follows:

```
DEFINE ADJUSTMENT "$(COLUMN=1)C" -
  /INHERIT=NEXT/CURRENT=-1/UNIT
```

Another approach is to ignore the comment marker and take the visible indentation from the text that follows on the line. To do this, use the /PREFIX qualifier, as follows:

```
DEFINE ADJUSTMENT "$(COLUMN=1)*$(PREFIX)" -
  /PREFIX = (FOLLOWING,CURRENT)
```

See Section 17.2.4 for FORTRAN-specific information.

17.2.3 Basic Rules for Pattern Matching

Pattern matching is insensitive to case. For example:

```
DEFINE ADJUSTMENT "begin"
```

LSE matches this definition to any of the following strings:

begin
BEGIN
Begin

Pattern matching completes with a word break. For example:

```
DEFINE ADJUSTMENT "begin"
```

LSE does not match the definition "begin" with the string "beginstuff".

You must explicitly specify any blank space in the definition pattern. For example:

```
DEFINE ADJUSTMENT "begin"
```

LSE does not match the definition "begin" with the string "beg in".

Table 17–1 contains named pattern elements that are valid in LSE.

Table 17–1: Named Pattern Elements

Named Pattern	Description
COLUMN	Limits the columns in which the text may start. See Section 17.2.3.3.
IDENTIFIER	Indicates a sequence of identifier characters.
FORMFEED	Indicates a form-feed character.
LINE_END	Specifies the end of a line, preceded by optional white space.
NUMBER	Indicates a sequence of digits.
OPTIONAL_SPACE	Indicates a sequence of spaces and tabs
PREFIX	See Section 17.2.2.9
FORTRAN_FUNCTION FORTRAN_COMMENT	See Section 17.2.4

You must use the $( ) convention to enclose named pattern elements. Thus, the following definition matches any sequence of identifier characters:

```
    DEFINE ADJUSTMENT "$(IDENTIFIER)"
```

To match any blank line, use the following definition:

```
    DEFINE ADJUSTMENT "$(LINE_END)"
```

17.2.3.1 Multiple Word Patterns

An adjustment pattern may consist of multiple words, as in the following example:

```
    DEFINE ADJUSTMENT "end if"
```

A pattern may include a combination of literal text and named patterns. For example, the following definition matches any identifier that is followed by a colon:

```
    DEFINE ADJUSTMENT "$(IDENTIFIER):"
```

Use LINE_END at the end of a pattern to specify that the pattern text is the only text on a matching source line. For example, the following definition matches any line that contains the word *case* without any text after it.

```
DEFINE ADJUSTMENT "CASE$(LINE_END)"
```

17.2.3.2 Blank Space and Adjustment Patterns

Blank space in adjustment pattern specifications is always explicit. If blank space occurs in an adjustment pattern, then matching buffer text must also have blank space between words. The number of blank columns does not matter and both tabs and spaces are interpreted as blank space. For example:

```
DEFINE ADJUSTMENT "end if"
```

This definition matches each of the following strings:

```
end if
end  if
end          if
```

However, the same definition does not match

```
endif
```

If blank space is not included in the definition, the matching buffer text cannot have blank space. For example, the following definition matches xxx: but does not match xxx :, that is, the same string with a blank separating the last x and the colon:

```
DEFINE ADJUSTMENT "$(IDENTIFIER):"
```

To indicate that blank space is optional, use the predefined OPTIONAL_SPACE pattern. For example:

```
DEFINE ADJUSTMENT "end$(OPTIONAL_SPACE)if"
```

This definition matches each of the following strings:

```
endif
end if
end         if
```

Another definition example follows:

```
DEFINE ADJUSTMENT "$(IDENTIFIER)$(OPTIONAL_SPACE):"
```

This definition matches each of the following strings:

```
xxx:
xxx :
xxx     :
```

17.2.3.3 Specifying Columns

To indicate that a pattern must begin in a particular column, use the predefined COLUMN pattern. For example, the following definition matches *end* only if it begins in column 1:

```
DEFINE ADJUSTMENT "$(COLUMN=1)end"
```

The next definition matches any identifier that begins in column 3, 4, or 5:

```
DEFINE ADJUSTMENT "$(COLUMN=3,5)$(IDENTIFIER)"
```

When you use the predefined COLUMN pattern, you must place it at the beginning of the pattern parameter and follow it with some specification of visible text, that is, a literal string or any named pattern element except $(OPTIONAL_SPACE) or $(LINE_END).

17.2.3.4 Pattern Matching Precedence

Long patterns take precedence over short ones and more restrictive patterns take precedence over less restrictive ones. In each of the following example pairs, the first definition takes precedence over the second:

```
DEFINE ADJUSTMENT "end if"
DEFINE ADJUSTMENT end

DEFINE ADJUSTMENT "$(IDENTIFIER):="
DEFINE ADJUSTMENT "$(IDENTIFIER):"

DEFINE ADJUSTMENT end
DEFINE ADJUSTMENT "$(IDENTIFIER)"
```

Matching is word by word; thus, the source string endmorestuff matches $(IDENTIFIER), not end. The precedence rules are subject to word breaks.

The patterns have the following order of decreasing precedence:

1. COLUMN
2. Literal strings
3. Required white space
4. OPTIONAL_SPACE
5. LINE_END
6. FORMFEED

7. FORTRAN_COMMENT
8. FORTRAN_FUNCTION
9. NUMBER
10. IDENTIFIER

If multiple patterns are distinguished only by their column values, they are ordered by the first column number, then the second. In the following example, the first definition has precedence over the second:

```
DEFINE ADJUSTMENT "$(COLUMN=1,2)xyz"
DEFINE ADJUSTMENT "$(COLUMN=2,7)xyz"
```

Patterns are matched from left to right, so a definition for which there is no match is possible. The following example has no blank space; if a string consists of a sequence of identifiers followed by an x, the match for IDENTIFIER consumes the x.

```
DEFINE ADJUSTMENT "$(IDENTIFIER)x"
```

In the following example, the OPTIONAL_SPACE named pattern element matches all the blank space before the z, leaving nothing to go with the required space.

```
DEFINE ADJUSTMENT "y$(OPTIONAL_SPACE) z"
```

17.2.3.5 Using Precedence to Hide Patterns

You can use the relative precedence of patterns to hide definitions. For example, if you want to treat an identifier followed by a colon in a special way, and only when the identifier is not the word “default”, use the following pattern definitions:

```
DEFINE ADJUSTMENT "default:"
DEFINE ADJUSTMENT "$(IDENTIFIER):"
```

The first definition takes precedence over the second, so any source line that begins with “default:” uses the action qualifiers specified on the first definition (or the defaults, if you do not specify any qualifiers). Any source line that begins with another sequence of identifier characters followed by a colon uses the second definition. (Note that the order of the definitions is *not* significant; however, the relative precedence of the patterns *is* significant.)

Similarly, you can use pattern length to hide definitions. For example, if you want to treat an identifier followed by a colon in a special way, unless an equal sign follows the colon, use the following definitions:

```
DEFINE ADJUSTMENT "$(IDENTIFIER):"
DEFINE ADJUSTMENT "$(IDENTIFIER):="
```

The second pattern takes precedence because it is longer.

17.2.3.6 Rules for Pattern Strings

A pattern string is a sequence of literal strings and named pattern elements subject to the following rules:

- Blank space may not appear between the parentheses of named pattern specifications. The following is invalid:

  ```
  "$(COLUMN = 1)"
  ```

- If a pattern string includes the COLUMN named pattern specification, COLUMN must appear first. The following is invalid:

  ```
  "x $(COLUMN=7) y"
  ```

- The COLUMN named pattern specification always requires at least one value.
- A pattern may not begin with the OPTIONAL_SPACE named pattern or any explicit white space. Thus, the following patterns are invalid:

  ```
  "$(OPTIONAL_SPACE)end"
  "   end"
  "$(COLUMN=4)$(OPTIONAL_SPACE)end"
  ```

- A pattern may not begin with the PREFIX named pattern. The following is invalid:

  ```
  "$(PREFIX) x"
  ```

- The LINE_END named pattern must be the last pattern. The following is invalid:

  ```
  "$(LINE_END)x"
  ```

- If a pattern includes PREFIX, then the definition must also include the /PREFIX qualifier.
- FORTRAN_COMMENT named patterns may be used only with the FORTRAN language.
- Patterns must not include line breaks. If a pattern includes multiple words, the whole pattern must appear on one line.

17.2.3.7 Using the Pattern Parameter

By default, the pattern parameter is the same as the adjustment name. You should specify the pattern parameter if the pattern string is not suitable as a name. For example, you may find a name difficult to remember when you want to issue the DELETE, SHOW, or EXTRACT command on the definition. The following example shows a pattern that is too long to be a good adjustment name:

```
"$(IDENTIFIER)$(OPTIONAL_SPACE):$(OPTIONAL_SPACE)$(PREFIX)begin"
```

To avoid this problem, create a distinctive adjustment name, as in the following example:

```
DEFINE ADJUSTMENT label -
  "$(IDENTIFIER)$(OPTIONAL_SPACE):$(OPTIONAL_SPACE)$(PREFIX)begin"
```

You can then use the name *label* to refer to the definition. Or you can create a naming convention to group similar definitions, as follows:

```
DEFINE ADJUSTMENT end
DEFINE ADJUSTMENT end1 "$(COLUMN=1)end"
```

You can then use the wildcard name *end** to refer to the definitions with end as the first visible text.

17.2.4 Special Processing for FORTRAN

In the FORTRAN language, the adjustment actions LSE applies depend on whether a line is a comment line, a continuation line, or neither of these.

A continuation line is treated as though defined with DEFINE ADJUSTMENT /INHERIT=MAXIMUM/CURRENT=1/UNIT. If the line is a comment line, LSE matches against adjustments that contain the FORTRAN_COMMENT pattern element (described later). If the line is neither a comment nor a continuation line, LSE matches against adjustments for noncomments (adjustments that do not contain the FORTRAN_COMMENT pattern element) and examines only the text in the statement field. COLUMN=1 refers to the first column in the statement field.

Pattern elements defined for FORTRAN are as follows:

FORTRAN_FUNCTION

LSE defines a special pattern, called FORTRAN_FUNCTION, for matching the first line of a FORTRAN function subprogram. The format of such a line is as follows:

type [*number] FUNCTION

type
Is one of the keywords in Table 17–2.

Table 17–2: Type Keywords

BYTE	DOUBLE COMPLEX	LOGICAL
CHARACTER	DOUBLE PRECISION	REAL
COMPLEX	INTEGER	

number
Is either a series of digits or an asterisk (*).

FORTRAN_COMMENT

LSE recognizes a special pattern element, FORTRAN_COMMENT. This pattern element may appear only as the first element of a pattern and means "Apply this definition only if the source line is known to be a comment." This meaning is conceptually similar to COLUMN, which means "Apply this definition if the text starts in the correct column."

An adjustment is a comment adjustment if the pattern begins with FORTRAN_COMMENT. Otherwise, it is a noncomment adjustment.

Examples

Each of the following examples starts in column 1. All examples are for ANSI format.

Example 1:

```
DEFINE ADJUSTMENT "$(FORTRAN_COMMENT)C**"
```

This definition matches each of the following lines of code:

```
    C**
    c**
    C**stuff
```

However, the definition does not match any of the following lines:

```
C stuff
C **
CC**
    C**
***
!**
12345 x = y
           end
     1continuation
```

Example 2:

```
DEFINE ADJUSTMENT "$(FORTRAN_COMMENT)!"
```

This definition matches each of the following lines of code:

```
!stuff
!
!*
```

However, it does not match either of the following lines:

```
C
    !
```

Example 3:

```
DEFINE ADJUSTMENT "endif"
```

This definition matches each of the following lines of code:

```
12345 endif
      endif
     0endif
52             endif
           endif
```

However, it does not match either of the following lines:

```
C endif
     1endif
```

Example 4:

```
DEFINE ADJUSTMENT "$(COLUMN=1)end"
```

This definition matches each of the following lines of code:

```
12345 end
      end
     0end
```

However, it does not match any of the following lines:

```
12345      end
           end
C end
```

17.3 Tab Increments and the DEFINE ADJUSTMENT Command

The DEFINE ADJUSTMENT command adjusts indentation by columns. When you write definitions, you need to make assumptions about the size of a tab.

For example:

```
IF x GTR y
THEN
    max := x
```

Consider the following definition:

```
DEFINE ADJUSTMENT then /CURRENT=2
```

With a tab increment of 4, this definition yields the following adjusted result:

```
IF x GTR y
  THEN
    max := x
```

However, if the tab increment is 2 instead of 4, the actual indentation is as follows:

```
IF x GTR y
THEN
  max := x
```

The following is the adjusted indentation:

```
IF x GTR y
  THEN
  max := x
```

This is not a desirable result. The solution is to inform LSE what tab increments the definitions work with.

The DEFINE LANGUAGE and MODIFY LANGUAGE commands have an /OVERVIEW_OPTIONS=TAB_RANGE qualifier that specifies the tab increment for buffers associated with a given language. TAB_RANGE takes two numbers. The second number must be at least twice as big as the first number. The range specifies the tab increment values that work with the DEFINE ADJUSTMENT command. If the first number is M and the second number is N, then the DEFINE ADJUSTMENT command is assumed to be correct for tab increments from M to N, inclusive.

For example, the following definition instructs LSE to indent Ada code with a tab increment of 2, and indicates that the relevant DEFINE ADJUSTMENT commands were written assuming a tab increment of 4:

```
DEFINE LANGUAGE ADA /TAB_INCREMENT=2 /OVERVIEW_OPTIONS=(TAB_RANGE=(4,8))
```

LSE makes the appropriate adjustments so the definitions will work with the smaller tab increment. It does this by doubling the indentation of each source line before applying any other indentation adjustments.

For best performance and format of the VIEW SOURCE/DEBUG output, make sure the /TAB_INCREMENT value is within the range specified on /OVERVIEW_OPTIONS=TAB_RANGE. For most uses, the recommended definition is as follows:

```
/TAB_INCREMENT=4 /OVERVIEW_OPTIONS=(TAB_RANGE=2,8)
```

The /TAB_INCREMENT value is usually 4. If you choose a tab range of (2,8), be wary of definitions such as the following:

```
DEFINE ADJUSTMENT x /SUBSEQUENT=6
DEFINE ADJUSTMENT y /CURRENT=-2
```

You choose the /SUBSEQUENT and /CURRENT values because they work well with a tab of 4. But the x definition will probably be wrong with a tab value of 8. The y definition will probably be wrong with a tab value of 2. To fix this, change the tab range and/or change the DEFINE ADJUSTMENT qualifier values so they work well together.

17.4 Debugging

There are two ways to debug DEFINE ADJUSTMENT definitions. One way is to use the viewing commands on sample text and experiment with DEFINE ADJUSTMENT definitions until you are satisfied with the results.

Alternatively, you can debug your definitions by using the VIEW SOURCE/DEBUG command. This command generates a representation of the source buffer, indented as LSE perceives the indentation, as specified by the adjustment definitions. The result is displayed in a system buffer, $OVERVIEW.

For example:

```
    DEFINE ADJUSTMENT "!"       /CURRENT = -1 /UNIT
    DEFINE ADJUSTMENT BEGIN     /SUBSEQUENT = 2 /NOCOMPRESS
    DEFINE ADJUSTMENT END       /CURRENT = -1   /SUBSEQUENT = -2
```

Suppose you have executed the VIEW SOURCE/DEBUG command when the cursor is in a buffer containing the following source text (assuming each line begins in column 1):

```
    ! Swap the numbers
    BEGIN
    temp = a;
    a = b;
    ! Get the value from temporary storage.
    b = temp
    END
```

This puts the following text in the $OVERVIEW buffer:

```
U          3  0  0 ! Swap the numbers.
 NC        4  0  1  BEGIN
           1  2  3    temp = a;
           1  2  3    a = b;
U          3  2  2   ! Get the value from temporary storage.
           1  2  3    b = temp
           5  2  2   END
---------------------------------------------------------------------------
    Left margin numbers (in order):
        adjustment number
        /SUBSEQUENT running total
        effective indentation

    U = /UNIT  NC = /NOCOMPRESS  NO = /NOOVERVIEW  N# = /NOCOUNT
    IP = /INHERIT=PREV  IN = /INHERIT=NEXT, IO = /INHERIT=NEITHER
    I< = /INHERIT=MINIMUM,  I> = /INHERIT=MAXIMUM

   Adjustment
  number:   name                                        use count

       1: default                                          3
       2: default blank                                    0
       3: !                                                2
       4: BEGIN                                            1
       5: END                                              1
```

The information in this buffer explains the following:

- The first line used adjustment #3 (the adjustment for !). This adjustment includes a /UNIT qualifier. This line's effective indentation is 0 because its visible indentation is 1 and its adjustment includes /CURRENT=-1.
- The second line used adjustment #4 (the adjustment for BEGIN). This adjustment includes a /NOCOMPRESS qualifier. This line's effective indentation is 1. The adjustment for this line also includes /SUBSEQUENT=2, so the /SUBSEQUENT running total for the next line is 2.
- The third line used the default adjustment (adjustment #1) because its text does not match the pattern of any defined adjustment. This line's effective indentation is 3 because its visible indentation is 1 and the running total of /SUBSEQUENT values is 2.
- The fourth and sixth lines are similar to the third line.
- The fifth line used the same adjustment as the first line. Since the running total of /SUBSEQUENT values is 2 when you reach this line, its effective indentation is 3.
- The seventh line used adjustment #5 (the adjustment for END). This line's effective indentation is 2. Its visible indentation is 1. This is adjusted by the /SUBSEQUENT running total of 2, and the adjustment qualifier /CURRENT = -1. The calculation is: 1 + 2 - 1 = 2.

You can judge the quality of your definitions by looking at the relative indentation of the lines. A line hides all the lines that follow it until a line occurs that has the same or less indentation. In this example: the first comment hides all the other lines; the second comment hides only one line; and none of the assignment lines hide any lines.

When some lines have an adjusted indentation that is negative, the display is appropriately altered.

Chapter 18

Customizing Reports

This chapter provides guidance for those who need to customize reports or create new reports. Section 18.1 is an introduction. Section 18.2 describes how the REPORT command invokes VAXTPU. Section 18.3 describes how the reports are organized. Section 18.4 describes how to customize 2167A reports.

18.1 Introduction

The SCA command REPORT uses the VAX Text Processing Utility (VAXTPU) to generate reports. You can customize the reports and/or create your own reports by writing additional VAXTPU routines. You can access the VAXTPU source files for the SCA command REPORT in the SYS$LIBRARY directory.

Customizing reports requires that you understand how to invoke VAXTPU procedures. You should study the VAXTPU code provided by Digital and then make minor changes by modifying that code. With experience, you can write complete reports from scratch.

The VAXTPU procedures use the SCA callable interface. See the appendix that describes the SCA callable interface in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual*. You must understand the callable interface in order to retrieve SCA data for your reports.

The interface between VAXTPU and SCA is built into the version of VAXTPU that is embedded in LSE. Therefore, you must use LSE for access to the VAXTPU/SCA interface, even if you do not use LSE for other editing tasks. You cannot use the version of VAXTPU provided with VMS for this purpose.

18.2 How the REPORT Command Invokes VAXTPU

When you issue the REPORT command, SCA first checks to see whether or not LSE has been activated, and if necessary, activates LSE. Then the command is parsed and a number of global VAXTPU variables are initialized, based on the settings of the qualifiers and parameters on the command. Table 18–1 shows these variables and their defaults.

Table 18–1: VAXTPU Variables

Variables	Values
SCA$REPORT_NAME	The name of the report. There is no default value.
SCA$REPORT_DOMAIN_QUERY	The name of the query to use as the domain for the report, as a string. The default for this is the null string, which by convention implies that the entire SCA database is the domain.
SCA$REPORT_TARGET	The type of target file to produce. The default value for this is the null string. The interpretation of this depends upon the particular report being produced.
SCA$REPORT_FILL	The setting of the /FILL qualifier. This value can be either the integer 1, for true, or the integer 0, for false. The default value is 1, meaning that FILL is turned on for those reports for which it is meaningful.
SCA$REPORT_OUTPUT	The output file for the report. The default value is the null string, which by convention is interpreted to mean that the report name is used as the file name, while the file type is implied by the target type.
SCA$REPORT_REST_OF_LINE	The remainder of the REPORT command line. This is a single string. The default value is the empty string. The interpretation of this value depends upon the particular report being produced.

(continued on next page)

Table 18–1 (Cont.): VAXTPU Variables

Variables	Values
SCA$REPORT_HELP_LIBRARY	The name of the help library to use as the value of the /HELP_LIBRARY qualifier on all DEFINE PACKAGE commands generated by this report. This value is used only by the PACKAGE report and is ignored for other reports. If omitted, the PACKAGE report will omit the /HELP_LIBRARY qualifier from any DEFINE PACKAGE commands it generates.
SCA$REPORT_LANGUAGES	A list of one or more languages to use as the value of the /LANGUAGE qualifier on all DEFINE PACKAGE commands generated by this report. This value is used only by the PACKAGE report and is ignored for other reports. If omitted, the PACKAGE report inserts a placeholder for the value of the /LANGUAGE qualifier on the DEFINE PACKAGE commands it generates. You must then replace that placeholder manually before you can use the DEFINE PACKAGE command.

Reports supplied by Digital inspect the value of SCA$REPORT_DOMAIN_QUERY to see whether or not it is empty. If it is empty, then the entire SCA database is used as the domain for the report. If it is not empty, it is presumed to be the name of an SCA query that was qualified by the SCA query attributes SYMBOL=FILE AND OCCURRENCE=COMMAND. (Refer to the appendix that describes SCA query expressions in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for more details on these query attributes.) The report is then limited to just the programs contained within the files determined by this query.

After these variables have been set, SCA and LSE construct the name of the VAXTPU procedure that implements the report by concatenating the string SCA_REPORT_ to the name of the report, as you specified on the REPORT command.

If you specify REPORT *myreport*, SCA and LSE look for a VAXTPU procedure named SCA_REPORT_MYREPORT. If that procedure does not exist, an error is issued. If it does exist, it is invoked with no parameters. Thus, if you want to create a new report called *myreport*, you write a VAXTPU procedure SCA_REPORT_MYREPORT, taking no parameters, to implement the report. You get the other information from the REPORT command by inspecting the values of the VAXTPU variables that were initialized by SCA and LSE.

For the reports provided by Digital, the SCA_REPORT_xxx routines load the remainder of the report tool. They do this by checking for the existence of a known procedure (SCA$REPORT_OUTPUT_MODULE_IDENT), and if it is not present, load a fixed set of TPU files from SYS$LIBRARY. These files are described in the next section. If the SCA$REPORT_OUTPUT_MODULE_IDENT procedure already exists, then no further loading is performed.

18.3 How Reports are Organized

Source code for the reports are in the SYS$LIBRARY directory. Each file has a name of the form SCA$REPORT_module.TPU, with the exception of the file SCA$QUERY_CALLABLE.TPU. As indicated previously, the top-level routines for each report have a name of the form SCA_REPORT_report_name. All other routines for the reports provided by Digital have names of the form SCA$REPORT_routine_name. Digital suggests that if you want to modify one of the routines furnished by Digital, you first rename it by changing the dollar sign to an underscore. This is an iterative process, since all routines that call a changed routine have to be changed as well.

The files are organized as follows:

- SCA$REPORT_GLOBALS.TPU—This file contains a number of global constants and variables that are used in the process of generating the report. In particular, it contains constants that represent the names of tags that correspond to particular sections of reports.
- SCA$REPORT_INTERNALS.TPU—This file implements the INTERNALS report.
- SCA$REPORT_2167A_DESIGN.TPU—This file implements the 2167A report.
- SCA$REPORT_HELP.TPU—This file implements the HELP report.
- SCA$REPORT_PACKAGE.TPU—This file implements the PACKAGE report.
- SCA$REPORT_VALIDATE.TPU—This file contains routines that are used to validate the parameters passed to the reports.

- SCA$REPORT_FORMAT.TPU—This file contains routines that format output based on the target text processor (VAX DOCUMENT, RUNOFF, or ordinary text). Each routine dispatches to a lower-level routine based on the current target.
- SCA$REPORT_FORMAT_DOCUMENT.TPU—This file contains the routines that format text for VAX DOCUMENT output.
- SCA$REPORT_FORMAT_RUNOFF.TPU—This file contains routines that format text for RUNOFF output.
- SCA$REPORT_FORMAT_TEXT.TPU—This file contains routines that format text for ordinary text output. (There are no formatting routines for the HELP and PACKAGE reports because they do not have distinct concepts, such as chapters, sections, or headings.)
- SCA$REPORT_OUTPUT.TPU—This file contains generic routines for inserting text into the output file. The output file is implemented as one or more VAXTPU buffers, which are merged and written when the entire report is complete.
- SCA$REPORT_UTILITIES.TPU—This file contains a variety of general-purpose routines that are used for retrieving design data from the SCA database.

If you wish to modify parts of one or more of these files, the simplest approach is to copy the relevant files from SYS$LIBRARY into a personal directory and make the changes you want. Then, by defining logical names for the modified files, you can get the SCA_REPORT_ routines to load your modified version.

For example, suppose you want to add a new tag called DEVELOPERS, and change the reports so that DEVELOPERS is treated synonymously with the AUTHORS tag. Simply defining the tag, as described in Section 12.5, does not tell the reports how to interpret the tag. To make the change for the report, you copy over the SCA$REPORT_GLOBALS.TPU file. In that file there is a set of tag synonym definitions. The one for the AUTHORS tag looks similar to the following:

```
!
!   Tag synonyms:  These let us have several different tags that
!   map onto the same section of a report.  For example, AUTHOR and
!   AUTHORS are synonyms, as are FORMAL PARAMETERS and FORMAL
!   ARGUMENTS.
!

CONSTANT
    sca$report_authors :=
            '("AUTHOR"'
        + 'OR "AUTHORS")',
```

You could change this to be the following:

```
CONSTANT
    sca$report_authors :=
            '("AUTHOR"'
        + 'OR "AUTHORS"'
        + 'OR "DEVELOPERS")',
```

Then define (at the DCL level) the logical name SCA$REPORT_GLOBALS to point at the new version of the file. When you run the reports, your version of this file is loaded. When the report code generates the AUTHORS section of the report, it creates an SCA query from the following TPU string:

```
'SYMBOL=TAG AND ' + sca$report_authors
```

This evaluates to the following SCA query expression:

```
 SYMBOL=TAG AND ("AUTHOR" OR "AUTHORS" OR "DEVELOPERS")
```

For more complicated changes to the report tool, read the source code as provided by Digital.

18.4 Customizing 2167A Reports

Because the relationships between your program design or code and the DOD 2167A Software Design Document are highly dependent upon policies and procedures established at your site, it is expected that you will need to customize the report.

In this section, three simple types of customizations are presented. The first example adds a section to a report. The next example shows how you can generate the INPUT/OUTPUT section of a report automatically from data declarations in the code, instead of getting the information directly from comment tags. The final section shows how to change the mapping of UNITS to files to mapping UNITS to packages.

18.4.1 Adding a Section to a 2167A Report

In the case of the 2167A design reports, sections for *components* are written by the TPU procedure SCA$REPORT_2167A_DESIGN_BODY. Sections for *units* are written by the TPU procedure SCA$REPORT_2167A_UNITS. These routines are contained in SYS$LIBRARY:SCA$REPORT_2167A_DESIGN.TPU.

Each of these routines uses a pair of TPU arrays to determine the sections to write. The arrays are initialized in the procedure SCA$REPORT_2167A_GLOBALS_INITIALIZE, which is contained in the file SYS$LIBRARY:SCA$REPORT_GLOBALS.TPU.

One array is used to contain the names of relevant tags. The other contains the title strings to output as the heading for the corresponding section of the report.

For units, the arrays are called SCA$REPORT_2167A_UNIT_SECTION_HEADINGS, and SCA$REPORT_2167A_UNIT_TAGS. For components, the arrays are called SCA$REPORT_2167A_COMPONENT_SECTION_HEADINGS, and SCA$REPORT_2167A_COMPONENT_TAGS.

If you wanted to add a section called PERFORMANCE CONSIDERATIONS for units, you would begin by defining a UNIT PERFORMANCE CONSIDERATIONS tag by using the DEFINE TAG command, as follows:

```
DEFINE TAG "UNIT PERFORMANCE CONSIDERATIONS"/TYPE=TEXT -
    /LANGUAGE=your_language
```

You can also modify your LSE templates to make this tag available at an appropriate point in the comment header.

Next, you edit the SCA$REPORT_GLOBALS file to add the tag and the corresponding section heading to the appropriate arrays. If you look in the SCA$REPORT_2167A_GLOBALS_INITIALIZE procedure, you will find a sequence of assignment statements for the arrays corresponding to units. Each set looks similar to the following:

```
sca$report_2167A_unit_section_headings {section_index}
    := 'Heading to use for the item';
sca$report_2167A_unit_tags {section_index}
    := '("TAG TO USE FOR THE ITEM")';
section_index := section_index + 1;
```

Simply add three more statements for the section you want to add, as follows:

```
sca$report_2167A_unit_section_headings {section_index}
    := 'Performance considerations';
sca$report_2167A_unit_tags {section_index}
    := '("UNIT PERFORMANCE CONSIDERATIONS")';
section_index := section_index + 1;
```

Digital recommends that you use distinct names for tags at the component and unit level. In this case PERFORMANCE CONSIDERATIONS would be an appropriate tag to use at the component level. Since the correct level for section headings is apparent from the context in the report output, it is not necessary to use distinct section headings, although you may do so.

18.4.2 Using Program Code For Report Information

As supplied by Digital, the 2167A design report does not use any of the declarations in your design or code for its data; only tag information is used. Depending on your particular needs, it may be quite reasonable for you to use declarations for some sections of the report. The following example shows how you can get the INPUT/OUTPUT DATA ELEMENTS section for a given unit by using your knowledge of the relationship between program elements and the 2167A concept.

For the purpose of this example, suppose that you are using a language such as Ada or Pascal, where imported and exported data are explicitly declared. In Ada, this is done with packages. In Pascal it is done with GLOBAL and EXTERNAL attributes. Either way, SCA can identify such data with the following query expression:

```
SYMBOL=VARIABLE AND OCCURRENCE=DECLARATION AND DOMAIN=MULTI_MODULE
```

If this is precisely the set of data you want to identify as INPUT/OUTPUT DATA ELEMENTS for each unit, the question now is twofold: How do you locate the data for a given unit? How do you put out the information for that data?

By looking at the TPU procedure that is responsible for writing unit data, SCA$REPORT_2167A_UNITS, you can determine that units are identified by their file name, which is available in the local TPU variable unit_file_name. Thus, the following TPU expression creates a string with the appropriate SCA query expression.

```
'SYMBOL=VARIABLE AND OCCURRENCE=DECLARATION ' -
+ 'AND DOMAIN=MULTI_MODULE AND FILE= ' + unit_file_name
```

This is the basic idea behind the report tool, namely the generation of SCA query expressions that correspond to meaningful sets of data.

Having identified the correct query expression, you must create the query in the TPU code, walk through all occurrences found by the query, and write something reasonable for each occurrence. To do this, use the SCA callable interface, a subset of which is available directly in TPU. (See the appendix describing the SCA callable interface in the *VAX Language-Sensitive Editor and VAX Source Code Analyzer Reference Manual* for details.) Queries must be initialized, parsed, found, and cleaned up. Thus, the following code fragment is a suitable outer shell:

```
sca$query_initialize(
    lse$sca_command_context,    ! Always use this when initializing
                                ! queries from TPU
    input_output_query);
sca$query_parse(
    input_output_query,
    'SYMBOL=VARIABLE AND OCCURRENCE=DECLARATION '
    + ' AND DOMAIN=MULTI_MODULE AND FILE= ' + unit_file_name);
sca_status := sca$query_find (input_output_query);

IF NOT INT (sca_status)
THEN
    ! No input/output data was found
    !
    sca$report_start_header_document (
        csc_level + 2,      ! Headings for units are two levels deeper
                            ! than headings for the component (CSC)
        'INPUT/OUTPUT DATA ELEMENTS');
    sca$report_paragraph ('No Input/Output data elements were found')
ELSE
    ! We found some.  Let's loop through them.
    !
    LOOP
        sca_status :=
            sca$query_get_occurrence (
                input_output_query,
                input_output_query_entity);
        EXITIF NOT INT (sca_status);
        «Write appropriate data»
    ENDLOOP
ENDIF;
sca$report_cleanup(input_output_query);
```

Depending on exactly what information you want to write, you can get suitable information by using an SCA containment operation to find tags that are associated with the declaration of this item, or by using an SCA typing operation to find the type of the item.

18.4.3 Changing the Mapping of Files

The 2167A design report implements the mapping of files to units via SCA queries, in the procedure SCA$REPORT_2167A_UNITS, contained in file SYS$LIBRARY:SCA$REPORT_2167A_DESIGN.TPU. This procedure takes as its parameters the name of the current component, and the heading level for the component. (The heading level is used to insure that chapter and section headings in the output are numbered correctly. It is not relevant to the discussion that follows.) The procedure constructs an SCA containment query using the component name to determine the units of the query, and then steps through the units.

The key principle is that the connection between program elements and 2167A concepts is created by building an SCA query and walking through it. In some cases, the result of the SCA query exactly matches the 2167A concept you want, in which case every occurrence found by the query is reported. In more complicated cases, the occurrences in the query must be filtered further. The occurrences in the query can be filtered by doing one or more of the following:

- Asking SCA for some of the attributes of the occurrence
- Asking SCA for tags that are associated with the occurrence
- Using the SCA information to locate the source position and looking for further information at that point in the source file
- Using naming conventions
- Applying other particular knowledge you may have about the organization and relationships of your code

In this case, the query that is constructed by this procedure is as follows:

```
CONTAINED_BY ( -
    END     = "UNIT OF" AND SYMBOL=TAG, -
    BEGIN   = component_name AND SYMBOL=KEYWORDS, -
    DEPTH   = 1, -
    RESULT  = BEGIN);
```

This query returns all occurrences of the component name that is contained by the UNIT OF tag. That is, it returns all occurrences that look like the following:

```
-- UNIT OF: component_name
```

This locates the units that are associated with the given *component_name*, but does not identify the program structure that is the unit. As shipped, the 2167A design report identifies that program structure as the file containing the occurrence. The file name is determined by the routine SCA$REPORT_GET_FILE_FROM_ENTITY (from SYS$LIBRARY:SCA$REPORT_UTILITIES.TPU). Then, for each relevant tag in the unit, a query of the following form is computed to find the text related to the tag for this unit:

```
SYMBOL=TAG AND specific_tag_name AND FILE=unit_file_name
```

This is done implicitly by the routine SCA$REPORT_WRITE_TAGS_SUBSECTION (also in SYS$LIBRARY:SCA$REPORT_UTILITIES.TPU). A string of the form FILE=unit_file_name and a string containing a particular tag of interest are passed to this routine, which then looks for the tag inside the file and writes the appropriate section if the tag is found.

If, instead of basing units on files, you want to base them on packages, you need to create a query that determines the package in which the UNIT OF tag occurred. First, you must build a query that represents the single occurrence of the UNIT OF tag that is of interest. Assuming that the current occurrence of the UNIT OF tag is indicated by the variable *unit_entity*, the following TPU code fragment creates the query and gets its name:

```
SCA$QUERY_INITIALIZE (lse$sca_command_context, unit_entity_query);
SCA$SELECT_ENTITY (unit_entity_query, unit_entity);
SCA$QUERY_GET_NAME(unit_entity_query, unit_entity_query_name);
```

Now you can build the query that determines the associated package. Use the following TPU code fragment:

```
SCA$QUERY_INITIALIZE (lse$sca_command_context, package_query);
SCA$QUERY_PARSE (
    package_query,
      'CONTAINED_BY ('
    +   'END= SYMBOL=MODULE AND OCCURRENCE=DECLARATION,'
    +   'BEGIN= @' + unit_entity_query_name + ','
    +   'DEPTH=1,'
    +   'RESULT=END');
IF NOT INT SCA$QUERY_FIND(package_query)
THEN
    ! This UNIT OF occurrence is not directly contained by a PACKAGE.
    « Take appropriate other action »

ELSE
    ! We have the package that contains this UNIT OF occurrence.
    ! Get the occurrence of this package
    SCA$QUERY_GET_OCCURRENCE (package_query, package_entity);
    SCA$QUERY_INITIALIZE (lse$sca_command_context, package_entity_query)
    SCA$QUERY_SELECT_OCCURRENCE (package_entity_query, package_entity);
    SCA$QUERY_GET_NAME (package_entity_query, package_entity_query_name);
    unit_query_expression := package_entity_query_name;
    «Write the sections corresponding to this package »
ENDIF
```

Note that once the package has been found, you must use SCA$QUERY_SELECT_OCCURRENCE to create a query that contains just that occurrence. You must also determine the name of that query. In this case, the routine that writes out individual sections, SCA$REPORT_WRITE_TAGS_SUBSECTION, takes the name of the query that refers to the *package_entity_query_name* package as its first parameter and looks for the given tags inside that query. Thus, instead of building the preceding query, SCA$REPORT_WRITE_TAGS_SUBSECTION builds a query of the following form:

```
    CONTAINED_BY ( -
        END = @package_entity_query_name, -
        BEGIN = specific_tag_name AND SYMBOL=TAG, -
        DEPTH = 1, -
        RESULT = BEGIN);
```

You can construct much more complicated reports by using the SCA callable interface and the report tool procedures, while relying on particular conventions or knowledge of your own code. If you want to do advanced report customizations, read the TPU files for the report tool with names of the form SCA$REPORT_*.TPU in SYS$LIBRARY.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	**Call**	**Contact**
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Language-Sensitive Editor and VAX Source Code Analyzer User Manual
AA–PAJLA–TK

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™

No Postage
Necessary
if Mailed
in the
United States

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

Reader's Comments

VAX Language-Sensitive Editor and VAX Source Code Analyzer User Manual
AA–PAJLA–TK

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™

No Postage
Necessary
if Mailed
in the
United States

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