VMS Software

VSI OpenVMS

VSI DECset for OpenVMS Guide to Detailed Program Design

Document Number: DO–PROGDS–01A

Publication Date: December 2021

Revision Update Information: This is a new manual.

Operating System and Version: VSI OpenVMS I64 Version 8.4-1H1
VSI OpenVMS Alpha Version 8.4-2L1 and 8.4-2L2

Software Version: DECset Version 12.8 for OpenVMS

VMS Software, Inc. (VSI)
Burlington, Massachusetts, USA

VSI DECset for OpenVMS Guide to Detailed Program Design

Legal Notice

Preface

This guide introduces the Program Design Facility, which provides an integrated method for designing, creating, compiling, correcting, and inspecting source code. The guide also explains how to get started using its basic features.

1. Intended Audience

This guide is for experienced programmers and technical managers involved in detailed program design. The user should be familiar with both LSE and SCA.

2. Document Structure

The *Guide to Detailed Program Design for OpenVMS Systems* contains the following chapters:

- Chapter 1 provides an overview of the VSI DECset tools, LSE, SCA, and program design.
- Chapter 2 shows how features of LSE, SCA, and OpenVMS compilers are used to design programs and generate reports.
- Chapter 3 provides in-depth information on entering detailed program design.
- Chapter 4 provides information on running standard reports, the available options, and sample outputs for each report.
- Chapter 5 provides information on customizing reports and creating new reports.

3. Associated Documents

The following documents might also be helpful when using LSE:

- *Using VSI DECset for OpenVMS Systems* describes how to use the Software Engineering Tools (VSI DECset) with other OpenVMS facilities to create an effective software development environment on OpenVMS systems.
- *VSI DECset for OpenVMS Guide to Language-Sensitive Editor* contains instructions on how to use the DECwindows LSE.
- *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* contains command dictionary, parameter glossary,command summary and translation table information for both the LSE and SCA components.
- *VSI DECset for OpenVMS Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual* contains LSE command-line interface information and OpenVMS-specific information.
- *VSI DECset for OpenVMS Guide to Source Code Analyzer* contains instructions on how to use the DECwindows SCA.
- *VSI DECset for OpenVMS Source Code Analyzer Command-Line Interface and Callable Routines Reference Manual* contains SCA command-line interface information and OpenVMS-specific information.

4. References to Other Products

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

Note

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

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

5. Typographical Conventions

Table 1 lists the conventions used in this guide.

Table 1. Typographical Conventions

Convention	Description
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
KP*n*	The phrase KP*n* indicates that you must press the key labeled with the number or character *n* on the numeric keypad, for example, KP3 or KP-.
file-spec, ...	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates additional parameters, options, or values you can enter.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being described.
( )	In format descriptions, if you choose more than one option, parentheses indicate that you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
bold text	Bold text represents the introduction of a new term.
italic text	Italic text represents book titles, parameters, arguments, and information that can vary in system messages (for example, Internal error *number*).
UPPERCASE	Uppercase indicates the name of a command,routine, file, file protection code, or the abbreviation of a system privilege.
lowercase	Lowercase in examples indicates that you are to substitute a word or value of your choice.

Chapter 1. Introduction

This chapter is an introduction to the Software Engineering Tools (VSI DECset), designing programs, and generating reports.

The Program Design Facility provides an integrated method for designing, creating, compiling, correcting, and inspecting source code. This is not a product, or even a tool—it is an abstract title for a set of features implemented across Language-Sensitive Editor (LSE), Source Code Analyzer (SCA), and the compilers.

1.1. VSI DECset

The DECset tools provide an integrated method for designing, creating,compiling, correcting, and inspecting your source code. You can include design information that can be processed, analyzed, and preserved throughout the software development cycle. You can review and modify the source code for your software project, and access all your project files through LSE.

DECset is a group of the following VSI layered components:

- Language-Sensitive Editor/Source Code Analyzer (LSE/SCA)
- Code Management System (CMS)
- Module Management System (MMS)
- VSI Digital Test Manager
- Performance and Coverage Analyzer (PCA)
- DECset Environment Manager

The combination of features from two of these tools, LSE and SCA, provide the ability to design programs and retrieve program design.

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

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

1.2. Source Code Analyzer (SCA)

The Source Code Analyzer (SCA) is an interactive cross-reference and static analysis tool that works with many languages. It can help you understand the complexities of a large software project. Because it allows you to analyze and understand an entire system, SCA is extremely useful during the implementation and maintenance phases of a project.

SCA provides the following capabilities:

- **Cross-referencing—**Gives you an index to information in your source code
- **Static analysis—**Enables you to extract information about program structures and the relationship of routines, symbols, and files
- **Library creation and maintenance—**Takes the information generated by supported compilers and merges these files together into libraries to create a picture of your entire project
- **Graphical user interface—**Provides a DECwindows-based user interface from which you can easily access all SCA capabilities

1.3. Language-Sensitive Editor (LSE)

The Language-Sensitive Editor (LSE) is an advanced text editor with language-specific features. 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 outlining
- Documentation extraction

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

1.4. Designing Programs and Generating Reports

LSE and SCA provide features to help you develop your detailed design and retain it as you move from design to implementation. LSE overviews and SCA reports can retrieve the retained design information.

You can also use LSE and SCA to extract detailed design information from existing source files. You can customize LSE and SCA to improve the design information extracted from source files that were not created using the standard LSE templates.

Chapter 2. Getting Started Designing Programs

This chapter shows, with simple examples, how features of LSE, SCA, and OpenVMS compilers are used to design programs and generate reports. You can use these features to enter and retrieve a wide variety of program design information, including:

- Routine interfaces
- Data types
- Algorithms
- Routine calls

Design information is information about the low-level design of your software. There are two kinds of design information:

- Structural (modules, routines and their arguments, variables)
- Textual (description of data and algorithms in the form of comments or pseudocode)

With DECset, you specify design information in program source files. This is sometimes called **embedded program design**. You can gradually transform your design into a completed program, retaining design information in comments, or you can set aside the design and create your implementation in a new source file.

You can choose any of the programming languages supported by DECset to express your design. Your implementation can be in the same language as your design or in other languages. DECset can supply extracted design information in a variety of forms, such as LSE templates, HELP text, and hardcopy reports.

To create, modify, and retrieve low-level design information, use the following tools:

- LSE to create and modify design information (in comments and pseudocode)
- A compiler to check the syntax of your design
- A compiler and SCA to transfer design information to an SCA library
- LSE overviews, SCA queries, and SCA reports to retrieve design information

The output from the PACKAGE and HELP reports can then assist in further design and development work.

2.1. Entering Design Information with LSE

This section explains the steps for entering design information for a new or existing program. The examples are in C, but the same steps can be applied to designs and programs written in any language supported by DECset. Customer-written compilers, and even some obsolescent compilers, might not support the /DESIGN qualifier. Therefore, refer to the specific compiler's documentation to determine whether it supports the entering of design information. For more detailed information for entering designs, see Chapter 3.

This section describes the following:

- Entering design information for a module
- Adding design information for a routine
- Entering pseudocode
- Moving pseudocode to comments
- Using overview operations
- Generating an implementation from the design

2.1.1. Entering Design Information for a Module

To create a new design or program, do the following:

1. Create a source file with LSE.
2. Expand the placeholders to get the template for module header comments.

If you are adding design information to an existing source file, get the template by typing the appropriate token name (for the C language, *module_level_comments* or *MLC*), then enter the EXPAND (Ctrl/E) command. You can customize the template to match your own source formatting standards by using the LSE TOKEN commands.

The following is an example of the standard C template for module header comments:

```
/*
**++
**  FACILITY:  {@tbs@}
**
**  MODULE DESCRIPTION:
**
**      {@tbs@}
**
**  AUTHORS:
**
**      {@tbs@}
**
**  CREATION DATE:  {@tbs@}
**
**  DESIGN ISSUES:
**
**      {@tbs@}
**
**  [@optional module tags@]...
**
**  MODIFICATION HISTORY:
**
**      {@tbs@}...
**--
*/
```

Complete the header comment block by filling in text to replace the `[{@tbs@}` placeholders and delete unnecessary sections. For example:

```
/*
**++
**  FACILITY:  Sample facility 1
**
**  MODULE DESCRIPTION:
**
**      This is a sample module used to show how to use LSE and SCA to
 create
**      a detailed design.
**
**  AUTHORS:
**
**      Jane Smith
**
**  CREATION DATE:  June 27, 1998
**
**  KEYWORDS:
**
**      Examples, sample design
**
**  MODIFICATION HISTORY:
**
**      {@tbs@}...
**--
*/
```

2.1.2. Adding Design Information for a Routine

To add design information for a routine, do the following:

1. Expand additional placeholders to get a function definition template.
2. Fill in the header comment block and calling sequence information.
3. Add the type declarations needed for the function return value and argument types.

The following is an example of a typical result:

```
#include
<stdlib>typedef int integer_matrix[10][10]; ❶
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right);
/*
**++
**  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
 to
**      see if the matrices conform.
**
**  FORMAL PARAMETERS:
**
**      left:
```

```
**          The left operand.
**
**      right:
**          The right operand.
**
**  RETURN VALUE:
**
**      The result of multiplying the two matrices.
**
**--
*/
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right) ❷
{
    [@block declaration@]...;
    {@statement@}...;
}
```

Key to the example:

❶ Shows the type declaration used for return value and argument types.
❷ Shows the function definition, including function name, return value type, arguments, and their types.

Note

You have now entered enough design information so you can use SCA to retrieve design information in the form of LSE package definitions, HELP text, and a preliminary INTERNALS report. See Section 2.2 for more information.

2.1.3. Entering Pseudocode

You can enter your algorithm's design in the form of pseudocode. Pseudocode is a textual description of the actions to be performed by a piece of software. In DECset, pseudocode is bracketed by pseudocode delimiters. By default, these delimiters are « ». They can be customized by using the SET LANGUAGE PSEUDOCODE DELIMIT command.

You can mix pseudocode and final source code, as shown below. Enter pseudocode by issuing the ENTER PSEUDOCODE (PF1 Space) command, then type your pseudocode text. For example:

```
 #include
<stdlib>typedef int integer_matrix[10][10];
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right);
/*
**++
**  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
**      to see if the matrices conform.
**
**  FORMAL PARAMETERS:
**
**      left:
**          The left operand.
```

```
**
**      right:
**          The right operand.
**
**  RETURN VALUE:
**
**      The result of multiplying the two matrices.
**
**--
*/
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right)
{
    integer_matrix result_matrix; ❶
    «Loop over the rows of the left matrix» ❷
    return result_matrix; ❸
}
```

Key to the example:

❶ Shows the final source code
❷ Uses pseudocode as a loop statement
❸ Shows final source code

2.1.4. Moving Pseudocode to Comments

LSE enables you to preserve the descriptive pseudocode text for later use as design information. To change pseudocode to comment lines, issue the ENTER COMMENT command. For example, if you apply the ENTER COMMENT (PF1 B) command to the pseudocode line shown in the example code in Section 2.1.3, the result is as follows:

```
/*
** Loop over the rows of the left matrix
*/
{@tbs@}
return result_matrix;
```

You can now fill in the next level of detail of your design as follows:

```
/*
**  Loop over the rows of the left matrix
*/
for (i = 1;  i < «matrix size»;  i++)
{
 «Loop over the columns of the right matrix» ❶
};
return result_matrix;
```

Note that ❶ shows a new pseudocode line.

2.1.5. Using Overview Operations

You can use overview operations to retrieve design information stored in comments, as in this code example:

```
/*
**  Loop over the rows of the left matrix
*/
```

```
for (i = 1;  i < 10;  i++)
{
    /*
    **  Loop over the columns of the right matrix ❶
     */
     for (j = 1;  j < 10;  j++)
     {
         /*
         **  Compute the inner product of the current row and column
          */
          for (k = 1;  k < 10;  k++)
          {
              *result_matrix[i][j] =
                  *result_matrix[i][j] + *left[i][k] * *right[k][j];
           }
      };
};
return result_matrix;
```

Do an overview operation on ❶ by issuing the COLLAPSE (Ctrl \) command.

The following is the result of using the overview operation with the COLLAPSE command:

```
/*
**  Loop over the rows of the left matrix
*/
for (i = 1;  i < 10;  i++)
{
«** Loop over the columns of the right matrix ...» ❶};
return result_matrix;
```

Key to the example:

❶ Shows the retrieved design information that was originally entered as pseudocode in the example in Section 2.1.4.

2.1.6. Completing the Implementation

The source code is complete when all pseudocode placeholders are expanded. (This file is available online in SCA$EXAMPLES:DESIGN_EXAMPLE.C.)

The complete example is as follows:

```
/**
**++
**  FACILITY:  Sample facility 1
**
**  MODULE DESCRIPTION:
**
**      This is a sample module used to show how to use LSE and SCA to
**      create a detailed design.
**
**  AUTHORS:
**
**      Jane Smith
**
**  CREATION DATE:  June 27, 1998
**
**  DESIGN ISSUES:
```

```
**
**      {@tbs@}
**
**  KEYWORDS:
**
**      Examples, sample design
**
**  MODIFICATION HISTORY:
**
**      {@tbs@}
**--
*/
#include <stdlib>typedef int integer_matrix[10][10];
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right);
/**
**++
**  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
**      to see if the matrices conform.
**
**  FORMAL PARAMETERS:
**
**      left:
**          The left operand.
**
**      right:
**          The right operand.
**
**  RETURN VALUE:
**
**      The result of multiplying the two matrices.
**
**--
*/
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right)
{
    integer_matrix *result_matrix;
    int i, j, k;
    /*
    **  Allocate and initialize the result matrix
    */
    result_matrix = malloc (i*j);
    for (i = 1;  i < 10;  i++)
    {
        for (j = 1;  j < 10;  j++)
        {
            *result_matrix[i][j] = 0;
        }
    };
    /*
    **  Loop over the rows of the left matrix
    */
    for (i = 1;  i < 10;  i++)
```

```
    {
        /*
        **  Loop over the columns of the right matrix
        */
        for (j = 1;  j < 10;  j++)
        {
            /*
            **  Compute the inner product of the current row and column
            */
            for (k = 1;  k < 10;  k++)
            {
                *result_matrix[i][j] =
                    *result_matrix[i][j] + *left[i][k] * *right[k][j];
            }
        };
    };
    return result_matrix;
}
```

2.2. Retrieving Design Information

You can retrieve design information throughout the development cycle. This section describes the following topics:

- Transferring design information to an SCA library
- Using an SCA keyword query
- Generating reports, using the supported languages shown in the example in Section 2.2.3.
- Using LSE packages and help text

2.2.1. Transferring Design Information to an SCA Library

To transfer the design information to an SCA library, you must first compile your source file to generate an analysis data file. You create the file containing the comment design information by typing the following command:

```
$ CC/ANA/DESIGN/NOOBJECT design_example
```

To create an SCA library and load the analysis data file, type the following:

```
$ CREATE/DIR [.scalib]
$ SCA CREATE LIBRARY [.scalib]
$ SCA SET LIBRARY [.scalib]
$ SCA LOAD design_example
```

Some compilers generate .XREF files that can be converted to .ANA files by means of the SCA IMPORT command. For more information, see the *VSI DECset for OpenVMS Guide to Source Code Analyzer*.

2.2.2. Using an SCA Keyword Query

You can use an SCA keyword query to find which modules contain keywords. A **keyword** or keyword phrase associates a concept with a routine or module. These keyword phrases can then be used for

indexing. For example, you can label your modules and routines with keyword phrases such as User interface, OpenVMS-specific, AST reentrant, or Examples. Each module or routine can have several keyword phrases associated with it, as many as are applicable.

For example, to see which modules have the keyword Examples associated with them, enter the following:

```
SCA> FIND CONTAINING (SYMBOL=KEYWORD AND "Examples", SYMBOL=MODULE)
```

2.2.3. Generating Reports

To generate a package definition, help file, and INTERNALS report, type the following:

```
$ SCA
SCA> SET LIB [.scalib]
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME PACKAGE
SCA> SET REPORT LANGUAGE [Ada|BASIC|C|COBOL|FORTRAN|Pascal|PL/I]
SCA> SET REPORT HELP_LIBRARY   mydisk:[mydir]design_example_help
SCA> SET REPORT OUTPUT design_example
SCA> REPORT
SCA> SET REPORT NAME HELP
SCA> SET REPORT OUTPUT design_example
SCA> REPORT
SCA> SET REPORT NAME INTERNALS
SCA> SET REPORT OUTPUT design_example
SCA> REPORT
SCA> EXIT
```

An LSE package definition gives calling sequence information for routines, and can also be linked to help text for the routines and their parameters. To create an LSE environment file from the package definition, type the following:

```
$ LSEDIT/NODISP/INIT=SYS$INPUT: design_example.lse
  SET COMMAND LANGUAGE PORTABLE
  EXECUTE BUFFER LSE
  SAVE ENVIRONMENT CHANGES mydisk:[mydir]design_example.env
  EXIT
```

An OpenVMS help file contains a hierarchy of topics, along with text explaining each topic. The help files generated as DECset reports contain information about the modules in your SCA library, and about each routine and its parameters. To load the help text into a help library named example_help.hlb, type the following:

```
$ LIBR/CREATE/HELP design_example_help design_example.hlp
```

An INTERNALS report contains detailed design information about your software. It is useful to anyone who needs to understand the software in order to make changes to it (enhancements or bug fixes). To generate a PostScript output file from the INTERNALS report, type the following:

```
$ DOCUMENT design_example.sdml software.reference PS
```

2.2.4. Using LSE Packages and Help Text

For further development work, you can load your package definition into LSE by entering the following commands:

```
$ DEFINE LSE$ENVIRONMENT mydisk:[mydir]design_example.env
$ LSEDIT new.c
```

Type the routine name, matrix_multiply, then expand it by issuing the EXPAND command to get the calling sequence for the routine, as follows:

```
     matrix_multiply (
     {@left@},
     {@right@})
```

To get help text, place the cursor on the matrix_multiply name and enter the HELP INDICATED command. The following text is displayed:

```
matrix_multiply
  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.

  Returns: pointer to integer_matrix

  Additional information available:
  left       right
```

You can get a description of a routine parameter by entering its name. For example, if you type "left" you will see the following text:

```
matrix_multiply
  left
    Type: pointer to integer_matrix
    The left operand.
```

You can now fill in the parameter values and continue development.

Chapter 3. Entering Design Information Using LSE

This chapter provides information on how to enter detailed program design. It includes information about:

- Introductory material
- Storing design information in program structure
- Storing design information in comments
- Using pseudocode to enter design information

3.1. Introduction

In many software engineering environments, the last step before actual coding is to generate a detailed design. With DECset, a detailed design consists of all the following:

- Ordinary programming language syntax
- Ordinary and tagged comments
- Pseudocode placeholders

3.2. Expressing Design Information Using Program Structure

Part of your design is expressed in ordinary programming language constructs. This includes the following:

- Names of routines
- Return value type of each routine
- Names and types of parameters to the routines
- Type definitions and data declarations for high-level types and variables that are known at design time

For example, in the matrix_multiply example used in Chapter 2, the design information describing the calling sequence for the routine matrix_multiply uses ordinary C syntax:

```
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right)
```

In addition, the definition of the type integer_matrix also uses ordinary C syntax:

```
typedef int integer_matrix[10][10];
```

Given only this design information (without additional information stored in comments), you can get reports that describe the calling sequence for the routine, including type information. With the addition

of comments describing the routine, you can get reports that provide additional explanation of what the routine does, how the parameters are used, and so on.

3.3. Expressing Design Information Using Comments

A significant amount of design information is expressed using tagged comments. A **tagged comment** is a specially formatted comment consisting of a tag and related text. A tagged comment begins with a defined tag or annotation that characterizes the text of the comment. The following is an example of a tagged comment, where FUNCTIONAL DESCRIPTION is the tag:

```
-- FUNCTIONAL DESCRIPTION:
--
--     Find the arithmetic mean of a list of integers.
```

With LSE, you can easily enter tagged comments into your programs. The templates for LSE include a standard set of comment tags. You can change these tags and add new tags.

When programs are compiled with the /DESIGN qualifier, the compiler performs the following:

- Scans comment blocks, looking for tagged comments
- Inserts data about those comments into the SCA analysis file

This information can be retrieved by SCA and matched with corresponding identifiers, such as routine names that appear in the code, and be used to generate design reports.

Section 3.3.1 and Section 3.3.2 explain how to use tagged comments, and how to associate comments with objects.

3.3.1. Using Tagged Comments

Tags are defined within LSE and are saved in an LSE environment file that is read by the compiler. Default tags are in the LSE$SYSTEM_ENVIRONMENT file, where they are also available to compilers. The LSE$ENVIRONMENT logical name must be defined to make your tags available to compilers. There are several types of tags, and the value of these tags is parsed differently depending on the tag type. There are also a number of special tags, each beginning with a dollar sign ($).

The compiler groups comments into **comment blocks**. Comment blocks are separated either by code (any visible text that is not contained in a comment) or by a blank line (any blank line that is not contained in a comment). Within each comment block, the compiler looks for tagged comments. A tag must be the first text on the line of the comment, not including the comment delimiters. If the tag is not the only text on the line, it must be followed by a tag terminator (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.

There are three types of tagged comments: **text**, **keyword**, and **structured**. The following sections describe these types in detail.

3.3.1.1. Text Comments

Text comments contain ordinary text and are the most common type of tagged comments. No special processing occurs for these comments. For example:

```
-- FUNCTIONAL DESCRIPTION:
--
--     This function multiplies two matrices.
```

3.3.1.2. Keyword Lists

A keyword list contains keyword phrases used to identify sections of code. A keyword tag may be associated with keyword phrases from a predefined list, which in turn is defined with the DEFINE KEYWORDS command, or might take arbitrary keyword phrases. In either case, keyword phrases are separated by commas, and can contain space characters. For example:

```
-- FACILITY:
--
--     Sample facility
--
-- KEYWORDS:
--
--     Sample, matrix arithmetic
```

3.3.1.3. Structured Comments

The body of a structured comment consists of a sequence of one or more **subtags** and their associated text. Unlike ordinary tags, subtags need not be predefined. For example, a FORMAL PARAMETERS structured comment consists of a sequence in which each parameter name is a subtag, and is followed by the description of the parameter. A blank line is required before the first subtag, and between subsequent subtags. For example:

```
-- FORMAL PARAMETERS:
--                  ❶
--     P1:          ❷
--         The first parameter
--                  ❸
--     P2:          ❹
--         The second parameter
--                  ❺
-- RETURN VALUE: ❻
--
--     Text about return value
--
```

Key to the example:

❶ Shows the required blank line.
❷ The parameter name (P1) must be followed by a tag terminator (in this case a colon).
❸ Shows the required blank line.
❹ The parameter name (P2) must be followed by a tag terminator.
❺ Shows the required blank line.
❻ The RETURN VALUE tag name must be indented less than (to the left of) P2, and no more than FORMAL PARAMETERS.

Fully expand the *[subtags]* and *[more-subtags]* placeholders produced by the language templates to automatically get the correct formatting.

Two implicit tags are defined for all languages. These are the $UNTAGGED tag and the $REMARK tag. The $UNTAGGED tag is associated with any comment text that occurs at the beginning of a comment block, before the first tag within the comment block.

The $REMARK tag is associated with the first line of text in a comment block, not including any tag names. The PACKAGE report, for example, uses the $REMARK tagged comment for each routine as the description text for the routine.

For example:

```
function function_1 (...)
--
-- This function computes the integer function of the P1,  ❶
-- with or without P2s.
--
-- FORMAL PARAMETERS:
     ...
```

❶ These two lines are associated with the $UNTAGGED tag; the first line is also associated with the $REMARK tag.

3.3.2. Associating Comments with Declarations

You can use comments to associate design information with declarations in your program. Comments that occur in executable portions of your code, where there are no adjacent declarations, are not used for SCA reports.

SCA looks for the closest declaration that is adjacent to the comment block. If there is no adjacent declaration, the comment is associated with the outer-level declaration containing the comment, if any.

This comment association can be ambiguous. For example, suppose you have the following C fragment:

```
int x;
/* This comment describes a variable */
int y;
```

The declarations of *x* and *y* are equally close to the comment. You can put in blank lines to get the association you want. For example:

```
int x;
/* This comment describes a variable */
int y;
```

This results in the comment being associated with *x*. Consider the following example:

```
int x;
/* This comment describes a variable */
int y;
```

This results in the comment being associated with *y*, because the declaration of y is now closer to the comment than is the declaration of x.

If you leave an ambiguous situation in your code, SCA uses the setting from the LSE SET LANGUAGE COMMENT ASSOCIATION command. (See the entries for SETLANGUAGE commands for more details.)

When you use the /DESIGN=COMMENTS qualifier to compile your source program, the compiler uses your LSE$ENVIRONMENT files and the LSE$SYSTEM_ENVIRONMENT file to determine the setting of the SET LANGUAGE COMMENT ASSOCIATION command. That setting is stored in your analysis data file. SCA performs the comment association, using that setting, at the time you load the file into the SCA library. If you want 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 analysis data file into SCA.

3.4. Expressing Design Information Using Pseudocode

Pseudocode is easy to write and provides a way to sketch your design ideas. You can convert pseudocode to comments, thus providing a way to preserve design information.

Each compiler that supports the /DESIGN qualifier has a set of conventions describing in what context pseudocode is allowed. Check the documentation for your compiler to learn where pseudocode is allowed in your design. See also Section 4.2.1 for more detail.

While working on your detailed design, you can compile to verify your program syntax, as follows:

```
/DESIGN=(PLACEHOLDERS,COMMENTS)/ANA
```

When your implementation is complete, you can compile with these qualifiers:

```
/DESIGN=(NOPLACEHOLDERS,COMMENTS)/ANA
```

This verifies that no placeholders or pseudocode remain in your source file,and, at the same time, stores information about comments in the analysis datafile.

To create designs for individual routines, expand the LSE placeholders as necessary. Use tagged comments and pseudocode placeholders to contain design information that is still at an abstract level, and use actual code for those portions that are known.

To enter an algorithm or data declaration design, use the ENTER PSEUDOCODE command. See the command dictionary in the *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* for further information regarding pseudocode commands.

3.4.1. Using Pseudocode in Data Declarations

The design of data structures can be a part of a detailed design. An example of the design of an Ada record is 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, as follows:

❶ LSE generates the *{discrete_range}...* placeholder.
❷ Shows a pseudocode placeholder. This is created using the ENTER PSEUDOCODE (PF1 Space) command, then typing the contents.
❸ Shows a pseudocode placeholder, which describes the next field of the record in general terms.

Two important points concerning pseudocode placeholders are shown in this example:

- A pseudocode placeholder typically contains ordinary text, not source code.
- It is preferable to fill in as much detail of the design as possible as actual source code.

You can declare the previous example as follows:

```
type record_type is
    «a complicated record definition»;
```

However, this format provides little information for your SCA database. For example, 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.

3.4.2. Using Pseudocode in Algorithms

The following example shows a routine body in Ada:

```
  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;
```

Key to the example:

❶ Uses pseudocode as the conditional expression in the *if* statement.
❷ Uses pseudocode to represent the entire body of the *then* clause of the statement.
❸ Shows a procedure call. The procedure specification (not shown) must also be present for Ada to recognize this procedure call. With the completed design, you can use SCA to get information about calls to this routine, including this call.
❹ Contains an LSE list placeholder. You can use placeholders as part of a design in any context in which they normally appear as the result of an LSE expansion operation. In this example, the *[statement]...* placeholder remains as a convenience because the algorithm is not yet complete.

3.4.3. Refining the Design

As the design is refined, you replace pseudocode by more detailed pseudocode,and then by final source code. To preserve the original design information, use the ENTER COMMENT (PF1 B) command. This applies both to the low-level design phase and the implementation phase.

For example, the *if* statement used in the previous example can be refined as follows:

```
  if P2 /= null_P2 then       -- 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;
```

Key to the example:

❶ Uses the ENTER COMMENT LINE command to move the pseudocode for the *if* statement over to the right, before writing the condition.

❷ Uses the ENTER COMMENT BLOCK command 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 generic `{tbs}` placeholder.

Chapter 4. Retrieving Design Information

This chapter provides information on retrieving design information by running standard reports, the options available for each report, and sample outputs for each report.

This chapter describes the following topics:

- Using overviews to retrieve design information
- Using the OpenVMS compilers and SCA to get design information into an SCA library
- Finding keywords
- Generating design reports

4.1. Using Overviews to Retrieve Design Information

A powerful feature of the DECset design environment is the ability to display overviews of your code, hiding low-level details to give you a better view of a larger section of code. You can refine the overall results with the LSE NEW ADJUSTMENT command. You can see varying levels of detail when you edit a source file by using the LSE OVERVIEW commands: VIEW SOURCE, COLLAPSE, FOCUS, and EXPAND (see the LSE documentation for more information on these commands). The INTERNALS report uses these LSE commands to produce the body section for each routine.

The following example uses Ada:

```
package body example is

type integer_matrix is array (integer range <>, integer range <>) of
integer;

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 to
--  see if the matrices conform.
--
-- FORMAL PARAMETERS:
--
--  left:
--          The left operand.
--
--  right:
--          The right operand.
--
```

```
-- RETURN 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;
end example;
```

Do a COLLAPSE operation on the first line of the function. The result is a single line overview of the function, as in the following:

```
«function matrix_multiply (left, right : in integer_matrix) ...»
```

Do an EXPAND operation on the overview line to get additional detail, as in the following:

```
function matrix_multiply (left, right : in integer_matrix)
    return integer_matrix is
«-- FUNCTIONAL DESCRIPTION: ...»
    «result_matrix : ...»
«begin ...»
end matrix_multiply;
```

Now do an EXPAND on the «`begin ...`» line to see the next level of detail:

```
function matrix_multiply (left, right : in integer_matrix)
    return integer_matrix is
«-- FUNCTIONAL DESCRIPTION: ...»
    «result_matrix : ...»
begin
    -- Loop over the rows of the left matrix
    --
    «outer_loop: for i in left'range loop ...»
    return result_matrix;
end matrix_multiply;
```

You can continue doing overview operations to selectively display details of the function, and to hide details that are not currently of interest. You can also perform editing functions on overview lines. (See the command dictionary in the *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* for information on using editing commands with overviews.)

4.2. Transferring Design Information to an SCA Library

Once there is a partial or complete design, you can process the design by using a compiler and SCA.

4.2.1. Compiling Design Information

With OpenVMS compilers that support SCA, use the /DESIGN qualifier to tell the compiler to process design information. In addition, you can use the /DESIGN qualifier with the SCA ANALYZE command to process design information in your source code if your compiler does not support SCA. This qualifier takes two keyword values, as follows:

- [NO]COMMENT—Tells the compiler to search inside comments for program design information
- [NO]PLACEHOLDERS—Tells the compiler to recognize placeholders (including pseudocode placeholders) as valid program syntax

While working on your detailed design, you can compile to verify your program syntax:

```
/DESIGN=(PLACEHOLDERS,COMMENTS)/ANA
```

This process checks the syntax of your source code and stores information about comments in the analysis data file. When your implementation is complete, you can compile with these qualifiers:

```
/DESIGN=(NOPLACEHOLDERS,COMMENTS)/ANA
```

This verifies that no placeholders or pseudocode remain in your source file, and, at the same time, stores information about comments in the analysis datafile.

4.2.2. Loading Design Information into an SCA Library

To load analysis data files into an SCA library, use the SCA command LOAD. SCA does not recognize any differences between an analysis data file containing design information and one containing final code information. If a design evolves directly into an implementation, you can use the same arrangement of libraries during design as during implementation.

To refer to your design while continuing implementation, you can set up two, parallel SCA libraries:

- One for keeping design information
- One for holding the implementation information

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 modules being implemented occlude their designs. For those modules still in the design stage, the designs are still available.

To refer to both the code and 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 recognizes only one module of a given name in any library.

- You can move back and forth, using the SCA command SET LIBRARY.

4.3. Using Keyword Queries

Once the analysis data files containing design information are loaded into an SCA library, you can use SCA queries to retrieve information, as with any other SCA library. The symbol classes defined by SCA specifically for design information are keyword, placeholder, and tag. To get design information, you use these classes with the SYMBOL=construct of the SCA query language.

For example, if you want to find all routines that are marked with the keyword interface, use the following SCA command:

```
SCA> FIND CONTAINED_BY(SYMBOL=routine, 'interface' AND SYMBOL=KEYWORD,
DEPTH=1)
```

4.4. Generating Reports

In addition to getting information directly from SCA queries, you can retrieve design information by generating reports. A report covers all or a designated part of your SCA database and presents information in a structured way. You must have both LSE and SCA on your system to generate reports.

You generate reports with the SCA command REPORT. SCA provides four reports that can be customized, or you can add new reports. See Chapter 5 for more information.

The output for each standard report is controlled by the use of report options. The following sections explain how to set report options, and list the available options for each standard report.

4.4.1. Using the Report Commands

Use the REPORT commands for reports provided by SCA and for customized reports that you create. The SCA commands are as follows:

- SET REPORT
- SHOW REPORT
- RESET REPORT
- REPORT

SET REPORT option-name option-value

This command sets an option value. The options available for each standard report are listed in Section 4.4.10.You can also obtain a list of all the current option values by using the command SHOW REPORT *.

Before issuing any SET REPORT, RESET REPORT, or SHOW REPORT commands,you must select that report. This is done by using the SET REPORT NAME command. For example,if you want to run a PACKAGE report with trace messages enabled, use the following set of commands:

```
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME PACKAGE
SCA> SET REPORT TRACE_MESSAGES ON
SCA> REPORT PACKAGE
```

RESET REPORT option-expression

This command restores the default values of a set of options. The default option values for each standard report are listed in the Section 4.4.10. The option-expression can be the name of a single option or a wildcard name. For example, the following command resets all options whose names begin with the letter T:

```
SCA> RESET REPORT T*
```

The following command resets all report options for the current report:

```
SCA> RESET REPORT *
```

SHOW REPORT option-expression

This command displays the values of a set of options. The option-expression can be the name of a single option or a wildcard name. For example, the following command displays the values of all options whose names begin with the letter T.

```
SCA> SHOW REPORT T*
```

The following command displays all option values for the current report.

```
SCA> SHOW REPORT *
```

REPORT [report-name]

This command generates a report. If a SET REPORT NAME command has been given, the report-name is optional; if specified, it must match the name given in the SET REPORT NAME command. When the report is complete, all option values are reset to their default values.

The report formats provided by SCA are as follows:

- HELP—An OpenVMS 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

4.4.2. General Report Information

The output of the REPORT command is usually not in its final state. HELP reports must be loaded by the OpenVMS Librarian utility 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 DECdocument, or with DIGITAL Standard Runoff.

Because reports perform many SCA queries, they can be time consuming. For this reason, VSI recommends that you use the REPORT command from batch jobs.

Reports are based on two types of information from your design or program:

- Program declarations (modules, routines, types, and variables)
- Design information stored in comments

Reports 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 SCA reports use tags that are included in the system environment file supplied with LSE. You can use the SHOW TAGS command in LSE to show the tags for a particular language.

In general, the tags applicable for an entire file or module are distinct from the tags applicable for a single subroutine. For example, the ABSTRACT tag describes a module, whereas 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.

Make sure your SCA library is consistent with the current state of your source files. Otherwise, the report tool cannot locate the comment text in your source files. If you move your source files after they have been compiled, use the LSE$SOURCE logical name to indicate to the report tool where the source files are. This logical name definition performs the equivalent task as the SET DIRECTORY SOURCE LSE command. Your definition of this logical will have he same attributes and capabilities. (Refer to the *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* for information about the SET DIRECTORY SOURCE command.) For example:

```
$ DEFINE LSE$SOURCE MYDISK:[MYDIR]
```

This definition tells the report generator to look in the directory MYDISK:[MYDIR] if it is unable to find a source file in the directory where it was compiled.

4.4.3. DOMAIN Option

The report tool steps through the files in the domain one at a time and steps through the routines within each file one at a time. The default domain for the report is the set of all files that have command-line references in your SCA library, as follows:

```
SCA> FIND (SYMBOL=FILE AND OCCURRENCE=COMMAND_LINE)
```

To limit reports to specific files in your system, perform 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 command with the -name option.
3. Use the query name as the domain for the report.

The following example limits the report to only those files containing the string *matrix* as part of the file name:

```
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> FIND -name MYQUERY *matrix*  AND SYMBOL=FILE AND
OCCURRENCE=COMMAND_LINE

SCA> SET REPORT NAME report_name
SCA> SET REPORT DOMAIN MYQUERY
SCA> REPORT
```

4.4.4. FILL Option

The FILL option is applied to INTERNALS and 2167A reports. In cases where comment text is copied into the report, the FILL option determines whether the text will be filled. Use FILL OFF if your comments typically contain tables or other formatted output that should not be filled.

4.4.5. DESIGN_EXAMPLE Source File

The source file at the end of Chapter 2 is used to generate sample output in the following sections. This file is also available online in SCA$EXAMPLES:DESIGN_EXAMPLE.C. Information on customizing reports to get different output is given in Chapter 5.

Each report is described in separate sections, as follows:

- Description
- Sample output
- Table indicating where the information in the output file comes from. These sources are as follows:
  - Your program structure
  - Comments in your source files
  - Report options that you enter when you generate the report

In the case of tagged comments, the table also indicates what tag names are used for the information, and, if applicable, the name of the TPU variable that contains the list of tag names. These TPU variables used for reports all start with *sca$report_*. In the table, this prefix is replaced by ellipses (...).

The variables and constants referred to in the following sections are defined in the file SYS$LIBRARY:SCA$REPORT_CUSTOMIZATIONS.TPU.

The following is a portion of the DESIGN_EXAMPLE source file used to generate the sample report output:

Module Header Comments

```
/*
**++
**  FACILITY:  Sample facility 1
**
**  MODULE DESCRIPTION:
**
**      This is a sample module used to show how to use LSE and SCA to
**      create a detailed design.
**
**  AUTHORS:
**
**      Jane Smith
**
**  CREATION DATE:  June 27, 1998
**
**  DESIGN ISSUES:
**
**      {@tbs@}
**
**  KEYWORDS:
**
**      Examples, sample design
**
**  MODIFICATION HISTORY:
```

```
**
**      {@tbs@}
**--
*/
```

Include Files and Type Declarations

```
#include <stdlib>typedef int integer_matrix[10][10];
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right);
```

Routine Header Comments

```
/*
**++
**  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
**      to see if the matrices conform.
**
**  FORMAL PARAMETERS:
**
**      left:
**          The left operand.
**
**      right:
**          The right operand.
**
**  RETURN VALUE:
**
**      The result of multiplying the two matrices.
**
**--
*/
```

Routine Definition

```
integer_matrix *matrix_multiply (integer_matrix *left,
integer_matrix *right)
```

4.4.6. Creating Online HELP

The HELP report produces an .HLP file that the OpenVMS Librarian utility loads into a standard OpenVMS HELP library. See the *VSI OpenVMS Librarian Utility Manual* for information on help libraries. See also Section 4.4.10 for standard report options reference tables.

The following is an example of a command sequence that generates a HELP report:

```
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME HELP
SCA> SET REPORT OUTPUT X.HLP
SCA> REPORT
```

After generating a HELP report, you can load the .HLP file into an OpenVMS HELP library by using the following command:

```
$ LIBRARY/HELP/CREATE help-library-name help-file-name
```

In the previous command, *help-library-name* is the name of the HELP library that you are creating, and *help-file-name* is the name of the .HLP file generated by the REPORT command.

You can tell the OpenVMS Librarian the maximum key size allowed within a given library by specifying the CREATE=KEYSIZE qualifier. For HELP libraries, the default is 15. For REPORT help files, this might be too small because it limits the top level help key to 15 characters. To increase the key size,use the following command:

```
$ LIBRARY/HELP/CREATE=KEYWORD:nn help-library-name help-file-name
```

If the key size you specify is too small, the OpenVMS Librarian displays the following message when you load the .HLP file: “Key XXX name length illegal”. To find the key size of an existing library, use the LIBRARY/LIST command to read the maximum key length in the header information.

The following is the output of the HELP report for the example program used in Chapter 2. The callouts in the example are described in Table 4.1.

```
1 DESIGN_EXAMPLE ❶
This is a sample module used to show how to use LSE and SCA to create ❷
a detailed design.
2 matrix_multiply ❸
This function computes the matrix product of two integer matrices. ❹

It uses a simple, triple-nested loop, and does not do any checking to
see if the matrices conform.

Returns: pointer to integer_matrix ❺
3 left ❻
Type: pointer to integer_matrix ❼

The left operand. ❽
3 right
Type: pointer to integer_matrix

The right operand.
```

Table 4.1. HELP Report

Callout	Source of Information	TPU Variable
❶	Program structure—module name, or file name if there is no module name	
❷	Tagged comment: PROGRAM DESCRIPTION PACKAGE DESCRIPTION MODULE DESCRIPTION ABSTRACT	...module_descriptions
❸	Program structure—routine name	
❹	Tagged comment: FUNCTIONAL DESCRIPTION	...routine_description_tags
❺	Program structure—return value type	

Callout	Source of Information	TPU Variable
❻	Program structure—parameter name	
❼	Program structure—parameter type	
❽	Untagged comment associated with parameter declaration or tagged comment: FORMAL PARAMETERS FORMAL ARGUMENTS PARAMETERS ARGUMENTS	...routine_parameters

4.4.7. Creating LSE Package Definitions

The PACKAGE report produces an .LSE file, used by LSE to define LSE packages for your program.

The following is an example of a command sequence that generates a PACKAGE report:

```
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME PACKAGE
SCA> SET REPORT OUTPUT name.LSE
SCA> REPORT
```

To use the result of a PACKAGE report, issue the following LSE commands, then define the LSE$ENVIRONMENT logical name to point to the new environment file that you created. You can then use the package definitions within subsequent editing sessions.

```
LSE> SET COMMAND LANGUAGE LSE
LSE> OPEN FILE package-name.LSE
LSE> EXECUTE BUFFER LSE
LSE> SAVE ENVIRONMENT CHANGES file_name
```

The following is the output of the PACKAGE report for the example program used in Chapter 2. The callouts in the example are described in Table 4.2.

```
LSE NEW PACKAGE DESIGN_EXAMPLE ❶
    LSE SET PACKAGE ROUTINE EXPAND LSE$PKG_EXPAND_ROUT_
    LSE SET PACKAGE PARAMETER EXPAND LSE$PKG_EXPAND_PARM_
    LSE SET PACKAGE HELP LIBRARY SYS$LOGIN_DEVICE:[]DESIGN_EXAMPLE_HELP ❷
    LSE SET PACKAGE HELP TOPIC "DESIGN_EXAMPLE" ❸
    LSE SET PACKAGE LANGUAGE ADA     ❹
    LSE SET PACKAGE LANGUAGE BASIC   ❹
    LSE SET PACKAGE LANGUAGE C       ❹
    LSE SET PACKAGE LANGUAGE COBOL   ❹
    LSE SET PACKAGE LANGUAGE FORTRAN ❹
    LSE SET PACKAGE LANGUAGE PASCAL  ❹
    LSE SET PACKAGE LANGUAGE PLI     ❹

LSE NEW ROUTINE "matrix_multiply" ❺
    LSE SET ROUTINE HELP TOPIC "matrix_multiply" ❻
    LSE SET ROUTINE DESCRIPTION -
    "This function computes the matrix product of two integer matrices." ❼
    LSE NEW ROUTINE PARAMETER left ❽
    LSE SET ROUTINE PARAMETER value
    LSE NEW ROUTINE PARAMETER right ❾
    LSE SET ROUTINE PARAMETER value
```

Table 4.2. PACKAGE Report

Callout	Source of Information
❶	Program structure—module name, or file name if there is no module name
❷	Help library file spec comes from HELP_LIBRARY option
❸	Program structure—module name, or file name if there is no module name
❹	Language names come from LANGUAGES option
❺	Program structure—routine name
❻	Program structure—routine name
❼	$REMARK comment for the routine
❽	Program structure—parameter name
❾	Program structure—parameter name

4.4.8. Creating INTERNALS Reports

The INTERNALS report is a comprehensive report on your system, on a module-by-module, routine-by-routine basis. The INTERNALS report extracts information from tagged comments to describe the various aspects of your program. For example, information under the FUNCTIONAL DESCRIPTION tag is used to describe each routine, whereas 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. The output is a file suitable for processing by DECdocument. The default output file name is INTERNALS.SDML.
- RUNOFF—The output is a file suitable for processing by DIGITAL Standard Runoff (DSR). The default file name is INTERNALS.RNO.
- TEXT—The output is a file that you can read directly. The default file name is INTERNALS.TXT.

For example, to produce an INTERNALS report that can be processed by DECdocument, type the following commands:

```
SCA> SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME INTERNALS
SCA> SET REPORT TARGET DOCUMENT
SCA> REPORT
```

Process the resulting file with DECdocument. You must use the SOFTWARE.REFERENCE doctype, as follows:

```
$ DOCUMENT INTERNALS.SDML SOFTWARE.REFERENCE destination
```

The SDML files generated by the REPORT command are suitable for printable books. They are not directly usable by the Bookreader. To make an optional Book reader-compatible file, do the following:

1. Add a front-matter section to the SDML files you are processing. For example:

   ```
   $ SCA REPORT INTERNALS
   ```

Edit INTERNALS.SDML and add the following text at the beginning of the file, after the <COMMENT> line:

```
<FRONT_MATTER>
<TITLE_PAGE>
<TITLE>(INTERNALS Report)
<ENDTITLE_PAGE>
<ENDFRONT_MATTER>
```

Note

You can supply whatever title you choose in place of "INTERNALS Report".

2. Add symbol names to your SDML files.

```
$ DOCUMENT/GENERATE_SYMBOL INTERNALS.SDML
```

3. Process the resulting file using the SOFTWARE.ONLINE doctype and the BOOKREADER destination, and specify /CONTENTS.

```
$ DOCUMENT/CONTENTS INTERNALS.SDML SOFTWARE.ONLINE BOOKREADER
```

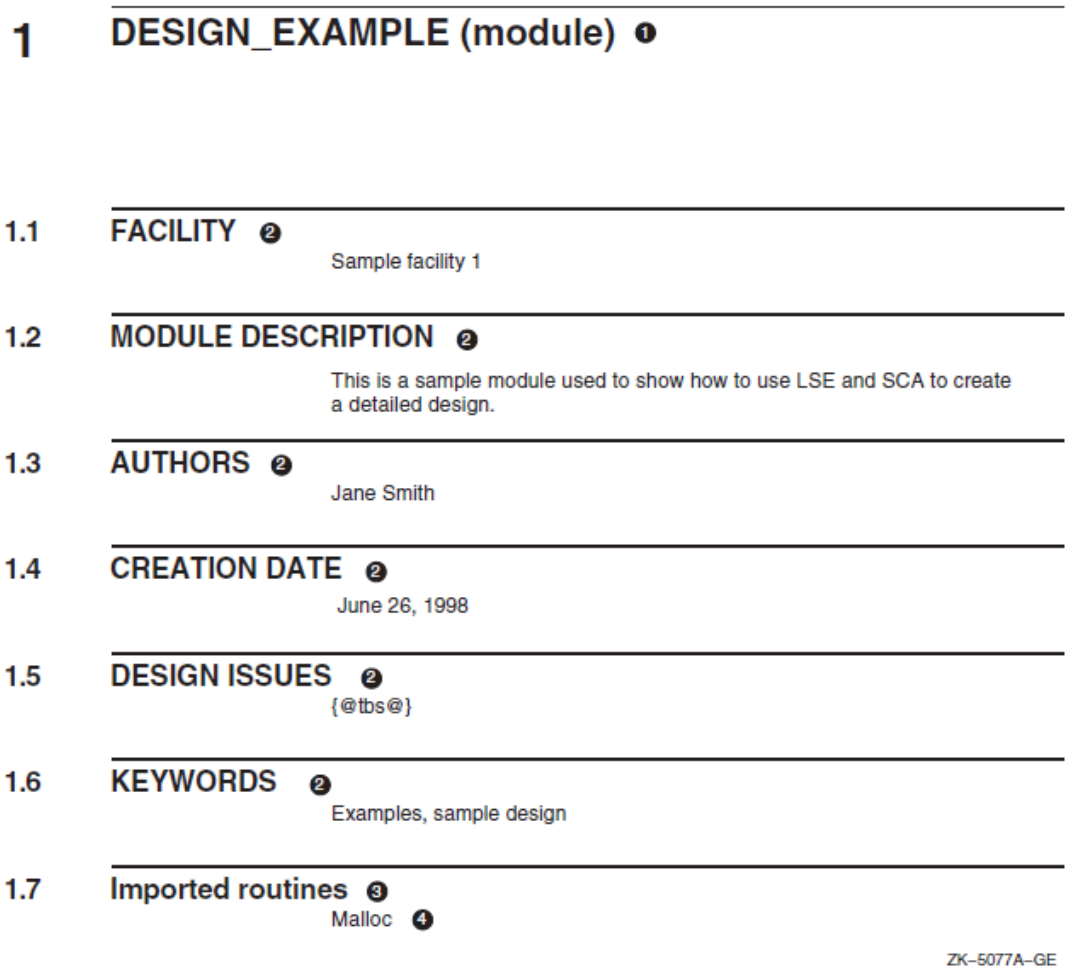
An INTERNALS report contains a chapter for each file in the report domain. Each chapter contains the following:

- Information from module or file level tags, such as ABSTRACT
- 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*. In addition, the body of the routine is presented in a hierarchical fashion, using overviews to hide details at the upper layers, and proceeding until the entire body has been produced.

Figure 4.1 is an example of an INTERNALS report for compilation units.

Figure 4.1. INTERNALS Report Information for Compilation Units Example



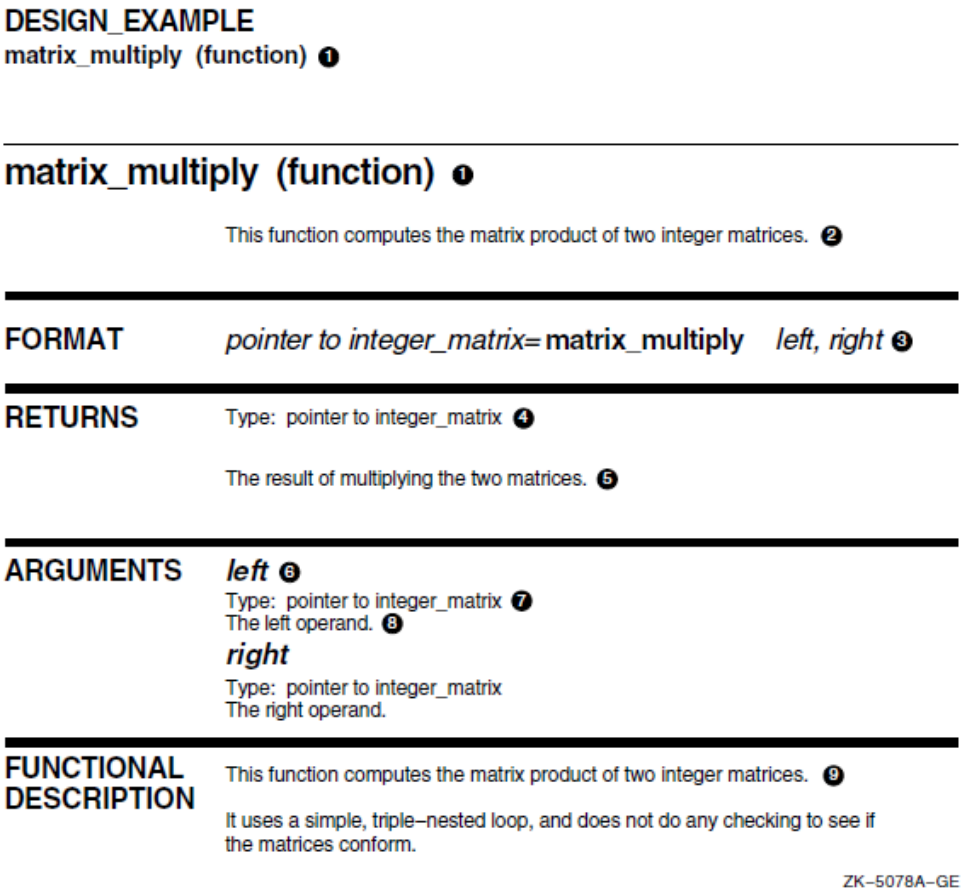
The following table describes the sources of information for the callouts in Figure 4.1.

Callout	Source of Information
❶	Program structure—module name, or file name if there is no module name plus declaration class of module.
❷	Tagged comment—one section for each module-level tagged comment.[1]
❸	Sections on module-level program structure (imported and exported routines, variables, and types; COMMON blocks, and soon)—name and (if applicable) type information comes from program structure. Object description (for exported objects and module-wide objects) comes from the comment associated with object's declaration. Section title comes from the constant string in SCA$REPORT_CUSTOMIZATIONS.TPU.
❹	Program structure—routine name.

[1]These tagged comments are typically in the module header comment block.

Figure 4.2 is an example of an INTERNALS report for a routines section.

Figure 4.2. INTERNALS Report Information for a Routines Section Example



The following table describes the sources of information for the callouts in Figure 4.2.

Callout	Source of Information	TPU Variable
❶	Program structure—routine name plus declaration class	
❷	$REMARK comment for routine	
❸	Program structure—routine name, parameter names, return value type	
❹	Program structure—return value type	
❺	Tagged comment: RETURN VALUE ROUTINE VALUE FUNCTION VALUE	...routine_return_value
❻	Program structure—parameter name	
❼	Program structure—parameter type	
❽	Untagged comment associated with parameter[1] declaration or tagged comment: FORMAL PARAMETERS FORMAL ARGUMENTS	...routine_parameters

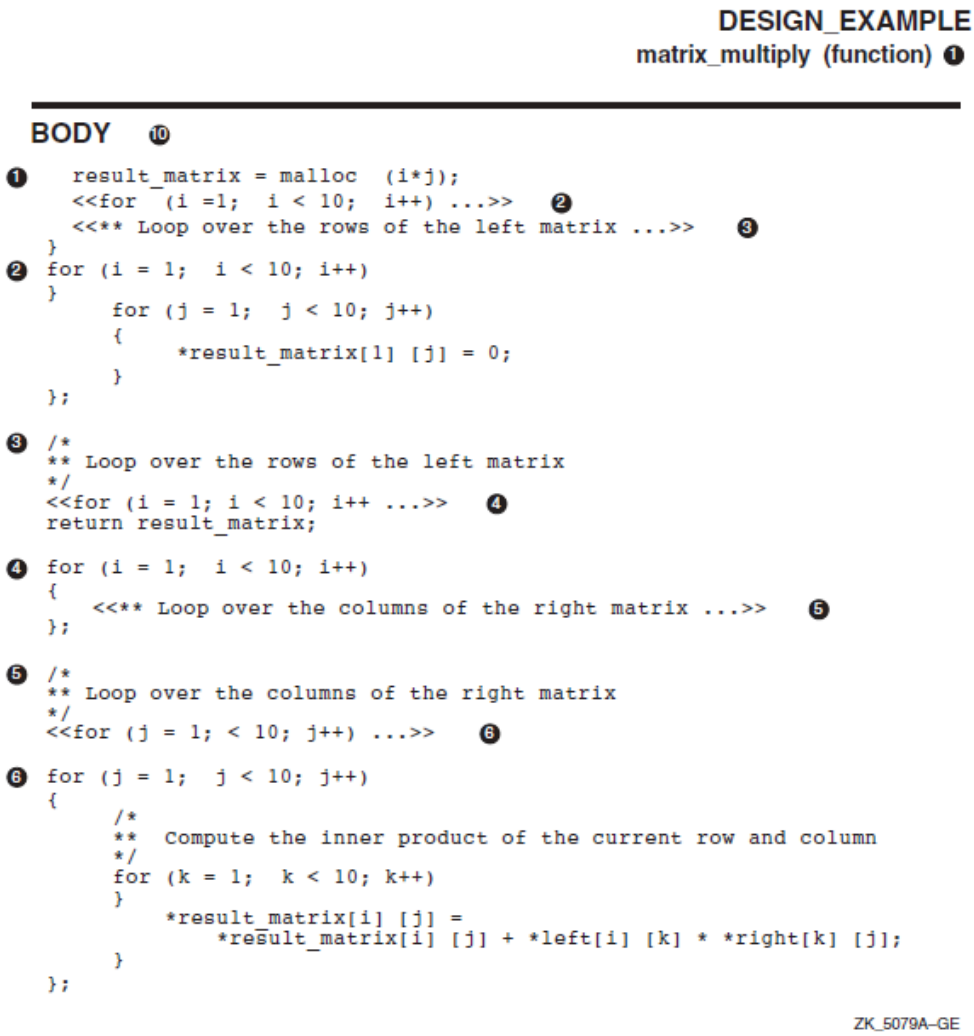
Callout	Source of Information	TPU Variable
	PARAMETERS ARGUMENTS	
❾	Tagged comment—one section for each routine-level tagged comment[2]	

[1]Each parameter description can be either from within a structured comment (under a subtag that matches the parameter name), or from the untagged comment associated with the parameter declaration.

[2]These tagged comments are typically in the routine header comment block. Tag names on the excluded list (*sca$report_excluded_tags* constant) and module-level tag names are excluded from this section. Sections are in the same order as the corresponding text appears in the source file.

Figure 4.3 is an example of an INTERNALS report for a body section.

Figure 4.3. INTERNALS Report Information for a Body Section Example



The following table describes the sources of information for the callouts in Figure 4.3.

Callout	Source of Information[1]
❿	Body section—Program structure gives the range of lines in the source file that compose the body of the routine. The LSE overview feature is used to display progressively detailed sections of the code.
❶	The body of the routine begins with a final source code statement.
❷	The second item of the body is pseudocode expanded to a nested for loop.

Callout	Source of Information[1]
❸	The third and final item of the body is pseudocode expanded with nested pseudocode. Note that the pseudocode description is included in the expansion.
❹	A for loop expansion is embedded with the prior left matrix loop and also expands to include an additional pseudocode loop for the right matrix.
❺	The right matrix pseudocode is expanded and includes one final pseudocode entry. Note that the pseudocode description is included in the expansion.
❻	The final expansion occurs with a nested for loop.

[1]Final expanded items are marked with the callout figures seen on the left side. A right-side callout shows pseudocode to be expanded.

4.4.9. 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 specified by MIL-STD-2167A. The report tool creates the design section, Section 4 of the 2167A Software Design Report. You can include this output file in your complete Software Design Report, as follows:

- Use the DECdocument `<INCLUDE>` or `<ELEMENT>` tag for DECdocument reports.
- Use the .REQUIRE directive for DIGITAL Standard Runoff (DSR) reports.
- Merge the output of the REPORT command manually 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
```

These profile files use the appropriate commands to include the lower-level files in the report. The SCA $2167A 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 another design tool.

Type the following SCA command to generate the file 2167A_DESIGN.SDML:

```
$ 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 option, you must also change the profile file using the new file name.

Invoke DECdocument to generate a Software Design report, as follows:

```
$ DOCUMENT /CONTENTS SCA$2167A.SDML MILSPEC destination_type
```

4.4.9.1. 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 sublevel components, or units. A **unit** is the lowest level entity described in the design. For SCA 2167A_DESIGN reports, use tagged comments to represent this structure.

The 2167A_DESIGN report treats each file in your system as a unit of the 2167A design. You specify design information for each unit in a comment block in the source file. Because a 2167A component

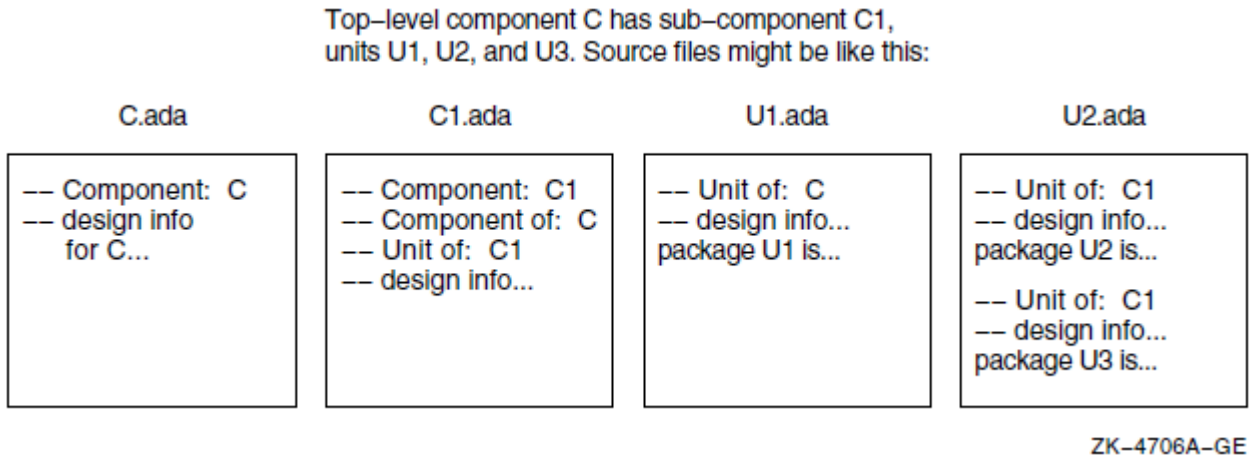
is a collection of units and subcomponents, the 2167A_DESIGN report maps a set of files into each component. However, it is redundant to duplicate all of the component design information in each file of the component. Instead,select one file as the main design file of the component and put the design information there. The other files in the component contain a single tagged comment that names the component to which they belong.

The special tags used to designate 2167A relationships are as follows:

- COMPONENT—Used in each file that you designate as the main design file for a component, either top level or sublevel; the comment names that component.
- COMPONENT OF—Used in sublevel components to name the parent of the sublevel component.
- UNIT OF—Used in each unit (each file of your system) and names the component to which the file belongs.

Figure 4.4 shows a basic layout for a set of source files and how the special tags are used.

Figure 4.4. Source Files of Special Tags



You can find a set of template files in the SCA$2167A directory, if you choose this option when SCA is installed. For the example, perform 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 commands:

```
$ SCA
SCA SET COMMAND LANGUAGE PORTABLE
SCA> SET REPORT NAME 2167A_DESIGN
SCA> SET REPORT OUTPUT mydir:2167a_design
SCA> REPORT
```

To process the report with DECdocument, perform the following steps:

1. Copy the SDML profile files from SCA$2167A into your directory, as follows:

```
$ COPY SCA$2167A:*.SDML mydir:
```

2. Define 2167A_DESIGN to point at the report output file you generated, as follows:

```
$ DEFINE 2167A_DESIGN mydir:2167a_design.sdml
```

3. Invoke DECdocument with a recognized destination_type, such as POST or LINE, as follows:

```
$ DOCUMENT /CONTENTS 2167A_PROFILE MILSPEC destination_type
```

4.4.9.2. 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 (COMPONENT, COMPONENT OF, and UNIT OF) 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:

```
FIND CONTAINED_BY ( -
    END = "UNIT OF" AND SYMBOL=TAG, -
    BEGIN = "Component 1" AND SYMBOL=KEYWORD, -
    DEPTH = 1, -
    RESULT = BEGIN))
```

Similarly, you can use queries on the COMPONENT OF tag to find sublevel components of a given component.

The 2167A_DESIGN report uses this information to create the report. It starts by finding the names of all the components of the system.

The report goes through the components one at a time, and writes the component section for each. It finds the units that belong to each component, and writes a unit subsection for each unit.

The data in the 2167A Software Design Report is obtained from the tagged comments in your source files. Table 4.3 shows the tags corresponding to paragraphs in the report.

Table 4.3. Tags for Component and Unit Information

TAGS FOR COMPONENT INFORMATION:	
Tag:	**Description of corresponding section:**
COMPONENT DESCRIPTION	General description of the component
INPUT/OUTPUT DATA	Input and output data for the component
ALGORITHMS	Algorithms used by the component
ERROR HANDLING	Error detection and 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:**
UNIT DESCRIPTION	General description of the unit
INPUT/OUTPUT DATA ELEMENTS	Input and output data for the unit
LOCAL DATA ELEMENTS	Data used only in this unit

INTERRUPTS AND SIGNALS	Interrupts and signals handled by this unit
UNIT ALGORITHMS	Algorithms used by this unit
UNIT ERROR HANDLING	Error detection and 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 conventions, the 2167A report makes no attempt to use program elements (packages, routines, and so forth.). All information in the report is obtained from tagged comments. However, it is 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 5 for more information.)

4.4.10. Options for Standard Reports

The options for 2167A_DESIGN reports are shown in Table 4.4.

Table 4.4. 2167A_DESIGN Report Options

Option Name	Report Description	Default Value
DOMAIN_QUERY	Name of query to be used as the domain.	null string
FILL	Controls whether comment text is filled.	ON
OUTPUT	The output file specification.	2167A_DESIGN.target-type
TARGET	Indicates the type of output file to generate.	SDML (other valid values are TEXT, TXT, RUNOFF, RNO, DSR, DOCUMENT)
DESCRIPTION_INDENT_ WIDTH	Gives the width of the first column of two-column tables.	32
LIST_STYLE	Gives the list style to use.	*sca$report_k_list_numbered*
LITERAL_ANGLE_BRACKETS	Controls whether text that contains words in angle brackets, which might otherwise be interpreted as tags, should include the LITERAL tags. ON means to add the LITERAL tags.	ON

Option Name	Report Description	Default Value
SCA_DEBUG_MESSAGES	Controls whether to enable debugging messages giving SCA status values.	OFF
STATUS_MESSAGES	Controls whether to enable status messages.	ON
TRACEBACK_FLAG	Controls whether to enable TPU traceback messages during report execution.	ON
TRACE_MESSAGES	Controls whether to enable trace messages generated by calling *sca $report_trace_message*.	OFF
USE_SOURCE_SPELLING	Controls whether the spelling of routine, variable, and type names are taken from the source file or are supplied by SCA.	ON

The options for INTERNALS reports are shown in Table 4.5.

Table 4.5. INTERNALS Report Options

Option Name	Report Description	Default Value
DOMAIN_QUERY	Name of query to be used as the domain.	null string
FILL	Controls whether comment text is filled.	ON
OUTPUT	The output file specification.	INTERNALS.target-type
TARGET	Indicates the type of output file to generate.	SDML(other valid values are TEXT, TXT, RUNOFF, RNO, DSR, DOCUMENT)
DECL_CLASS_MODULES	Controls whether declaration class information appears in report output.	ON, but overridden for some languages
DECL_CLASS_ROUTINES	Controls whether declaration class information appears in report output.	ON, but overridden for some languages
DESCRIPTION_INDENT_ WIDTH	Gives the width of the first column of two-column tables.	32
INTERNALS_MAXIMUM_ FRAGMENT_SIZE	Gives the maximum number of lines of source code to be displayed in a single code fragment of an INTERNALS routine body.	12
LIST_STYLE	Gives the list style to use.	*sca$report_k_list_numbered*
OVERVIEW_LEVEL	Controls what depth of detail to display for the body section.	-1
ROUTINE_DEPTH	Controls whether nested routines are reported. For example, a	1 (top-level routines only)

Option Name	Report Description	Default Value
	routine_depth of 2 means to report on top-level routines plus the first level of nested routines.	
SCA_DEBUG_MESSAGES	Controls whether to enable debugging messages giving SCA status values.	OFF
STATUS_MESSAGES	Controls whether to enable status messages.	ON
TRACEBACK_FLAG	Controls whether to enable TPU traceback messages during report execution.	ON
TRACE_MESSAGES	Controls whether to enable trace messages generated by calling *sca $report_trace_message*.	OFF
USE_SOURCE_SPELLING	Controls whether the spelling of routine, variable, and type names are taken from the source file or are supplied by SCA.	ON

The options for HELP reports are shown in Table 4.6.

Table 4.6. HELP Report Options

Option Name	Report Description	Default Value
DOMAIN_QUERY	Name of query to be used as the domain.	null string
FILL	Controls whether comment text is filled.	ON
OUTPUT	The output file specification.	HELP.HLP
TARGET	Indicates the type of output file to generate.	HLP (the valid values are HELP and HLP)
SCA_DEBUG_MESSAGES	Controls whether to enable debugging messages giving SCA status values.	OFF
ROUTINE_DEPTH	Controls whether nested routines are reported. For example, a routine_depth of 2 means to report on top-level routines plus the first level of nested routines.	1 (top-level routines only)
STATUS_MESSAGES	Controls whether to enable status messages.	ON
TRACEBACK_FLAG	Controls whether to enable TPU traceback messages during report execution.	ON
TRACE_MESSAGES	Controls whether to enable trace messages generated by calling *sca $report_trace_message*.	OFF

Option Name	Report Description	Default Value
USE_SOURCE_SPELLING	Controls whether the spelling of routine, variable, and type names are taken from the source file or are supplied by SCA.	ON

The options for PACKAGE reports are shown in Table 4.7.

Table 4.7. PACKAGE Report Options

Option name	Report Description	Default value
DOMAIN_QUERY	Name of query to be used as the domain.	null string
OUTPUT	The output file specification.	PACKAGE.LSE
TARGET	Indicates the type of output file to generate. For details, see the REPORT command description in the *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual*.	LSE (the valid values are LSE, VMSLSE, PLSE, PORTABLE, and LSEDIT as a synonym for LSE)
LANGUAGES	The list of languages in which the package definition should be available.	({language-name1}...)
HELP_LIBRARY	The file specification used as the help library.	nullstring
ROUTINE_DEPTH	Controls whether nested routines are reported. For example, a routine_depth of 2 means to report on top-level routines plus the first level of nested routines.	1 (top-level routines only)
SCA_DEBUG_MESSAGES	Controls whether to enable debugging messages giving SCA status values.	OFF
STATUS_MESSAGES	Controls whether to enable status messages.	ON
TRACEBACK_FLAG	Controls whether to enable TPU traceback messages during report execution.	ON
TRACE_MESSAGES	Controls whether to enable trace messages generated by calling *sca $report_trace_message*.	OFF
USE_SOURCE_SPELLING	Controls whether the spelling of routine, variable, and type names are taken from the source file or are supplied by SCA.	ON

Chapter 5. Customizing Reports

This chapter provides information on how to customize reports or create new reports.

The following topics are described:

- An introduction to customizing reports
- Organizing reports
- Processing reports
- Modifying a report source file
- Changing the default value of an option
- Modifying query expressions
- Adding new tags and keyword lists
- Modifying tag names
- Modifying section headers and other fixed text
- Deleting information from a report
- Customizing 2167A reports

5.1. Introduction

The SCA REPORT command uses TPU in conjunction with the SCA callable interface to generate reports. You can customize the reports or create your own reports (or both) by modifying the TPU source code. The TPU source files for the SCA command REPORT are located in the SYS$LIBRARY directory. For more information on TPU, see the *Guide to the DEC Text Processing Utility*.

The interface between TPU and SCA is built into LSE. Therefore, you must use LSE for access to the TPU/SCA interface.

5.2. How Reports are Organized

Source files for the reports are in the SYS$LIBRARY directory. Each file has a name in the form SCA$REPORT_module. TPU, with the exception of the file SCA$QUERY_CALLABLE.TPU. The top-level procedure for each report has a name in the form *sca$report_report_name.* All other routines for the reports have names of the form *sca$report_routine_name*.

The source files that you will customize most often are as follows:

- SCA$REPORT_INTERNALS.TPU—Implements the INTERNALS report.
- SCA$REPORT_2167A_DESIGN.TPU—Implements the 2167A report.
- SCA$REPORT_HELP.TPU—Implements the HELP report.
- SCA$REPORT_PACKAGE.TPU—Implements the PACKAGE report.

- SCA$REPORT_UTILITIES.TPU—Contains general-purpose routines. It is organized into several sections, with a table of contents at the beginning of the file.
- SCA$REPORT_CUSTOMIZATIONS.TPU—Contains variables and constants that provide easy report customization. It also contains comments about customizing reports. It is organized into several sections, with a table of contents at the beginning of the file.
- SCA$REPORT_PORTABLE_SYNTAX.TPU—Contains procedures used for processing the portable report commands and defining report options.

Each source file contains header comments summarizing the file's contents, and each procedure within the source file is commented.

5.3. Overview of Report Processing

Report processing consists of three major parts, as follows:

1. Issuing SCA queries
2. Storing query results
3. Generating report output on the basis of the query results

Query processing (parts 1 and 2) is defined by the report definition. Output generation (part 3) is done by the action routines specified in the report definition. A report definition specifies the following:

- What queries are issued for the report, in what order the queries are issued, and what the dependencies are between queries.
- What information is stored for each query result.
- What action routines are invoked to generate the report output. Query result information is passed to the action routines by means of result arrays.

The following are the steps in the main procedure for each of the standard reports:

1. Standard initialization (processes command options, creates buffers needed for report generation, and initializes variables)
2. Builds a report definition
3. Generates the report by processing the report definition
4. Standard cleanup (deletes report buffers, deletes the report definition, deletes result arrays, and so on)

The main procedure for a supplied report is named *sca$report_xxx*, where *xxx* is the report name. For example, the main procedure for the INTERNALS report is named *sca$report_internals*, and it is in the source file SYS$LIBRARY:SCA$REPORT_INTERNALS.TPU.

For each standard report, the report definition is created by the procedure *sca$report_build_definition_xxx*, where *xxx* is the report name. The procedures called by the build_definition procedure are defined in the source file SCA$REPORT_UTILITIES.TPU. Information on how to call each of these procedures is found in the procedure's header comments.

A set of utility routines is provided to fetch the results stored for each query. These *sca$report_fetch_** routines are also in SCA$REPORT_UTILITIES.TPU, with calling sequence information in the header comments of each procedure. For examples, see the action routines in SCA$REPORT_PACKAGE.TPU.

5.4. Modifying Report Source Files

The supplied source files for the REPORT code are written in the TPU language and reside in SYS$LIBRARY:SCA$REPORT_*.TPU. For some report customizations, you need to modify one or more of these source files. To modify a source file and then use it during report generation, perform the following steps:

1. Make a local copy of the source file.

```
$ COPY SYS$LIBRARY:SCA$REPORT_xxx.TPU   mydisk:[mydir]
```

2. Edit the local copy to make the desired customizations.

```
$ LSEDIT mydisk:[mydir]SCA$REPORT_xxx.TPU
```

3. Define a logical name to point to the modified file.

```
$ DEFINE SCA$REPORT_xxx mydisk:[mydir]SCA$REPORT_xxx
```

4. Generate a report from SCA.

```
$ SCA REPORT report-name
```

5.5. Changing the Default Value of an Option

Each report option is represented by a TPU variable with the name *sca$report_option_xxx*. The option name is *xxx*. For example, the TPU variable corresponding to the *overview_level*option is *sca$report_option_overview_level*. The option variables are defined and initialized to their default values in the option definition procedure for each report.

To change the default value for an option, modify theoption definition code in the option definition procedure. For example, if you do not want to have Body sections in your INTERNALS reports by default, you can change this option definition by modifying code in SCA$REPORT_INTERNALS.TPU. In the procedure *sca$report_define_options_internals*, change the following line:

```
sca$report_add_valid_option ('overview_level', -1, '');
```

The revised line is as follows:

```
sca$report_add_valid_option ('overview_level', 0, '');
```

In this example, you changed the default value of the overview_level option from –1 to 0.

To use the modified file, follow the procedure described in Section 5.4.

5.6. Modifying Query Expressions

The query expressions issued by the standard reports are defined as string constants in SCA$REPORT_CUSTOMIZATIONS.TPU. The constant names begin with *sca$report_k_query_*. Some query expressions are formed during report execution (for example, queries that include the name of the source file currently being processed). The constant portions of these query expressions are also defined in SCA$REPORT_CUSTOMIZATIONS.TPU.

To modify a query expression, edit the string constant definition for the query in SCA$REPORT_CUSTOMIZATIONS.TPU. For example, the standard PACKAGE report uses the

$REMARK comment text for a description of each routine defined. To use the text from a comment with the OVERVIEW tag, change the definition of the constant *sca$report_k_query_routine_remark* in SCA$REPORT_CUSTOMIZATIONS.TPU to the following:

```
sca$report_k_query_routine_remark :=
     'FIND CONTAINED_BY (@SCA$REPORT_CURRENT_ROUTINE,' +
     'SYMBOL=TAG AND "OVERVIEW", DEPTH=1, RESULT=BEGIN)'
```

5.7. Adding New Tags and Keyword Lists

LSE provides a set of standard tag definitions for each language. You can define additional tags by using 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). These tags are then available when compiling programs with the /DESIGN qualifier.

Note that it is the compiler and not SCA that checks the program to ensure that any keywords used are appropriate for a given keyword tag.

The following example shows how to label each module with a list of requirements supplied by the module:

```
SET COMMAND LANGUAGE VMSLSE
DEFINE TAG  REQUIREMENTS /LANGUAGE=ADA /TYPE=KEYWORD -
/KEYWORDS=Requirements_list
DEFINE KEYWORDS Requirements_list
    "AST reentrant"
    "Execution time under 1 millisecond"
    "Accepts dynamic strings"
END DEFINE
```

To save these definitions in an environment file, type the following commands:

```
LSE> SET COMMAND LANGUAGE LSE
LSE> SAVE ENVIRONMENT CHANGES MYDISK:[MYDIRECTORY]MYTAGS
```

The CHANGES option tells LSE to save only the new definitions that you 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 and the REQUIREMENTS keyword tag will be recognized in your source file. For example, your source file might contain the following lines:

```
-- Requirements:
--     AST Reentrant, accepts dynamic strings
```

5.8. Modifying Tag Names

The standard reports do special processing of certain tagged comments. For example, tagged comments that contain the routine description and descriptions of routine parameters are given special treatment for HELP and INTERNALS reports. The tag names for these specially processed comments are given by string constants in SCA$REPORT_CUSTOMIZATIONS.TPU. If you define new tag names,modify these string constants.

For example, if you want to use the RESULT tag to label the text description of the return value of a function, add the following tag definition to your LSE environment file:

```
DEFINE TAG "RESULT" /TYPE=TEXT /LANGUAGE=language-name
```

Change the definition of the constant *sca$report_routine_return_value* in SCA$REPORT_CUSTOMIZATIONS.TPU as follows:

```
sca$report_routine_return_value :=
'("RETURN VALUE" OR "ROUTINE VALUE" OR "FUNCTION VALUE" OR "RESULT")'
```

The DEFINE TAG command tells the compiler that RESULT is a valid tag name to be recorded in the .ANA file and loaded into the SCA library. Changing the constant definition in the report code modifies report processing to recognize the RESULT tag as receiving special handling for HELP and INTERNALS reports.

5.9. Modifying Section Headers and Other Fixed Text

The text for section headers and other fixed text used within reports is defined as string constants in SCA$REPORT_CUSTOMIZATIONS.TPU.

To modify one of these strings, edit the string constant definition in SCA$REPORT_CUSTOMIZATIONS.TPU. For example, to change the Intramodule variables heading to Module-wide variables, change the definition of the constant *sca$report_section_variables* in SCA$REPORT_CUSTOMIZATIONS.TPU as follows:

```
sca$report_section_variables    := 'Module-wide variables'
```

5.10. Deleting Information from a Report

You delete a definition entry by either commenting out the code or by marking it as an *ignore* entry. To comment out the code, add an exclamation point (!) at the beginning of each line. To mark the definition as an *ignore* entry, add the following code to the entry definition:

```
sca$report_definition_add_option (sca$report_k_option_ignore);
```

For example, if you do not want to have an Imported Types section in your INTERNALS reports, edit the file SCA$REPORT_INTERNALS.TPU to remove this section. In the procedure *sca$report_build_definition_internals*, you need to delete the definition labelled *imported types entry*. The code that defines this entry begins with the following lines:

```
!-------------------------------------------------------------------------!
  level 3 - imported types
!    attribute - type NAME, location object_name
!
sca$report_definition_add_entry (3, 2, 1, 1);
sca$report_definition_add_label ('imported types entry');
```

You can make this entry be ignored by adding the following code directly afterthe call to *sca$report_definition_add_label*:

```
sca$report_definition_add_option (sca$report_k_option_ignore);
```

5.11. Customizing 2167A Reports

Because the relationships between your program design or code and the DOD 2167A Software Design Document are dependent upon policies established at your site,you will probably need to customize the report.

In this section, two simple types of customizations are presented. The first example adds a section to a report. The next example shows how to generate the INPUT/OUTPUT section of a report automatically from data declarations in the code, instead of getting the information directly from tagged comments.

5.11.1. Adding a Section to a 2167A Report

The report definition for 2167A_DESIGN reports is given by the procedure *sca$report_build_definition_2167a_design*, in the file SCA$REPORT_2167A_DESIGN.TPU. This report definition contains a series of entries for component and unit information. Each entry specifies the tag that labels the information in your source files, and the title to be used for the corresponding section of the report. You might want to have additional report sections that correspond to other information in your source files. This section describes how to add such a report section, as follows:

1. To add a section called PERFORMANCE CONSIDERATIONS for units, 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
   ```

2. Add the definition entry for the new section in SCA$REPORT_2167A.TPU. If you want the new section to be added after all the other sections,add the definition entry after the one labelled UNIT REQ ENTRY. If you want it in a different position, add the new definition entry between the sections where you want the new section to appear.

   The definition entry appears as follows:

   ```
   !-------------------------------------------------------------------------
      !    level 3 - unit performance entry
      !    unit_perf query
      !    attribute - type TAG_NAME, location tag_name
      !    attribute - type TAG_TEXT, location tag_text
      !    attribute - type CONSTANT, location section_label
      !
      sca$report_definition_add_entry (3, 1, 3, 0);
      sca$report_definition_add_label ('unit performance entry');
      ! This dynamic query specifies that the name of the current file
      ! will be inserted at run-time.
   ```

```
      ! The variable sca$report_unit_file_name is filled in by the action
      ! routine for the units entry - it is the file name for the current
      ! unit.
      sca$report_add_query_dynamic (
          sca$report_k_query_2167a_unit_perf + '"',
          'sca$report_unit_file_name',
          '"');
      sca$report_definition_add_info (sca$report_k_info_type_tag_name,
                                      sca$report_location_tag_name);
      sca$report_definition_add_info (sca$report_k_info_type_tag_text,
                                      sca$report_location_tag_text);
      sca$report_definition_add_info (sca$report_k_info_type_constant,
                                      sca$report_location_section_label,
                                      'Performance considerations');
```

In the file SCA$REPORT_CUSTOMIZATIONS.TPU, add a query string for your new section, as follows:

```
CONSTANT
    sca$report_k_query_2167a_unit_perf :=
        'FIND SYMBOL=TAG AND "PERFORMANCE CONSIDERATIONS" AND FILE=';
```

You can also modify your LSE templates to make this tag available at an appropriate point in the comment header.

VSI recommends that you use distinct names for tags at the component and unit level. In this case, COMPONENT PERFORMANCE CONSIDERATIONS is an appropriate tag to use at the component level. Because the correct level for section headings is apparent from the context in the report output, it is not necessary to use section headings, although you can do so.

5.11.2. Using Program Code For Report Information

The 2167A design report is based solely on tagged comment text; it does not use your program structure. This section shows how to modify this report to get information about global variables from your program structure, rather than from a tagged comment.

The standard 2167A report gets information about a component's INPUT/OUTPUTDATA ELEMENTS (global variables) from tagged comments. To get this information from your program structure, you need to make several changes in the report definition entry labeled *component i/o entry*.

First, change the query used in this entry. The query that finds all the global variables in the file name .ext is as follows:

```
FIND SYMBOL=VARIABLE AND OCCURRENCE=DECLARATION AND DOMAIN=MULTI_MODULE -
    AND FILE="name.ext"
```

To use this query in your report, change the definition of the constant *sca$report_k_query_component_io* contained in the file SCA$REPORT_CUSTOMIZATIONS.TPU. The original definition is as follows:

```
sca$report_k_query_2167a_component_io :=
     'FIND SYMBOL=TAG AND "INPUT/OUTPUT DATA" AND FILE=',
```

Change it to the following:

```
sca$report_k_query_component_io :=
     'FIND SYMBOL=VARIABLE AND OCCURRENCE=DECLARATION AND -
         DOMAIN=MULTI_MODULE AND FILE=',
```

Next, change the entry to gather the name and type of each global variable by changing the calls to *sca $report_definition_add_info*, as follows:

```
sca$report_definition_add_info (sca$report_k_info_type_name,
       sca$report_location_object_name);
sca$report_definition_add_info (sca$report_k_info_type_variable_type,
       sca$report_location_object_type);
sca$report_definition_add_info (sca$report_k_info_type_source_location,
       sca$report_location_source_location);
```

You also need to add a subentry to get the description of each variable. Add an entity name to the *component i/o* entry to form the appropriate query to get the description text associated with each variable, as follows:

```
sca$report_definition_add_entity_name ('SCA$REPORT_CURRENT_OBJECT');
```

Add the subentry (modeled after similar entries in the INTERNALS report definition), as follows:

```
!------------------------------------------------------------------------
! level 3 - object description
!    attribute - type TAG_NAME, location tag_name
!    attribute - type TAG_TEXT, location tag_text
!
sca$report_definition_add_entry (3, 1, 2, 1);
sca$report_definition_add_option (sca$report_k_option_all_occurrences);
sca$report_definition_add_query (sca$report_k_query_object_description);
sca$report_definition_add_info (sca$report_k_info_type_tag_name,
    sca$report_location_tag_name);
sca$report_definition_add_info (sca$report_k_info_type_tag_text,
    sca$report_location_tag_text);
! Action routine for after query is processed - do comment filtering.
!sca$report_definition_add_action_routine (
    'sca$report_do_comment_filter',
    sca$report_k_invoke_after_all_entities);
```

The last change invokes a different action routine for the component i/o entry. Write and invoke an action routine that writes the variable name, type, and description with the necessary target-specific formatting information surrounding it. Model this action routine after the procedure *sca $report_do_internals_exported_variables* (contained in the SCA$REPORT_INTERNALS.TPU file), which writes this information for exported variables in the INTERNALS report. The commands are as follows:

```
! Invoke after all entities - write component section.
!
sca$report_definition_add_action_routine (
    'sca$report_do_io_section',
    sca$report_k_invoke_after_all_entities);
```

Your new definition entry for the *component i/o* section looks like the following:

```
!------------------------------------------------------------------------
!  level 2 - component i/o tag
!    component_io query
!    attribute - type NAME, location object_name
!    attribute - type VARIABLE_TYPE, location object_type
!    attribute - type SOURCE_LOCATION, location source_location
!    attribute - type CONSTANT, location section_label
!
```

```
sca$report_definition_add_entry (2, 1, 4, 2);
sca$report_definition_add_label ('component i/o entry');
! The variable sca$report_CSC_file_name is filled in by a CSC action
! routine - it is the file name for the current CSC.
sca$report_definition_add_query_dynamic (
    sca$report_k_query_component_io + '""',
    'sca$report_CSC_file_name',
    '"' );
sca$report_definition_add_info (sca$report_k_info_type_name,
                                sca$report_location_object_name);
sca$report_definition_add_info (sca$report_k_info_type_variable_type,
                                sca$report_location_object_type);
sca$report_definition_add_info (sca$report_k_info_type_source_location,
                                sca$report_location_source_location);
sca$report_definition_add_info (sca$report_k_info_type_constant,
                                sca$report_location_section_label,
                                'Input/output data');
sca$report_definition!add_entity_name ('SCA$REPORT_CURRENT_OBJECT');
! Invoke before query - write header for all component sections.
!
sca$report_definition_add_action_routine (
        'sca$report_start_component_sections',
        sca$report_k_invoke_before_query);
! Invoke after all entities - write component section.
!
sca$report_definition_add_action_routine (
        'sca$report_do_io_section',
        sca$report_k_invoke_after_all_entities);

!-----------------------------------------------------------------------
!  level 3 - object description
!    attribute - type TAG_NAME, location tag_name
!    attribute - type TAG_TEXT, location tag_text
!
sca$report_definition_add_entry (3, 1, 2, 1);
sca$report_definition_add_option (sca$report_k_option_all_occurrences);
sca$report_definition_add_query (sca$report_k_query_object_description);
sca$report_definition_add_info (sca$report_k_info_type_tag_name,
                                sca$report_location_tag_name);
sca$report_definition_add_info (sca$report_k_info_type_tag_text,
                                sca$report_location_tag_text);
! Action routine for after query is processed - do comment filtering.
!sca$report_definition_add_action_routine (
        'sca$report_do_comment_filter',
        sca$report_k_invoke_after_all_entities);
```