HP DECset for OpenVMS

Source Code Analyzer Command-Line Interface and Callable Routines Reference Manual

Order Number: AA–QJEYC–TK

July 2005

This reference manual provides information on using the HP Source Code Analyzer callable interface and the HP Source Code Analyzer Query Language.

Revision/Update information:	This is a revised manual.
Operating System Version:	OpenVMS I64 Version 8.2
	OpenVMS Alpha Version 7.3–2 or 8.2
	OpenVMS VAX Version 7.3
Windowing System Version:	DECwindows Motif for OpenVMS I64 Version 1.5
	DECwindows Motif for OpenVMS Alpha Version 1.3–1 or 1.5
	DECwindows Motif for OpenVMS VAX Version 1.2–6
Software Version:	HP DECset Version 12.7 for OpenVMS
	HP Language-Sensitive Editor/Source Code Analyzer Version 5.0

Hewlett-Packard Company
Palo Alto, California

Printed in the US

The HP OpenVMS documentation set is available on CD-ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Tables

Preface

This reference manual provides information about using the HP Source Code Analyzer (SCA) callable interface and the SCA Query Language.

Intended Audience

This manual is intended for experienced programmers and technical managers.

Document Structure

This reference contains the following chapters:

- Chapter 1 describes the SCA concepts and provides platform-specific information for using SCA on OpenVMS systems[1].
- Chapter 2 describes the SCA callable interface and provides information on message handling and rules for calling SCA routines. It also describes the SCA callable routines.
- Chapter 3 describes the features of the SCA Query Language and demonstrates its use for simple and advanced operations.
- Chapter 4 describes the rules governing the use of the SCA Query Language and provides information for evaluating query expressions.

Associated Documents

The following documents might be helpful when using SCA:

- *Guide to DIGITAL Source Code Analyzer for OpenVMS Systems*—Describes the HP Source Code Analyzer (SCA) and explains how to get started using its basic features.

[1] OpenVMS systems refers to OpenVMS VAX, OpenVMS Alpha and OpenVMS I64 systems.

- *HP DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual*—Provides a command dictionary for the LSE and SCA command language.

SCA is a component of the HP DECset tool kit. For more information on other HP DECset components, see the reference manuals for the individual components.

References to Other Products

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

Note

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

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

Conventions

Table 1 lists the conventions used in this manual.

Table 1 Conventions Used in This Reference

Convention	Description
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
Return	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, Return.
Ctrl/*x*	The 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.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Reference

Convention	Description
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.
boldface text	Boldface text represents the introduction of a new term.
`monospaced boldface text`	Boldface, monospace text represents user input in interactive examples.
italic text	Italic text represents book titles, parameters, arguments, attributes, 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.
mouse	The term **mouse** refers to any pointing device, such as a mouse, puck, or stylus.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Reference

Convention	Description
MB1,MB2,MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button.

1 Introduction

HP Source Code Analyzer (SCA) is a multilanguage, interactive cross-reference and static analysis tool. SCA helps you understand software projects by enabling you to make inquiries about the contents of the code. With SCA, you can quickly locate information about any program symbol and see the relationships to other symbols.

SCA stores compiler-generated information about a set of source files in an SCA library. You can perform various queries on the library and then, based on query results, navigate to locations of interest in the original source code.

1.1 Platform-Specific Information for Using SCA on OpenVMS

This section describes information about logical names, private SCA files, the command language syntax, and default settings for OpenVMS systems.

1.1.1 SCA$LIBRARY Logical Name

Whenever you set an SCA library from within SCA, it defines the logical name SCA$LIBRARY to refer to the library set. To retain the setting of your SCA library from one SCA invocation to another, leave this logical name pointing to your SCA library directory. Alternatively, you can set this logical name before an SCA invocation to make SCA point to an appropriate library automatically.

1.1.2 SCA Private Files

SCA creates private files that should not be modified or removed.

After you create an SCA library, you might notice the SCA$EVENT.DAT file in your directory.

Note

This file comprises your SCA library. Do not delete or modify this file. To delete an SCA library, use the SCA command DELETE LIBRARY.

1.1.3 Command Language Syntax

SCA has two command languages available: VMS and Portable. The VMS command language is the original SCA command language that has always been present in SCA, and has remained the most used. The Portable command language is a more recent command language devised for use in environments other than OpenVMS.

Some commands, such as REPORT, are available only in Portable, whereas others, like SPAWN and ATTACH, are available only in VMS. The choice of default command language is made at HP DECset installation time, but can always be changed.

This guide presents examples in the VMS command language setting.

Note

The online SCA Help provides either individual command descriptions or a list of all command descriptions.

To get help on a topic, use one of the following methods:

- At the SCA character-cell format's command line, type `HELP`, then enter the desired topic name.
- In the SCA HP DECwindows format, choose Enter Commands . . . from the Commands menu, then issue the following command :

```
SCA> HELP Command_Definitions
```

If you often use commands that can be invoked in only one command language, you might want to change the default command language setting. To set the default command language, enter one of the following commands at the SCA> prompt:

```
SCA> SET COMMAND LANGUAGE VMS
SCA> SET COMMAND LANGUAGE PORTABLE
```

1.1.4 Default Settings on OpenVMS Systems

The following settings are active by default on OpenVMS systems:

- OpenVMS-style, command language syntax
- Case-insensitive querying
- Hyphen-insensitive querying
- OpenVMS-style, wildcard-character searching
- Elimination of duplicate occurrences and symbols resulting from multiple inclusions of a single include file

2 Using the SCA Callable Interface

With the SCA callable interface, you can use SCA within independent application programs. You can integrate SCA into alternative user-interfaces and generate specialized reports based on SCA information.

The SCA callable interface provides a set of callable routines. You should have an understanding of basic SCA concepts before using these routines.

This chapter contains the following information:

- An overview of the SCA callable interface
- A description of message handling
- The rules for calling SCA routines
- A description of the callable command interface
- A description of the callable query interface
- The callable SCA routines

2.1 Overview

The SCA callable interface contains two components. The first is a set of callable command routines. The callable command routines comprise a high-level interface that you must always use, regardless of the type of application. This provides a very simple callable interface to SCA that is sufficient for most applications.

The second component is a set of callable query routines. The callable query routines comprise a lower-level interface to the FIND command. When you use this interface, your application has control over the specification of queries and the manipulation of query results.

2.2 Message Handling

The SCA callable interface handles all messages the same way—it signals them. If you want control over the display of such messages, you must establish a condition handler. Establishing a condition handler is optional.

2.3 Rules for Calling SCA Routines

Programs that call SCA routines must follow these rules:

- To free all dynamic memory associated with the interface routines and data structures, you must call SCA$CLEANUP to terminate callable SCA.
- Most SCA routines are not asynchronous system trap (AST) reentrant; therefore, you should not call an SCA routine (except SCA$ASYNCH_TERMINATE) from an AST routine that might currently be interrupting an SCA routine.
- Your program must not disable ASTs. If they are disabled in the user code prior to the SCA routine call, SCA will not behave as expected.
- If your program uses event flags, you must use the OpenVMS Run-Time Library (RTL) routines (LIB$RESERVE_EF, LIB$GET_EF, and LIB$FREE_EF) to coordinate the use of event flags between your program and SCA.
- Except for SCA$ASYNCH_TERMINATE, do not call SCA from within an SCA callback routine or from within a routine that is handling a condition signaled by SCA.
- Your program must not unwind when handling a condition signaled by SCA.

2.3.1 Rules for Calling SCA Routines from LSE/HP DECTPU

A subset of the SCA callable interface routines can be called from HP Text Processing Utility (HPTPU) code executed from within the HP Language-Sensitive Editor (LSE). The following list describes rules for calling SCA routines from HP DECTPU:

- You must use LSE to execute your HP DECTPU code. The SCA routines are not available from the version of HP DECTPU provided with OpenVMS software.

- LSE uses the HP DECTPU variable LSE$SCA_COMMAND_CONTEXT to store the command context created when initializing SCA. When you call the SCA routine SCA$QUERY_INITIALIZE, you must pass this variable as the command-context parameter. You should not change this variable.
- Only the following routines are available:
  - SCA$QUERY_CLEANUP
  - SCA$QUERY_COPY
  - SCA$QUERY_FIND
  - SCA$QUERY_GET_ATTRIBUTE
  - SCA$QUERY_GET_ATTRI_KIND_T
  - SCA$QUERY_GET_ATTRI_VALUE_T
  - SCA$QUERY_GET_DESCRIPTION
  - SCA$QUERY_GET_OCCURRENCE
  - SCA$QUERY_GET_NAME
  - SCA$QUERY_INITIALIZE
  - SCA$QUERY_PARSE
  - SCA$QUERY_SELECT_OCCURRENCE

  Note that you cannot call SCA$INITIALIZE or SCA$CLEANUP from HP DECTPU. LSE will handle the calls to the following routines for you automatically.
- SCA query contexts and entity handles are represented as HP DECTPU integers.
- SCA attribute constants, of the form SCA$K_ATTRI_attribute_name, are defined as TPU constants in the following file:
  - SYS$LIBRARY:SCA$QUERY_CALLABLE.TPU

  You must read and execute this file before using any of the SCA$K_ATTRI_attribute_name constants.

2.4 Callable Command Interface

The callable command routines comprise the high-level interface that you must always use, regardless of the type of application. This provides a simple callable interface to SCA that is sufficient for most applications.

Use the SCA$INITIALIZE routine to initialize SCA. You must call this routine before any other SCA routine. SCA$INITIALIZE creates a **command-context**. A command-context is represented as a longword-sized value that SCA$INITIALIZE returns to the application. This command-context value must be passed to all of the other command routines, and to the SCA$QUERY_INITIALIZE routine.

Use the SCA$CLEANUP routine to terminate a command-context. It frees the SCA internal data structures and severs all connections to SCA libraries and servers.

Typically, an application has only one command-context at a time. However, you can call SCA$INITIALIZE more than once without calling SCA$CLEANUP. Each call to SCA$INITIALIZE creates a separate command-context. Each separate command-context behaves independently from all other command-contexts. With more than one command-context, an application can query two different virtual libraries at the same time without having to reset the virtual library. Note that process quotas might place limits on the number of command-contexts you can run simultaneously.

For each command-context created by a call to SCA$INITIALIZE, there must be a call to SCA$CLEANUP in order to properly shut down that context. The process invoking SCA will not terminate normally until such a call to clean up the context is made.

Use the SCA$DO_COMMAND routine to execute SCA commands. For all commands except FIND, SCA$DO_COMMAND is the only available routine for executing an SCA command. It gives the application control over the character-cell user interface of any SCA command.

In addition, there are two routines, SCA$LOCK_LIBRARY and SCA$UNLOCK_LIBRARY, for creating and deleting write-locks on current virtual library lists. With the SCA$LOCK_LIBRARY routine, you can lock the virtual library so it cannot be modified between successive calls to the SCA$DO_COMMAND or SCA$QUERY_FIND routines.

Example of the SCA Callable Interface

The following is an example of an application routine using the SCA callable command interface. This example shows a simple routine that automatically loads .ANA files following compilation.

```
ROUTINE load_ana_file( ana_file_spec : _string ) =
!++
! FUNCTIONAL DESCRIPTION:
!
!   LOAD_ANA_FILE loads the .ANA file specified as ANA_FILE_SPEC
!   into the current SCA library.
!
! MACROS:
!
!   _COPY_STRING concatenates several strings into a single
!       destination string.  It is similar to STR$CONCAT.
!   _DYNAMIC_STRING declares and initializes a dynamic string descriptor.
!   _FREE_STRING releases the memory described by a dynamic string
!        descriptor.
!   _STRING declares a string descriptor, but does not initialize it.
!--
    BEGIN
    LOCAL
        cmdline : _dynamic_string
        sca_context;

    ! Initialize SCA.
    !
    sca$initialize( sca_context );

    ! Load the .ANA file.
    !
    _copy_string( cmdline, 'LOAD ', .ana_file_spec )
    sca$do_command( sca_context, cmdline );

    ! Free the memory associated with the dynamic string descriptor.
    !
    _free_string( cmdline );

    ! Cleanup SCA.
    !
    sca$cleanup( sca_context )
    END;
```

2.5 Callable Query Interface

The callable query routines comprise the second component of the SCA callable interface. This component is a lower-level interface to the FIND command. Using this interface, an application has control over the specification of queries and the manipulation of query results.

As with the callable command interface, you begin using the callable query interface by entering an SCA$QUERY_INITIALIZE routine. However, instead of creating a command-context, this routine initializes a **query-context**. As with the callable command interface, SCA$QUERY_INITIALIZE must be called before any other query routine. A query-context is represented as a longword-sized value that SCA$QUERY_INITIALIZE returns to the application. This query-context value must be passed to many of the other query routines.

Similarly, you use the SCA$QUERY_CLEANUP routine to end a query-context using SCA. It frees the internal data structures that represent the query.

Typically, an application uses several query-contexts at the same time. Each call to SCA$QUERY_INITIALIZE creates a separate query-context.

In SCA, the term **query** refers to both a question and its corresponding result. There is a one-to-one correspondence between a query and a query-context. A query-context first expresses a question. The SCA$QUERY_FIND routine then evaluates the query. This creates a **query result**. Finally, the result is analyzed. You can then enter the SCA$QUERY_SELECT_OCCURRENCE routine to use the results of one query in forming the question of another query.

Definitions required to use the SCA callable query interface are stored in files called SYS$LIBRARY:SCA$QUERY_CALLABLE.*. Each language has its own definition file. For example, the Pascal definition file is called SYS$LIBRARY:SCA$QUERY_CALLABLE.PAS.

2.5.1 Data Models

The callable query interface uses an entity-relationship-attribute data model. Entities are atomic elements of the database. Two entities can have a relationship. A relationship has associated with it two entities—one is the source, the other is the target. Both entities and relationships have attributes.

In the SCA callable query interface, the only kind of entity that is presented is **occurrence**. These occurrences have many attributes; for example, an occurrence can have a **name**.

2.5.2 Handles

The callable query interface uses **handles** to refer to entities and attributes. Handles are generic. You can use a particular handle to refer to an entity or an attribute, or with any query-context.

A handle is a longword-value. Before using a handle for the first time, it must be set to zero. Thereafter, you can use it in a routine that requires a handle as an output parameter, such as SCA$QUERY_GET_OCCURRENCE. After that you can use it either as an output or input parameter.

After you use a handle to refer to something, it becomes dependent on the query-context that contains its referent. If a query-context is deleted, any handles associated with that query-context become invalid. An invalid handle must be reinitialized before you use it again; that is, it must be set to zero. Any use of a query-context as an output parameter also invalidates any associated handles. This includes any question-building routine, such as SCA$QUERY_PARSE and SCA$QUERY_FIND.

SCA attempts to detect the use of an invalid handle and signal an error message. Such use is considered a programmer error on the part of the application.

2.5.3 Entities

Entities are atomic elements of the database and have attributes. In SCA, entities are occurrences of symbols. For example, if a routine declares a local variable STATUS and uses it in calls to other routines, the declaration as well as the uses of this variable are distinct occurrences of a symbol named STATUS. Each such occurrence is an entity.

An entity is referred to with a handle. A handle that refers to an entity is called an **entity handle**. You can use the SCA$QUERY_GET_OCCURRENCE routine to visit all of the entities in a particular query result.

2.5.4 Attributes

Entities have attributes that provide information about the entity. You use attributes to do different things, such as express questions and retrieve results.

The SCA callable query interface supports a number of different kinds of attributes. Each **attribute-kind** has associated with it an attribute-value of a particular data type. All attribute-values are returned as character strings.

The following is the list of attribute-kinds:

- SCA$K_ATTRI_APPEARANCE
- SCA$K_ATTRI_DOMAIN
- SCA$K_ATTRI_EXPRESSION
- SCA$K_ATTRI_LANGUAGE
- SCA$K_ATTRI_NAME
- SCA$K_ATTRI_OCCURRENCE_CLASS
- SCA$K_ATTRI_SYMBOL_CLASS
- SCA$K_ATTRI_MACH_DSIZE
- SCA$K_ATTRI_MACH_DTYPE
- SCA$K_ATTRI_PASSING_MECHANISM
- SCA$K_ATTRI_FILE_SPEC
- SCA$K_ATTRI_BEGIN_RECORD_NUMBER
- SCA$K_ATTRI_NAME_RECORD_NUMBER
- SCA$K_ATTRI_BEXE_RECORD_NUMBER
- SCA$K_ATTRI_END_RECORD_NUMBER
- SCA$K_ATTRI_BEGIN_CHAR_OFFSET
- SCA$K_ATTRI_NAME_CHAR_OFFSET
- SCA$K_ATTRI_BEXE_CHAR_OFFSET
- SCA$K_ATTRI_END_CHAR_OFFSET
- SCA$K_ATTRI_ALL

You can use the SCA$K_ATTRI_ALL attribute-kind in calls to SCA$QUERY_GET_ATTRIBUTE to find all attributes, regardless of kind.

Some attribute-kinds correspond to attribute-names in the Query Language that have similar names. For example, the SCA$K_ATTRI_NAME attribute-kind corresponds to the NAME attribute-name. Other attribute-kinds have a different representation at the command line, and some attribute-kinds, such as SCA$K_ATTRI_PASSING_MECHANISM, have no equivalent in the Query Language.

Attribute Retrieval

Attribute-retrieval is the process of getting attributes of entities in a query result. Attributes are retrieved using the SCA$QUERY_GET_ATTRIBUTE and SCA$QUERY_GET_ATTRI_VALUE_T routines.

There are two ways to use these routines. The first way is to use the get-attribute-value routine to find the value of a particular attribute-kind for an entity. For example, if OCC is a handle to an occurrence whose name you want to retrieve into the string-descriptor NAME, use the following call:

```
sca$query_get_attri_value_t( occ, name, %REF(sca$k_attri_name) );
```

The string-descriptor NAME now contains the name of the occurrence, for example, XYZ.

You retrieve the value of any attribute as a character-string. For example, you can use the following call to retrieve a string representation of the symbol-class in the string-descriptor SYMBOL_CLASS:

```
sca$query_get_attri_value_t( occ,
                             symbol_class,
                             %REF(sca$k_attri_symbol_class) );
```

The string-descriptor SYMBOL_CLASS now describes the character string VARIABLE.

The second way is to use the SCA$QUERY_GET_ATTRIBUTE routine to get more than one attribute for an entity, by returning a handle to an attribute. You can then use that handle in calls to SCA$QUERY_GET_ATTRI_VALUE_T to get the value of the attribute.

In addition, with the SCA$QUERY_GET_ATTRI_KIND_T routine, you can get the **kind** of the attribute. The following example shows how to use this routine, and highlights the manipulation of attributes:

```
ROUTINE dump_entity( entity ) =
!++
! FUNCTIONAL DESCRIPTION
!
!       Display all of the attributes of an entity.
!
! FORMAL PARAMETERS:
!
!   entity
!       The address of an SCA handle describing the entities whose
!       attributes are to be displayed.
!
! MACROS:
!
!   _COPY_STRING concatenates several strings into a single
!       destination string.  It is similar to STR$CONCAT.
!   _DYNAMIC_STRING declares and initializes a dynamic string descriptor.
!   _FREE_STRING releases the memory described by a dynamic string
!        descriptor.
!--
    BEGIN
    LOCAL
        attribute : INITIAL(0),             ! handle for attributes
        display_line : _dynamic_string,     ! one line of display output
        iteration_ctx : INITIAL(0),         ! iteration-context
        kind : _dynamic_string,             ! attribute-kind as a string
        value: _dynamic_string;             ! attribute-value as a string

    ! Loop once for every attribute of the specified entity.
    !
    WHILE sca$query_get_attribute( .entity,
                                   %REF(sca$k_attri_all),
                                   attribute,
                                   iteration_ctx ) DO
        BEGIN
        ! Get the attribute-kind and the attribute-value.
        !
        sca$query_get_attri_kind_t( attribute, kind );
        sca$query_get_attri_value_t( attribute, value );

        ! Create a line of output (e.g., "NAME=FOO") and write it out.
        !
        _copy_string( display_line, kind, '=', value );
        lib$put_output( display_line );
        END;

    _free_string( display_line, kind, value )
    END;
```

The possible attributes are as follows:

- **Appearance**—Specify the **occurrence-appearance** attribute with the SCA$K_ATTRI_APPEARANCE attribute-kind. The **occurrence-appearance** attribute indicates that the name of the symbol appears in the source code, or that it was hidden.

  The **occurrence-appearance** attribute corresponds to portions of the **occurrence-class** attribute in the Query Language. In particular, it corresponds to the attribute-selection expressions occ=visible and occ=hidden.

- **Symbol Domain**— Specify the **symbol-domain** attribute with the SCA$K_ATTRI_DOMAIN attribute-kind. The **symbol-domain** attribute indicates the range of source code in which a symbol has the potential of being used.

  The **symbol-domain** attribute corresponds to the domain=symbol-domain-value attribute-selection expression in the Query Language.

- **Expression**— Specify the **occurrence-expression** attribute with the SCA$K_ATTRI_EXPRESSION attribute-kind. The **occurrence-expression** attribute indicates that the occurrence explicitly appears in the source file, or that the occurrence was implicit. For example, in Fortran, you can implicitly declare a variable. As a Fortran programmer, you might think that the variable is not declared. The SCA view is that it is declared, but it is declared implicitly.

  The **occurrence-expression** attribute corresponds to the portions of the **occurrence-class** attribute in the Query Language. In particular, it corresponds to the attribute-selection expressions occ=explicit and occ=implicit.

- **Language**— Specify the **language** attribute with the SCA$K_ATTRI_LANGUAGE attribute-kind. The **language** attribute indicates the programming language of the occurrence.

  The **language** attribute does not correspond to anything in the Query Language.

- **Name—** Specify the **name** attribute with the SCA$K_ATTRI_NAME attribute-kind. The **name** attribute indicates the name of the symbol.
- **Occurrence Class—** Specify the **occurrence-class** attribute with the SCA$K_ATTRI_OCCURRENCE_CLASS attribute-kind. The **occurrence-class** attribute indicates the kind of occurrence.

  The **occurrence-class** attribute corresponds to the occurrence=occurrence-class-value attribute-selection expression.
- **Symbol Class—** Specify the **symbol-class** attribute with the SCA$K_ATTRI_SYMBOL_CLASS attribute-kind. The **symbol-class** attribute indicates the kind of symbol.

  The **symbol-class** attribute corresponds to the symbol=symbol-class-value attribute-selection expression.
- **Passing Mechanism—** Specify the **passing-mechanism** attribute with the SCA$K_ATTRI_PASSING_MECHANISM attribute-kind. The **passing-mechanism** attribute applies only to argument declarations. It indicates the mechanism by which an argument is passed to a routine.

  The **passing-mechanism** attribute does not correspond to anything in the Query Language.
- **File Specification—** Specify the **file-specification** attribute with the SCA$K_ATTRI_FILE_SPEC attribute-kind. The **file-specification** attribute describes the file specification of the source file that contains the occurrence.

  The **file-specification** attribute does not correspond to anything in the Query Language.

- **Begin Record Number—** Specify the **begin-record-number** attribute with the SCA$K_ATTRI_BEGIN_RECORD_NUMBER attribute-kind. The **begin-record-number** describes the source-file record number at the beginning of the lexical range of a declaration. It is not applicable to references, which are assumed to have a single lexical location.

  The **begin-record-number** attribute does not correspond to anything in the Query Language.

- **Name Record Number—** Specify the **name-record-number** attribute with the SCA$K_ATTRI_NAME_RECORD_NUMBER attribute-kind. The **name-record-number** attribute describes the source-file record number where the name of the symbol appears.

  The **name-record-number** attribute does not correspond to anything in the Query Language.

- **Begin Executable Record Number—** Specify the **begin-executable-record-number** attribute with the SCA$K_ATTRI_BEXE_RECORD_NUMBER attribute-kind. The **begin-executable-record-number** attribute describes the source-file record number where the executable part of a routine begins. This attribute is primarily applicable to primary routine declarations.

  The **begin-executable-record-number** attribute does not correspond to anything in the Query Language.

- **End Record Number—** Specify the **end-record-number** attribute with the SCA$K_ATTRI_END_RECORD_NUMBER attribute-kind. The **end-record-number** attribute describes the source-file record number at the end of the lexical range of a declaration. It is not applicable to references, which are assumed to have a single lexical location.

  The **end-record-number** attribute does not correspond to anything in the Query Language.

- **Begin Character Offset**— Specify the **begin-character-offset** attribute with the SCA$K_ATTRI_BEGIN_CHAR_OFFSET attribute-kind. The **begin-character-offset** attribute describes the source-file character offset at the beginning of the lexical range of a declaration. It is not applicable to references, which are assumed to have a single lexical location.

  The **begin-character-offset** attribute does not correspond to anything in the Query Language.

- **Name Character Offset**— Specify the **name-character-offset** attribute with the SCA$K_ATTRI_NAME_CHAR_OFFSET attribute-kind. The **name-character-offset** attribute describes the source-file character offset where the name of the symbol appears.

  The **name-character-offset** attribute does not correspond to anything in the Query Language.

- **Begin Executable Character Offset**— Specify the **begin-executable-character-offset** attribute with the SCA$K_ATTRI_BEXE_CHARACTER_OFFSET attribute-kind. The **begin-executable-character-offset** attribute describes the source-file character offset where the executable part of a routine begins. This attribute is primarily applicable to primary routine declarations.

  The **begin-executable-character-offset** attribute does not correspond to anything in the Query Language.

- **End Character Offset**— Specify the **end-character-offset** attribute with the SCA$K_ATTRI_END_CHAR_OFFSET attribute-kind. The **end-character-offset** attribute describes the source-file character offset of the end of the lexical range of a declaration. It is not applicable to references, which are assumed to have a single lexical location.

  The **end-character-offset** attribute does not correspond to anything in the Query Language.

2.5.5 Example of the Callable Query Interface

The following example is an application using the callable query interface. It shows the building of questions, their evaluation, and the use of the query results.

```
ROUTINE display_tags_of_decl( sca_cmd_ctx, decl_name : _string ) =
!++
! FUNCTIONAL DESCRIPTION
!
!       Display info about all the tags associated with primary
!       declarations with the specified name.
!
! FORMAL PARAMETERS:
!
!   sca_cmd_ctx
!           An SCA command-context.  It has already been initialized.
!           The SCA library has already been set.
!
!   decl_name
!           Name of the declaration.
!
! MACROS:
!
!   _COPY_STRING concatenates several strings into a single
!       destination string.  It is similar to STR$CONCAT.
!   _DYNAMIC_STRING declares and initializes a dynamic string descriptor.
!   _FREE_STRING releases the memory described by a dynamic string
!        descriptor.
!   _STRING declares a string descriptor, but does not initialize it.
!--
    BEGIN
    LOCAL
        all_tags_query,             ! query context for "SYMBOL=TAG"
        cmdline : _dynamic_string,  ! temp for holding FIND command line
        decl : INITIAL(0),          ! handle for the named declarations
        decl_query,                 ! query context for the named decls
        particular_decl_query,      ! query context for particular decl
        tag : INITIAL(0),           ! handle for tag declarations
        tag_name : _dynamic_string, ! holds the tag name strings
        tag_query;                  ! query context for tags

    ! Initialize the necessary query contexts.
    !
    sca$query_initialize( .sca_cmd_ctx, all_tags_query );
    sca$query_initialize( .sca_cmd_ctx, decl_query );
    sca$query_initialize( .sca_cmd_ctx, particular_decl_query );
    sca$query_initialize( .sca_cmd_ctx, tag_query );

    ! Lock the virtual library so that it cannot change out from under
    ! us.  If this is not done, there is a chance that the call to
    ! SCA$QUERY_SELECT_OCCURRENCE will fail, since the specified
    ! occurrence is made obsolete if its module is updated.
    !
    sca$lock_library( .sca_cmd_ctx );
```

```
! Form the question: all of the primary declarations with the
! specified name.
!
_copy_string( cmdline, 'occ=primary AND ', .decl_name );
sca$query_parse( decl_query, cmdline );
_free_string( cmdline );

! Evaluate that question.
!
sca$query_find( decl_query );

! Form the question: SYMBOL=TAG.
!
sca$query_parse( all_tags_query, _ascid( 'symbol=tag' ) );

! Now we visit each of the primary declarations described by DECL_QUERY.
!
WHILE sca$query_get_occurrence( decl_query, decl ) DO
    BEGIN
    ! So we have our hands on a named-declaration.  Now we want
    ! to find its corresponding tag declarations.  Begin by
    ! creating a query question that describes the name-decl.
    !
    sca$query_select_occurrence( particular_decl_query, decl );

    ! Then we get the name of that query.
    !
    sca$query_get_name( particular_decl_query,
                        particular_decl_query_name );

    ! Form the question:
    !   CONTAINING( @name-of-decl-query, SYMBOL=TAG, RESULT=BEGIN )
    !
    _copy_string(
        cmdline,
        'CONTAINING( BEGIN=SYMBOL=TAG, RESULT=BEGIN, END=@',
        .particular_decl_query_name );

    sca$query_parse( tag_query, cmdline );

    ! Evaluate that query.
    !
    sca$query_find( tag_query );
    ! Visit each of the tags
    !
    WHILE sca$query_get_occurrence( tag_query, tag ) DO
        BEGIN
        ! Now that we've finally got our hands on a tag declaration,
        ! let's check out its name.
        !
        sca$query_get_attri_value_t( tag,
                                     tag_name,
                                     %REF(sca$k_attri_name) );
        !
        ! TAG_NAME is now a string descriptor describing the
        ! name of the tag.

        [~etcetera~]
        END
    END;

_free_string( cmdline, tag_name, tag_query_name );

sca$unlock_library( .sca_cmd_ctx );

sca$query_cleanup( .sca_cmd_ctx, all_tags_query );
sca$query_cleanup( .sca_cmd_ctx, decl_query );
sca$query_cleanup( .sca_cmd_ctx, particular_decl_query );
sca$query_cleanup( .sca_cmd_ctx, tag_query );
END;
```

2.6 Callable SCA Routines

The callable SCA routines are described in this section. The callable SCA routines are divided into the following categories:

- Callable command-interface routines
- Callable query initialization and cleanup routines
- Callable query question-building routines
- Callable query result-manipulation routines
- Callable query miscellaneous routines

2.6.1 Callable Command Interface Routines

The following are callable command interface routines:

- SCA$ASYNCH_TERMINATE
- SCA$CLEANUP
- SCA$DO_COMMAND
- SCA$GET_INPUT

- SCA$INITIALIZE
- SCA$LOCK_LIBRARY
- SCA$PUT_OUTPUT
- SCA$UNLOCK_LIBRARY

2.6.2 Callable Query Initialization/Cleanup Routines

The following are callable query initialization and cleanup routines:

- SCA$QUERY_CLEANUP
- SCA$QUERY_INITIALIZE

2.6.3 Callable Query Question-Building Routines

The following are callable query question-building routines:

- SCA$QUERY_PARSE
- SCA$QUERY_SELECT_OCCURRENCE
- SCA$SELECT_OCCURRENCE

2.6.4 Callable Query Result Manipulation Routines

The following are callable query result manipulation routines:

- SCA$GET_ATTRIBUTE
- SCA$GET_ATTRI_KIND_T
- SCA$GET_ATTRI_VALUE_T
- SCA$GET_OCCURRENCE
- SCA$QUERY_GET_ATTRIBUTE
- SCA$QUERY_GET_ATTRI_KIND_T
- SCA$QUERY_GET_ATTRI_VALUE_T
- SCA$QUERY_GET_OCCURRENCE

2.6.5 Callable Query Miscellaneous Routines

The following are callable query miscellaneous routines:

- SCA$GET_CURRENT_QUERY
- SCA$QUERY_COPY
- SCA$QUERY_FIND
- SCA$QUERY_GET_NAME

SCA$ASYNCH_TERMINATE

Sets a flag indicating that a Ctrl/C has been issued.

Format

SCA$ASYNCH_TERMINATE   command_context

Argument

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context value

Condition Value Returned

SCA$_NORMAL	Normal successful completion

Description

The SCA$ASYNCH_TERMINATE routine sets a flag indicating that a Ctrl/C has been issued.

SCA$CLEANUP

Shuts down the SCA callable command interface, which frees all dynamic memory associated with the interface routines and data structures.

Format

SCA$CLEANUP   command_context

Argument

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context value

Condition Value Returned

SCA$_NORMAL	The SCA callable command interface has been successfully shut down.

Description

The SCA$CLEANUP routine shuts down the SCA callable command interface, which frees all dynamic memory associated with the interface routines and data structures.

SCA$DO_COMMAND

Parses an SCA subsystem command and invokes command processing, if the command is syntactically correct.

Format

```
SCA$DO_COMMAND   command_context,
                 command_string
                 [,parameter_routine]
                 [,continuation_routine]
                 [,continuation_prompt]
                 [,user_argument]
                 [,confirm_routine]
                 [,topic_routine]
                 [,display_routine]
```

Arguments

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context value.

command_string

type:	character string
access:	read-only
mechanism:	by descriptor

SCA subsystem command

parameter_routine

type:	procedure
access:	read-only
mechanism:	by reference

Routine that prompts for required parameters. You can specify SCA$GET_INPUT or a compatible routine. If this parameter is omitted or a routine address of 0 is specified, commands with missing parameters fail and display a command-line interface (CLI) error message.

continuation_routine

type:	procedure
access:	read-only
mechanism:	by reference

Routine that prompts for the remainder of a continued command (that is, a command that ends with a hyphen). You can specify SCA$GET_INPUT or a compatible routine. If this parameter is omitted or a routine address of 0 is specified, no continuation prompting is performed.

continuation_prompt

type:	character string
access:	read-only
mechanism:	by descriptor

Command continuation prompt string (for example, SCA>). This parameter must be specified if the **continuation_routine** parameter is specified.

user_argument

type:	longword
access:	read-only
mechanism:	by reference

User-specified value to be passed to any action routine (other than CLI prompt routines) called by this routine.

confirm_routine

type:	procedure
access:	read-only
mechanism:	by value

Command confirmation prompt routine to be used by commands supporting a /CONFIRM qualifier. You can specify SCA$GET_INPUT or a compatible routine. If this argument is omitted, the /CONFIRM qualifier is not supported.

topic_routine

type:	procedure
access:	read-only
mechanism:	by value

Help topic prompt routine. You can specify SCA$GET_INPUT or a compatible routine. If this routine returns an error, command processing is terminated. If this argument is omitted, no help prompting is performed.

display_routine

type:	procedure
access:	read-only
mechanism:	by value

Routine to be called to display one line of command output. You can specify SCA$PUT_OUTPUT or a compatible routine. If this routine returns an error, command processing is terminated. If this argument is omitted, no display routine is called.

Condition Values Returned

All SCA condition values and
many system values.

Description

The SCA$DO_COMMAND routine parses an SCA subsystem command and invokes command processing, if the command is syntactically correct.

SCA$GET_ATTRIBUTE

Gets a handle to an attribute of an entity.

Format

```
SCA$GET_ATTRIBUTE   entity,
                    attribute_kind,
                    attribute_handle,
                    [,iteration_context]
```

Arguments

entity

type:	$SCA_HANDLE
access:	read-only
mechanism:	by reference

SCA entity handle describing the entity or relationship whose attributes are being obtained

attribute_kind

type:	$SCA_ATTRIBUTE_KIND
access:	read-only
mechanism:	by reference

Kind of attribute to be obtained

You can specify any attribute-kind with this routine.

attribute_handle

type:	$SCA_HANDLE
access:	write-only
mechanism:	by reference

SCA attribute handle to describe the obtained attribute

iteration_context

type:	$SCA_ITERATION_CONTEXT
access:	read/write
mechanism:	by reference

Iteration-context. This longword must contain 0 on the first call to this routine for a particular iteration. This routine uses the longword to maintain the iteration context. The caller must not change the contents of the longword.

Condition Values Returned

SCA$_NORMAL	An attribute has been successfully returned.
SCA$_NONE	Warning. An attribute has not been returned. Either there are no such attributes in the entity, or there are no more attributes.

Description

The SCA$GET_ATTRIBUTE routine gets a handle to an attribute of an entity.

If the **iteration_context** parameter is not specified, this routine finds the first attribute of the specified kind **(attribute_kind)** and updates **attribute_handle** to describe that attribute.

In general, several attributes can be associated with a particular entity. With this routine, you can find all of those attributes by using the **iteration_context** parameter.

SCA$GET_ATTRI_KIND_T

Gets an attribute kind.

Format

```
SCA$GET_ATTRI_KIND_T   attribute_handle,
                       attribute_kind
```

Arguments

attribute_handle

type:	$SCA_HANDLE
access:	read-only
mechanism:	by reference

SCA handle describing an attribute whose attribute-kind is to be obtained.

attribute_kind

type:	character string
access:	write-only
mechanism:	by descriptor

Kind of attribute.

Condition Value Returned

SCA$_NORMAL	An attribute kind has been successfully returned.

Description

The SCA$GET_ATTRI_KIND_T routine returns the kind of any attribute as a character string.

SCA$GET_ATTRI_VALUE_T

Gets an attribute value.

Format

```
SCA$GET_ATTRI_VALUE_T   handle,
                        attribute_value
                        [,attribute_kind]
```

Arguments

handle

type:	$SCA_HANDLE
access:	read/write
mechanism:	by reference

SCA attribute handle describing either an attribute or an entity whose value is to be obtained.

attribute_value

type:	character string
access:	read/write
mechanism:	by descriptor

Value of the attribute being selected.

attribute_kind

type:	$SCA_ATTRIBUTE_KIND
access:	read/write
mechanism:	by reference

Kind of attribute to be obtained.

Condition Values Returned

SCA$_NORMAL	An attribute value has been successfully returned.
SCA$_NONE	Warning. An attribute-value has not been returned. There are no such attributes in the entity. This condition can be returned only if this routine is processing an entity.

Description

The SCA$GET_ATTRI_VALUE_T routine returns the value of any attribute as a character string.

If the handle describes an attribute, this routine returns the value of that attribute. In this case, the **attribute_kind** parameter must not be specified.

If the handle describes an entity, this routine returns the value of the first attribute of that entity that is of the kind specified by the **attribute_kind** parameter. In this case, the **attribute_kind** parameter must be specified.

If you want to get more than one attribute value of a particular kind for an entity, use the routine SCA$GET_ATTRIBUTE. This applies only to the attribute-kinds SCA$K_ATTRI_NAME and SCA$K_ATTRI_ALL.

The value of any kind of attribute can be returned by this routine, except for SCA$K_ATTRI_ALL. This routine will convert to character string those attributes whose data type is not character string.

This routine does not accept the attribute-kind SCA$K_ATTRI_ALL as the value of the **attribute_kind** parameter. It is not meaningful to get just the first attribute without regard to attribute-kind.

SCA$GET_CURRENT_QUERY

Gets the name of the current query in the given command-context.

Format

```
SCA$GET_CURRENT_QUERY   command_context,
                        query_name
```

Arguments

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context.

query_name

type:	character string
access:	write-only
mechanism:	by descriptor

Name of the current query in the context of the given command-context.

Condition Value Returned

SCA$_NORMAL	The name of the current query has been successfully retrieved.

Description

The SCA$GET_CURRENT_QUERY routine gets the name of the current query in the given command-context.

SCA$GET_INPUT

Gets one record of ASCII text from the current controlling input device specified by SYS$INPUT.

Format

```
SCA$GET_INPUT    get_string,
                [,prompt_string]
                [,output_length]
                [,user_argument]
```

Arguments

get_string

type:	character string
access:	write-only
mechanism:	by descriptor

Buffer to receive the line read from SYS$INPUT. The string is returned by a call to STR$COPY_DX.

prompt_string

type:	character string
access:	read-only
mechanism:	by descriptor

Prompt message displayed on the controlling terminal. A valid prompt consists of text followed by a colon (:), a space, and a no carriage-return and line-feed combination. The maximum size of the prompt message is 255 characters. If the controlling input device is not a terminal, this argument is ignored.

output_length

type:	word
access:	write-only
mechanism:	by reference

Word to receive the actual length of the GET-STRING line, not counting any padding in the case of a fixed string. If the input line is truncated, this length reflects the truncated string.

user_argument

type:	_UNSPECIFIED
access:	read-only
mechanism:	by reference

User-specified value passed to the routine calling this action routine.

Condition Values Returned

SCA$_NORMAL	An input line was returned.
Failure completion code from LIB$GET_INPUT	

Description

The SCA$GET_INPUT routine gets one record of ASCII text from the current controlling input device specified by SYS$INPUT.

SCA$GET_OCCURRENCE

Returns an occurrence from the query specified by the **query_name** argument.

Format

```
SCA$GET_OCCURRENCE   command_context,
                     query_name,
                     occurrence
                     [,order]
```

Arguments

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context.

query_name

type:	character string
access:	read-only
mechanism:	by descriptor

Name of the query in the command-context.

occurrence

type:	$SCA_HANDLE
access:	read/write
mechanism:	by reference

SCA occurrence handle that describes an occurrence.

order

type:	$SCA_OCC_SORT_ORDER
access:	read-only
mechanism:	by reference

Order of retrieval of occurrences from the query result.

Condition Values Returned

SCA$_NORMAL	An occurrence has been successfully returned.
SCA$_NEWNAME	An occurrence has been successfully returned. This occurrence has a different name from the occurrence that was returned by the previous call to this routine with this query context. This condition implies that this new occurrence is also of a different symbol.
SCA$_NEWITEM	An occurrence has been successfully returned. This new occurrence is of a different symbol from the occurrence that was returned by the previous call to this routine with this query context.
SCA$_NOMORE	Warning. An occurrence has not been returned. The traversal of the query result has been exhausted.

Description

The SCA$GET_OCCURRENCE routine returns an occurrence from the query specified by the **query_name** argument.

If the occurrence handle supplied is 0, the routine returns a handle to the first occurrence in the query represented by the **query_name** argument. If the occurrence handle supplied on input represents a valid occurrence, the routine returns a handle to the next occurrence in the query result. To be valid, the occurrence handle supplied on input must refer to an occurrence in the query represented by the **query_name** argument.

The query name supplied is interpreted in the context of the command-context identified by the **command_context** argument.

The order of retrieval of the occurrences is defined by the optional **order** argument. The possible values for this argument are as follows:

- SCA$K_OCCURRENCE_ORDER_DEFAULT
- SCA$K_OCCURRENCE_ORDER_LEXICAL

When SCA$K_OCCURRENCE_ORDER_LEXICAL is used for this argument, the order of retrieval of occurrences from any module coincides with their lexical order within that module.

When SCA$K_OCCURRENCE_ORDER_DEFAULT is used for this argument, the order of retrieval of occurrences is undefined.

The default value for the **order** argument is SCA$K_OCCURRENCE_ORDER_DEFAULT.

SCA$INITIALIZE

Initializes the SCA callable command interface.

Format

SCA$INITIALIZE   command_context

Argument

command_context

type:	$SCA_COMMAND_CONTEXT
access:	write-only
mechanism:	by reference

SCA command-context value to be initialized. This value is passed as an argument to other SCA$xxx routines.

Condition Value Returned

SCA$_NORMAL	The SCA callable command interface has been successfully initialized.

Description

The SCA$INITIALIZE routine initializes the SCA callable command interface. Note that corresponding to each call to this routine, a call to SCA$CLEANUP must be made so the process invoking SCA terminates normally.

SCA$LOCK_LIBRARY

Locks all the physical libraries in the current virtual library list so they cannot be modified.

Format

SCA$LOCK_LIBRARY   command_context

Argument

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

SCA command-context.

Condition Value Returned

SCA$_NORMAL	The libraries have been successfully locked.

Description

The SCA$LOCK_LIBRARY routine locks all the physical libraries in the current virtual library list so they cannot be modified.

SCA$PUT_OUTPUT

Writes a record to the current controlling output device specified by SYS$OUTPUT.

Format

```
SCA$PUT_OUTPUT   string,
                 user_argument
```

Arguments

string

type:	character string
access:	read only
mechanism:	by descriptor

String to be written to SYS$OUTPUT. You can concatenate one or more additional character strings with the primary string to form a single output record. You can specify a maximum of 20 strings. The maximum resulting record length is 255 characters.

user_argument

type:	_UNSPECIFIED
access:	read only
mechanism:	by reference

User-specified value passed to the routine calling this action routine.

Condition Values Returned

SCA$_NORMAL	The string was successfully written to SYS$OUTPUT.
Failure completion code from the HP RMS $PUT service	

Description

The SCA$PUT_OUTPUT routine writes a record to the current controlling output device specified by SYS$OUTPUT.

SCA$QUERY_CLEANUP

Cleans up an SCA query context, which frees all dynamic memory associated with the query.

Format

SCA$QUERY_CLEANUP query_context

Argument

query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

SCA query context to be cleaned up.

Condition Value Returned

SCA$_NORMAL	The query context has been successfully cleaned up.

Description

The SCA$QUERY_CLEANUP routine cleans up an SCA query context, which frees all dynamic memory associated with the query.

SCA$QUERY_COPY

Copies a query from SRC_QUERY_CONTEXT to DST_QUERY_CONTEXT.

Format

```
SCA$QUERY_COPY   src_query_context,
                 dst_query_context
```

Arguments

src_query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

SCA query context that describes the query to be copied.

dst_query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

SCA query context into which the query is to be copied.

Condition Value Returned

SCA$_NORMAL	The query expression has been successfully copied.

Description

The SCA$QUERY_COPY routine copies a query from SRC_QUERY_CONTEXT to DST_QUERY_CONTEXT. This will copy whatever is in SRC_QUERY_CONTEXT, whether that is a question, or a question and a result.

SCA$QUERY_FIND

Finds the occurrences that match the query expression specified by QUERY_CONTEXT.

Format

SCA$QUERY_FIND   query_context

Argument

query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

SCA query context that describes a query expression to be evaluated.

Condition Values Returned

SCA$_NORMAL	The query expression has been successfully evaluated.
SCA$_NOOCCUR	No occurrences match the query expression.
SCA$_RESULTEXISTS	The query already has a result prior to this call.

Description

The SCA$QUERY_FIND routine finds the occurrences that match the query expression specified by QUERY_CONTEXT.

SCA$QUERY_GET_ATTRIBUTE

Gets the handle to an attribute of an entity.

Format

```
SCA$QUERY_GET_ATTRIBUTE   entity,
                          attribute_kind,
                          attribute_handle,
                          [,iteration_context]
```

Arguments

entity

type:	$SCA_HANDLE
access:	read-only
mechanism:	by reference

SCA entity handle describing the entity or relationship whose attributes are being obtained.

attribute_kind

type:	$SCA_ATTRIBUTE_KIND
access:	read-only
mechanism:	by reference

Kind of attribute to be obtained.

Any attribute-kind can be specified on this routine.

attribute_handle

type:	$SCA_HANDLE
access:	write-only
mechanism:	by reference

SCA attribute handle to describe the obtained attribute.

iteration_context

type:	$SCA_ITERATION_CONTEXT
access:	read/write
mechanism:	by reference

The iteration-context. This longword must contain 0 on the first call to this routine for a particular iteration. This routine uses the longword to maintain the iteration context. The caller must not change the contents of the longword.

Condition Values Returned

SCA$_NORMAL	An attribute has been successfully returned.
SCA$_NONE	Warning. An attribute has not been returned. Either there are no such attributes in the entity, or there are no more attributes.

Description

The SCA$QUERY_GET_ATTRIBUTE routine gets a handle to an attribute of an entity.

If the **iteration_context** parameter is not specified, this routine finds the first attribute of the specified kind **(attribute_kind)** and updates **attribute_handle** to describe that attribute.

In general, several attributes can be associated with a particular entity. With this routine, you can find all of those attributes by using the **iteration_context** parameter.

SCA$QUERY_GET_ATTRI_KIND_T

Gets an attribute kind.

Format

```
SCA$QUERY_GET_ATTRI_KIND_T   attribute_handle,
                             attribute_kind
```

Arguments

attribute_handle

type:	$SCA_HANDLE
access:	read-only
mechanism:	by reference

SCA handle describing an attribute whose attribute-kind is to be obtained.

attribute_kind

type:	character string
access:	write-only
mechanism:	by descriptor

Kind of the attribute.

Condition Value Returned

SCA$_NORMAL	An attribute kind has been successfully returned.

Description

The SCA$QUERY_GET_ATTRI_KIND_T routine returns the kind of any attribute as a character string.

SCA$QUERY_GET_ATTRI_VALUE_T

Gets an attribute value.

Format

```
SCA$QUERY_GET_ATTRI_VALUE_T   handle,
                              attribute_value
                              [,attribute_kind]
```

Arguments

handle

type:	$SCA_HANDLE
access:	read/write
mechanism:	by reference

SCA handle describing either an attribute or an entity whose value is to be obtained.

attribute_value

type:	character string
access:	read/write
mechanism:	by descriptor

The (string) value of the attribute being selected.

attribute_kind

type:	$SCA_ATTRIBUTE_KIND
access:	read/write
mechanism:	by reference

The kind of attribute to be obtained.

Condition Values Returned

SCA$_NORMAL	An attribute value has been successfully returned.
SCA$_NONE	Warning. An attribute-value has not been returned. There are no such attributes in the entity. This condition can be returned only if this routine is processing an entity.

SCA$QUERY_GET_ATTRI_VALUE_T

Description

The SCA$QUERY_GET_ATTRI_VALUE_T routine returns the value of any attribute as a character string.

If the handle describes an attribute, this routine returns the value of that attribute. In this case, the **attribute_kind** parameter must not be specified.

If the handle describes an entity, this routine returns the value of the first attribute of that entity that is of the kind specified by the **attribute_kind** parameter. In this case, the **attribute_kind** parameter must be specified.

If you want to get more than one attribute value of a particular kind for an entity, use the routine SCA$QUERY_GET_ATTRIBUTE. This applies only to the attribute-kinds SCA$K_ATTRI_NAME and SCA$K_ATTRI_ALL.

The value of any kind of attribute can be returned by this routine except for SCA$K_ATTRI_ALL. This routine will convert to character string those attributes whose data type is not character string.

This routine does not accept the attribute-kind SCA$K_ATTRI_ALL as the value of the **attribute_kind** parameter. It is not meaningful to get just the first attribute without regard to attribute-kind.

SCA$QUERY_GET_NAME

Returns the name of a query.

Format

```
SCA$QUERY_GET_NAME   query_context,
                     query_name
```

Arguments

query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

SCA query context whose name is to be obtained.

query_name

type:	character string
access:	write-only
mechanism:	by descriptor

The name of the query.

Condition Value Returned

SCA$_NORMAL	The query name has been successfully returned.

Description

The SCA$QUERY_GET_NAME routine returns the name of a query.

SCA$QUERY_GET_OCCURRENCE

Gets the next occurrence in the query result that is specified as the **query_context** argument.

Format

```
SCA$QUERY_GET_OCCURRENCE   query_context,
                           entity_handle
```

Arguments

query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

An SCA query context whose occurrences are to be obtained.

entity_handle

type:	$SCA_HANDLE
access:	read/write
mechanism:	by reference

An SCA entity handle that describes an entity.

Condition Values Returned

SCA$_NORMAL	An occurrence has been successfully returned.
SCA$_NEWNAME	An occurrence has been successfully returned. This occurrence has a different name from the occurrence that was returned by the previous call to this routine with this query context. This condition implies that this new occurrence is also of a different symbol.
SCA$_NEWITEM	An occurrence has been successfully returned. This new occurrence is of a different symbol from the occurrence that was returned by the previous call to this routine with this query context.

SCA$_NOMORE	Warning. An occurrence has not been returned. The traversal of the query result has been exhausted.

Description

The SCA$QUERY_GET_OCCURRENCE routine successively returns every occurrence in a query result. It provides one pass through all the occurrences.

SCA$QUERY_INITIALIZE

Initializes an SCA query context.

Format

```
SCA$QUERY_INITIALIZE   command_context,
                       query_context
```

Arguments

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read-only
mechanism:	by reference

An SCA command-context.

query_context

type:	$SCA_QUERY_CONTEXT
access:	write-only
mechanism:	by reference

An SCA query-context to be initialized. This value is passed as an argument to other SCA query routines (SCA$QUERY_xxx).

Condition Value Returned

SCA$_NORMAL	The query context has been successfully initialized.

Description

The SCA$QUERY_INITIALIZE routine initializes an SCA query context. This routine must be called before any other SCA query routines (SCA$QUERY_xxx).

SCA$QUERY_PARSE

Parses a query-expression command string and sets up a query context, if the command is syntactically correct.

Format

```
SCA$QUERY_PARSE   query_context, query_expression_string
                  [,query_expression_length]
```

Arguments

query_context

type:	$SCA_QUERY_CONTEXT
access:	read-only
mechanism:	by reference

An SCA query context that describes the indicated query expression.

query_expression_string

type:	character string
access:	read-only
mechanism:	by descriptor

A query expression string.

query_expression_length

type:	longword
access:	write-only
mechanism:	by reference

Length of the query expression returned from the parser.

Condition Values Returned

SCA$_NORMAL	The query expression string has been successfully parsed.
SCA$_MORETEXT	Warning. The query expression string has been successfully parsed, but the text following the query expression is not a legal part of the query expression. This condition is returned only if the **query_expression_length** parameter is specified. If the **query_expression_length** parameter is not specified, this routine insists that the whole **query_expression_string** argument be a legal query expression; in this case, all errors are signaled.

Description

The SCA$QUERY_PARSE routine parses a query expression string and sets up a query context, if the command is syntactically correct.

SCA$QUERY_SELECT_OCCURRENCE

Creates a query expression that matches a specific entity.

Format

```
SCA$QUERY_SELECT_OCCURRENCE   query_context,
                              entity_handle
```

Arguments

query_context

type:	$SCA_QUERY_CONTEXT
access:	read/write
mechanism:	by reference

An SCA query context that describes a specific entity.

entity_handle

type:	$SCA_HANDLE
access:	read/write
mechanism:	by reference

An SCA entity handle describing the entity that the newly defined query context is to match.

Condition Value Returned

SCA$_NORMAL	A query expression has been successfully defined.

Description

The SCA$QUERY_SELECT_OCCURRENCE routine creates a query expression that matches a specific entity. You use this routine to specify queries based on the results of previous queries. The **entity_handle** parameter is obtained by traversing the results of a previous query evaluation. Typically, the query context of the **entity_handle** parameter is not the same as the **query_context** parameter. However, they can be the same. If they are the same query context, then that previous query is replaced with the query defined by this routine and, as a result, **entity_handle** becomes invalid.

SCA$SELECT_OCCURRENCE

Creates a query that matches a specific occurrence.

Format

```
SCA$SELECT_OCCURRENCE   occurrence,
                        query_name
```

Arguments

occurrence

type:	$SCA_HANDLE
access:	read-only
mechanism:	by reference

An SCA occurrence handle describing the occurrence that the newly created query is to match.

query_name

type:	character string
access:	write-only
mechanism:	by descriptor

The name of the newly created query. This query is created in the context of the same command-context as that in which the input occurrence handle is defined.

Condition Values Returned

SCA$_NORMAL	A query expression has been successfully defined.

Description

Use this routine to create new queries based on the results of previous queries. The occurrence handle parameter is obtained by traversing the results of a previous query evaluation.

SCA$UNLOCK_LIBRARY

Unlocks all the physical libraries in the current virtual library list so they can be modified.

Format

SCA$UNLOCK_LIBRARY   command_context

Argument

command_context

type:	$SCA_COMMAND_CONTEXT
access:	read/write
mechanism:	by reference

An SCA command-context.

Condition Value Returned

SCA$_NORMAL	The libraries have been successfully unlocked.

Description

The SCA$UNLOCK_LIBRARY routine unlocks all the physical libraries in the current virtual library list so they can be modified.

3 Using the SCA Query Language

The SCA Query Language is an enhancement to the FIND command. By entering queries, you can both broaden and refine your use of SCA. With the SCA Query Language, you can make explicit queries of a large system and selectively limit queries to the results of previous query operations. Specifically, you can do the following:

- Analyze source code by using both file and symbol information.
- Use names to select symbols.
- Use other attributes to select symbols.
- Specify precise search parameters.
- Use relationship functions to query relationships between symbols.

Additional features of the SCA Query Language are also described in this chapter.

3.1 Basic Concepts

The SCA Query Language is based on the concept of a **query expression**. A query expression is a general algebraic expression in the form of a parameter to the FIND command. It is used to extract specific information from SCA libraries. A query expression can contain subexpressions joined by query operators or functions. A **query operator** is either a logical operator (such as AND or OR), or a relationship function (such as CONTAINING or CALLING).

The SCA Query Language uses two primary elements: **symbols** and **occurrences**. A symbol is an abstract entity in a program. An occurrence is any use of a symbol in source code.

A symbol can be a variable, routine, field within a record, or any other clearly distinguishable item in a program. Each symbol has a name. Symbols also have attributes called **symbol attributes**. One symbol attribute is its class. Symbol classes include variable, literal, macro, function, or task. Symbols also have **domain attributes**, for example, global or inheritable.

A symbol has associated with it a set of occurrences. In addition, every occurrence has a corresponding symbol.

An occurrence has attributes called **occurrence attributes**. Occurrence attributes supply additional information about the use of the symbol. For example, an occurrence can be a declaration or reference.

3.2 SCA Query Language Tutorial

With the SCA Query Language, you can perform a wide range of operations from simple to complex queries. This section contains a set of these operations, which are based on a C module. It begins with simple queries and gradually introduces more sophisticated ways to use the query language. The FIND commands demonstrated in this section are entered relative to an SCA library describing the following C module:

```
     #include <stdio.h>
     #include <strng.h>
     #include "openfiles.h"
     #include "types.h"

     extern int trnlit__openin,
         trnlit__openout,
         trnlit__badorig,
         trnlit__badrepl,
         trnlit__null,
         trnlit__compnull,
         trnlit__complete;


     int expand_string ( param_string str, code_vector codes,
       param_string error_text );


     void build_table (code_vector orig_vector, code_vector repl_vector,
      code_vector_length orig_length, code_vector_length repl_length,
      boolean complement, trans_table table );

     void copy_file (FILE *in_file, FILE *out_file, trans_table table);

     void write_error (int error_status, int extra_status, char *err_text);

     int read_command_line (int argc, char *argv[],
              FILE *in_file, FILE *out_file, trans_table table );

     main (int argc, char *argv[])
     {
1        FILE *in_file;
1        FILE *out_file;
1        trans_table table;
```

```
531    1
532    1        read_command_line (argc, argv, in_file, out_file, table);
533    1        copy_file (in_file, out_file, table);
534    1        return trnlit__complete;
535    1    }
536
537         int read_command_line (int argc, char *argv[],
538                  FILE *in_file, FILE *out_file, trans_table table )
539         {
540    1        param_string in_file_name;
541    1        param_string orig_chars_string;
542    1        param_string repl_chars_string;
543    1        param_string out_file_name;
544    1        code_vector orig_vector;
545    1        code_vector repl_vector;
546    1        code_vector_length orig_len;
547    1        code_vector_length repl_len;
548    1        int status;
549    1        param_string err_text;
550    1        boolean complement_originals;
551    1
552    1        orig_len = 0;
553    1        repl_len = 0;
554    1
555    1        in_file_name = argv[1];
556    1        orig_chars_string = argv[2];
557    1        repl_chars_string = argv[3];
558    1        out_file_name = argv[4];
559    1
560    1        open_in (in_file, in_file_name, "", trnlit__openin);
561    1
562    1        err_text = "";
563    1        if (strlen (orig_chars_string) == 0)
564    1        {
565    2     status = trnlit__null;
566    2        }
567    1        else
568    1        {
569    2     complement_originals = (orig_chars_string[0] == '-');
570    2     if (complement_originals)
571    2     {
572    3         strncpy (orig_chars_string, &orig_chars_string[1],
573    3           strlen(orig_chars_string) - 1);
574    3     };
575    2     if (strlen(orig_chars_string) == 0)
576    2     {
577    3         status = trnlit__compnull;
578    3     }
579    2     else
580    2     {
581    3         status = expand_string (orig_chars_string, orig_vector, err_text);
582    3     };
```

```
  583    2       };
  584    1
  585    1       if (status != 1)
  586    1       {
  587    2        write_error (trnlit__badorig, status, err_text);
  588    2    return status;
  589    2       };
  590    1
  591    1       status = expand_string (repl_chars_string, repl_vector, err_text);
  592    1       if (status != 1)
  593    1       {
  594    2        write_error (trnlit__badrepl, status, err_text);
  595    2    return status;
  596    2       };
  597    1
  598    1       build_table (orig_vector, repl_vector, orig_len, repl_len,
  599    1                                 complement_originals, table);
  600    1
  601    1       open_out (out_file, out_file_name, "", in_file, trnlit__openout);
  603    1   }
```

The examples shown in this chapter are based on using SCA in standalone mode. When you use LSE, some results appear differently. For the sake of clarity, the following description is presented in the context of case sensitivity being turned off. If that is not the case, make sure you enter query expressions in the proper case.

3.2.1 Simple Queries

The simplest query specifies symbols based on the name of the symbol. For example, to get information about all the occurrences of symbols named ORIG_VECTOR, enter the following command:

```
FIND orig_vector
```

The result is as follows:

```
ORIG_VECTOR argument
    TRANSLIT\515         function parameter declaration
ORIG_VECTOR variable
    TRANSLIT\544         variable definition declaration
    TRANSLIT\581         read reference
    TRANSLIT\598         read reference
%SCA-S-OCCURS, 4 occurrences found (2 symbols, 1 name)
```

This display shows that SCA found two symbols named ORIG_VECTOR. The first symbol is the argument defined on line 515 of the sample program. This symbol has only one occurrence.

The second symbol is a variable and has three occurrences. The first occurrence is the declaration of the variable and indicates that it is the ORIG_VECTOR symbol defined on line 544 of the sample program. The next two occurrences are references to this variable.

You can restrict the query in several ways. For example, to get only declarations of symbols named ORIG_VECTOR, enter the following command:

```
FIND orig_vector AND occurrence=declaration
```

The result is as follows:

```
ORIG_VECTOR argument
    TRANSLIT\515         function parameter declaration
ORIG_VECTOR variable
    TRANSLIT\544         variable definition declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 1 name)
```

To get only read references of symbols named ORIG_VECTOR, enter the following command:

```
FIND orig_vector AND occurrence=read
```

The result is as follows:

```
ORIG_VECTOR variable
    TRANSLIT\581         read reference
    TRANSLIT\598         read reference
%SCA-S-OCCURS, 2 occurrences found (1 symbol, 1 name)
```

The previous two examples show query selection based on occurrence attributes.

With the SCA Query Language, you can use wildcards. For example, to display information about procedures and functions without regard to their names, enter the following command:

```
FIND * AND symbol=routine
```

Because the name attribute is a wildcard, it can be left out completely. The following command is equivalent to the previous one:

```
FIND symbol=routine
```

The result of this command is similar to the following sample:

```
BUILD_TABLE procedure
    TRANSLIT\515         void function declaration
    TRANSLIT\598         call reference
COPY_FILE procedure
    TRANSLIT\519         void function declaration
    TRANSLIT\533         call reference
```

```
...
VSPRINTF function
    TRANSLIT\132         function declaration
WRITE_ERROR procedure
    TRANSLIT\521         void function declaration
    TRANSLIT\587         call reference
    TRANSLIT\594         call reference
%SCA-S-OCCURS, 87 occurrences found (73 symbols, 73 names)
```

This example shows query selection based on symbol attributes. You can combine both symbol and occurrence attributes in one query. For example, to find the primary declarations of routines, enter the following command:

```
FIND symbol=routine AND occurrence=primary
```

The result is as follows:

```
MAIN function
    TRANSLIT\526         function definition declaration
READ_COMMAND_LINE function
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

The previous examples also show the use of logical operators to form more complex queries based on subqueries. For more information on logical operators, see the section Section 3.2.3 later in this chapter.

You can further restrict the previous query by adding an expression that distinguishes module-specific symbols from those that (potentially) span multiple modules. To display the primary declaration of intermodule routines (only those that have the potential of spanning multiple modules), enter the following command:

```
FIND symbol=routine AND occurrence=primary AND domain=multi_module
```

Because the example is simple, the domain selection does not alter the earlier result, which is as follows:

```
MAIN function
    TRANSLIT\526         function definition declaration
READ_COMMAND_LINE function
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You can abbreviate attribute-selection expressions. For example, you can abbreviate the preceding command as follows:

```
FIND symb=rout AND occ=prim AND doma=mult
```

3.2.2 Using the EXPAND Function to Find Related Occurrences

The process of expanding a set of occurrences to include all the occurrences of the corresponding set of symbols is called **expansion**.

To perform an expansion operation, SCA first finds all the occurrences that match the given expression. SCA then finds all the symbols that correspond to that set of occurrences. Finally, the result of the query is all the occurrences of those symbols.

For example, to display all the occurrences of routines that have primary declarations in the current SCA library, enter the following command:

```
FIND EXPAND (occurrence=primary AND symbol=routine)
```

The result is as follows:

```
MAIN function
    TRANSLIT\526         function definition declaration
READ_COMMAND_LINE function
    TRANSLIT\523         function declaration
    TRANSLIT\532         call reference
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 4 occurrences found (2 symbols, 2 names)
```

The parenthetical expression is the same query used earlier. The addition of the expansion operation causes the result to contain all occurrences of the symbols found, not just those specified by the subexpression.

You can follow an expansion with more restrictions. For example, to display the call references of routines that have primary declarations in the SCA library being queried, enter the following command:

```
FIND EXPAND (occurrence=primary AND symbol=routine) AND occ=call
```

This is an example of a nested query expression. The inner query expression, EXPAND (occurrence=primary AND symbol=routine), is evaluated first, which results in a set of all the occurrences of routines for which there are primary declarations. That set of occurrences is the input to the outer query expression, which has the following form:

```
query-expression AND occ=call
```

The outer query expression removes all occurrences that are not call-references.

The result is as follows:

```
READ_COMMAND_LINE function
    TRANSLIT\532         call reference
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

In another example of expansion, to display declarations of symbols that have write references, enter the following command:

```
FIND EXPAND (occ=write) AND occ=decl
```

To evaluate this query, SCA begins by finding the set of write reference occurrences. Next, SCA expands this set to include all occurrences of these symbols. Finally, the new set is intersected with the set containing all declaration occurrences.

The result is as follows:

```
COMPLEMENT_ORIGINALS variable
    TRANSLIT\550         variable definition declaration
ERR_TEXT variable
    TRANSLIT\549         variable definition declaration
IN_FILE_NAME variable
    TRANSLIT\540         variable definition declaration
ORIG_CHARS_STRING variable
    TRANSLIT\541         variable definition declaration
ORIG_LEN variable
    TRANSLIT\546         variable definition declaration
OUT_FILE_NAME variable
    TRANSLIT\543         variable definition declaration
REPL_CHARS_STRING variable
    TRANSLIT\542         variable definition declaration
REPL_LEN variable
    TRANSLIT\547         variable definition declaration
STATUS variable
    TRANSLIT\548         variable definition declaration
%SCA-S-OCCURS, 9 occurrences found (9 symbols, 9 names)
```

3.2.3 Using Logical Operators to Select Information

The SCA Query Language can apply **logical operators** to the results of other query expressions. The logical operator expressions that are supported are union (OR), intersection (AND), negation (NOT), and exclusive-or (XOR), as follows:

- **Union**—Uses the OR logical operator to merge two sets, which results in a set containing all occurrences that exist in either set.
- **Intersection**—Identifies occurrences that exist in two different sets, which results in a set containing each occurrence that exists in both sets; occurrences that appear in only one of the sets are not included. The intersection operator is AND.
- **Negation**—Identifies occurrences that are not in the set. The negation operator is NOT.

- **Exclusive-or**—Selects the unique occurrences in two different sets, which results in a set containing all occurrences that exist in only one of the sets. The exclusive-or operator is XOR.

For example, if you want to find all the symbols that have TABLE in their name, but you want to exclude those symbols whose name is TABLE, enter the following command:

```
FIND *table* AND NOT table
```

The result is as follows:

```
BUILD_TABLE procedure
    TRANSLIT\515         void function declaration
    TRANSLIT\598         call reference
TRANS_TABLE type
    TRANSLIT\495         typedef definition declaration
    TRANSLIT\517         reference
    TRANSLIT\519         reference
    TRANSLIT\524         reference
    TRANSLIT\530         reference
    TRANSLIT\538         reference
%SCA-S-OCCURS, 8 occurrences found (2 symbols, 2 names)
```

If you want to find all symbols with names that begin with ORIG, but not those symbols that have read or write references, enter the following command:

```
FIND orig* AND NOT EXPAND(occ=read OR occ=write)
```

The result is as follows:

```
ORIG_LENGTH argument
    TRANSLIT\516         function parameter declaration
ORIG_VECTOR argument
    TRANSLIT\515         function parameter declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

The previous query can also be written as follows:

```
FIND orig* AND NOT EXPAND occ=(read,write)
```

To display the declarations of all symbols that are both read and written, enter the following command:

```
FIND (EXPAND(occ=read) AND EXPAND(occ=write)) AND occ=decl
```

The result is as follows:

```
COMPLEMENT_ORIGINALS variable
    TRANSLIT\550         variable definition declaration
ERR_TEXT variable
    TRANSLIT\549         variable definition declaration
IN_FILE_NAME variable
    TRANSLIT\540         variable definition declaration
ORIG_CHARS_STRING variable
    TRANSLIT\541         variable definition declaration
ORIG_LEN variable
    TRANSLIT\546         variable definition declaration
OUT_FILE_NAME variable
    TRANSLIT\543         variable definition declaration
REPL_CHARS_STRING variable
    TRANSLIT\542         variable definition declaration
REPL_LEN variable
    TRANSLIT\547         variable definition declaration
STATUS variable
    TRANSLIT\548         variable definition declaration
%SCA-S-OCCURS, 9 occurrences found (9 symbols, 9 names)
```

Alternatively, to display the declarations of all symbols that are either read or written, but not both, enter the following command:

```
FIND (EXPAND(occ=read) XOR EXPAND(occ=write)) AND occ=decl
```

The result is as follows:

```
ARGC argument
    TRANSLIT\526         function parameter definition declaration
ARGV argument
    TRANSLIT\526         function parameter definition declaration

...

TRNLIT__OPENIN variable
    TRANSLIT\502         variable declaration
TRNLIT__OPENOUT variable
    TRANSLIT\503         variable declaration
%SCA-S-OCCURS, 20 occurrences found (20 symbols, 17 names)
```

To find all the symbols that are declared but never referenced, enter the following command:

```
FIND NOT EXPAND occ=ref
```

This finds all occurrences of the symbols that are never referenced. The result is as follows:

```
pointer
    TRANSLIT\419         pointer definition declaration (hidden)
pointer
    TRANSLIT\419         pointer definition declaration (hidden)

...

__STRING_LOADED macro
    TRANSLIT\369         macro definition declaration (hidden)
__WHENCE argument
    TRANSLIT\279         function parameter declaration
%SCA-S-OCCURS, 557 occurrences found (557 symbols, 167 names)
```

If you enter this command and realize that your chosen languages and code practices tend to give some unimportant cases of declared but not referenced symbols (like modules and formal parameters), you might want to further qualify your request by entering the following command:

```
FIND (NOT EXPAND occ=ref) AND NOT "" AND NOT -
                                      symbol=(module,argument,type,macro)
```

The result is a display that removes symbols with null names, as well as modules, parameters (called ARGUMENTS), types, and macros from the set of occurrences of the symbols that are never referenced. The result in the example library is still large; segments of it follow:

```
COMPRESS component
    TRANSLIT\494         struct or union member definition declaration
CTERMID function
    TRANSLIT\349         function declaration

...

_FLAG component
    TRANSLIT\19          struct or union member definition declaration
_PTR component
    TRANSLIT\17          struct or union member definition declaration
%SCA-S-OCCURS, 76 occurrences found (76 symbols, 76 names)
```

3.2.4 The Current Query

SCA maintains a **current query**. A current query is the result of the previously entered query, or the one to which you are set using the NEXT QUERY, PREVIOUS QUERY, and GOTO QUERY commands. The current query is specified the same way as any other query. The name of the current query is SCA$CURRENT_QUERY.

The following is an example of a command sequence using the current query:

```
FIND *table* AND NOT table
FIND ( NOT @sca$current_query ) AND symbol=routine AND occurrence=decl
```

The @ function defaults to the current query. Consequently, you can also write the previous commands as follows:

```
FIND *table* AND NOT table
FIND ( NOT @() ) AND symbol=routine AND occ=decl
```

You can assign a name to a query by using the NAME option of the FIND command. A query remains available for use throughout a given invocation of SCA, unless it is explicitly deleted using the DELETE QUERY command. If you do not name a query, SCA automatically assigns a name to it. For example, the previous command sequence can be rewritten as follows:

```
FIND -NAME table *table* AND NOT table
FIND -NAME routine_decls symbol=routine AND occ=decl
FIND (NOT @table) AND @routine_decls
```

You can use the SHOW QUERY command to display all currently available queries. For example, if you follow the previous set of FIND commands with a SHOW QUERY command, the result is as follows:

```
    Name      Query expression         Description

    table     *table* AND NOT table    (none)
    routine_decls
              symbol=routine AND occ=decl
                                       (none)
(*) 1         (NOT @table) AND @routine_decls
                                       (none)
```

3.2.5 Structured Relationship Expressions

With the SCA Query Language, you can select occurrences based on their relationship to other occurrences. For example, enter the following command:

```
FIND CALLED_BY read_command_line
```

The result is as follows:

```
READ_COMMAND_LINE function calls
   BUILD_TABLE procedure
   EXPAND_STRING function
   OPEN_IN procedure
   OPEN_OUT procedure
   STRLEN function
   STRNCPY function
   WRITE_ERROR procedure
%SCA-S-OCCURS, 12 occurrences found (8 symbols, 8 names)
```

You can interpret the previous command as, "Find what is called by READ_COMMAND_LINE."

You can ask the reverse question, "Find what is calling READ_COMMAND_LINE," by entering the following command:

```
   FIND CALLING read_command_line
```

The result is as follows:

```
MAIN function calls
   READ_COMMAND_LINE function
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You can use the depth parameter to request that more than one level of structure be displayed. The depth parameter sets the number of levels of structure that you want SCA to trace. By default, one level of structure is traced.

To request a depth level of 2, enter the following command:

```
FIND CALLED_BY( main, depth=2 )
```

The result is as follows:

```
MAIN function calls
   COPY_FILE procedure
   READ_COMMAND_LINE function calls
      BUILD_TABLE procedure
      EXPAND_STRING function
      OPEN_IN procedure
      OPEN_OUT procedure
      STRLEN function
      STRNCPY function
      WRITE_ERROR procedure
%SCA-S-OCCURS, 15 occurrences found (10 symbols, 10 names)
```

You can use the DEPTH=ALL option to specify that all levels of call relationship are to be traced. To trace all call relationships from MAIN down, enter the following command:

```
FIND CALLED_BY( main, depth=all )
```

Because the example program is so simple, this command gives the same result as the previous command.

If you want to know if one routine can be called from within another directly or indirectly (for example, to display all of the paths of the call-graph that lead from MAIN to STRLEN), enter the following command:

```
FIND CALLED_BY( main, strlen, depth=all )
```

The result is as follows:

```
MAIN function calls
   READ_COMMAND_LINE function calls
      STRLEN function
%SCA-S-OCCURS, 6 occurrences found (3 symbols, 3 names)
```

You can interpret the previous command as, "Find what is called by MAIN, tracing any number of levels of structure, but include only the paths that lead to STRLEN."

One common problem with call-tree displays is that they often contain a large percentage of lines describing routines that are not a part of the application under development. These are routines supplied as part of your operating system, as runtime support for your programming language, or some other set of routines that are viewed by the developer as being a part of the base system. Having these utility routines in a call-tree is often a nuisance, because they make it difficult to see the most important structure of the call tree.

The primary declarations of such utility routines are not described in the SCA library of an application.

```
FIND CALLED_BY( main, EXPAND( occ=primary ), depth=all )
```

You can interpret the preceding command as, "Find what is called by MAIN, tracing any number of levels of structure, but include only the paths that lead to routines that have primary declarations." A more succinct interpretation is, "Trace the calls from MAIN through the routines that have primary declarations."

The result is as follows:

```
MAIN function calls
   READ_COMMAND_LINE function
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

You have even more control over the tracing of relationships by using the TRACE parameter. The TRACE parameter specifies a query expression. As the CALLED_BY function iteratively traces the calls, it continues tracing the called-by relationship only through the occurrences that match the trace-expression, which is specified as the value of the TRACE parameter.

For example, to display all of the paths of the call graph from MAIN down, except the call relationships traced through the routine READ_COMMAND_LINE, enter the following command:

```
FIND CALLED_BY( main, depth=all, trace=(NOT read_command_line) )
```

The result is as follows:

```
MAIN function calls
   COPY_FILE procedure
   READ_COMMAND_LINE function
%SCA-S-OCCURS, 3 occurrences found (3 symbols, 3 names)
```

Note that READ_COMMAND_LINE is included in the result, but the tracing of the called-by relationship does not continue through READ_COMMAND_LINE.

The TRACE parameter does not affect the first iteration. That first iteration is controlled by the BEGIN parameter.

Note that you can terminate tracing the called-by relationship through READ_COMMAND_LINE and you can exclude calls to READ_COMMAND_LINE, by entering the following command:

```
FIND CALLED_BY( build_table, -
                NOT read_command_line, -
                depth=all, -
                trace=(NOT read_command_line) )
```

This command displays all the paths of the call-graph from MAIN down, except it will not match calls to the routine READ_COMMAND_LINE. As a result, READ_COMMAND_LINE is not included in the display. The result is as follows:

```
MAIN function calls
   COPY_FILE procedure
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

3.2.6 Nonstructured Relationship Expressions

There is a simple but important difference between the commands about to be described here and those described in the previous section. Both sets of commands use information about the relationships between occurrences. However, the commands described in the previous sections use that relationship information to create a collection of occurrences and the relationships between those occurrences. The commands described in the following sections discard the relationship information from the query result. Consequently, these relationship-query expressions are considered nonstructured because the result is solely a flat set of occurrences.

Nonstructured relationship expressions are realized by using the result-parameter of the relationship functions. For example, you saw in the previous section the results of the following command:

```
FIND CALLED_BY read_command_line
```

The result is as follows:

```
READ_COMMAND_LINE function calls
   BUILD_TABLE procedure
   EXPAND_STRING function
   OPEN_IN procedure
   OPEN_OUT procedure
   STRLEN function
   STRNCPY function
   WRITE_ERROR procedure
%SCA-S-OCCURS, 12 occurrences found (8 symbols, 8 names)
```

A nonstructured version of the same command is as follows:

```
FIND CALLED_BY( read_command_line, result=nostructure )
```

The result is as follows:

```
BUILD_TABLE procedure
    TRANSLIT\598        call reference
EXPAND_STRING function
    TRANSLIT\581        call reference
    TRANSLIT\591        call reference
OPEN_IN procedure
    TRANSLIT\560        call reference
OPEN_OUT procedure
    TRANSLIT\601        call reference
READ_COMMAND_LINE function
    TRANSLIT\537        function definition declaration
STRLEN function
    TRANSLIT\563        call reference
    TRANSLIT\573        call reference
    TRANSLIT\575        call reference
STRNCPY function
    TRANSLIT\572        call reference
WRITE_ERROR procedure
    TRANSLIT\587        call reference
    TRANSLIT\594        call reference
%SCA-S-OCCURS, 12 occurrences found (8 symbols, 8 names)
```

You can also use the result-parameter to restrict the result to just the beginning or just the end of the relationship expression. For example, if you want to identify the routines that call C RTL routines for string manipulation, enter the following command:

```
FIND CALLING( str*, result=begin )
```

The result is as follows:

```
READ_COMMAND_LINE function
    TRANSLIT\537        function definition declaration
%SCA-S-OCCURS, 1 occurrence found (1 symbol, 1 name)
```

You can interpret this command as, “Find what is calling STR*, and report only the caller, READ_COMMAND_LINE, not the callee, STR*.”

If you want to find all the occurrences of routines that call RTL routines, expand the result of the previous query expression by entering the following command:

```
FIND EXPAND CALLING( str*, result=begin )
```

The result is as follows:

```
READ_COMMAND_LINE function
    TRANSLIT\523         function declaration
    TRANSLIT\532         call reference
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 3 occurrences found (1 symbol, 1 name)
```

You can also find routines that call STR* indirectly by entering the following command:

```
FIND CALLING( str*, depth=all, result=begin )
```

This command asks for the routines that call an RTL routine either directly or indirectly. The result is as follows:

```
MAIN function
    TRANSLIT\526         function definition declaration
READ_COMMAND_LINE function
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 2 occurrences found (2 symbols, 2 names)
```

3.2.7 Other Relationships

The TYPING and CONTAINING commands also enable you to query a software system. For example, you can determine the type of the routine parameter TABLE by entering the following command:

```
FIND TYPING table
```

The result is as follows:

```
TRANS_TABLE type types
   TABLE argument
   TABLE argument
   TABLE argument
   TABLE variable
   TABLE argument
%SCA-S-OCCURS, 10 occurrences found (6 symbols, 2 names)
```

You can trace these type relationships through multiple levels. Suppose TRANS_TABLE is defined as follows:

```
#define min_code 0
#define max_code 258

typedef int code_value; /* min_code .. max_code */

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

You can trace multiple levels by entering the following command:

```
FIND TYPING( table, depth=all )
```

The result is as follows:

```
TABLE argument is typed by
   TRANS_TABLE type is typed by
      array is typed by
         array index is typed by
         .  UNSIGNED INT scalar type
         array component is typed by
            record is typed by
               COMPRESS component is typed by
               .  enumeration type is typed by
               .     FALSE constant is typed by
               .     .  INT scalar type
               .     TRUE constant is typed by
               .        INT scalar type  (See above)
               TRANS_VALUE component is typed by
                  INT scalar type  (See above)
TABLE argument is typed by
   TRANS_TABLE type  (See above)
TABLE argument is typed by
   TRANS_TABLE type  (See above)
TABLE variable is typed by
   TRANS_TABLE type  (See above)
TABLE argument is typed by
   TRANS_TABLE type  (See above)
%SCA-S-OCCURS, 26 occurrences found (17 symbols, 9 names)
```

You can also find all variables of type INT by entering the following command:

```
FIND TYPED_BY( int, symbol=variable, result=begin )
```

The result is as follows:

```
ORIG_LEN variable
    TRANSLIT\546        variable definition declaration
REPL_LEN variable
    TRANSLIT\547        variable definition declaration
STATUS variable
    TRANSLIT\548        variable definition declaration
TRNLIT__BADORIG variable
    TRANSLIT\504        variable declaration
TRNLIT__BADREPL variable
    TRANSLIT\505        variable declaration
TRNLIT__COMPLETE variable
    TRANSLIT\508        variable declaration
TRNLIT__COMPNULL variable
    TRANSLIT\507        variable declaration
TRNLIT__NULL variable
    TRANSLIT\506        variable declaration
TRNLIT__OPENIN variable
    TRANSLIT\502        variable declaration
TRNLIT__OPENOUT variable
    TRANSLIT\503        variable declaration
%SCA-S-OCCURS, 10 occurrences found (10 symbols, 10 names)
```

You can interpret the previous command as, “Find the symbols that are typed-by INT and that match the query expression, SYMBOL=VARIABLE, and return just the beginning of the typed-by relationship.”

There is a general containment relationship. You can use it to get a declaration tree. To show the (primary) declaration structure of routines and variables in the module TRANSLIT, enter the following command:

```
FIND CONTAINED_BY( translit,
     symbol=(routine, variable) and occurrence=primary,
     depth=all )
```

The first parameter says, “Begin at TRANSLIT.” The second parameter says, “End with primary declarations of routines or variables.” The third parameter says, “Repeat any number of levels.” The result is as follows:

```
TRANSLIT module contains
   MAIN function contains
   .  IN_FILE variable
   .  OUT_FILE variable
   .  TABLE variable
   READ_COMMAND_LINE function contains
      COMPLEMENT_ORIGINALS variable
      ERR_TEXT variable
      IN_FILE_NAME variable
      ORIG_CHARS_STRING variable
      ORIG_LEN variable
      ORIG_VECTOR variable
      OUT_FILE_NAME variable
      REPL_CHARS_STRING variable
      REPL_LEN variable
      REPL_VECTOR variable
      STATUS variable
%SCA-S-OCCURS, 17 occurrences found (17 symbols, 17 names)
```

You can use the containment relationship to specify more precisely which symbol you want. To request all the occurrences of symbols named READ_COMMAND_LINE that are directly contained by the module TRANSLIT, enter the following command:

```
FIND CONTAINED_BY( translit, read_command_line, result=begin )
```

The result is as follows:

```
READ_COMMAND_LINE function
    TRANSLIT\523         function declaration
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 2 occurrences found (1 symbol, 1 name)
```

Alternatively, to request all the occurrences of symbols named READ_COMMAND_LINE that are directly or indirectly contained by the module TRANSLIT, enter the following command:

```
FIND CONTAINED_BY( translit, read_command_line, depth=all, result=begin )
```

The result is as follows:

```
READ_COMMAND_LINE function
    TRANSLIT\523         function declaration
    TRANSLIT\532         call reference
    TRANSLIT\537         function definition declaration
%SCA-S-OCCURS, 3 occurrences found (1 symbol, 1 name)
```

3.2.8 The IN Function

The CONTAINED_BY function is so general that even the most common queries involve the specification of several parameters. Therefore, the IN function has been defined as a special case of the CONTAINED_BY function.

The IN function returns all of the specified occurrences that are contained directly or indirectly (DEPTH=ALL) by a specified (set of) occurrences. See Chapter 4 for more information.

As an example, the FIND CONTAINED_BY command in the previous section can be more simply written as follows:

```
FIND IN( translit, read_command_line )
```

You can interpret this command as, “Find all occurrences in TRANSLIT named READ_COMMAND_LINE.”

To find all occurrences in TRANSLIT, including those nested within declarations of TRANSLIT, you omit the second parameter, as shown in the following command:

```
FIND IN translit
```

As another example, imagine that you are working on a compiler project and you have defined a query named PARSER to contain the list of all the modules that make up the parser. To find all the occurrences of symbols named STRLEN contained directly or indirectly within the parser modules, enter the following command:

```
FIND IN( @parser, strlen )
```

3.2.9 Path Names

The SCA query language accepts path-name notation. For example, to find only symbols named IN_FILE that are declared directly within MAIN, enter the following command:

```
FIND main\in_file
```

The result is as follows:

```
IN_FILE variable
    TRANSLIT\528         variable definition declaration
    TRANSLIT\532         read reference
    TRANSLIT\533         read reference
%SCA-S-OCCURS, 3 occurrences found (1 symbol, 1 name)
```

Alternatively, you can enter the following command:

```
FIND read_command_line\in_file
```

The result is as follows:

```
IN_FILE argument
    TRANSLIT\538         function parameter definition declaration
    TRANSLIT\560         read reference
    TRANSLIT\601         read reference
%SCA-S-OCCURS, 3 occurrences found (1 symbol, 1 name)
```

You can build up path names repeatedly to increase precision. For example, the previous display can be produced by entering the following command:

```
FIND translit\read_command_line\in_file
```

You can use wildcards within path names. For example, the previous display could also have been produced by entering the following command:

```
FIND translit\*\in_file
```

You can interpret this command as, "Find symbols named IN_FILE that are declared within any primary declaration that is contained within the primary declaration of TRANSLIT."

You can specify that any number of containment levels are acceptable. This is done by leaving out a path name, as in the following command:

```
FIND translit\\in_file
```

You can interpret this command as, "Find symbols named IN_FILE that are declared directly or indirectly by the primary declaration of TRANSLIT." The result is as follows:

```
IN_FILE variable
    TRANSLIT\528         variable definition declaration
    TRANSLIT\532         read reference
    TRANSLIT\533         read reference
IN_FILE argument
    TRANSLIT\538         function parameter definition declaration
    TRANSLIT\560         read reference
    TRANSLIT\601         read reference
%SCA-S-OCCURS, 6 occurrences found (2 symbols, 1 name)
```

You can include a general query expression as a path name by entering the following command:

```
FIND (mod1 OR mod2)\\code
```

You can interpret this command as, "Find symbols named CODE that are declared directly or indirectly by the primary declaration of either MOD1 or MOD2."

Similarly, if you are working on a compiler project, and you have defined a query named PARSER to contain the list of all the modules that make up the parser, then to find all of the symbols named CODE that are declared in the parser, enter the following command:

```
FIND @parser\\code
```

3.2.10 Combined Relationship Examples

You can combine more than one relationship function into one query. If you want to know whether it is possible for a call to the routine READ_COMMAND_LINE to modify a global variable, you need to consider not only READ_COMMAND_LINE itself, but also the whole call tree from READ_COMMAND_LINE down. You can find out by entering the following command:

```
FIND IN( CALLED_BY( read_command_line, depth=all ), -
         sym=var AND occ=write AND domain=multi )
```

The result is as follows:

```
%SCA-W-NOOCCUR, no symbol occurrence matches your selection criteria
```

In a still more complicated query, you might want to find all the occurrences of symbols of type TRANS_TABLE that are contained within the call-tree from MAIN down. To make this query, enter the following command:

```
FIND IN( CALLED_BY( main, depth=all ), -
         EXPAND TYPED_BY( trans_table, result=begin ) )
```

The result is as follows:

```
TABLE variable
    TRANSLIT\530        variable definition declaration
    TRANSLIT\532        read reference
    TRANSLIT\533        read reference
TABLE argument
    TRANSLIT\538        function parameter definition declaration
    TRANSLIT\599        read reference
%SCA-S-OCCURS, 5 occurrences found (2 symbols, 1 name)
```

You can use the TRACE parameter to restrict a query to a particular subset of a program.

If you were a developer working on a LOAD command, and the module LOAD contained one entry point, LOAD_FILE, you could show all the call relationships within the module LOAD beginning with the routine LOAD_FILE by entering the following command:

```
FIND CALLED_BY( load_file, depth=all, trace=IN(load) )
```

This command shows all calls that occur within module LOAD, but traces through only the routines whose primary declaration is within LOAD. For example, if LOAD_FILE calls to a routine READ_EVENT whose primary declaration is outside of LOAD, the call to READ_EVENT shows up in the display, but no calls within READ_EVENT are included. This is because tracing is turned off outside of the module LOAD.

Alternatively, to trace call relationships from LOAD_FILE down, but display calls only to routines whose primary declaration occurs within LOAD, enter the following command:

```
FIND CALLED_BY( load_file, IN(load), depth=all, trace=IN(load) )
```

You can interpret this command (through the TRACE parameter) as, "Continue tracing the called-by relationship only through routines whose primary declaration occurs within LOAD;" and (through the BEGIN parameter), "Include only the paths that end with routines whose primary declaration occurs within LOAD." Consequently, this command does not include calls to READ_EVENT.

You can trace all call relationships from LOAD_FILE down, subject to only one limitation—each path must end with a routine whose primary declaration occurs within LOAD. Because the TRACE expression has been defaulted (to *), this command traces the called-by expression through any routine as long as the path eventually leads back to a routine declared in LOAD. You can perform this operation by entering the following command:

```
FIND CALLED_BY( load_file, IN(load), depth=all )
```

4 Evaluating SCA Query Expressions

This chapter describes the rules governing the use of the SCA Query Language. The following tables provide an overview of the components of the SCA Query Language.

Table 4–1 lists attribute selection expressions.

Table 4–1 Attribute Selection Expressions

Attribute	Syntax	Example
name	name	foo
symbol class	**symbol=**symbol_class	symbol=argument
symbol domain	**domain=**symbol_domain	domain=global
occurrence class	**occurrence=**occ_class	occurrence=primary
file specification	**file=**file_spec	file="foo.c"

See the Section 4.5 section later in this chapter for a list of attribute-selection values and their meanings.

Table 4–2 lists the binary operators.

Table 4–2 Binary Operators

Type	Syntax	Example
path name	exp1\exp2	subrx\y
	exp1\\exp2	routa\\y
intersection	exp1 **AND** exp2	a AND occurrence=declaration
union	exp1 **OR** exp2	symbol=argument OR symbol=variable

(continued on next page)

Table 4–2 (Cont.) Binary Operators

Type	Syntax	Example
exclusive or	exp1 **XOR** exp2	occurrence=read XOR occurrence=write

Table 4–3 lists nonrelationship function expressions.

Table 4–3 Nonrelationship Function Expressions

Function	Syntax	Example
negation	**NOT** query expression	l* AND NOT lib*
expansion	**EXPAND** query expression	EXPAND (occurrence=primary)
indicated	**INDICATED( )**	EXPAND INDICATED( )
query usage	**@**query name	@any_query and domain=module

Relationship function expressions have the following general syntax:

```
rel_function_name(end=query_expression,
                  begin=query_expression,
                  depth=n,
                  result=result_keyword,
                  trace=query_expression)
```

Table 4–4 and Table 4–5 describe the syntax in more detail.

Table 4–4 Function Names

Function Name/Example	Description
CALLING X	Displays what routines call X
CALLED_BY (A,B,DEPTH=ALL)	Displays the call tree from A to B
TYPING (Y,DEPTH=ALL)	Displays the type information of Y
TYPED_BY REAL	Displays all symbols of type REAL
CONTAINED (X,SYMBOL=ROUTINE)	Displays all routines in X
CONTAINING (DOMAIN=GLOBAL, SYMBOL=MODULE, RESULT=BEGIN, DEPTH=ALL)	Displays modules that contain globally defined symbols

Table 4–5 Function Parameters

Parameter	Type	Default
END	query expression	*
BEGIN	query expression	*
DEPTH	integer	1
RESULT	keyword value	STRUCTURE
TRACE	query expression	*

See the Section 4.7.6.2 section later in this chapter for a list of RESULT keyword values and their meanings.

4.1 Query Expression Syntax

This section defines the syntax of a query expression. The following example is a high-level description of the syntax. It defines, for example, the form of a function call, but it does not describe which functions are available. The low-level details are described in later sections.

```
query-expression ::= attribute-selection-expression |
                     binary-op-expression |
                     function-call-expression |
                     (query-expression )

attribute-selection-expression ::= actual-parameter

binary-op-expression ::= query-expression binary-operator query-expression

binary-operator ::= AND | OR | XOR | \ | \\

function-call-expression ::= function-name actual-parameter |
                             function-name ([actual-parameter],... )

function-name ::= nonwildcard-string

actual-parameter ::= named-actual-parameter | positional_actual_parameter

named-actual-parameter ::= formal-parameter-name = actual-parameter-value

positional_actual_parameter ::= actual-parameter-value

formal-parameter-name ::= nonwildcard-string

actual-parameter-value ::= query-expression | name-expression |

                           keyword-list | range-list | number

keyword-list ::= keyword | (keyword,... )

keyword ::= nonwildcard-string

number ::= digit... | ALL
```

```
range-list ::= range | ( range,... )

range ::= number | number:number

name-expression ::= simple-string | "complex-string"

nonwildcard-string ::= {letter | digit | graphic-character}...

simple-string ::= {letter | digit | graphic-character |
                   wildcard-character | escape-character}...

complex-string ::= any-character...

letter ::= any-alphabetic-character

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

graphic-character ::= - | _ | $

wildcard-character  ::= * | %

escape-character ::= &
```

4.2 Operator Precedence and Associativity

Table 4–6 is a syntax diagram that gives the forms of query expressions. The forms are grouped into priority levels, and an associativity is given for each priority level. In Table 4–6:

```
exp1 = query-expression
exp2 = query-expression
```

Table 4–6 Query Expression Forms

Priority	Operator Expression	Associates from
highest	function-name actual-parameter	right to left
	exp1 \ exp2, exp1 \\ exp2	left to right
	exp1 AND exp2	left to right
	exp1 OR exp2	left to right
lowest	exp1 XOR exp2	left to right

4.3 Default Parenthesizing

Default parenthesizing for query expressions is determined by the operator priorities and associativity given in the previous diagram. The following rules apply:

- Parenthesize the functions and operators of a given expression in order of descending priority. That is, first parenthesize all function calls (highest priority), parenthesize path name expressions (\ and \ \) (next highest priority), and so on.

- If an expression contains several occurrences of the same operator, then parenthesize those operators in the order indicated by their associativity.

When an operator is parenthesized, the parentheses surround the operator and its operands.

As an example of the application of these rules, consider the following query expression:

```
CALLING CALLED_BY x OR y AND NOT z
```

This expression contains two binary operators (OR and AND) and three function calls (CALLING, CALLED_BY, and NOT). There are many ways in which it could be explicitly parenthesized, including the following:

```
CALLING CALLED_BY(x) OR y AND NOT (z)
CALLING( CALLED_BY(x) ) OR y AND NOT(z)
CALLING( CALLED_BY(x) ) OR (y AND NOT(z))
(CALLING( CALLED_BY(x) ) OR (y AND NOT(z)))
```

4.4 Semantics

SCA evaluates a query expression as follows:

1. Evaluate the operands of the expression.
2. Calculate an expression by applying the operator. The value obtained from this step is the value of the expression.

The order in which SCA evaluates the operands of a query expression is not defined. Because query expressions have no side effect, the order of evaluation does not matter. Furthermore, expressions are not necessarily evaluated from the innermost to the outermost, but they are evaluated in a semantically equivalent way.

The value of a query expression is a collection of symbol occurrences and, possibly, relationships between occurrences. A query result that has information about relationships between occurrences is called a **structured query result**. A nonstructured collection of occurrences has no interoccurrence relationship information.

4.5 Attribute-Selection Expressions

An **attribute-selection** expression selects occurrences based on the setting of occurrence and symbol attributes. An **attribute-selection** expression has the following form:

```
attribute-selection-exp ::= [ attribute-name = ] actual-parameter
```

If no attribute name is specified, Name is assumed.

SCA supports the following types of attribute selection:

- Name
- Symbol class
- Symbol domain
- Occurrence class
- File specification

The rest of this section describes attribute selection in more detail.

4.5.1 Name Selection

A **name-selection** expression selects occurrences that have names matching a specified name expression.

A **name-selection** expression has the following form, where a **name-expression** is a string of characters, possibly including wildcards:

```
name-selection-exp ::= name-expression |
                       name=name-expression |
                       name=( name-expression,... )
```

A name expression that includes a wildcard character is equivalent to a union of all the names that match the **name-selection** expression. A list of name expressions is equivalent to a union of **name-selection** expressions, each having a single name expression. Given these rules, the following three examples are equivalent:

```
name=( namexp1, namexp2 )
```

```
name=namexp1 OR name=namexp2
```

```
namexp1 OR namexp2
```

When a string is enclosed in quotation marks, the string can contain any ASCII character except a quotation mark. If you want a quotation mark in such a string, it must be represented by two successive quotation marks. For example, to find the name x"y, specify the following:

```
FIND "x""y"
```

You can override the wildcard characters (% and *) using the ampersand (&). If you want an ampersand in a string, it must be represented by two successive ampersands. For example:

- Use the name expression &* to find the name consisting of a single asterisk.
- Use the name expression && to find the name consisting of a single ampersand.

Enclosing a complex string in quotation marks itself does not affect the case sensitivity of the matching. If case sensitivity has been turned off, string matching is not sensitive to the case of the string specified in the name expression. If case sensitivity has been turned on, string matching is sensitive to the case of the string specified, regardless of whether it is quoted.

Note that although a hyphen (-) is allowed in a simple name, a command line that ends in a hyphen is a continued command.

4.5.2 Symbol-Class Selection

A **symbol-class-selection** expression selects occurrences whose symbol class is one of those specified in the **symbol-class-selection** expression. A **symbol-class-selection** expression has the following form:

```
symbol-class-selection-exp ::= symbol=symbol-class |
                               symbol=( symbol-class,... )
```

The **symbol-class** is one of the following keywords:

- ARGUMENT—Formal argument (such as a routine argument or macro argument)
- CLASS—Any C++ class object construct defined by union, structure, or class statements
- COMPONENT, FIELD—Component of a record
- CONSTANT, LITERAL—Named compile-time constant value
- EXCEPTION—Exception
- FILE—File

- FUNCTION, PROCEDURE, PROGRAM, ROUTINE, SUBROUTINE—Callable program function
- GENERIC—Generic unit
- KEYWORD—Keyword (as defined in the LSE environment file for comment processing)
- LABEL—User-specified label
- MACRO—Macro
- MODULE, PACKAGE—Collection of logically related elements
- PLACEHOLDER—Marker where program text is needed
- PSECT—Program section
- TAG—Comment heading
- TASK—Task
- TYPE—User-defined type
- UNBOUND—Unbound name
- VARIABLE—Program variable
- OTHER—Any other class of symbol

You use one or more of the generic (multilanguage) keywords to request specific classes of symbols. Because different languages use different terminology, several alternatives are provided for some classes of symbols.

A list of symbol classes is equivalent to a union of **symbol-class-selection** expressions, each having a single symbol class.

4.5.3 Symbol Domain Selection

A **symbol-domain-selection** expression selects occurrences whose symbol domain is one of those specified in the **symbol-domain-selection** expression.

A symbol's domain is the range of source code in which the symbol has the potential of being used. For example, a C static declaration creates a symbol that has a module-specific symbol domain; it cannot be used outside of that module. On the other hand, a regular C module-level declaration creates a symbol that has a multimodule symbol domain; it has the potential of being used in more than one module. The symbol domain of a GLOBAL is multimodule regardless of how many modules there are in which the symbol is used.

A **symbol-domain-selection** expression has the following form:

```
symbol-domain-selection-exp ::= domain=symbol-domain |
                                domain=( symbol-domain,... )
```

The **symbol-domain** is one of the following keywords:

- INHERITABLE—Able to be inherited into other modules (for example, by means of Pascal environment or Ada compilation system mechanisms)
- GLOBAL—Known to multiple modules through linker global symbol definitions
- PREDEFINED—Defined by the language (examples: FORTRAN sin, Pascal writeln)
- MULTI_MODULE—Domain spans more than one module (domain=multi_module is equivalent to domain=(inheritable,global, predefined)
- MODULE_SPECIFIC—Domain is limited to one module
- INCLUDE_FILE—Symbol whose most significant declaration occurs in a source include file.

A list of symbol domains is equivalent to a union of **symbol-domain-selection** expressions, each having a single symbol domain.

4.5.4 Occurrence Selection

An **occurrence-selection** expression selects occurrences whose occurrence class is one of those specified in the **occurrence-selection** expression. An **occurrence-selection** expression has the following form:

```
occurrence-selection-exp ::= occurrence=occurrence-class |
                             occurrence=( occurrence-class,...)
```

The **occurrence-class** is one of the following keywords:

Declarations

- PRIMARY—Most significant declaration (such as FUNCTION)
- ASSOCIATED—Associated declaration (such as EXTERNAL)
- DECLARATION—Both PRIMARY and ASSOCIATED declarations

References

- READ, FETCH—Fetch of a symbol value:
  – DIRECT_FETCH
  – INDIRECT_FETCH

- WRITE, STORE—Assignment of a symbol value:
  - DIRECT_STORE
  - INDIRECT_STORE
- ADDRESS, POINTER—Reference to the location of a symbol:
  - DIRECT_ADDRESS
  - INDIRECT_ADDRESS
- CALL—Call to a routine or macro:
  - DIRECT_CALL
  - INDIRECT_CALL
- COMMAND_LINE—Command-line file reference
- INCLUDE, REQUIRE—Source-file include reference
- PRECOMPILED, ENVIRONMENT, LIBRARY—Precompiled file include reference
- BASE—Any base class of a C++ class
- FRIEND—Any friend of a C++ class
- MEMBER—Any member of a C++ class
- SEPARATE—Any Ada package or subprogram unit defined as SEPARATE
- USE—Any USE of an Ada package or subprogram unit, or USE of a HP Fortran 90 module
- WITH—Any WITH of an Ada package or subprogram unit
- REFERENCE—All of the previous references
- OTHER—Any other kind of reference (such as a macro expansion or use of a constant)

Other Occurrence Classes

- EXPLICIT—Explicitly declared
- IMPLICIT—Implicitly declared
- VISIBLE—Occurrence appears in the source
- HIDDEN—Occurrence does not appear in the source
- COMPILATION_UNIT—Occurrence is a compilation unit
- LIMITED—Any Ada limited private type

- PRIVATE—Any private C++ object, or Ada private type
- PROTECTED—Any protected C++ object
- PUBLIC—Any public C++ object
- VIRTUAL—Any virtual C++ object

4.5.5 File Specification Selection

A **file-specification-selection** expression selects occurrences whose source position is in one of the files specified in the **file-specification-selection** expression. A **file-specification-selection** expression has the following form:

```
file-spec-selection-exp ::= file_spec=name-expression |
                            file_spec=( name-expression,... )
```

The **name-expression** is a name expression that is interpreted as a file specification.

4.6 Operator Expressions

This section describes the operators you use with the SCA Query Language. The value of an operator expression is a set of occurrences and relationships. SCA query expression operators are similar to functions in high-level languages, such as Pascal and Ada. Operator expressions have an operator name, enclosed by two operands, which are query expressions. The result of an operator expression is a query-expression result.

4.6.1 Path-Name Expressions

A **path-name** expression identifies symbols based on the nesting of primary declarations. A **path-name** expression has the following form:

```
pathname-expression ::= exp1 \ exp2 |
                        exp1 \\ exp2
```

The path name operators (\ and \\) are special cases of the general CONTAINED_BY function. The expression **exp1 \ exp2** is equivalent to the following expression:

```
EXPAND CONTAINED_BY( exp1 AND occ=primary,
                     exp2 AND occ=primary,
                     result=begin,
                     depth=all,
                     trace="" )
```

The expression **exp1 \ \ exp2** is equivalent to the following expression:

```
EXPAND CONTAINED_BY( exp1 AND occ=primary,
                     exp2 AND occ=primary,
                     result=begin,
                     depth=all )
```

4.6.2 Intersection Expressions

An **intersection** expression identifies occurrences that exist in two different sets. The **intersection** expression has the following form:

```
intersection-expression ::= exp1 AND exp2
```

The value of this expression is a set containing each occurrence that exists in both sets (**exp1** and **exp2**); occurrences that appear in only one of the sets are not included.

4.6.3 Union Expressions

A **union** expression merges two sets. The **union** expression has the following form:

```
union-expression ::= exp1 OR exp2
```

The value of this expression is a set containing all occurrences that exist in either set (**exp1** or **exp2**).

4.6.4 Exclusive-Or Expressions

An **exclusive-or** expression selects the unique occurrences in two different sets. The **exclusive-or** expression has the following form:

```
exclusive-or-expression ::= exp1 XOR exp2
```

The value of this expression is a set containing all occurrences that exist in exactly one of the sets (**exp1** and **exp2**); occurrences that appear in both sets are not included.

4.7 Function-Call Expressions

The form of a **function-call** expression is as follows:

```
function-name( [actual-parameter],... )
```

A parameter list consists of zero, one, or more parameters. Each parameter has a data type and a default value. A data type is either a query expression, name expression, keyword list, range list, or number.

An actual parameter list can be empty if the function has no parameters or if you are using default values for all of the parameters. In this case, the **function-call** expression is written as function-name( ).

If the actual parameter list consists of exactly one parameter, the parentheses can be dropped. This form of the **function-call** expression is as follows:

```
function-name actual-parameter
```

4.7.1 Parameter Association

A function call must pass one actual parameter for each formal parameter. The actual parameter is either listed explicitly in the function call, or supplied by means of a default value.

One way to establish the correspondence between actual and formal parameters is to give the parameter in each list the same position. That is, the association of the actual and formal parameters proceeds from left to right, item by item, through both lists. This form of association is called **positional**.

Another way of establishing correspondence is to specify the formal parameter name and the actual parameter being passed to it. You can associate an actual parameter with a formal parameter by using the assignment operator (=). The actual parameters in the call do not have to appear in the same order that the formal parameters appeared in the declaration. This form of association is called **named**.

You can use both positional and named actual parameters in the same call. However, you must still supply at most one actual parameter for any formal parameter, and you must list the positional parameters first.

4.7.2 Negation Function

The negation function finds occurrences that do not match a query expression. The negation function has the following form:

```
FUNCTION NOT( query_expression : query-expression = * )
```

The result of a call to this function is a set containing all occurrences that are not contained in **query_expression**. Note that the expression NOT( ) evaluates to the empty set.

4.7.3 Expansion Function

The expansion function expands a set of occurrences to include all the occurrences of the symbols that correspond to the original occurrence set.

The expansion function has the following form:

```
FUNCTION EXPAND( query_expression : query-expression = * )
```

See the section Section 3.2.2 section in Chapter 3 section for an example of the expansion function.

4.7.4 Indicated Function

The indicated function is available only from within LSE. The indicated function matches the occurrence at which the cursor is pointing. The indicated function has the following form:

```
FUNCTION INDICATED
```

The indicated function has no parameters. Thus, a call to the indicated function must have the following form:

```
INDICATED()
```

An indicated function can be nested within other query expressions.

4.7.5 Query Usage Function

A query usage function incorporates the results of a previous query into a new query expression. The query usage function has the following form:

```
FUNCTION @( query_name : query-name = sca$current_query )
```

The value of this expression is that of the expression that is specified as **query_name**. The default query is the current query, SCA$CURRENT_QUERY.

4.7.5.1 The Current Query

The current query specifies the result of the previous query, or the one to which you are set using the NEXT QUERY, PREVIOUS QUERY, and GOTO QUERY commands. The name of the current query is SCA$CURRENT_QUERY. The current query is used as follows:

```
@sca$current_query
```

Because the @ function defaults to the current query, this expression can also be written as follows:

```
@()
```

The following is an example of a command sequence using the current query:

```
FIND *table*
FIND @sca$current_query AND symbol=routine
```

Alternatively, you can write the previous commands as follows:

```
FIND *table*
FIND @() AND symbol=routine
```

4.7.6 Relationship Functions

A relationship function selects occurrences based on relationships between occurrences. There are two kinds of relationship expressions: **structured-relationship** and **nonstructured-relationship**.

A **structured-relationship** expression selects both occurrences and relationships between them. A **structured-relationship** expression preserves relationship information in the value of the expression. The result of such an expression is a structured query result.

A **nonstructured-relationship** expression selects occurrences based on relationships between occurrences. A **nonstructured-relationship** expression uses information about the relationships between occurrences, but does not preserve relationship information in the result of the expression. The result of such an expression is a nonstructured query result.

All the relationship functions have the same set of parameters (see Table 4–5). A relationship function has the following form:

```
FUNCTION function-name( end : query-expression = *,
                        begin : query-expression = *,
                        depth : number = 1,
                        result : keyword-list = structure,
                        trace : query-expression = * )
```

4.7.6.1 Individual Relationship Functions

This section describes the individual relationship functions. There are two kinds of relationship functions: **basic functions** and **inverse functions**. You can transform every basic function into its corresponding inverse function by removing the ING at the end of the function and adding ED_BY.

For example, you can transform the basic function CALLING to the inverse function, CALLED_BY. The two commands, CALLING(y,x) and CALLED_BY(x,y), produce the same result: a graph of call relationships from X to Y.

Relationship Functions

- CALLING
  CALLED_BY

  The CALLING relationship finds those occurrences in *begin-exp* that are call occurrences in *end-exp*. Typically, if not exclusively, declarations in *begin-exp* call references in *end-exp*. The CALLED_BY relationship finds those occurrences in *begin-exp* that are called by occurrences in *end-exp*.

- CONTAINING
  CONTAINED_BY

  The CONTAINING relationship finds those occurrences in *begin-exp* that contain occurrences in *end-exp*. The CONTAINED_BY relationship finds those occurrences in *begin-exp* that contain occurrences in *end-exp*.

- TYPING
  TYPED_BY

  The TYPING relationship finds each occurrence in *begin-exp* that determines the type of an occurrence in *end-exp*. The TYPED_BY relationship finds each occurrence in *begin-exp* whose type is one of the occurrences in *end-exp*.

4.7.6.2 Relationship Parameters

Relationship parameters determine the precise semantics of a relationship expression. The following is a list of relationship parameters and how you use them.

END=end-expression
Specifies those occurrences at which the tracing of relationships can end. Only paths that end on one of these occurrences are included in the result. The default is **end=***.

BEGIN=begin-expression
Specifies those occurrences at which the tracing of relationships can begin. Only paths that begin on one of these occurrences are included in the result. The default is **begin=***.

DEPTH=depth-level
Specifies the number of levels of structure to be traced. The default depth-level is 1. **Depth=all** indicates that there is no limit to the number of levels of structure that are to be traced.

RESULT=result-keyword-list
In this syntax, **result-keyword-list** is one or more of the following keywords:

- [NO]STRUCTURE—Indicates whether relationship information is to be preserved in the query result.
- ANY_PATH—Indicates that any path that traces from the **begin-expression** to the **end-expression** will satisfy the query. SCA will return the first complete paths it finds. By default, all such paths are returned.
- BEGIN—Indicates that only those occurrences that begin the relationship graph are to be included in the result. **RESULT=BEGIN** implies a NOSTRUCTURE result.
- END—Indicates that only those occurrences that end the relationship graph are to be included in the result. **RESULT=END** implies a NOSTRUCTURE result.

The default is **RESULT=STRUCTURE**.

TRACE=trace-expression
A query expression whose result specifies those occurrences through which relationship tracing is to be continued. The **trace** parameter does not affect the first iteration. That first iteration is controlled by the **begin** parameter. The default is **trace=***.

4.7.7 The IN Function

The IN function restricts a set of occurrences to those occurrences that are directly or indirectly contained by another set of occurrences.

The IN function has the following form:

```
FUNCTION IN( end : query-expression = *,
             begin : query-expression = * ) =
```

The IN function is a special case of the CONTAINED_BY function. It has been included in the set of predefined functions because it provides a particularly useful subset of the capabilities of the CONTAINED_BY function. For example:

```
IN (end=x, begin=y)
```

The expression is equivalent to the following:

```
CONTAINED_BY (end=x and DECLARATION=PRIMARY,
              begin=y,
       result=begin,
       depth=all)
```

4.8 Abbreviation Rules

You can abbreviate attribute-selection formal parameter names and attribute-selection actual parameter keywords to their first four characters. You can truncate these names and keywords to fewer characters as long as the truncation is unique. For example, the symbol-class-selection attribute name is the only such name that begins with S. Therefore, you can abbreviate the **symbol** attribute name to just one character.

Attribute-selection, actual parameter keywords work the same way.

Special considerations apply when you use these names and keywords in command procedures. To ensure readability, you should not abbreviate at all. If you do abbreviate, never abbreviate to fewer than four characters, or you risk the possibility that your command procedure might not be compatible with future releases of SCA.

You can abbreviate only attribute-selection formal parameter names and attribute-selection actual parameter keywords.

Index