VMS Software

VSI OpenVMS

Command Definition, Librarian, and Message Utilities

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS x86-64 Version 9.2-1 or higher

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

Command Definition, Librarian, and Message Utilities

Legal Notice

Preface ..... **v**
1. About VSI v
2. Intended Audience v
3. Document Structure v
4. Related Documents vi
5. VSI Encourages Your Comments vi
6. OpenVMS Documentation vi
7. Typographical Conventions vi

Chapter 1. Command Definition Utility **1**
1.1. Command Processing 1
1.1.1. Command String Components 1
1.1.2. System and Process Command Tables 2
1.2. Using CDU 2
1.3. Choosing a Table 2
1.3.1. Modifying Your Process Command Table 3
1.3.2. Adding a System Command 3
1.3.3. Creating an Object Module 4
1.4. Writing a Command Definition File 4
1.4.1. Defining Syntax 5
1.4.2. Defining Values 6
1.4.2.1. Built-In Value Types 6
1.4.2.2. User-Defined Keywords 7
1.4.3. Defining Command Verbs 8
1.4.4. Disallowing Entity Combinations 8
1.4.4.1. Specifying Expression Entities 9
1.4.4.2. Operators 12
1.4.5. Identifying Object Modules 13
1.5. Processing Command Definition Files 14
1.5.1. Adding Command Definitions to a Command Table 14
1.5.2. Deleting Command Definitions 14
1.5.3. Creating Object Modules 15
1.5.4. Creating New Command Tables 15
1.6. Using Command Language Routines 16
1.6.1. CDU Usage Summary 16
1.6.2. CDU File Statements 17
1.6.3. CDU Qualifiers 32
1.6.4. CDU Examples 38

Chapter 2. Librarian Utility **43**
2.1. LIBRARIAN Description 43
2.1.1. Types of Libraries 43
2.1.2. Structure of Libraries 44
2.1.3. Character Case of Library Keys 45
2.1.4. Shareable Image Libraries 45
2.1.5. Help Libraries 46
2.1.5.1. Creating Help Files 46
2.1.5.2. Formatting Help Files 47
2.1.5.3. Help Text Example 48
2.1.5.4. Retrieving Help Text 50
2.1.6. Using the Librarian Utility to Save Disk Space 51
2.1.7. Librarian Utility (LBR) Routines 52
2.1.8. LIBRARIAN Usage Summary 52

2.1.9. LIBRARIAN Qualifiers 54
Chapter 3. Message Utility 77
3.1. MESSAGE Description 77
3.1.1. Message Format 77
3.1.2. Constructing Messages 78
3.1.2.1. The Message Source File 78
3.1.2.2. Compiling the Message Source File 79
3.1.2.3. Linking the Message Object Module 79
3.1.3. Using Message Pointers 80
3.1.4. The SET MESSAGE Command 81
3.1.5. Message Source Files 82
3.1.6. MESSAGE Usage Summary 83
3.1.7. MESSAGE Qualifiers 84
3.1.8. MESSAGE Commands 87
3.1.9. MESSAGE Examples 97

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for programmers and general users of the OpenVMS operating system.

3. Document Structure

This manual is divided into three parts.

Chapter 1 describes the Command Definition Utility (CDU) and consists of the following sections:

- Description—Provides a full description of CDU.
- Usage Summary—Outlines the following information:
  - Invoking the utility
  - Exiting from the utility
  - Directing output
  - Restrictions or privileges required
- File Statements—Describes the statements used in building command definition files, including statement formats, parameters, and examples.
- Qualifiers—Describes qualifiers, including format, parameters, and examples.
- Examples—Provides additional CDU examples.

Chapter 2 describes the Librarian utility (LIBRARIAN) and consists of the following sections:

- Description—Provides a full description of LIBRARIAN.
- Usage Summary—Outlines the following information:
  - Invoking the utility
  - Exiting from the utility
  - Directing output
- Qualifiers—Describes qualifiers, including format, parameters, and examples.

Chapter 3 describes the Message utility (MESSAGE) and consists of the following sections:

- Description—Provides a full description of MESSAGE.

- Usage Summary—Outlines the following information:
  - Invoking the utility
  - Exiting from the utility
- Qualifiers—Describes qualifiers, including format, parameters, and examples.
- Commands—Describes source file statements, including format, parameters, and examples.
- Examples—Provides additional examples for using message files and pointer files.

4. Related Documents

For related information about these utilities, refer to the following documents:

- *VSI OpenVMS DCL Dictionary*
- *VSI OpenVMS Linker Utility Manual*

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

6. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

7. Typographical Conventions

The following conventions are used in this manual:

Convention	Meaning
Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key (*x*) or a pointing device button.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.

Convention	Meaning
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the name of an argument, an attribute, or a reason. Bold type also represents the introduction of a new term.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
–	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated.

Chapter 1. Command Definition Utility

CDU Description

The Command Definition Utility (CDU) creates, deletes, or changes command definitions in a command table. Command tables are data structures created by CDU and used by the command language interpreter (CLI) to parse and evaluate DIGITAL Command Language (DCL) commands.

There are two types of command tables: system command tables used to parse system commands and process command tables used to parse process-specific commands. CDU creates command tables from command definition files, from existing command tables, or from a combination of these sources. The new table can be either executable code or an object module.

The following sections describe:

- How DCL processes commands
- How to write command definitions
- How to modify command tables
- How to process command definitions
- How to use command language routines in programs

1.1. Command Processing

To write command definitions and modify command tables, you must understand how the DCL command interpreter processes commands. The process begins when DCL prompts you for a command and you enter an appropriate command string. Then DCL processes the command string from left to right using definitions in your process command table. Your process command table contains a list of valid commands and their attributes.

To parse a command string, DCL calls the CLI$DCL_PARSE routine to check each entity in the command string. If each entity is valid, DCL sets up an internal representation of the command string. Then DCL uses the CLI$DISPATCH routine to invoke the image or routine that executes the command. If the command string is not valid, DCL issues an error message.

The image or routine that executes a command must call the CLI$PRESENT and CLI$GET_VALUE routines to get information about the entities that were present in the command string. The image or routine uses this information to determine how to execute the command.

1.1.1. Command String Components

A command string can contain a **verb** that specifies the command to be executed, a **parameter** that specifies the verb object, and a **qualifier** that describes or modifies the action taken by the verb.

The DCL command definitions describe the allowable parameter values for each command. The command definitions also indicate whether or not qualifiers can take values and the value types that can be specified. Examples of qualifier values include file specifications, integer values, keywords, and

character strings. Some commands (SET and SHOW) accept keywords as parameters. A keyword is a predefined string that can be used as a value for a parameter, qualifier, or another keyword.

The following example illustrates the components of a DCL command string:

```
$ DIFFERENCES/MODE=ASCII MYFILE.DAT YOURFILE.DAT
```

DIFFERENCES is the verb and /MODE is a qualifier that has as its value the keyword ASCII. MYFILE.DAT and YOURFILE.DAT are file specifications that function as the command parameters.

The next example shows a command that uses a keyword as a parameter value:

```
$ SHOW DEFAULT
```

Here, SHOW is the verb and DEFAULT is a keyword used as a parameter.

1.1.2. System and Process Command Tables

When you log in, the system command table in SYS$LIBRARY:DCLTABLES.EXE is copied to your process and DCL uses this process command table to parse command strings. Changing your process command table does not affect SYS$LIBRARY:DCLTABLES.EXE. To change the DCL tables, you need the CMKRNL privilege.

The system command table is created from source files called command definition files. A command definition file contains statements that name and describe verbs. VSI maintains the command definition files for DCL; they are not shipped with your system.

1.2. Using CDU

To use CDU:

- Determine which table you want to create or modify. In general, you modify your process command table or the DCL table in SYS$LIBRARY, or you create an object module for a new table.
- Choose a name and syntax for the command you define. Use a text editor to create a command definition file that defines the command.
- Use the DCL command SET COMMAND to add your command definition to the appropriate command table. You can modify your process command table or a specified command table file. You can also create an object module from your command definition file.
- Write the code for the image or routine that is invoked by the command you are adding to the command table.

Note that the foreign command facility is an alternate way to define command verbs. The foreign command allows you to pass information about a command string to an image. However, if you use the foreign command facility, your program must parse the command string; DCL does not parse the command string for you. For information about how to define a foreign command, see the *VSI OpenVMS User's Manual*.

1.3. Choosing a Table

The type of table you are modifying or creating affects the way that you write a command definition, process this definition, and write the code that executes your command.

The most common tables that you modify or create include your process command table, the DCL table in SYS$LIBRARY, and new tables that allow your programs to process commands.

1.3.1. Modifying Your Process Command Table

To add a command to your process command table, define the new command in a command definition file, specifying the name of an image for the command to invoke. Then use SET COMMAND to add the new command to your process command table and to copy the new table back to your process. For example, the following command adds a command in NEWCOMMAND.CLD to your process command table:

```
$ SET COMMAND NEWCOMMAND.CLD
```

Now you can enter the new command after the DCL prompt. DCL will parse the command and then invoke the image that executes the command. Note that, when you write the source code for the new command, you must use the command language routines CLI$PRESENT and CLI$GET_VALUE to obtain information about the command string. Refer to the *VSI OpenVMS Utility Routines Manual* for additional information.

The first example in the Section 1.6.4 section shows how to add a new command to your process command table and how to write the program that executes the new command.

To make the command in NEWCOMMAND.CLD available to you each time you log in, include the SET COMMAND command in your LOGIN.COM file.

1.3.2. Adding a System Command

Following are the instructions to add a command to the DCL command table in SYS$LIBRARY:

- Define the command in a command definition file, specifying the name of an image for the command to invoke.
- Use SET COMMAND to add the new definition to the DCL command table and copy the new table back to SYS$LIBRARY:

  ```
  $ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES.EXE -
  _$ /OUTPUT=SYS$COMMON:[SYSLIB]DCLTABLES.EXE NEWCOMMAND.CLD
  ```

- To make the new command available to other users, use the INSTALL utility:

  ```
  $ INSTALL REPLACE SYS$LIBRARY:DCLTABLES.EXE
  ```

- To make the new command available to existing interactive processes, you can log out and log in again, or execute the following command:

  ```
  $ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES.EXE
  ```

Note

To ensure that the modified tables are written to the cluster common root, the output file specification is: SYS$COMMON:[SYSLIB]. This ensures that the new command is available to all systems sharing the same system disk. This also avoids potential problems with future changes to the command tables due to copies of DCLTABLES being present in the SYS$SPECIFIC:[SYSLIB] and SYS$COMMON:[SYSLIB] areas referenced by SYS$LIBRARY:

To locate potentially errant copies of the command tables, use the following command:

```
$ DIR SYS$SPECIFIC:[SYSLIB]DCLTABLES.EXE
```

1.3.3. Creating an Object Module

To create an object module for a new command table, define the commands in a command definition file, specifying the name of a routine in a program that executes the command. Then use SET COMMAND with the /OBJECT qualifier to create an object module from the command definition file. For example:

```
$ SET COMMAND/OBJECT NEWCOMMAND
```

Now link this object module with the program that uses the table. Note that, when you link a command table with your program, the program must perform the functions of a command interpreter. That is, the program must obtain the command string and call the parsing routine CLI$DCL_PARSE to verify and create an internal representation of it. The program must also call CLI$DISPATCH to invoke the appropriate routine. Each command routine must use the DCL interface routines CLI$PRESENT and CLI$GET_VALUE to get information about the command string that invoked the routine.

The second example in the Section 1.6.4 section shows how to write and process command definitions for an object module and how to write a program that parses commands and invokes routines.

1.4. Writing a Command Definition File

A command definition file contains information that defines a command and its parameters, qualifiers, and keywords. In addition, the command definition file provides information about the image or routine that is invoked after the command string is successfully parsed.

Use a text editor to create a command definition file that contains the statements you need to describe your new command; you can use clauses to specify additional information for statements. The default file type for a command definition file is .CLD.

Use exclamation points to delimit comments. An exclamation point causes all characters that follow it on a line to be treated as comments.

Any statement and its clauses can be coded using several lines. No continuation character is necessary. (However, you cannot split names across two lines.) If you place a statement on one line, you can separate clauses in the statement with either commas or spaces.

You cannot abbreviate statement or clause names in the command definition language. All names (for example, DEFINE SYNTAX, PARAMETER, and so on) must be spelled out completely.

Most statements and clauses accept user-supplied information such as verb names, qualifier names, image names, and so on. You can specify this information as a symbol or as a string.

If the statement requires that a term be specified as a string, enclose the term in quotation marks. A string can contain any alphanumeric or special characters. To include quotation marks within a string, use two quotation marks (""). For example, PARAMETER P1, LABEL=PORT, PROMPT="Enter ""one"" value" produces the following:

```
Enter "one" value
```

Note

To maintain compatibility with earlier releases, CDU accepts character strings that are not enclosed in quotation marks. However, VSI recommends that you surround character strings in quotation marks. If you do not enclose a string in quotation marks, all alphabetic characters are converted to uppercase characters (capital letters).

If a statement requires that a term be specified as a symbol, do not enclose the term in quotation marks. A symbol name must start with a letter or a dollar sign. It can contain from 1 to 31 letters, numbers, dollar signs, and underscore characters.

The Command Definition Language includes the following statements:

- DEFINE SYNTAX syntax-name [verb-clause[,...]]
- DEFINE TYPE type-name [type-clause[,...]]
- DEFINE VERB verb-name [verb-clause[,...]]
- IDENT ident-string
- MODULE module-name

The following sections provide an overview of each CDU statement. See the Section 1.6.2 section for more detailed descriptions of each type of statement.

1.4.1. Defining Syntax

The DEFINE SYNTAX statement allows a command verb to use alternative syntax depending on the parameters, qualifiers, and keywords that are present in the command string. It redefines the syntax for a command verb previously defined by a DEFINE VERB or DEFINE TYPE statement, or it can be used to redefine the syntax for a command verb *redefined* by a previous DEFINE SYNTAX statement.

To define a syntax change, you must provide two DEFINE statements: a primary DEFINE statement and a secondary DEFINE statement. The primary DEFINE statement defines the affected command verb and it must include a SYNTAX=syntax-name verb clause to point to the secondary DEFINE statement. The secondary DEFINE statement defines the alternate syntax.

For example, you can write a command definition that uses a different syntax for a command verb when a particular qualifier is explicitly present, that is, not by default. When you include the specified qualifier in the command string, the syntax defined in the secondary DEFINE statement applies to the command verb described by the primary DEFINE statement.

This is the format for the DEFINE SYNTAX statement:

```
DEFINE SYNTAX syntax-name [verb-clause,[,...]]
```

The **syntax-name** verb clause is the name of the alternate syntax definition. The verb clause specifies additional information about the syntax. You can use the same verb clauses in a DEFINE SYNTAX statement as are allowed in a DEFINE VERB statement, with one exception: you cannot use the SYNONYM verb clause with DEFINE SYNTAX.

The following example shows how a syntax change is used to specify an alternate command syntax when the /LINE qualifier is specified:

```
DEFINE VERB ERASE
         IMAGE "DISK1:[MYDIR]ERASE"
         QUALIFIER SCREEN
         QUALIFIER LINE, SYNTAX=LINE ❶
DEFINE SYNTAX LINE ❷
         IMAGE "DISK1:[MYDIR]LINE"
         QUALIFIER NUMBER, VALUE(REQUIRED)
```

❶ The DEFINE VERB statement defines the verb ERASE. This verb accepts two qualifiers, /SCREEN and /LINE. The qualifier /LINE uses an alternate syntax, specified with the SYNTAX=LINE clause. If you enter the command ERASE/LINE, the definitions in the DEFINE SYNTAX LINE statement override the definitions in the DEFINE VERB ERASE statement. However, if you enter the command ERASE/SCREEN or if you do not specify any qualifiers, the definitions in the DEFINE VERB ERASE statement apply.

❷ The DEFINE SYNTAX statement defines an alternate syntax called LINE. If you enter the command ERASE with the /LINE qualifier, the image DISK1:[MYDIR]LINE.EXE is invoked. The new syntax allows the qualifier /NUMBER, which requires a value.

1.4.2. Defining Values

To define values for parameters, qualifiers, or keywords, use the VALUE clause. When you use the VALUE clause, you can further define the value type with the TYPE clause.

With the TYPE clause, you can specify that a value type must be a built-in type (for example, a file specification), or you can specify that a value must be a user-defined keyword. Section 1.4.2.1 lists the built-in value types; Section 1.4.2.2 describes how to specify a user-defined keyword.

When you use the VALUE clause and do not define a value type, DCL processes the value in the following way. If the value is not enclosed in quotation marks, then DCL converts letters to uppercase and compresses multiple spaces and tabs to a single space. If the value is enclosed in quotation marks, then DCL removes the quotation marks, preserves the case of letters, and does not compress tabs and spaces. To include quotation marks within a quoted string, use two contiguous quotation marks ("") in the place you want the quotation marks to appear.

1.4.2.1. Built-In Value Types

The Command Definition Language provides the following built-in value types:

$ACL	The value must be an access control list.
$DATETIME	The value must be an absolute time or a combination time. DCL converts truncated time values, combination time values, and keywords for time values (such as TODAY) to absolute time format. DCL fills blank date fields from the current system date and fills omitted time fields with zeros.
$DELTATIME	The value must be a delta time. DCL fills missing fields with zeros.
$EXPRESSION	The value must be a DCL-style expression. DCL evaluates the expression and provides the results.
$FILE	The value must be a valid file specification.
$NUMBER	The value must be an integer represented by either decimal, octal, or hexadecimal numbers.
$PARENTHESIZED_VALUE	The value must be enclosed in parentheses. Note that DCL does not remove the parentheses.

$QUOTED_STRING	The value must be a string enclosed in quotation marks. Note that DCL does not remove the quotation marks.
$REST_OF_LINE	DCL treats the rest of the line literally as the specified value, ignoring spaces or punctuation marks. DCL does not remove quotation marks when processing the string.

The following example shows a parameter that must be specified as a file specification:

```
DEFINE VERB PLAY
        IMAGE "DISK1:[MYDIR]PLAY"
        PARAMETER P1, VALUE(TYPE=$FILE)
```

1.4.2.2. User-Defined Keywords

The DEFINE TYPE statement defines keywords that are acceptable for use as values for various command entities, including parameters, qualifiers, or other keywords.

To indicate that a command entity requires a keyword, use a VALUE clause of the following form in a definition statement:

```
DEFINE SYNTAX VALUE(TYPE=type-name)
```

The **type-name** points to the DEFINE TYPE statement that specifies the allowable keywords for the entity.

This is the format for the DEFINE TYPE statement:

```
DEFINE TYPE type-name [type-clause[,...]]
```

The **type-name** is the name of the keyword list, and the **type-clause** lists the acceptable keywords. Each type clause begins with the keyword KEYWORD, followed by one or more keywords that can be used with the parameter, qualifier, or keyword that references the keyword list. The next example includes two type-clauses in the DEFINE TYPE statement:

```
KEYWORD FAST, DEFAULT
KEYWORD SLOW
```

The following example illustrates the use of a DEFINE TYPE statement in conjunction with a DEFINE VERB statement:

```
DEFINE VERB SKIM ❶
        IMAGE "USER:[TOOLS]SKIM"
        QUALIFIER SPEED, VALUE(TYPE=SPEED_KEYWORDS) ❷

DEFINE TYPE SPEED_KEYWORDS ❸
        KEYWORD FAST, DEFAULT
        KEYWORD SLOW
```

❶ The DEFINE VERB statement defines a verb, SKIM, which invokes the image [TOOLS]SKIM.EXE and accepts the qualifier /SPEED.

❷ The VALUE clause indicates that the /SPEED qualifier accepts a list of keywords as defined by the DEFINE TYPE SPEED_KEYWORDS statement.

❸ The DEFINE TYPE statement lists the keywords that can be used with the /SPEED qualifier; you can specify SKIM/SPEED=FAST or SKIM/SPEED=SLOW. If you specify the /SPEED qualifier without a value, the default is FAST.

1.4.3. Defining Command Verbs

The DEFINE VERB statement defines a new command verb and specifies its characteristics. You can define any number of verbs in a single command definition file.

The format for the DEFINE VERB statement is as follows:

```
DEFINE VERB verb-name [verb-clause[,...]]
```

The verb name is the name of the command. A verb clause specifies additional information about the verb. Verb clauses can appear in any order in the command definition file. Verb clauses are optional.

You can specify the following verb clauses:

DISALLOW	Controls the use of an entity or a combination of entities.
NODISALLOWS	Permits all entities and entity combinations.
IMAGE	Specifies an image to be invoked by the verb.
PARAMETER	Defines a command parameter.
NOPARAMETERS	Disallows parameters.
QUALIFIER	Defines a command qualifier.
NOQUALIFIERS	Disallows qualifiers.
ROUTINE	Specifies a routine to be invoked by the verb.
SYNONYM	Specifies a verb synonym.

The following example illustrates a DEFINE VERB statement:

```
DEFINE VERB SEARCH ❶
          IMAGE "SEARCH" ❷
          PARAMETER P1, LABEL=SOURCE, PROMPT="File", VALUE(REQUIRED) ❸
```

❶ The DEFINE VERB statement names the verb SEARCH.
❷ The IMAGE verb clause identifies the image to be invoked at run time.
❸ The PARAMETER verb clause defines the first parameter to appear after the verb in the command string. LABEL, PROMPT, and VALUE are parameter clauses that further define the parameter. LABEL defines a name that the image uses to refer to the parameter. PROMPT indicates the prompt string to be issued if you do not specify the parameter in the command string. VALUE uses the REQUIRED clause to indicate that the parameter must be present in the command string.

1.4.4. Disallowing Entity Combinations

When you define a verb, you can use the DISALLOW verb clause to selectively disallow the use of one or more entities with the verb.

The DISALLOW verb clause has the following format:

```
DISALLOW expression
```

The **expression** in the clause specifies the disallowed entities and you can use any of the various logical operators (exclusive-OR, AND, OR, and so forth) to define them. When a command string is parsed, each entity in the expression is tested to determine if the entity is present (true) or absent (false). If an entity is present by default but is not explicitly present in the command string, the entity is evaluated as absent (false).

After each entity is evaluated, the logical operations are performed. If the result is true, the command string is disallowed. If the result is false, the command string is valid.

For example, a command definition might contain a DEFINE VERB statement that defines the verb SPORTS with three qualifiers: /TENNIS, /BOWLING, and /BASEBALL. However, you might want to make the qualifiers mutually exclusive. The following example shows how to use the DISALLOW verb clause to put this restriction into the command definition file:

```
DEFINE VERB SPORTS
          IMAGE "DISK3:[WILSON]SPORTS"
          QUALIFIER TENNIS
          QUALIFIER BOWLING
          QUALIFIER BASEBALL
          DISALLOW ANY2(TENNIS, BOWLING, BASEBALL)
```

The DISALLOW verb clause indicates that a command string is invalid if it contains more than one of the qualifiers /TENNIS, /BOWLING, or /BASEBALL.

Note that, when you specify any entity in a DISALLOW expression, the search context is the entire command string. Therefore, local qualifiers are treated as if they were global. The following example shows the global context of the search:

```
DEFINE VERB TEST
          IMAGE "DISK3:[WORK]TEST"
          PARAMETER P1
          PARAMETER P2
          QUALIFIER QUAL1
          QUALIFIER QUAL2,PLACEMENT=LOCAL
          QUALIFIER QUAL3,PLACEMENT=LOCAL
          DISALLOW P1 AND QUAL1
          DISALLOW QUAL2 AND QUAL3
```

Thus, the following two commands would be disallowed:

```
TEST P1 P2/QUAL1

TEST P1/QUAL2 P2/QUAL3
```

The global search context applied to local qualifiers is used only with DISALLOW processing, not with normal command parsing.

1.4.4.1. Specifying Expression Entities

When you specify entities in an expression, you need to uniquely identify the entities that are disallowed. You can specify an entity using one of the following:

- A parameter, qualifier, or keyword name or label
- A keyword path
- A definition path

Names and Labels

You can refer to a parameter or qualifier using its name or label if the entity is defined in the current definition. To refer to a keyword, you can use its name or label if the keyword is in a keyword path that

starts from the current definition, and if the keyword name or label is unique. (See the next subsection for more information about keyword paths.)

If the LABEL=label-name keyword is used to assign a label to an entity, use the label name to refer to the entity. Otherwise, use the entity name.

The following example disallows combinations of entities:

```
DEFINE VERB COLOR
          IMAGE "WORK:[JUDY]COLOR"
          QUALIFIER RED
          QUALIFIER BLUE
          QUALIFIER GREEN, VALUE(TYPE=GREEN_AMOUNT)
          DISALLOW RED AND ALL
          DISALLOW BLUE AND ALL

DEFINE TYPE GREEN_AMOUNT
          KEYWORD ALL
          KEYWORD HALF
```

In this example, you can use the qualifier names RED and BLUE in the DISALLOW verb clause because both names are used in the current definition. You can use the keyword ALL because it is in a keyword path that starts within the current definition (the TYPE=GREEN_AMOUNT qualifier clause starts the path) and the keyword name is unique.

The DISALLOW clauses indicate that the following command strings are not valid:

```
$ COLOR/RED/GREEN=ALL
$ COLOR/BLUE/GREEN=ALL
```

To refer to a parameter or qualifier in another definition or to refer to a keyword whose path begins in another DEFINE statement, you must use a definition path.

Keyword Paths

A keyword path provides a way to uniquely identify a keyword. You can refer to a keyword using a keyword path if the keyword is in a path that starts from the current definition and if the keyword name or label is not unique. You can also use a keyword path if the same keyword can be used with more than one parameter or qualifier.

A keyword path contains a list of entity names or labels that are separated by periods. The first name in a keyword path is the name (or label) of the first entity that references the keyword's value type definition. A keyword path can contain up to eight names (the first parameter or qualifier definition, plus seven DEFINE TYPE keyword definitions).

If a keyword is assigned a label name, use the label name in the keyword path. Otherwise, use the keyword name. You can omit names that are not needed to resolve a keyword reference from the beginning of a keyword path. However, you must include enough names to uniquely reference the keyword.

The following command string illustrates a situation that requires keyword paths to uniquely identify keywords. In this command string, you can use the same keywords with more than one qualifier. (In the command definition file, two qualifiers refer to the same DEFINE TYPE statement.)

```
$ NEWCOMMAND/QUAL1=(START=5,END=10)/QUAL2=(START=2,END=5)
```

The keyword path QUAL1.START identifies the keyword START when it is used with QUAL1; the keyword path QUAL2.START identifies the keyword START when it is used with QUAL2. The name START is an ambiguous reference if used alone.

To disallow use of the keyword QUAL1.START when a third qualifier (QUAL3) is present, use the following line in the command definition file:

```
DISALLOW QUAL1.START AND QUAL3
```

Although you cannot use QUAL1.START when QUAL3 is present, you can still use QUAL2.START with QUAL3.

The following example contains a keyword (ALL) that appears in two DEFINE TYPE statements:

```
DEFINE VERB COLOR
       IMAGE "WORK:[JUDY]COLOR"
       QUALIFIER RED, VALUE(TYPE=RED_AMOUNT)
       QUALIFIER GREEN, VALUE(TYPE=GREEN_AMOUNT)
       DISALLOW RED AND GREEN.ALL
       DISALLOW GREEN AND RED.ALL

DEFINE TYPE RED_AMOUNT
       KEYWORD ALL
       KEYWORD MIXED

DEFINE TYPE GREEN_AMOUNT
       KEYWORD ALL
       KEYWORD HALF
```

In this example, you must use the keyword path RED.ALL to refer to the ALL keyword when it is used in the value type definition RED_AMOUNT; you must use the keyword path GREEN.ALL to refer to the ALL keyword when it is used in the value type definition GREEN_AMOUNT.

Definition Paths

A definition path links a syntax definition to an entity that is defined in another DEFINE statement. For example, a definition path is needed when a syntax definition provides new disallow clauses for parameters or qualifiers that are defined in a primary definition.

A definition path has the following format:

```
<definition-name>entity-spec
```

The definition name is the name of the DEFINE statement where the entity is defined or the keyword path begins. The entity specification can be an entity name, a label, or a keyword path. The angle brackets are required.

For example:

```
DISALLOW <SKIP>FIRST
```

This clause disallows a command string if the entity FIRST (specified in the DEFINE VERB statement for the command verb SKIP) is present.

The next example uses a keyword path and a definition path:

```
DISALLOW <FILE>BILLS.ELECT AND GAS
```

This clause disallows a command string if the entity described by the keyword path BILLS.ELECT (which originates in the DEFINE VERB statement for the command verb FILE) is present.

CDU does not check a definition path to determine whether the path refers to an entity that is valid in a given context. If you use a definition path to specify an entity that is not valid in a particular context, results are unpredictable. For example, if you try to disallow the qualifier NOTES in the DEFINE SYNTAX statement, the entity NOTES would not be recognized as valid because the path to BILL_TYPES is not established in the DEFINE VERB statement for the command verb READ.

```
DEFINE VERB FILE
        QUALIFIER BILLS, SYNTAX=BILL_TYPES
        QUALIFIER RECEIPTS

DEFINE VERB READ
        QUALIFIER NOTES

DEFINE SYNTAX BILL_TYPES
        DISALLOW <READ>NOTES
```

Although the DISALLOW clause correctly identifies an entity in the command definition file, this entity is not valid in the DEFINE SYNTAX statement. However, the clause DISALLOW `<FILE>RECEIPTS` is valid in the DEFINE SYNTAX statement. The DEFINE SYNTAX statement inherits the qualifier RECEIPTS from the primary DEFINE statement (FILE) because no qualifiers are specified.

See the description of the DEFINE SYNTAX statement in the Section 1.6.2 section for more information about how entities are inherited by DEFINE SYNTAX statements.

1.4.4.2. Operators

A command definition can include one or more expressions of the relationship between an action verb and one or more objects of the verb (entities) that can be qualifiers, parameters, or keywords in various combinations. For example, the following expression states that the command is disallowed if it contains *both* of the previously defined qualifiers SINCE and BEFORE:

```
DISALLOW SINCE AND BEFORE
```

The logical operator AND stipulates that the command is invalid only when both qualifiers are present. When an expression contains logical operators, the operators are evaluated after the related command entities are determined to be present (logical true) or absent (logical false). If the *result* of the expression is true (that is, if both qualifiers are present), the command is disallowed. Conversely, if the result is false (one or none of the qualifiers is present), the command is accepted.

Table 1.1 shows the operators you can use in command definition expressions and the order in which CDU evaluates these operators. The highest precedence value is 1. When an expression contains two or more operators of equal precedence, CDU evaluates the leftmost operator first.

Table 1.1. Summary of CDU Operators

Operator	Precedence	Meaning
ANY2	1	True if any two or more of the entities listed are present
NEG	1	True if the negated form of the entity is present
NOT	1	True if the entity is not present or if an entity is present by default
AND	2	True if both entities are present

Operator	Precedence	Meaning
OR	3	True if either entity is present

The following example shows how to use the AND operator:

```
DISALLOW TERMINAL AND PRINTER
```

This statement disallows the command string if both entities (TERMINAL and PRINTER) are present.

You can use parentheses to override the order in which operations are evaluated; operations within parentheses are evaluated first. For example:

```
DISALLOW FAST AND (SLOW OR STILL)
```

The parentheses force the OR operator to be evaluated before the AND operator. Therefore, if the result of SLOW OR STILL is true, and if FAST is present in the command string, then the string is disallowed.

1.4.5. Identifying Object Modules

Use the MODULE and IDENT statements to provide identifying information if your command definition file is to create an object module. (You can create an object module from a command definition file with the command SET COMMAND/OBJECT. The object module contains a command table that you can link with your program.)

The MODULE statement assigns a symbolic name to the object module containing the command table. This is the format for the MODULE statement:

```
MODULE module-name
```

The **module-name** is the symbolic name for the object module.

The IDENT statement provides additional information in a quoted string format to identify the module. Typically, this might be the date the module was created or the name of the creator. This is the format for the IDENT statement:

```
IDENT ident-string
```

The **ident-string** is a quoted string having up to 31 characters.

The following sample command definition file illustrates the use of the MODULE and IDENT statements:

```
MODULE TABLE  ❶
IDENT "Updated 4/15/92"  ❷

DEFINE VERB SAVE    ❸
      ROUTINE SAVE_ROUT

DEFINE VERB GET    ❸
      ROUTINE GET_ROUT
```

❶ The MODULE statement assigns the name TABLE to the command table that CDU creates when you use the command SET COMMAND/OBJECT to develop an object module for the new command.

❷ The IDENT statement provides additional identifying information. In this example, it shows the date when the command definition file was updated.

❸ The DEFINE VERB statements define command verbs that can be used by the main program to invoke appropriate routines.

1.5. Processing Command Definition Files

A command definition file must be translated into an executable command table before the commands in the table can be parsed and executed. To perform this translation, use the DCL command SET COMMAND to invoke the Command Definition Utility.

The command SET COMMAND has the following modes:

SET COMMAND/DELETE	Deletes command definitions from a command table
SET COMMAND/OBJECT	Creates an object file from a command definition file
SET COMMAND/REPLACE	Adds or replaces definitions in a command table using definitions from a command definition file

The /DELETE, /OBJECT, and /REPLACE qualifiers are mutually exclusive; you can use only one SET COMMAND mode in a command string. In addition to the qualifiers that specify modes, SET COMMAND provides the following qualifiers:

/[NO]LISTING	Controls whether an output listing is created
/[NO]OUTPUT	Controls where the modified command table should be written
/TABLE	Specifies the command table that is to be modified

See the Section 1.6.3 section for additional information.

1.5.1. Adding Command Definitions to a Command Table

Use the /REPLACE qualifier to add or replace verbs in the command table. By default, SET COMMAND uses the /REPLACE mode to add commands to your process command table and to return the modified command table to your process.

The following example shows how to add the new command SKIP to your process command table:

```
$ SET COMMAND SKIP
```

In this example, SET COMMAND adds the definitions from the command definition file SKIP.CLD to your process command table. The modified table replaces your original process command table. The /REPLACE qualifier is present by default, so you do not need to explicitly specify it in the command string.

To modify a command table other than your process table, use the /TABLE qualifier and the /OUTPUT qualifier.

1.5.2. Deleting Command Definitions

Use the /DELETE qualifier to delete a command name from a command table. By default, commands are deleted from your process command table. The following example shows how to delete the command SKIP from your process command table:

```
$ SET COMMAND/DELETE=SKIP
```

1.5.3. Creating Object Modules

Use the /OBJECT qualifier to create an object module from a command definition file. When you enter the following sample command, CDU creates an object module, NEWCOMS.OBJ, containing a command table with the verb definitions from NEWCOMS.CLD:

```
$ SET COMMAND/OBJECT NEWCOMS
```

You can then link NEWCOMS.OBJ with a program that parses commands using the new command table.

1.5.4. Creating New Command Tables

You cannot use the /OBJECT qualifier to create an object module from a command definition file that contains the IMAGE clause. However, you can create an empty command table to which you can add verbs that invoke images. The following is a step-by-step example of how to do this:

1. Create an empty command table by developing a command definition file that contains only a MODULE statement to define the module name and an IDENT statement. In the following example, CDU creates the empty command table, TEST_TABLE, from a command definition file named TEST_TABLE.CLD:

   ```
   MODULE TEST_TABLE
   IDENT "New command table"
   ```

2. Create an object module (TEST_TABLE.OBJ) from TEST_TABLE.CLD:

   ```
   $ SET COMMAND/OBJECT TEST_TABLE.CLD
   ```

3. Link TEST_TABLE.OBJ to create a shareable image, TEST_TABLE.EXE:

   ```
   $ LINK/SHARE/NOTRACEBACK TEST_TABLE
   ```

4. Create a command definition file that defines verbs that invoke images. In the following example, the command definition file VERBS.CLD includes two statements that call existing images:

   ```
   DEFINE VERB PASS
           IMAGE "DISK4:[ROSEN]PASS"
   DEFINE VERB THROW
           IMAGE "DISK4:[ROSEN]THROW"
       .
       .
       .
   ```

5. Add the new commands in VERBS.CLD to the empty command table in TEST_TABLE.EXE and write the modified table back to the file TEST_TABLE.EXE. The /TABLE and /OUTPUT qualifiers specify the input and output table files. For example:

   ```
   $ SET COMMAND/TABLE=TEST_TABLE.EXE/OUTPUT=TEST_TABLE.EXE VERBS
   ```

   Note that the version number of the output file is one greater than the version number of the input file. If you do not explicitly specify an output file using the /OUTPUT qualifier, CDU replaces your process command table with the modified command table.

1.6. Using Command Language Routines

A program invoked by a command that you have added to your process (or system) command table needs information about the command string that invoked it. The program can obtain this information by calling the appropriate command language routine:

CLI$PRESENT	Determines if an entity is present in the command string
CLI$GET_VALUE	Gets the value of the next entity in the command string
CLI$DCL_PARSE	Parses a command string
CLI$DISPATCH	Invokes the routine that corresponds to the verb most recently parsed

When you use CDU to add a new command, use the CLI$PRESENT and CLI$GET_VALUE routines from the program invoked by the command to get information about the command string that called the program.

When you use CDU to create and link an object module that includes a command table, use the CLI$DCL_PARSE and CLI$DISPATCH routines to parse the command string and to execute the command. Then use the CLI$PRESENT and CLI$GET_VALUE routines within the routines that execute the command.

The Section 1.6.4 section shows two programs that call these routines. For more information about the command language routines, see the *VSI OpenVMS Utility Routines Manual*.

1.6.1. CDU Usage Summary

The Command Definition Utility (CDU) creates, deletes, or changes command definitions in a command table. CDU uses either an existing command table, a file that contains command definitions, or a combination of these, to create a new command table. The output table can be part of an executable image or an object module.

You invoke CDU with the DCL command SET COMMAND together with the appropriate qualifiers.

Format

`SET COMMAND [filespec[,...]]`

Command Parameter

`filespec[,...]`
Specifies the name of one or more command definition files (default file type .CLD). If you specify two or more files, separate them with commas.

Wildcard characters are allowed in the file specification.

Usage Summary

Use the DCL command SET COMMAND to invoke CDU. SET COMMAND has the following modes:

SET COMMAND/DELETE	Deletes command definitions from a command table
SET COMMAND/OBJECT	Creates an object module from a command definition file

SET COMMAND/REPLACE	Adds or replaces definitions in a command table using definitions from a command definition file

The /DELETE, /OBJECT, and /REPLACE qualifiers establish the various SET COMMAND modes and are mutually exclusive; that is, you can use only one of these qualifiers in a command string.

The DCL prompt reappears on your screen when CDU finishes processing the command definition file or table.

By default, SET COMMAND/DELETE and SET COMMAND/REPLACE modify your process command table and return the modified table to your process. You can modify a different input command table by using the /TABLE command qualifier.

Note

You need CMKRNL privilege to modify the system command table in SYS$LIBRARY:DCLTABLES.EXE.

You can write the command table to an output file by using the /OUTPUT command qualifier along with the /TABLE qualifier.

SET COMMAND/OBJECT creates an object module with the same name as the command definition file unless you specify an alternate file name.

1.6.2. CDU File Statements

This section provides complete information about the statements that can be used in a command definition file. The statements are as follows:

DEFINE SYNTAX syntax-name [verb-clause[,...]]
DEFINE TYPE type-name [type-clause[,...]]
DEFINE VERB verb-name [verb-clause[,...]]
IDENT ident-string
MODULE module-name

DEFINE SYNTAX

DEFINE SYNTAX — Defines a syntax change that replaces a command's syntax (as defined in a DEFINE VERB, DEFINE TYPE, or another DEFINE SYNTAX statement). A syntax change allows a verb to use different syntax depending on the parameters, qualifiers, and keywords present in the command string.

Description

DEFINE statements that refer to changed syntax are called primary DEFINE statements; DEFINE SYNTAX statements that define new syntax are called secondary DEFINE statements.

When a command string is parsed, its components are scanned from left to right. The string is parsed according to the current definition until CDU encounters an entity that specifies a syntax change. The remainder of the string is parsed using the new definition. DCL does not rescan the entities that appear before the entity that specified the syntax change.

Table 1.2 shows how the DEFINE SYNTAX statement modifies the current command definition if an entity specifies a syntax change. After parsing the command string, DCL saves the command definition

to determine if any entities in the command string are not allowed. Then, DCL invokes the image or routine specified by the command definition and uses the definition to process CLI$PRESENT and CLI$GET_VALUE calls.

Table 1.2. How the DEFINE SYNTAX Statement Modifies the Primary DEFINE Statement

DEFINE SYNTAX Specifies	Result
An image	An image overrides the image in the primary DEFINE statement. DCL invokes the new image after it parses the command string.
A routine	A routine overrides the routine in the primary DEFINE statement. DCL invokes the new routine when CLI$DISPATCH is called.
One or more disallows	One or more disallows are used during command parsing and they override disallows in the primary DEFINE statement. This applies to all entities in the command that have not been invalidated by the new syntax definition.
No disallows	Disallows from the primary DEFINE statement are used during command parsing.
The NODISALLOWS clause	No disallows are permitted, regardless of definitions in the primary DEFINE statement.
One or more parameters	Parameters that were already parsed are not reparsed according to the new definitions. However, parameters to the right of the entity that specified the new syntax are parsed according to the new definitions. DCL uses the new parameter definitions when processing CLI$PRESENT and CLI$GET_VALUE calls. Note that, in the DEFINE SYNTAX statement, P1 refers to the first parameter in the command string. To define additional parameters, use the PARAMETER clause in a secondary DEFINE statement to first enter the definitions for the original parameters exactly as they appear in the primary DEFINE statement. Then, enter the definitions for the additional parameters.
No parameters	Parameter definitions from the primary DEFINE statement are used when DCL parses the remainder of the command string. DCL also uses these parameter definitions when processing CLI$PRESENT and CLI$GET_VALUE calls.
The NOPARAMETERS clause	Parameters previously parsed are not reparsed to the new definitions. However, no parameters are allowed when DCL parses entities to the right of the entity that specifies the new syntax. DCL uses the NOPARAMETERS definition when processing CLI$PRESENT and CLI$GET_VALUE calls.
One or more qualifiers	If any qualifiers have been previously parsed, they are ignored, and DCL issues an informational message. Qualifiers that appear in the command string after the entity specifies the new syntax are parsed according to the new definition. DCL uses the new qualifier definitions when processing CLI$PRESENT and CLI$GET_VALUE calls. Note that the qualifier that causes the syntax change cannot be retrieved from the CLI routines. VSI recommends the use of either

DEFINE SYNTAX Specifies	Result
	the IMAGE or ROUTINE clause to determine which syntax is in use.
No qualifiers	Qualifier definitions from the primary DEFINE statement are used when DCL parses the remainder of the command string. DCL also uses these qualifier definitions when processing CLI$PRESENT and CLI$GET_VALUE calls.
The NOQUALIFIERS clause	Qualifiers previously parsed are ignored. No qualifiers are allowed when DCL parses entities to the right of the entity that specifies the new syntax. DCL uses the NOQUALIFIERS definition when processing CLI$PRESENT and CLI$GET_VALUE calls.

Format

`DEFINE SYNTAX syntax-name [verb-clause[,...]]`

`syntax-name`
The name of the syntax change. The name is required and must immediately follow the DEFINE SYNTAX statement.

`[verb-clause[,…]`
Optional verb clauses that define attributes of the command string.

DEFINE SYNTAX accepts the following verb clauses:

- DISALLOW, NODISALLOWS
- IMAGE
- PARAMETER, NOPARAMETERS
- QUALIFIER, NOQUALIFIERS
- ROUTINE

The following text describes these clauses. Note that, if the syntax list contains only an IMAGE or ROUTINE clause, it affects only the specified clause in the primary DEFINE statement. If the list contains any qualifiers or the NOQUALIFIERS keyword, all qualifiers in the primary DEFINE statement are replaced by the qualifiers in the syntax list. If the syntax list contains neither qualifiers nor the NOQUALIFIERS keyword, the qualifiers in the primary DEFINE statement apply. Similarly, if the syntax list contains any parameter or the NOPARAMETERS keyword, all parameters in the primary DEFINE statement are replaced.

`DISALLOW expression`
`NODISALLOWS`
Disallows a command string if the result of the expression is true. The NODISALLOWS clause indicates that all entities and entity combinations are allowed.

The **expression** specifies an entity or a combination of entities connected by operators. Each entity in the expression is tested to see if it is present (true) or absent (false) in a command string. If an entity is present by default but not explicitly provided in the command string, the entity is false.

After each entity is evaluated, the operations indicated by the operators are performed. If the result is true, the command string is disallowed. If the result is false, the command string is valid.

You can specify entities in an expression using an entity name or label, a keyword path, or a definition path. See Section 1.4.4.1 for more information about entities. You can also specify the operators AND, ANY2, NEG, NOT, or OR. See Section 1.4.4.2 for more information about these operators.

IMAGE image-string
Names an image to be invoked by the command. The **image-string** is the file specification (a maximum of 63 characters) of the image DCL invokes when you enter the command. The default device and directory is SYS$SYSTEM: and the default file type is .EXE.

If you do not specify the IMAGE verb clause and you use SET COMMAND/REPLACE to process the command definition file, the verb name is used as the image name. At run time, DCL searches for an image whose file name is the same as the verb name and whose device and directory names and file type are SYS$SYSTEM: and .EXE, respectively.

PARAMETER param-name [,param-clause[,...]]
NOPARAMETERS
Can be used to specify up to eight parameters in the command string. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** defines the position of the parameter in the command string. The name must be in the form P`n`, where `n` is the position of the parameter. The parameter names must be numbered consecutively from P1 to P8. The name must immediately follow the PARAMETER clause.

The **param-clause** specifies additional characteristics for the parameter. You can use the following parameter clauses:

- DEFAULT
- LABEL=label-name
- PROMPT=prompt-string
- VALUE[(param-value-clause[,...])]

DEFAULT indicates that a user-defined parameter keyword is present by default. You should use this clause only if you also use the VALUE clause to indicate that a user-defined keyword must be specified as the parameter value. See the description of the DEFINE TYPE statement for more information on defining a keyword that is present by default.

To designate a default parameter that is not a keyword, use the VALUE(DEFAULT=default-string) clause.

LABEL=label-name defines a label for referring to a parameter at run time. Specify the label name as a symbol. If you do not specify a label name, the parameter name (P1 through P8) is used as the label name.

PROMPT=prompt-string supplies a prompt string (a maximum of 31 characters) when a parameter is omitted from the command string. If you do not specify a prompt string and a required parameter is missing, DCL uses the parameter name as the prompt string.

When you define more than one parameter but only the first parameter is required, DCL prompts for the first parameter until the user either enters a value or aborts the command with Ctrl/Z. When the user enters a value for the first parameter, DCL prompts for the optional parameters. If the user presses Return without entering a value for an optional parameter, DCL executes the command.

VALUE[(param-value-clause[,...])] specifies additional characteristics for the parameter. When you specify parameter value clauses, enclose them in parentheses and separate items with commas.

VALUE accepts the following parameter value clauses:

CONCATENATE	Indicates that a parameter can be concatenated to another parameter with a plus sign.
DEFAULT=default-string	Specifies a default value to be used if a value for the parameter is not explicitly given. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 94 characters. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT clause.
LIST	Permits you to enter a list of parameters separated by commas or plus signs.
NOCONCATENATE	Indicates that the parameters cannot be concatenated.
REQUIRED	Indicates that the parameter is required. All required parameters must precede optional ones. If you use the REQUIRED clause, you should also specify a prompt string. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Gives either a built-in value type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the value type name as a symbol. See Section 1.4.2.1 for more information about built-in value types.

QUALIFIER qual-name [,qual-clause[,...]]
NOQUALIFIERS
Specifies a qualifier that can be included in the command string. You can use the QUALIFIER clause up to 255 times in a DEFINE SYNTAX statement. The NOQUALIFIERS clause indicates that no qualifiers are allowed.

The **qual-name** is the name of the qualifier. Specify the qualifier name as a symbol. The first four characters of the qualifier name must be unique.

The **qual-clause** specifies additional qualifier characteristics. You can use the following qualifier clauses:

- BATCH
- DEFAULT
- LABEL=label-name
- NEGATABLE, NONNEGATABLE
- PLACEMENT=placement-clause
- SYNTAX=syntax-name
- VALUE[(qual-value-clause[,...])]

BATCH indicates that the qualifier is present by default if the command is used in a batch job.

DEFAULT indicates that the qualifier is present by default in both batch and interactive jobs.

LABEL=label-name defines a label for requesting information about the qualifier at run time. Specify the label name as a symbol. If you do not specify a label name, the qualifier name is used as the label name.

NEGATABLE and NONNEGATABLE indicate whether the qualifier can be negated by adding NO to the qualifier name. The default is NEGATABLE; if you do not specify either NEGATABLE or NONNEGATABLE, NO can be added to the qualifier name to indicate that the qualifier is not present.

PLACEMENT=placement-clause indicates where the qualifier can appear in the command string. PLACEMENT accepts the following placement clauses:

GLOBAL	Indicates that the qualifier applies to the entire command and can be placed after the verb or after a parameter. This is the default if you do not specify the PLACEMENT clause.
LOCAL	Indicates that the qualifier can appear only after a parameter value and that it applies only to that parameter.
POSITIONAL	Indicates that the qualifier can appear anywhere in the command string, but the function of the qualifier depends on its position: if the qualifier is used after a parameter value, it applies only to that parameter; if it is used after the verb, it applies to all parameters.

SYNTAX=syntax-name specifies an alternate syntax definition to be invoked when the qualifier is present. The syntax name must correspond to the name used in a DEFINE SYNTAX statement. Specify the syntax name as a symbol.

VALUE[(qual-value-clause[,...])] specifies additional characteristics for the qualifier. When you specify qualifier value clauses, enclose the list in parentheses and separate items with commas. If you do not specify any qualifier value clauses, DCL converts letters in qualifier values to uppercase.

VALUE accepts the following qualifier value clauses:

DEFAULT=default-string	Specifies a default value to be used if a value for the qualifier is not explicitly given. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 94 characters. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT qualifier clause.
LIST	Indicates that a list of values separated by commas can be specified for the qualifier. This list must be enclosed in parentheses, and the items must be separated by commas. Note that plus signs cannot be used to separate items in a list of qualifier values.
REQUIRED	Indicates that the qualifier must have an explicitly specified value. No prompting is performed for a required qualifier value. The REQUIRED clause and the DEFAULT clause are mutually exclusive.

TYPE=type-name	Gives either a built-in value type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the value type name as a symbol. See Section 1.4.2.1 for more information about built-in value types.

`ROUTINE routine-name`
Names a routine in syntax. Use the ROUTINE clause to create an object module from the command definition file.

The **routine-name** provides the name of the routine to be executed when CLI$DISPATCH is called. Specify the routine name as a symbol.

If you do not specify a routine, the routine from the primary DEFINE statement is invoked, if applicable.

Examples

1.
```
DEFINE VERB WRITER
         IMAGE "WORK:[JONES]WRITER"
         QUALIFIER LINE, SYNTAX=LINE
         QUALIFIER SCREEN, SYNTAX=SCREEN

DEFINE SYNTAX LINE
         IMAGE "WORK:[JONES]LINE"
         QUALIFIER NUM

DEFINE SYNTAX SCREEN
         IMAGE "WORK:[JONES]SCREEN"
         QUALIFIER AUDIT
```

   This example illustrates a command definition file (WRITER.CLD) containing DEFINE SYNTAX statements that cause syntax changes depending upon the qualifiers specified in the command string. The verb WRITER invokes a text editor (WRITER.EXE). However, you can use the SCREEN and the LINE qualifiers to invoke alternate text editors.

   You can add the command definition to your process command table by issuing the following command:

```
$ SET COMMAND WRITER
```

   Then you can use the WRITER command to access different text editors. For example, assume you specify the following command:

```
$ WRITER/LINE
```

   Here you invoke the LINE editor instead of the default editor (WRITER). Syntax redefinition is done from left to right because parsing of the string is done from left to right. This order means that when you specify two qualifiers that invoke different syntax lists, the leftmost qualifier takes precedence (because it is parsed first).

2.
```
DEFINE VERB DISPLAY
         PARAMETER P1, LABEL=ITEM, VALUE(REQUIRED, TYPE=$FILE)
         QUALIFIER SAVE, SYNTAX=SAVE

DEFINE SYNTAX SAVE
         IMAGE "WORK:[NEWMAN]:SAVE_DISPLAY"
```

```
PARAMETER P1, LABEL=ITEM, VALUE(REQUIRED, TYPE=$FILE)
PARAMETER P2, LABEL=NAME
```

This example shows a syntax change that defines an additional parameter. The command definition file defines the verb DISPLAY. If the DISPLAY command is used without the /SAVE qualifier, then one parameter is required. This parameter indicates the name of the file to be displayed. If the DISPLAY command is used with the /SAVE qualifier, then two parameters are required: the name of the file to be displayed and the name of the file where the display should be saved. Note that you must repeat the definition of P1 in the DEFINE SYNTAX statement.

DEFINE TYPE

DEFINE TYPE — Describes the keywords referenced by the VALUE(TYPE=type-name) clause. You can use the VALUE clause in a DEFINE VERB, DEFINE SYNTAX, or DEFINE TYPE statement to indicate predefined values (keywords) for command parameters, qualifiers, or keywords.

Format

`DEFINE TYPE name [type-clause[,...]]`

`name`
The name of the DEFINE TYPE statement. This name must match the name used in the VALUE(TYPE=type-name) clause that references the DEFINE TYPE statement.

`[type-clause[,...]]` Defines a keyword that can be used as the value of the entity that referenced the DEFINE TYPE statement. The DEFINE TYPE statement accepts the following type clause:

```
KEYWORD keyword-name [,keyword-clause[,...]]
```

This clause specifies a keyword that can be used as the value type of the entity that references the DEFINE TYPE statement. Repeat the KEYWORD value type clause for each keyword that can be used. You can specify up to 255 keywords in a DEFINE TYPE statement.

The **keyword-name** is the name of the keyword. The optional **keyword-clause** specifies additional keyword characteristics.

You can use the following keyword clauses:

- DEFAULT
- LABEL=label-name
- NEGATABLE, NONNEGATABLE
- SYNTAX=syntax-name
- VALUE[(key-value-clause[,...])]

DEFAULT indicates that the keyword is present by default. For this keyword to be recognized as present by default, the parameter, qualifier, or keyword definition that references this DEFINE TYPE statement must also specify the DEFAULT clause.

LABEL=label-name defines a label for referencing the keyword at run time. By default, the keyword name is used as the label name.

NEGATABLE and NONNEGATABLE indicate whether the keyword can be negated by adding NO to the keyword name (the default is NONNEGATABLE). If you do not specify either NEGATABLE or NONNEGATABLE, NO cannot be used to negate the keyword name. Note that this differs from qualifiers, which, by default, are negatable.

SYNTAX=syntax-name specifies an alternate verb definition to be invoked when the keyword is present. The syntax name must match the name used in the corresponding DEFINE SYNTAX statement.

VALUE[(key-value-clause[,...])] specifies additional characteristics for the keyword.

VALUE accepts the following keyword value clauses:

DEFAULT=default-string	Specifies a default value to be used if a value for the keyword is not explicitly given. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 94 characters. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT clause with the entity that uses the keyword.
LIST	Indicates that a list of values for the keyword can be given. This list must be enclosed in parentheses, and the items must be separated by commas. Note that plus signs cannot be used to separate items in a list of keyword values.
REQUIRED	Indicates that the keyword must have an explicitly specified value. No prompting is performed for a required keyword value. If the keyword is specified without a value, an error is automatically issued by DCL. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Symbolically equates either a built-in value type or the name of a DEFINE TYPE statement that defines keywords that can be specified as the keyword value. The TYPE clause cannot be specified if the DEFAULT clause is specified. See Section 1.4.2.1 for more information about built-in value types.

Examples

1.
```
DEFINE VERB DISPLAY
         PARAMETER P1, LABEL=OPTION, PROMPT="What"
                       VALUE(REQUIRED, TYPE=DISPLAY_OPTIONS)

   DEFINE TYPE DISPLAY_OPTIONS
         KEYWORD ANIMALS, SYNTAX=DISPLAY_ANIMALS
         KEYWORD FLOWERS, SYNTAX=DISPLAY_FLOWERS

   DEFINE SYNTAX DISPLAY_ANIMALS
         IMAGE "USER:[JOHNSON]ANIMALS"
         PARAMETER P1, LABEL=OPTION, VALUE(REQUIRED)
         QUALIFIER SMALL
         QUALIFIER LARGE
         QUALIFIER ALL, DEFAULT

   DEFINE SYNTAX DISPLAY_FLOWERS
         IMAGE "USER:[JOHNSON]FLOWERS"
```

```
        PARAMETER P1, LABEL=OPTION, VALUE(REQUIRED)
        NOQUALIFIERS
```

This example shows how to define keywords that can be specified as parameters for the verb DISPLAY. Each keyword uses its own syntax definition to invoke an image to execute the command.

After you add the command definition to your process command table, you can enter the following DISPLAY commands:

```
$ DISPLAY ANIMALS
$ DISPLAY FLOWERS
```

In addition, the syntax definition DISPLAY_ANIMALS specifies three qualifiers that can be used only with the command DISPLAY ANIMALS. No qualifiers are allowed with the command DISPLAY FLOWERS.

2.
```
DEFINE VERB DRAW
        QUALIFIER COLOR, VALUE(TYPE=COLOR_NAMES)

DEFINE TYPE COLOR_NAMES
        KEYWORD RED
        KEYWORD BLUE
```

This example shows a verb definition that uses a DEFINE TYPE statement to define keywords that can be used with a qualifier. After you add the command definition for DRAW to your process command table, you can enter the following DRAW commands:

```
$ DRAW/COLOR=RED
$ DRAW/COLOR=BLUE
```

3.
```
DEFINE VERB RANDOM
        PARAMETER P1, VALUE(TYPE=THINGS), DEFAULT

DEFINE TYPE THINGS
        KEYWORD NUMBER, DEFAULT
        KEYWORD LETTER
```

This example defines a verb, RANDOM. RANDOM accepts a parameter, which must be one of the user-defined keywords NUMBER or LETTER. If a parameter is not specified with the verb RANDOM, then the default is NUMBER.

Note that, for the keyword NUMBER to be present by default, you must use the DEFAULT clause in two places. You must specify DEFAULT when you define the parameter in the DEFINE VERB statement, and you must also specify DEFAULT when defining the NUMBER keyword in the DEFINE TYPE statement.

DEFINE VERB

DEFINE VERB — Defines a new command, its parameters, its qualifiers, and the image or routine it invokes.

Format

DEFINE VERB verb-name [verb-clause[,...]]

verb-name

The name of the command verb. This parameter is required and must immediately follow the DEFINE VERB statement. The first four characters of the verb name must be unique.

verb-clause[,...]
Optional verb clauses that define attributes of the command string.

DEFINE VERB accepts the following verb clauses:

- DISALLOW, NODISALLOWS
- IMAGE
- PARAMETER, NOPARAMETERS
- QUALIFIER, NOQUALIFIERS
- ROUTINE
- SYNONYM

The following text describes these verb clauses.

DISALLOW expression
NODISALLOWS
Disallows a command string if the result of the expression is true. The NODISALLOWS clause indicates that all entities and entity combinations are allowed.

The **expression** specifies an entity or a combination of entities connected by operators. Each entity in the expression is tested to see if it is present (true) or absent (false) in a command string. If an entity is present by default but not explicitly provided in the command string, the entity is false.

After each entity is evaluated, the operations indicated by the operators are performed. If the result is true, the command string is disallowed. If the result is false, the command string is valid.

You can specify entities in an expression using an entity name or label, a keyword path, or a definition path. See Section 1.4.4.1 for more information about entities. You can also specify the operators AND, ANY2, NEG, NOT or OR. See Section 1.4.4.2 for more information about these operators.

IMAGE image-string
Names an image to be invoked by the command. The **image-string** is the file specification (a maximum of 63 characters) of the image DCL invokes when you enter the command. The default device and directory is SYS$SYSTEM: and the default file type is .EXE.

If you do not specify the IMAGE verb clause and you use SET COMMAND/REPLACE to process the command definition file, the verb name is used as the image name. At run time, DCL searches for an image whose file name is the same as the verb name and whose device and directory names and file type are SYS$SYSTEM: and .EXE.

PARAMETER param-name [,param-clause[,...]]
NOPARAMETERS
Can be used to specify up to eight parameters in the command string. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** defines the position of the parameter in the command string. The name must be in the form P`n`, where `n` is the position of the parameter. The parameter names must be numbered consecutively from P1 to P8. The name must immediately follow the PARAMETER clause.

The **param-clause** specifies additional characteristics for the parameter. You can use the following parameter clauses:

- DEFAULT
- LABEL=label-name
- PROMPT
- VALUE[(param-value-clause[,...])]

DEFAULT indicates that a user-defined parameter keyword is present by default. You should use this clause only if you also use the VALUE clause to indicate that a user-defined keyword must be specified as the parameter value. See the description of the DEFINE TYPE statement for more information about defining a keyword that is present by default.

To designate a default parameter that is not a keyword, use the VALUE(DEFAULT=default-string) clause.

LABEL=label-name defines a label for referring to a parameter at run time. Specify the label name as a symbol. If you do not specify a label name, the parameter name (P1 through P8) is used as the label name.

PROMPT=prompt-string supplies a prompt string (a maximum of 63 characters) when a parameter is omitted from the command string. If you do not specify a prompt string and a required parameter is missing, DCL uses the parameter name as the prompt string.

When you define more than one parameter but only the first parameter is required, DCL prompts for the first parameter until the user either enters a value or aborts the command with Ctrl/Z. When the user enters a value for the first parameter, DCL prompts for the optional parameters. If the user presses Return without entering a value for an optional parameter, DCL executes the command.

VALUE[(param-value-clause[,...])] specifies additional characteristics for the parameter. When you specify parameter value clauses, enclose them in parentheses and separate items with commas.

VALUE accepts the following parameter value clauses:

CONCATENATE	Indicates that a parameter can be concatenated to another parameter with a plus sign.
DEFAULT=default-string	Specifies a default value to be used if the value for the parameter is not explicitly given. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 94 characters. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT parameter clause.
LIST	Permits you to enter a list of parameters separated by commas or plus signs.
NOCONCATENATE	Indicates that the parameters cannot be concatenated.
REQUIRED	Indicates that the parameter is required. All required parameters must precede optional ones. If you use the REQUIRED clause, you

	should also specify a prompt string. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Gives either a built-in value type or the name of a DEFINE TYPE statement that lists keywords for the parameter. Specify the value as a symbol. See Section 1.4.2.1 for more information about built-in value types.

QUALIFIER qual-name [,qual-clause[,...]]
NOQUALIFIERS
Specifies a qualifier that can be included in the command string. You can use the QUALIFIER clause up to 255 times in a DEFINE VERB statement. The NOQUALIFIERS clause indicates that no qualifiers are allowed.

The **qual-name** is the name of the qualifier. The first four characters of the qualifier name must be unique. Specify the qualifier name as a symbol.

The **qual-clause** specifies additional qualifier characteristics. You can use the following qualifier clauses:

- BATCH
- DEFAULT
- LABEL=label-name
- NEGATABLE, NONNEGATABLE
- PLACEMENT=placement-clause
- SYNTAX=syntax-name
- VALUE[(qual-value-clause[,...])]

BATCH indicates that the qualifier is present by default if the command is used in a batch job.

DEFAULT indicates that the qualifier is present by default in both batch and interactive jobs.

LABEL=label-name defines a label for requesting information about the qualifier at run time. Specify the label name as a symbol. If you do not specify a label name, the qualifier name is used by default.

NEGATABLE and NONNEGATABLE indicate whether the qualifier can be negated by adding NO to the qualifier name. The default is NEGATABLE; if you do not specify either NEGATABLE or NONNEGATABLE, NO can be added to the qualifier name to indicate that the qualifier is not present.

PLACEMENT=placement-clause indicates where the qualifier can appear in the command string. PLACEMENT accepts the following placement clauses:

GLOBAL	Indicates that the qualifier applies to the entire command string and can be placed after the verb or after a parameter. This is the default if you do not specify the PLACEMENT clause.
LOCAL	Indicates that the qualifier can appear only after a parameter value and that it applies only to that parameter.

POSITIONAL	Indicates that the qualifier can appear anywhere in the command string, but the function of the qualifier depends on its position: if the qualifier is used after a parameter value, it applies only to that parameter; if it is used after the verb, it applies to all parameters.

SYNTAX=syntax-name specifies an alternate syntax definition to be invoked when the qualifier is present. The syntax name must correspond to the name used in the related DEFINE SYNTAX statement. This alternate syntax is useful for commands that invoke different images depending upon the particular qualifiers that are present. Specify the syntax name as a symbol.

VALUE[(qual-value-clause[,...])] specifies additional characteristics for the qualifier. When you specify qualifier value clauses, enclose the list in parentheses and separate items with commas. If you do not specify any qualifier value clauses, DCL converts letters in qualifier values to uppercase.

VALUE accepts the following qualifier value clauses:

DEFAULT=default-string	Specifies a default value to be used if a value for the qualifier is not explicitly given. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 94 characters. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT qualifier clause.
LIST	Indicates a list of values separated by commas can be specified for the qualifier. This list must be enclosed in parentheses, and the items must be separated by commas. Note that plus signs cannot be used to separate items in a list of qualifier values.
REQUIRED	Indicates that the qualifier must have an explicitly specified value. No prompting is performed for a required qualifier value. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Gives either a built-in value type or a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the value type as a symbol. See Section 1.4.2.1 for more information about built-in value types.

`ROUTINE routine-name`
Symbol that specifies a routine the command calls to create an object module from the command definition file.

The **routine-name** provides the name of a routine that is executed when CLI$DISPATCH is called.

If you do not specify a routine, no default is provided.

`SYNONYM synonym-name`
Defines a synonym for the verb name. Specify the synonym name as a symbol.

Examples

1.
```
DEFINE VERB ERASE
            PARAMETER,P1 VALUE(DEFAULT=DISK3:[JONES]STATS.DAT)
```

This definition tells the command language interpreter that ERASE is a valid verb and that it takes a parameter. If you do not enter a parameter value, the default is DISK3:[JONES]STATS.DAT.

Because no image name is specified, the verb ERASE invokes the image SYS$SYSTEM:ERASE.EXE.

2.
```
DEFINE VERB SCATTER
          IMAGE "WRKD$:[MORRISON]SCATTER"
          PARAMETER P1, LABEL=INFILE, PROMPT="Input_file?", -
            VALUE(REQUIRED)
          PARAMETER P2, LABEL=OUTFILE, PROMPT="Output_file?", -
            VALUE(REQUIRED)
          QUALIFIER SLOW, DEFAULT
          QUALIFIER FAST
          DISALLOW SLOW AND FAST
```

This example shows a command definition file that defines a new command called SCATTER that invokes the image WRKD$:[MORRISON]SCATTER.EXE. It has two required parameters, an input file and an output file. It has two mutually exclusive qualifiers, /SLOW and /FAST (the default is /SLOW).

IDENT

IDENT — Provides identifying information for an object module created from a command definition file.

Format

IDENT ident-string

ident-string
A string containing identifying information. The string has a maximum length of 31 characters.

Example

```
MODULE COMMAND_TABLE
IDENT "V04-001"
DEFINE VERB SPIN
    .
    .
    .
```

This command definition file uses the IDENT statement to identify the object module file.

MODULE

MODULE — Provides a name for an object module and for a global symbol that refers to the address of a command table within an image into which the object module is linked.

Format

MODULE module-name

module-name

The **module-name** is used to create a global symbol that refers to the address of the command table within the image into which the object module is to be linked.

By default, CDU uses the object file name specified with the /OBJECT command qualifier. If no object file is explicitly specified, then CDU uses the name of the first command definition file as the module name.

Example

```
$ CREATE TEST.CLD
MODULE TEST_TABLE
DEFINE VERB SEND
        ROUTINE SEND_ROUT
        PARAMETER P1
    .
    .
    .
DEFINE VERB SEARCH
        ROUTINE SEARCH_ROUT
        PARAMETER P1
^Z
$ SET COMMAND/OBJECT=TEST.OBJ TEST
$ LINK PROG,TEST
$ RUN PROG
```

TEST.CLD defines two commands (SEND and SEARCH) that call routines in PROG.EXE, a program that uses DCL to parse command strings and execute routines.

The SET COMMAND command creates a command table object module that is linked with the program object module (PROG.OBJ) to produce an image (PROG.EXE) that includes the code for the program and for the command table. TEST_TABLE refers to the address of the command table in the image.

When you run PROG.EXE, it calls DCL parsing routines to parse the command string using the command table in module TEST_TABLE.

1.6.3. CDU Qualifiers

The following pages describe the qualifiers that can be used with the DCL command SET COMMAND. The qualifiers are as follows:

- /ALPHA
- /DELETE
- /LISTING
- /OBJECT
- /OUTPUT
- /REPLACE
- /TABLE
- /VAX

The /DELETE, /OBJECT, and /REPLACE qualifiers indicate SET COMMAND modes; these qualifiers are mutually exclusive.

/ALPHA

/ALPHA — Causes CDU to create an OpenVMS Alpha object module when used with the /OBJECT qualifier. The default is to create OpenVMS Alpha object modules on OpenVMS Alpha systems and to create OpenVMS VAX object modules on OpenVMS VAX systems.

Format

SET COMMAND/ALPHA/OBJECT [=object-filespec] filespec

object-filespec
The file specification for the object file. If no file name is specified, default to the name of the first input (command definition) file; the default file type is .OBJ.

filespec
The command definition file to be processed (wildcard characters are allowed). The default type is .CLD.

Example

```
$ SET COMMAND /ALPHA /OBJECT=A  TEST
MODULE TEST_TABLE
```

In this example, the command definition file TEST.CLD is processed and the command table is written as an Open VMS Alpha object module to a file named A.OBJ.

/DELETE

/DELETE — Used to delete verb names or synonym names from the command table. If a verb name has synonyms, this qualifier deletes the specified verb or synonym name. If any synonyms remain, or if you delete synonyms and the original verb name remains, the remaining names still reference the verb definition. You can use the /DELETE qualifier to delete a verb in either your process command table or in a command table file specified with the /TABLE qualifier. If you do not use the /TABLE qualifier to specify an alternate command table, the default is to delete verbs from your process command table. If you do not use the /OUTPUT qualifier to specify an output file, the default is to return the modified command table to your process. You cannot use the /LISTING, /OBJECT, or /REPLACE qualifier with /DELETE.

Format

SET COMMAND/DELETE=(verb[,…])

verb
A verb or verb synonym to be deleted from the specified command table. If you specify two or more names, separate them with commas and enclose the list in parentheses.

Examples

1. `$ SET COMMAND/DELETE=DO`

   In this example, SET COMMAND deletes the verb DO from your process command table.

2. `$ SET COMMAND/DELETE=(PUSH,SHOVE)/TABLE=TEST_TABLE/OUTPUT=NEW_TABLE`

   The commands PUSH and SHOVE are deleted from the command table TEST_TABLE.EXE. The /OUTPUT qualifier writes the modified table to the file NEW_TABLE.EXE. If you do not include the /OUTPUT qualifier, CDU uses the modified table to overwrite your process command table.

/LISTING

/LISTING — Controls whether an output listing is created and optionally provides an output file specification for the listing file. A listing file contains a listing of the command definitions along with any error messages. The listing file is similar to a compiler listing. If you specify the /LISTING qualifier and omit the file specification, output is written to the default device and directory; the listing file has the same name as the first command definition file and a file type of .LIS. You can use the /LISTING qualifier only in /OBJECT or /REPLACE mode; you cannot create a listing in /DELETE mode. In /OBJECT and /REPLACE modes, the default is /NOLISTING.

Format

`SET COMMAND/LISTING [=listing-filespec] [filespec[,...]]`

`SET COMMAND/NOLISTING`

`listing-filespec`
The file specification for the listing file. The default file name is the name of the first command definition file. The default file type is .LIS.

`filespec`
The name of the command definition file to be processed (wildcard characters are allowed). The default file type is .CLD.

Examples

1. `$ SET COMMAND/LISTING TEST`

   In this example, the command definition file TEST.CLD is processed by CDU, and the new verbs are added to your process command table. (By default, SET COMMAND uses /REPLACE mode.) The modified table is returned to your process, and a listing file named TEST.LIS is created.

2. `$ SET COMMAND/LISTING=A TEST`

   The command definition file TEST.CLD is processed by CDU, and the verb definitions are added to your process command table. The modified table is returned to your process, and a listing file named A.LIS is created.

3. `$ SET COMMAND/LISTING/OBJECT GAMES`

   SET COMMAND is used to create an object module (GAMES.OBJ) that contains the command definitions in GAMES.CLD. The output object module can then be linked with a program. A listing file named GAMES.LIS is created.

/OBJECT

/OBJECT — Creates an object module from a command definition file and optionally provides an object file specification. You cannot use the /OBJECT qualifier to create an object module from a command

definition that contains the IMAGE clause. An object module containing a command table can be linked with the object modules from your program. This enables the program to use its own command table for parsing command strings and executing routines. On OpenVMS VAX systems, the /OBJECT qualifier creates a VAX module by default. Note that you cannot combine VAX modules and Alpha modules in the same object file. For more information, see the description of the /VAX qualifier. On OpenVMS Alpha systems, the /OBJECT qualifier creates an Alpha module by default. Note that you cannot combine Alpha modules and VAX modules in the same object file. For more information, see the description of the /ALPHA qualifier. You can specify only one command definition file when you use SET COMMAND/OBJECT. If you specify the /OBJECT qualifier and omit the file specification, output is written to the default device and directory; the object file has the same name as the input file and a file type of .OBJ. You cannot use the /DELETE, /OUTPUT, /REPLACE, or /TABLE qualifier with /OBJECT.

Format

SET COMMAND/OBJECT [=object-filespec]

object-filespec
The file specification for the object file. If no file name is specified, defaults to the name of the first input (command definition) file; the default file type is .OBJ.

filespec
The command definition file to be processed (wildcard characters are allowed). The default file type is .CLD.

Examples

1. `$ SET COMMAND/OBJECT TEST`

   In this example, the command definition file TEST.CLD is processed and a new command table is created. This table is written as an object module to a file named TEST.OBJ. (If not explicitly given, the name of the object module defaults to the name of the command definition file with a file type of .OBJ.)

2. `$ SET COMMAND/OBJECT=A TEST`

   In this example, the command definition file TEST.CLD is processed and the command table is written as an object module to a file named A.OBJ.

/OUTPUT

/OUTPUT — Controls where the modified command table should be placed. If you provide an output file specification, the modified command table is written to the specified file. If you do not provide an output file specification, the modified command table is placed in your process. The /NOOUTPUT qualifier indicates that no output is to be generated. You can use the /OUTPUT qualifier only in /DELETE or /REPLACE mode; the default is /OUTPUT with no file specification. You cannot use the /OUTPUT qualifier in /OBJECT mode.

Format

SET COMMAND/OUTPUT [=output-filespec] [filespec[,...]]

SET COMMAND/NOOUTPUT

output-filespec

The specification of the output file that contains the edited command table. The default file type is .EXE.

You can specify an output file only when you use the /TABLE=filespec qualifier to describe an input table.

filespec
The name of the command definition file to be processed (wildcard characters are allowed). The default file type is .CLD.

Examples

1. `$ SET COMMAND/OUTPUT TEST`

   The file TEST.CLD is processed and the definitions are added to your process command table. The modified table is returned to your process. (The result is the same as if you had issued the command SET COMMAND TEST.)

2. `$ SET COMMAND/TABLE=A/OUTPUT=A TEST`

   The definitions from TEST.CLD are added to command table A.EXE. CDU writes the modified table to the new A.EXE, which has a version number one greater than the input table file.

   If you use the /TABLE qualifier and do not provide an output file specification, the modified command table replaces your process command table.

3. `$ SET COMMAND/NOOUTPUT TEST`

   The definitions from TEST.CLD are added to your process command table, and the modified table is not written anywhere. You can use this command string to test whether a command definition file is written correctly.

/REPLACE

/REPLACE — Used to add or replace verbs in the command table. You can use the /REPLACE qualifier to either modify the process command table or, with the /TABLE qualifier, to modify a command table file. You cannot use the /REPLACE qualifier with the /OBJECT or /DELETE qualifier. If you do not explicitly specify /DELETE, /OBJECT, or /REPLACE, the default is /REPLACE.

Format

SET COMMAND/REPLACE [filespec [,...]]

filespec
The file to be processed (wildcard characters are allowed). The default file type is .CLD.

Examples

1. `$ SET COMMAND SCROLL`

   This command adds the command definitions from the file SCROLL.CLD to your process command table. The /REPLACE, /TABLE, and /OUTPUT qualifiers are present by default. The /REPLACE qualifier indicates /REPLACE mode; the /TABLE qualifier indicates that your process command table is to be modified; and the /OUTPUT qualifier indicates that the modified command table is to be written to your process.

2. `$ SET COMMAND/TABLE/OUTPUT SCROLL`

This command adds the command definitions from the file SCROLL.CLD to your process command table and returns the modified table to your process. (The /TABLE and /OUTPUT qualifiers, with no specified files, default to your process command table.) This command is the same as the command SET COMMAND SCROLL.

3. `$ SET COMMAND/TABLE=COMMAND_TABLE/OUTPUT=NEW_TABLE TEST`

   CDU adds command definitions from TEST.CLD to the command table in the file COMMAND_TABLE.EXE, and the modified command table is written to NEW_TABLE.EXE.

   If you use the /TABLE qualifier to provide an input command table, be sure to provide an output file specification. Otherwise, CDU uses the modified command table to replace your process command table.

4. `$ SET COMMAND/TABLE=TEST_TABLE MYCOMS`

   In this example, the definitions from MYCOMS.CLD are added to the command table in TEST_TABLE.EXE. The modified command table is written to your process and replaces your process command table. You should replace your process command table only if the new command table contains all the commands you need to perform your work. DCL commands copied to your process command table when you logged in are overwritten.

/TABLE

/TABLE — Specifies the command table to be modified. If you specify the /TABLE qualifier and omit the file specification, the current process command table is modified. Can be used with /DELETE or /REPLACE but not with /OBJECT; the default is /TABLE with no input file specification. If you include a file specification, the specified command table is modified. If you use the /TABLE qualifier without the /OUTPUT qualifier, the modified command table replaces your process command table.

Format

SET COMMAND/TABLE [=input-filespec][filespec [,...]]

SET COMMAND/NOTABLE

input-filespec
The input file that contains the command table to be edited. The default file type is .EXE.

filespec
The command definition file to be processed (wildcard characters are allowed). The default file type is .CLD.

Examples

1. `$ SET COMMAND/TABLE TEST`

   The commands from TEST.CLD are added to your process command table and the results are returned to your process. The /TABLE qualifier with no file specification indicates that your process command table is to be modified. This command is the same as the command SET COMMAND TEST.

2. `$ SET COMMAND/TABLE=A/OUTPUT=B TEST`

CDU adds the command definitions from TEST.CLD to the command table in A.EXE and writes the modified command table to B.EXE.

If you use the /TABLE qualifier to provide an input command table, be sure to provide an output file specification. Otherwise, the modified command table replaces your process command table.

3. `$ SET COMMAND/TABLE=A`

In this example, the command table in A.EXE is written to your process and replaces your process command table. You should replace your process command table only if the new command table contains all the commands you need to perform your work. DCL commands copied to your process command table when you logged in are overwritten.

/VAX

/VAX — Causes CDU to create an OpenVMS VAX object module when used with the /OBJECT qualifier. The default is to create OpenVMS Alpha object modules on OpenVMS Alpha systems and to create OpenVMS VAX object modules on OpenVMS VAX systems.

Format

SET COMMAND/VAX/OBJECT [=object-filespec] filespec

object-filespec
The file specification for the object file. If no file name is specified, defaults to the name of the first input (command definition) file; the default file type is .OBJ.

filespec
The command definition file to be processed (wildcard characters are allowed). The default file type is .CLD.

Example

```
$ SET COMMAND/VAX/OBJECT=A TEST
```

In this example, the command definition file TEST.CLD is processed and the command table is written as an OpenVMS VAX object module to a file named A.OBJ.

1.6.4. CDU Examples

Adding a Command to Your Process Command Table

This example shows how to add a command to your process command table and how to use command language routines in the image invoked by the new command.

The following command definition file defines a new verb called SAMPLE:

```
DEFINE VERB SAMPLE
       IMAGE  "USERDISK:[MYDIR]SAMPLE"
       PARAMETER  P1,LABEL=FILESPEC
       QUALIFIER  EDIT
```

To process this command definition file, use the DCL command SET COMMAND:

```
$ SET COMMAND SAMPLE
```

This command string invokes CDU to process the command definition file (SAMPLE.CLD) and to add the verb SAMPLE to your process command table. The modified table is returned to your process.

The following program illustrates a program called SAMPLE.BAS. It uses the CLI$PRESENT and CLI$GET_VALUE command language routines to obtain information about a command string parsed by DCL.

```
1    EXTERNAL INTEGER FUNCTION CLI$PRESENT,CLI$GET_VALUE

10   IF CLI$PRESENT('EDIT') AND 1%
     THEN
         PRINT '/EDIT IS PRESENT',A$

20   IF CLI$PRESENT('FILESPEC') AND 1%
     THEN
         CALL CLI$GET_VALUE('FILESPEC',A$)
         PRINT 'FILESPEC = ',A$

30   END
```

This source program must be compiled and linked before it can be invoked by a command verb. When you compile and link the source program, the output file (SAMPLE.EXE) contains an executable image.

You can now use the SAMPLE command to invoke the image SAMPLE.EXE, as follows:

```
$ SAMPLE
```

DCL processes this command in the same way it processes the DCL commands provided by VSI; that is, DCL checks the syntax and then invokes SAMPLE.EXE to execute the command.

You can include in the command string any parameters and qualifiers defined for the SAMPLE command verb. For example, you can enter the following command string:

```
$ SAMPLE MYFILE
```

In this case, you receive the following display on your screen:

```
FILESPEC = MYFILE
```

You can also include the /EDIT qualifier in the command string. For example:

```
$ SAMPLE MYFILE/EDIT
```

In this case, you receive the following display on your screen:

```
/EDIT IS PRESENT
FILESPEC = MYFILE
```

If you include a qualifier that is not accepted by the command verb, you receive a DCL error message. For example:

```
$ SAMPLE MYFILE/UPDATE
%DCL-W-IVQUAL, unrecognized qualifier - check validity, spelling, and -
 placement
   \UPDATE\
```

If you include two or more parameters in the command string for a verb that was defined to accept only one parameter, you receive an error message. For example:

```
$ SAMPLE MYFILE INFILE
%DCL-W-MAXPARM, too many parameters - reenter command with fewer parameters
   \INFILE\
```

Creating an Object Module Table for Your Program

This example shows how to create an object module table for your program. It also shows how to use command language routines to parse a command string and to invoke the correct program routine.

When you write a command definition file to create an object module table, specify routines (not images) for each command verb. Your program calls these routines when it processes command strings.

The following example illustrates a command definition file called TEST.CLD that defines three verbs: SEND, SEARCH, and EXIT. Each verb invokes a routine in the program USEREXAMP.BAS.

```
MODULE TEST_TABLE

DEFINE VERB SEND
        ROUTINE SEND_COMMAND
        PARAMETER P1, LABEL = FILESPEC
        QUALIFIER EDIT

DEFINE VERB SEARCH
        ROUTINE SEARCH_COMMAND
        PARAMETER P1, LABEL = SEARCH_STRING

DEFINE VERB EXIT
       ROUTINE EXIT_COMMAND
```

Process TEST.CLD by using SET COMMAND with the /OBJECT qualifier to create object module TEST.OBJ:

```
$ SET COMMAND/OBJECT TEST
```

You can then link TEST.OBJ with an object module that was created from your source program.

The following BASIC program, entitled USEREXAMP.BAS, invokes the routines listed in the command table in TEST.OBJ. It uses the command language routines CLI$DCL_PARSE and CLI$DISPATCH to parse command strings and to invoke the routine associated with the command. The program also uses CLI$PRESENT and CLI$GET_VALUE to obtain information about command strings.

```
10  SUB SEND_COMMAND
    EXTERNAL INTEGER FUNCTION CLI$PRESENT,CLI$GET_VALUE
```

```
    PRINT 'SEND COMMAND'
    PRINT ''

20  IF CLI$PRESENT ('EDIT') AND 1%
    THEN
         PRINT '/EDIT IS PRESENT'

30  IF CLI$PRESENT ('FILESPEC') AND 1%
    THEN
         CALL CLI$GET_VALUE ('FILESPEC',A$)
         PRINT 'FILESPEC = ',A$

90  SUBEND

100 SUB SEARCH_COMMAND
    EXTERNAL INTEGER FUNCTION CLI$PRESENT,CLI$GET_VALUE
    PRINT 'SEARCH COMMAND'
    PRINT ''

110 IF CLI$PRESENT('SEARCH_STRING') AND 1%
    THEN
         CALL CLI$GET_VALUE('SEARCH_STRING',A$)
         PRINT 'SEARCH_STRING = ',A$
190 SUBEND

200 SUB EXIT_COMMAND
    CALL SYS$EXIT(1% BY VALUE)

290 SUBEND

1   EXTERNAL INTEGER FUNCTION CLI$DCL_PARSE,CLI$DISPATCH
    EXTERNAL INTEGER FUNCTION SEND_COMMAND,SEARCH_COMMAND,EXIT_COMMAND
    EXTERNAL INTEGER TEST_TABLE,LIB$GET_INPUT

2   IF NOT CLI$DCL_PARSE(,TEST_TABLE,LIB$GET_INPUT,LIB$GET_INPUT,'TEST>') -
      AND 1%
    THEN
         GOTO 2

3   PRINT ''
    CALL CLI$DISPATCH
    PRINT ''
    GOTO 2
    END
```

This source program must be compiled before it can be linked with an object module created from the SET COMMAND/OBJECT command. To compile this program, invoke the VAX BASIC compiler:

```
$ BASIC USEREXAMP
```

You now have a USEREXAMP.OBJ file in addition to the original USEREXAMP.BAS source file. Link USEREXAMP.OBJ with TEST.OBJ by entering the following command:

```
$ LINK USEREXAMP,TEST
```

You now have a file containing an executable image (USEREXAMP.EXE). To execute the image, enter the following command:

```
$ RUN USEREXAMP
```

USEREXAMP.EXE displays the following prompt on your screen:

```
TEST>
```

You can now enter any of the commands you defined in TEST.CLD. For example:

```
TEST> SEND
```

The program calls CLI$DCL_PARSE to parse the command string SEND. SEND is a valid command, so CLI$DISPATCH transfers control to the SEND_COMMAND routine. This routine displays the following text:

```
SEND COMMAND
TEST>
```

You can also include a parameter with the SEND command. For example:

```
TEST> SEND MESSAGE.TXT
```

DCL invokes the SEND_COMMAND routine, which displays the following text:

```
SEND COMMAND
FILESPEC = MESSAGE.TXT
TEST>
```

You can also enter the /EDIT qualifier with SEND. For example:

```
TEST> SEND/EDIT MESSAGE.TXT

SEND COMMAND
/EDIT is present
FILESPEC = MESSAGE.TXT
TEST>
```

You can enter other commands that your program accepts. For example:

```
TEST> SEARCH
```

The SEARCH command string invokes a different routine from the one defined by SEND. In this case, the screen displays the following text:

```
SEARCH COMMAND
TEST>
```

Unlike the SEND command, the SEARCH command accepts no qualifiers. If you attempt to include a qualifier (such as /EDIT) in the SEARCH command string, CLI$DCL_PARSE signals the following error:

```
%CLI-W-NOQUAL, qualifier not allowed on this command
```

To exit from the USEREXAMP program and return to the DCL command level, issue the EXIT command:

```
TEST> EXIT
```

Chapter 2. Librarian Utility

2.1. LIBRARIAN Description

Libraries are files that you can use to store modules of code or text. The DCL LIBRARY command invokes the Librarian utility.

This chapter describes how to use the Librarian utility (LIBRARIAN) to create, access, and maintain libraries on all OpenVMS platforms. To use Librarian (LBR) routines to create and maintain libraries and their modules, see the following documentation:

- *HP OpenVMS Version 8.2 New Features and Documentation Overview* – Describes new LBR routines for Executable and Linking Format (ELF) object libraries and existing routines extended for ELF object libraries (OpenVMS I64 only).
- *VSI OpenVMS Utility Routines Manual* – Describes additional LBR routines that are the same on OpenVMS I64 systems and OpenVMS Alpha systems.

2.1.1. Types of Libraries

You can use LIBRARIAN to maintain the following types of libraries:

- Object libraries—Contain the object modules of frequently called routines. The OpenVMS Linker searches specified object module libraries when it encounters a reference it cannot resolve in one of its input files. For more information about how the linker uses libraries, see the *VSI OpenVMS Linker Utility Manual*.

  An object library has a default file type of .OLB and defaults the file type of input files to .OBJ.

- Macro libraries—Contain macro definitions used as input to the assembler. The assembler searches specified macro libraries when it encounters a macro that is not defined in the input file. See the *VAX MACRO and Instruction Set Reference Manual* for information about defining macros on OpenVMS VAX systems. For information on porting VAX MACRO code to an OpenVMS Alpha systems, see *VSI OpenVMS MACRO Compiler Porting and User's Guide*. For information on porting code to I64 systems, see *Porting Applications from VSI OpenVMS Alpha to VSI OpenVMS Industry Standard 64 for Integrity Servers*.

  A macro library has a default file type of .MLB and defaults the file type of input files to .MAR.

- Help libraries—Contain modules of help text that provide user information about a program. You can retrieve help text at DCL level by executing the DCL command HELP or, in your program, by calling the appropriate LBR routines. For information about creating help modules for insertion into help libraries, see Section 2.1.5.

  A help library has a default file type of .HLB and defaults the file type of input files to .HLP.

- Text libraries—Contain sequential record files that you want to retrieve as data for a program. For example, program source code can be stored in text libraries. Each text file inserted into the library corresponds to one library module. Your programs can retrieve text from text libraries by calling the appropriate LBR routines.

  A text library has a default file type of .TLB and defaults the file type of input files to .TXT.

- Shareable image libraries—Contain the symbol tables of shareable images used as input to the linker. For information about how to create a shareable image library, see Section 2.1.4

  A shareable image library has a default type of .OLB and defaults the file type of input files to .EXE.

You can create library files that do not have the default file type. For example, you can create the object library LIB.xxx by entering the following:

```
$ LIBRARY/CREATE/OBJECT LIB.xxx *.obj
```

You can then access the object library by entering the following:

```
$ LIBRARY/LIST LIB.xxx
```

Table 2.1 shows the libraries that are created by the Librarian utility for each OpenVMS platform:

Table 2.1. Libraries Created by OpenVMS Platform

OpenVMS VAX	OpenVMS Alpha	OpenVMS I64
VAX object	Alpha object	I64 object
VAX sharable image	Alpha sharable image	I64 sharable image
Alpha object[1]	VAX object[1]	
Alpha sharable image[2]	VAX sharable image[2]	
Macro	Macro	Macro
Text	Text	Text
Help	Help	Help

[1]Use /ALPHA qualifier to create and manipulate Alpha object and sharable image libraries.
[2]Use /VAX qualifier to create and manipulate VAX object and sharable image libraries.

2.1.2. Structure of Libraries

Every library contains a library header that describes the contents of the library; for example, its type, size, version number, creation date, and number of indexes.

Similarly, each module in the library has a module header that contains information about the module, including its type, attributes, and date of insertion into the library.

All libraries contain an index called the module name table (MNT); the keys in the MNT are the names of the modules in the library. Object module libraries also contain an index called the global symbol table (GST); the keys in the GST are the names of the symbols associated with each of the library modules.

For I64 systems, these symbols also include Unix-style weak symbol definitions and Group symbol definitions.

Note that the MNT catalogs modules by module name, rather than by the name of the input file that contained the inserted module. The only exception to this procedure occurs with text libraries, for which the file name of the input file containing the text automatically becomes the module name (unless you use the /MODULE qualifier).

2.1.3. Character Case of Library Keys

The character case of module names and symbols in libraries depends on the library type:

- Help libraries—Module names remain in the format they were entered; that is, individual uppercase and lowercase characters are preserved. However, a second, identically spelled module name (but of a different or mixed character case) to the same library replaces the previous module name, and character case is ignored for match operations. For example, *help Sample* and *help SAMPLE* access the same module.
- Object libraries — Module names and symbol names are in the format in which they were entered. Match operations require the character case to be identical. For example, for *SAMPLE*, you must enter *SAMPLE*, not *Sample* or *sample*.
- Text and macro libraries — By default, all module names are converted to uppercase characters. For example, *Sample* becomes *SAMPLE*. Likewise, for match operations, either *Sample* or *sample* matches *SAMPLE*.

  You can override this default behavior by using the CASE_SENSITIVE option to the /CREATE qualifier. If you specify CASE_SENSITIVE:YES, module names remain in the format they were entered, individual uppercase and lowercase characters are preserved, and match operations require the character case to be identical.

  To use the default behavior for macro and text libraries, do not include the CASE_SENSITIVE option with the /CREATE qualifier, or specify CASE_SENSITIVE:NO.

  The CASE_SENSITIVE option works only for macro and text libraries. If you try to use it for other library types, you will get an error message and the library creation operation will abort (no library is created).

2.1.4. Shareable Image Libraries

A shareable image library or ELF sharable image library is made up of only the symbol tables of shareable images, which serve as input to the linker. To create a shareable image library, use the LIBRARY command with the /SHARE qualifier, as follows:

```
$ LIBRARY/CREATE/SHARE MYSHARLIB MYSHRIMG1,MYSHRIMG2,SHRIMG3
```

You can then specify the library in the LINK command exactly as if it were an object library.

```
$ LINK/MAP/FULL MYPROG, MYSHARLIB/LIBRARY
```

The linker includes in the link operation whatever shareable images are needed from references to MYSHARLIB.

To explicitly include a shareable image, use the /INCLUDE qualifier.

```
$ LINK/MAP/FULL MYPROG, MYSHARLIB/INCLUDE=(MYSHRIMG1)/LIBRARY
```

For each shareable image found that either contains a necessary symbol or was specifically requested with the /INCLUDE qualifier, the linker looks up the image file (default file type is .EXE) and processes it as if it had been specified in an options file.

Unless the search is disabled with the /NOSYSSHR qualifier, the linker also searches the library SYS$LIBRARY:IMAGELIB.OLB after processing any user default libraries (LNK$LIBRARY). Modules found in IMAGELIB.OLB are opened with a default file specification of SYS$LIBRARY:.EXE.

The default file type for the LIBRARY/SHARE command is .OLB for the shareable image symbol table library and .EXE for the input shareable image files.

LIBRARIAN uses the GSMATCH identification numbers (IDs) of the shareable image as the module ID in the library. This provides a convenient way to verify that the shareable image information in the library matches the shareable image information contained in a map file from a link operation.

Note that a module inserted into a shareable image library contains only module header information, which is extracted from the shareable image. Consequently, although it is not an invalid action, there is little reason to extract modules from a shareable image library.

2.1.5. Help Libraries

Help text is a convenient means of providing specific information about a program to an interactive user. The help text is stored as modules in help libraries. You can access the help modules by using the DCL command HELP or by calling the appropriate LBR routines (described in the *VSI OpenVMS Utility Routines Manual*). In this way, a user can quickly retrieve relevant information about how to use your program.

You create help libraries the same way you create object, macro, and text libraries, using the LIBRARY/CREATE command. However, before you can insert modules into a help library, you must format the input file so that LIBRARIAN can catalog its individual modules. Section 2.1.5.1 and Section 2.1.5.2 describe how to create input files containing help modules.

2.1.5.1. Creating Help Files

The input files that you insert into help libraries are text files that you build with a program or a text editor. Each input file can contain one or more help modules. A help module is made up of the lines of help text that relate to the same topic, or key.

Each module within a help library contains a group of related keys, or topics, numbered key 1 through key 9. Each key represents a level within the hierarchy of the module. The key-1 name identifies the main topic of help information; for example, the name of a command in your program that requires explanation. The key-2 through key-9 names identify subtopics that are related to the key-1 name; for example, the command's parameters or qualifiers or both. This organization enables users of your program to find general information describing how to use the command and then to select subtopics that provide additional information about the command's parameters and qualifiers. The maximum length of a key-1 name is determined by the key size option of the /CREATE qualifier.

Index keywords remain in the format they were entered, that is, the character case is unchanged. A second keyword to the same library, identically spelled but of a different or mixed character case, replaces the previous preserved keyword. Character case is ignored for match operations. For example, *help Sample* and *help SAMPLE* access the same module.

The key names for help topics and subtopics can include any printable ASCII characters except those used by LIBRARIAN as either delimiters (space, horizontal tab, and comma) or comments (exclamation point).

VSI recommends that you restrict key names to the following characters:

- Uppercase and lowercase letters (A,a,B,b . . . Z,z)
- Digits (0,1,2 . . . 9)
- Dollar sign ($)

- Underscore (_)
- Hyphen (-)

VSI also recommends that you avoid—especially as the first character of a key name—certain characters that have special meaning to LIBRARIAN retrieval routines. If you use these characters in key names, you might not be able to specify them explicitly for retrieval.

The characters you should not use are as follows:

- Asterisk (*)
- Percent sign (%)
- Ellipsis (...)
- At sign (@)
- Slash (/)
- Question mark (?)
- Left parenthesis (() used as a first character
- Apostrophe (’) and quotation marks (")

If a key contains any of these characters, you might be able to retrieve its help text by abbreviating the key to avoid the special characters or by using wildcard characters in their place. For information about using wildcard characters, see the *VSI OpenVMS User's Manual*.

Also, note that you cannot abbreviate your retrieval key if it contains wildcard characters.

2.1.5.2. Formatting Help Files

Each line in a help module consists of the key number in the first column, followed by the name of the key. The subtopic lines that follow (key 2 through key 9) consist of the subkey number followed by the name of the subkey. For example, a help module for a command might have the following two key lines:

```
1 Command name
   .
   .
   .
   help text
   .
   .
   .
2 Parameters
```

Each help source file can contain several modules. LIBRARIAN recognizes a group of key-1 and subkey lines and their associated text as a module. A module is terminated either by another key-1 line or by an end-of-file record.

A help source file has the following format:

```
1 key-1 name
   .
   .
   .
   help text
   .
```

```
   .
   .
2 key-2 name
   .
   .
   .
   help text
   .
   .
   .
n key-n name
   .
   .
   .
1 key-1 name
```

LIBRARIAN stores the key-1 name in its module name table; therefore, the name of the module is the same as the key-1 name. The subsequent numbers in the first column indicate that the line is a subkey. A module can have several subkeys with the same number. For example, a help module describing a command might have the following key-2 lines:

```
2  parameters
2  arguments
```

You can insert comments anywhere in a module. When LIBRARIAN encounters an exclamation point as the first character on a line, it assumes that the line consists of comments. Comment lines that follow a key-1 line are included in the module. However, when your program retrieves help text, LIBRARIAN does not display the comments.

The help text can be any length; the only restriction to the text is that it cannot contain a number or a slash (/) in the first column of any line. A number in the first column of a line indicates that the line is a key. A slash in the first column indicates a qualifier line.

A qualifier line is similar to a key line, except that LIBRARIAN returns a list of all the qualifier lines when you request help either on a key-1 topic or on the key containing the qualifiers (usually a key-2 topic named “Qualifiers”). Therefore, if your help module describes a command that has qualifiers, LIBRARIAN provides a list of all the command's qualifiers whenever you request help on the command.

2.1.5.3. Help Text Example

The help module in Example 2.1 shows the organization of some of the help text for the DCL command LIBRARY.

Example 2.1. Help Text for LIBRARY Command

```
1 LIBRARY
  Invokes the Librarian utility to create, modify, or describe an
  object, macro, help, text, or shareable image library.

  Format:

    LIBRARY library-file-spec [input-file-spec[,...]]
2 Command Parameters

  library-file-spec

    Specifies the name of the library you want to create or modify.
```

```
    If the file specification does not include a file type, the LIBRARY
    command assumes a default type of .OLB, indicating an object library.

  input-file-spec[,...]

    Specifies the names of one or more files that contain modules you
    want to insert or replace in the specified library.

    If you specify more than one input file, separate the file
    specifications with commas. The input-file-spec parameter is
    required when you specify /REPLACE, which is the LIBRARY command's
    default operation, or /INSERT, which is an optional qualifier.

    When you use the /CREATE qualifier to  create  a  new  library,
    the input-file-spec parameter is optional. If you include an input
    file specification with /CREATE, the LIBRARY command first creates
    a new library and then inserts the contents of the input files into
    the library.

2 Command_Qualifiers

  /BEFORE
   /BEFORE[=time]

    Used in conjunction with the /LIST qualifier to  specify  that only
    those modules dated earlier than a particular time be listed. You
    can specify an absolute time or a combination of absolute and delta
    times.

     If you omit the /BEFORE qualifier, all modules are listed regardless
     of date. If you specify /BEFORE without a date or time, all modules
     created before today are listed by default.

  /COMPRESS
   /COMPRESS[=(option[,...])]

    Recovers space that had been occupied by modules deleted from the
    library. When you specify /COMPRESS, the LIBRARY command by default
    creates a new version of the library in your current default
    directory.

    You can use options to the /COMPRESS  qualifier to make some
    specifications in the new version of the library different from the
    original library.

  /CREATE
   /CREATE[=(option[,...])]

    Creates a new library. When you specify /CREATE, you can optionally
    specify a file or a list of files that contains modules to be placed
    in the library.

    By default, the LIBRARY command creates an  object  module  library;
    specify /SHARE, /MACRO, /HELP, or /TEXT to change the default
    library type.
        .
        .
        .
```

2.1.5.4. Retrieving Help Text

You can retrieve help text at DCL level by using the DCL command HELP or, in your program, by calling the appropriate Librarian utility (LBR) routines (as described in the *VSI OpenVMS Utility Routines Manual*.

By default, the HELP command retrieves help text from the system help library SYS$HELP:HELPLIB.HLB and from user help libraries associated with the logical names HLP$LIBRARY, HLP$LIBRARY_1, HLP$LIBRARY_2, and so forth. Using the /LIBRARY qualifier with the HELP command lets you search a library other than the default libraries. For more information, see the description of the HELP command in the *VSI OpenVMS DCL Dictionary*.

When you retrieve help text, you specify the key-1 topic followed by any subtopics that contain appropriate help information. LIBRARIAN returns the help text associated with the key path you specify. For example, the help text for the LIBRARY command is stored in the default system help library; thus, to retrieve the LIBRARY command's key-1 help information, you would enter the DCL command HELP LIBRARY. LIBRARIAN would return the associated help text, followed by the message “Additional information available:” and a list of all the key-2 names in the module. In this case, LIBRARIAN also returns a list of all the qualifiers specified in the qualifier lines. Example 2.2 displays the text returned by the HELP LIBRARY command.

Example 2.2. HELP LIBRARY Display

```
LIBRARY

     Invokes the Librarian utility to create, modify, or describe an
     object, macro, help, text, or shareable image library.

     Format:

       LIBRARY library-file-spec [input-file-spec[,...]]

  Additional information available:

  Command_Parameters   /ALPHA    /BEFORE    /COMPRESS   /CREATE
  /CROSS_REFERENCE     /DATA     /DELETE    /EXTRACT    /FULL     /GLOBALS
  /HELP    /HISTORY    /INSERT   /LIST      /LOG        /MACRO    /MODULE
  /NAMES   /OBJECT     /ONLY     /OUTPUT    /REMOVE     /REPLACE
  /SELECTIVE_SEARCH    /SHARE    /SINCE     /SQUEEZE    /TEXT     /VAX
  /WIDTH
```

Note that you cannot retrieve the key-2 level “Parameters” by entering HELP PARAMETERS. LIBRARIAN searches for a subkey only after finding the higher level keys. In other words, if you want to retrieve key-3 text, you have to specify the key-1 and key-2 lines that form a path to the key-3 line.

Also note that you can provide information about a qualifier that has more than one form by associating two or more qualifier lines with the same help text. That is, the text associated with the qualifiers follows the two or more qualifier lines. For example:

```
$ HELP LIBRARY/GLOBALS

LIBRARY

  /GLOBALS
```

```
   /GLOBALS
   /NOGLOBALS

    Controls, for object module libraries, whether the names of global
    symbols in modules being inserted in the library are included in the
    global symbol table.

    By default, the LIBRARY command places all global symbol names in
    the global symbol table. Use /NOGLOBALS if you do not want the
    global symbol names included in the global symbol table.
```

When LIBRARIAN successfully searches the key path to the requested key, it displays all the key names in that path, followed by the help text associated with the last specified key. For example:

```
$ HELP LIBRARY/HELP

LIBRARY

   /HELP

    Indicates that the library is a help library. When you specify the
    /HELP    qualifier, the library file type defaults to .HLB and the
    input file type defaults to .HLP.
```

If you try to retrieve help text for a key that is not in the module name table, LIBRARIAN issues a message. For example:

```
$ HELP FIRE

Sorry, no documentation on FIRE

Additional information available:
```

This message is followed by a list of all the module names in the module name table.

If you have correctly specified the key-1 line but have requested a subkey that does not exist, LIBRARIAN issues a message. For example:

```
$ HELP LIBRARY/FIRE

Sorry, no documentation on LIBRARY/FIRE

Additional information available:

Parameters Command_Qualifiers
/BEFORE    /COMPRESS
   .
   .
   .
```

The message includes a list of all the subkeys associated with the last key that was specified correctly.

2.1.6. Using the Librarian Utility to Save Disk Space

You can save disk space by using the Librarian utility to reduce data files. To save disk space, create a library, copy the data files you want to reduce into the library, and then use the LIBRARY/DATA=REDUCE command to reduce the size of library. When you subsequently want to use the files in

their expanded form, you simply extract the data files and they will be expanded automatically. Because data expansion takes time, leave frequently used libraries in their expanded format to increase access performance.

Large, infrequently accessed files are good candidates for this method when you do not want to write a program that uses the callable interface to reduce and expand data files.

Note

For I64 systems, VSI strongly recommends that DCX data-reduced ELF object libraries first be expanded before performing Linker operations.

See the description of the /DATA qualifier for more information.

2.1.7. Librarian Utility (LBR) Routines

Programs can call Librarian utility (LBR) routines to do the following:

- Initialize a library
- Open a library
- Look up a key in a library
- Insert a new key in a library
- Return the names of the keys
- Delete a key and its associated data
- Read and write information

For VAX and Alpha systems, the *VSI OpenVMS Utility Routines Manual* describes in detail each LBR routine. For information about LBR routines on I64 systems, see the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

2.1.8. LIBRARIAN Usage Summary

LIBRARIAN

LIBRARIAN — The Librarian utility (LIBRARIAN) gives you easy access to libraries. Libraries are files in which you can store frequently used modules of code or text. You can use the DCL command LIBRARY (or the LBR routines) to create a library, maintain the modules in a library, or display information about a library and its modules. Note that libraries are files, so you can use DCL commands to manipulate libraries in their entirety; for example, you can use the DELETE, COPY, and RENAME commands to delete, copy, and rename libraries. For more information about file maintenance, see the *VSI OpenVMS DCL Dictionary*.

Format

```
LIBRARY library-file-spec [input-file-spec[,...]]
```

Command Parameters

library-file-spec

The name of the library you want to create or modify. This parameter is required. If you do not specify a library file, you are prompted for one, as follows:

```
_Library:
```

No wildcard characters are allowed in the library file specification.

If the file specification does not include a file type and if the command string does not indicate a library type, the LIBRARY command assumes an object library type and a default file type of .OLB. You can change the default library file type by specifying the appropriate qualifier, as follows:

Qualifier	Default Library File Type
/HELP	.HLB
/MACRO	.MLB
/OBJECT	.OLB
/TEXT	.TLB
/SHARE	.OLB

input-file-spec[,...]

The names of one or more files that contain modules you want to insert into the specified library. If you specify more than one input file, separate the file specifications with commas.

The input file specification is required when you specify /REPLACE, which is the LIBRARY command's default operation, or /INSERT, which is an optional qualifier. If you do not specify an input file when you use these qualifiers, you are prompted for it, as follows:

```
_File:
```

When you use the /CREATE qualifier to create a new library, the input file specification is optional. If you include an input file specification with the /CREATE qualifier, LIBRARY first creates a new library and then inserts the contents of the input files into the library.

Note that the /EXTRACT qualifier does not accept an input file specification.

If any file specification does not include a file type and if the command string does not indicate one, LIBRARY assumes a default file type of .OBJ, designating an object file. You can control the default file type by specifying the appropriate qualifier, as follows:

Qualifier	Default Input File Type
/HELP	.HLP
/MACRO	.MAR
/OBJECT	.OBJ
/TEXT	.TXT
/SHARE	.EXE

Note also that the file type you specify with the library file specification determines the default file type of the input file specification, provided that you do not specify the /CREATE qualifier. For example, if the library file type is .HLB, .MLB, .OLB, or .TLB, the input file type default will

be .HLP, .MAR, .OBJ, or .TXT, respectively. (If you specify the /CREATE qualifier and you are not creating an object library, you must use the appropriate file type qualifier.)

Wildcard characters are allowed in the input file specifications.

Usage Summary

The DCL command LIBRARY invokes the Librarian utility. After the operations specified by LIBRARY have completed, the Librarian utility exits.

If you use the /LIST qualifier to request information about a library, the output is directed to the file specification associated with /LIST or, if you do not supply a file specification, to SYS$OUTPUT.

2.1.9. LIBRARIAN Qualifiers

When using LIBRARY, you can specify qualifiers that request more than one function in a single command, with some restrictions. Generally, you cannot specify multiple qualifiers that request incompatible functions. The qualifiers that perform library functions, related qualifiers, and qualifier incompatibilities are summarized in Table 2.2.

Table 2.2. LIBRARY Command Qualifier Compatibilities

Qualifier	Related Qualifiers	Incompatible Qualifiers
/COMPRESS	/OUTPUT	/CREATE, /EXTRACT
/CREATE[1]	/SQUEEZE,[2] /GLOBALS,[3] /SELECTIVE_SEARCH[3]	/COMPRESS, /EXTRACT
/CROSS_REFERENCE	/ONLY	/EXTRACT, /LIST
/DATA	/COMPRESS	—
/DELETE	—	/EXTRACT
/EXTRACT	/OUTPUT	/COMPRESS, /CREATE, /DELETE, /INSERT, /LIST, /REMOVE, /REPLACE
/INSERT	/SQUEEZE,[2] /GLOBALS,[3] /SELECTIVE_SEARCH[3]	/EXTRACT
/LIST	/FULL, /NAMES,[3] /ONLY, /HISTORY, /BEFORE, /SINCE	/EXTRACT, /CROSS_REFERENCE
/REMOVE[3]	—	/EXTRACT
/REPLACE	/SQUEEZE,[2] /GLOBALS,[3] / SELECTIVE_SEARCH[3]	/EXTRACT
/MODULE[4]	/TEXT	/EXTRACT, /DELETE, /REMOVE

[1]The /CREATE, /INSERT, and /REPLACE qualifiers are compatible; however, if you specify more than one, /CREATE takes precedence over /INSERT, and /INSERT takes precedence over /REPLACE. The related qualifiers for /CREATE are applicable only if you enter one or more input files.

[2]This qualifier applies only to macro libraries.

[3]This qualifier applies only to object libraries and shareable image libraries.

[4]This positional qualifier applies only to text libraries.

Note that all the qualifiers are command qualifiers except for /MODULE, which is a positional qualifier that modifies the input file specification parameter.

/ALPHA

/ALPHA (VAX and Alpha only) — Directs LIBRARIAN to work with an OpenVMS Alpha object library when used with the /OBJECT qualifier or to work with an OpenVMS Alpha shareable image library when used with the /SHARE qualifier. When used with the /CREATE qualifier, LIBRARIAN creates an OpenVMS Alpha library of either an object or shareable image type depending whether /OBJECT or /SHARE is specified. The default is /ALPHA on OpenVMS Alpha systems and /VAX on OpenVMS VAX systems. The OpenVMS I64 Librarian does not accept this qualifier because the I64 Librarian works exclusively with I64 libraries.

Format

/ALPHA

Description

The /ALPHA qualifier is used to create and manipulate OpenVMS Alpha object and shareable image libraries. Because the formats of macro, help, and text libraries are identical on both system architectures, using the /ALPHA qualifier with the /MACRO, /TEXT, and /HELP qualifiers has no effect.

Note

You cannot have both OpenVMS Alpha and OpenVMS VAX object modules in one object library, nor can you have OpenVMS Alpha and OpenVMS VAX shareable images in the same shareable image library.

Examples

1. `$ LIBRARY/ALPHA/CREATE TESTLIB ERRMSG.OBJ,STARTUP.OBJ`

   This LIBRARY command creates an OpenVMS Alpha object library named TESTLIB.OLB and places the files ERRMSG.OBJ and STARTUP.OBJ as modules in the library.

2. `$ LIBRARY/ALPHA/SHARE/CREATE SHARELIB.OLB`

   This LIBRARY command creates an OpenVMS Alpha shareable image library called SHARELIB.OLB.

/BEFORE

/BEFORE — Specifies that only those modules inserted earlier than a particular time be listed.

Format

/BEFORE[=time]

time
Limits the modules to be listed to those inserted in the library before a specified time.

You can specify an absolute time or a combination of absolute and delta times. For details about specifying times, see the *VSI OpenVMS DCL Dictionary*.

Description

This qualifier is used with the /LIST qualifier. If you omit the /BEFORE qualifier, you obtain all the modules regardless of the dates. If you specify /BEFORE without a date or time, the default is to provide the modules inserted before today.

Example

```
$ LIBRARY/LIST/BEFORE=15-APR-:15 MATHLIB
```

This LIBRARY command lists the modules that were inserted into MATHLIB.OLB before 3 p.m. on April 15.

/COMPRESS

/COMPRESS — Recovers space that was occupied by modules deleted from the library. When you specify /COMPRESS, LIBRARY creates a new library. You can use options to the /COMPRESS qualifier to make some specifications in the new version of the library different from the original library.

Format

/COMPRESS[=(option[,...])]

option
An option that alters the size or format of the library, overriding the values specified when the library was created. Options are listed in the Description section.

Description

When you specify /COMPRESS, LIBRARY creates a new library. By default, the new library is created in your current default directory and has the same file name as the existing library and a file type that is the default for the type of library created. You can use the /OUTPUT qualifier to specify an alternate file specification for the compressed library.

Specify one or more of the following options to alter the size or format of the library, overriding the values specified when the library was created (for the default values, see the description of the /CREATE qualifier):

BLOCKS:n	Specifies the number of 512-byte blocks to be allocated for the library. By default, LIBRARY allocates 100 blocks for a new library.
GLOBALS:n	Specifies the maximum number of symbols the library can contain initially. By default, LIBRARY sets a maximum of 512 symbols for an object module library. (Macro, help, and text libraries do not have a symbol directory; therefore, the maximum for these libraries defaults to 0.)
HISTORY:n	Specifies the maximum number of library update history records that the library is to maintain. The maximum number of library update records you can specify is 32,767. The default is 20.
KEEP	Copies library update history records and any additional user data in the module header to the compressed library.
KEYSIZE:n	Specifies the maximum name length of modules or symbols. On VAX systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in object or macro libraries, and 39

	characters for modules in text or shareable image libraries. The maximum length you can specify for these names is 128 characters. Also on VAX systems, when you specify a key size value, remember that the MACRO compiler and the linker do not accept module names or global symbol names in excess of 31 characters. On Alpha systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in macro libraries, 39 characters for modules in text libraries, and 128 characters for modules in object or shareable image libraries. The maximum length you can specify for these names is 128 characters. Also on Alpha systems, when you specify a key size value, remember that the MACRO compiler does not accept module names and global symbol names in excess of 31 characters, and the linker does not accept module names in excess of 31 characters or global symbol names in excess of 64 characters. On I64 systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in macro libraries, 39 characters for modules in text libraries, and 1024 characters for modules in object or shareable image libraries. The maximum length you can specify for these names is 1024 characters.
MODULES:n	Specifies the maximum number of modules the library can initially contain. By default, LIBRARY sets an initial maximum of 128 modules for all library types. A library's size can grow past its initial allocation. However, for optimum performance, it is best to allocate the maximum number of modules you expect to use.
VERSION:n	On VAX systems, specifies that the library is to be stored in VMS Version 2.0 library format, if `n` is 2; or VMS Version 3.0 library format, if `n` is 3. On Alpha and I64 systems, this parameter is ignored.

If you specify more than one option, separate them with commas and enclose the list in parentheses.

Example

```
$ LIBRARY/COMPRESS=(KEYSIZE:40,MODULES:80)/TEXT SOURCE
```

This LIBRARY command creates a new version of the text library SOURCE.TLB. Space left after modules were deleted from the old version is recovered in the new version. The new version can contain up to 80 modules; the maximum length of module names in the new version is 40.

/CREATE

/CREATE — Requests the DCL command LIBRARY to create a new library. When you specify /CREATE, you can optionally specify a file or a list of files that contains modules to be placed in the library.

Format

/CREATE[=(option[,...])]

option

An option that overrides the system defaults to control the size or format of the library. Options are listed in the Description section.

Description

By default, the /CREATE qualifier creates an object module library. To indicate that the library is a macro, help, text, or shareable image library, specify /MACRO, /HELP, /TEXT, or /SHARE.

On OpenVMS VAX systems, the /CREATE qualifier creates a VAX library by default when used to create object and shareable image libraries. Note that you cannot have VAX modules and Alpha modules in the same library. For more information, see the description of the /VAX (VAX and Alpha only) qualifier.

On OpenVMS Alpha systems, the /CREATE qualifier creates an Alpha library by default when used to create object and shareable image libraries. Note that you cannot have Alpha modules and VAX modules in the same library. For more information, see the description of the /ALPHA qualifier.

On OpenVMS I64 systems, the /CREATE qualifier creates an ELF library when used to create object and shareable image libraries.

Specify one or more of the following options to override the system defaults:

BLOCKS:n	Specifies the number of 512-byte blocks to be allocated for the library. By default, LIBRARY allocates 100 blocks for a new library.
CASE_SENSITIVE:[YES/NO]	Specifies whether key operations on macro or text libraries are case sensitive. The default, CASE_SENSITIVE:NO, causes all module names to be converted to uppercase. CASE_SENSITIVE:YES causes current and subsequent key operations to behave as they do for object libraries. See Section 2.1.3 for a full description. This option is valid only for macro and text libraries.
GLOBALS:n	Specifies the maximum number of symbols the library can contain initially. By default, LIBRARY sets a maximum of 512 symbols for an object module library. (Macro, help, and text libraries do not have a symbol directory; therefore, the maximum for these libraries defaults to 0.)
HISTORY:n	Specifies the maximum number of library update history records that the library is to maintain. The maximum number of library update records you can specify is 32,767. The default is 20.
KEYSIZE:n	Specifies the maximum name length of modules or symbols. On VAX systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in object or macro libraries, and 39 characters for modules in text or shareable image libraries. The maximum length you can specify for these names is 128 characters. Also on VAX systems, when you specify a key size value, remember that the MACRO compiler and the linker do not accept module names or symbol names in excess of 31 characters.

	On Alpha systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in macro libraries, 39 characters for modules in text libraries, and 128 characters for modules in object or shareable image symbol table libraries. The maximum length you can specify for these names is 128 characters. Also on Alpha systems, when you specify a key size value, remember that the MACRO compiler does not accept module names and symbol names in excess of 31 characters, and the linker does not accept module names in excess of 31 characters or global symbol names in excess of 64 characters. On I-64 systems, LIBRARY assigns default name lengths of 15 characters for help modules, 31 characters for modules in macro libraries, 39 characters for modules in text libraries, and 1024 characters for modules in object or shareable image symbol table libraries. The maximum length you can specify for these names is 1024 characters.
MODULES:n	Specifies the maximum number of modules the library can initially contain. By default, LIBRARY sets an initial maximum of 128 modules for all library types. A library's size can grow past its initial allocation. However, for optimum performance, it is best to allocate the maximum number of modules you expect to use.
VERSION:n	On VAX systems, specifies that the library is to be stored in OpenVMS Version 2.0 library format, if `n` is 2; or VMS Version 3.0 library format, if `n` is 3. On Alpha and I64 systems, this parameter is ignored.

If you specify more than one option, separate them with commas and enclose the list in parentheses.

Examples

1. `$ LIBRARY/CREATE TESTLIB ERRMSG,STARTUP`

   This LIBRARY command creates an object module library named TESTLIB.OLB and places the files ERRMSG.OBJ and STARTUP.OBJ as modules in the library.

2. ```
   $ LIBRARY/MACRO/CREATE=(BLOCKS:40,MODULES:100) MYMAC TEMP
   $ MACRO MYMAC/LIBRARY,CYGNUS/OBJECT
   ```

   This LIBRARY command creates a macro library named MYMAC.MLB from the macros in the file TEMP.MAR. The new library has room for 100 modules in a 40-block file. If the input file contains multiple macros, each macro is entered in the new library.

/CROSS_REFERENCE

/CROSS_REFERENCE — Requests a cross-reference listing of an object library.

Format

`[=(option[,...])]`
   ```

option
An option that produces a cross-reference listing that is not limited to only symbols by name and symbols by value. Options are listed in the Description section.

Description

If you omit this qualifier, cross-reference listings are not provided. If you specify /CROSS_REFERENCE without specifying an option, you get cross-reference listings that contain symbols by name and symbols by value. By default, the listing file is created in your current default directory and has the same file name as the library and a file type of .LIS. You can use the /OUTPUT qualifier to specify an alternate file specification for the listing file.

You can specify one or more of the following options:

ALL	All types of cross-references
MODULE	Cross-reference listing of both the symbol references in the module and the symbol definitions
NONE	No cross-reference listing
SYMBOL	Cross-reference listing by symbol name
VALUE	Cross-reference listing of symbols by value

If you specify more than one option, separate the options with commas and enclose the list in parentheses.

Example

```
$ LIBRARY/CROSS_REFERENCE=ALL/OUTPUT=SYS$OUTPUT LIBRAR
```

This LIBRARY command requests a cross-reference listing of the object library LIBRAR.OLB. The cross-reference listing is displayed at the terminal. The listing includes cross-references by symbol, by value, and by module.

/DATA

/DATA — Stores a library in data-reduced format or expands a library previously stored in data-reduced format.

Format

/DATA=option

option
The option REDUCE, which stores a library in data-reduced format, or the option EXPAND, which expands a library previously stored in data-reduced format. There is no default; you must specify one of the options.

Description

When you specify /DATA, the DCL command LIBRARY creates a new library. By default, the new library is created in your current default directory with the same file name as the existing library and a file type that is the default for the type of library created. You can use the /OUTPUT qualifier to specify an alternate file specification for the library.

You use the /DATA qualifier to control how data is stored in the library. If you specify /DATA=REDUCE, data in the library is stored in data-reduced format; less disk space is required for the library, but access to the library generally is slower.

Note

For I64 systems, VSI strongly recommends that DCX data-reduced ELF object libraries first be expanded before performing Linker operations.

If you omit this qualifier, data is stored in the library in standard, rather than data-reduced, format. You can change a data-reduced library back to the standard format by specifying /DATA=EXPAND.

When you use the /DATA qualifier (with either option), LIBRARIAN also recovers space in the library that had been occupied by modules deleted from the library, just as if you had specified /COMPRESS.

Example

```
$ LIBRARY/TEXT/DATA=REDUCE TEXTLIB
```

This LIBRARY command stores the data in the text library TEXTLIB.TLB in data-reduced format.

/DELETE

/DELETE — Requests the LIBRARY command to delete (physically remove) one or more modules from a library.

Format

/DELETE=(module[,...])

module
The name of the module to be deleted.

Description

You must specify the names of the modules to be deleted. If you specify more than one module, separate the module names with commas and enclose the list in parentheses.

Wildcard characters are allowed in the module specification.

If you specify the /LOG qualifier with /DELETE, LIBRARY issues the following message:

```
%LIBRAR-S-DELETED, MODULE module-name DELETED FROM library-name
```

Example

```
$ LIBRARY/DELETE=FREEZE/LOG THAW
```

This LIBRARY command physically removes the module FREEZE from the object library THAW. A message is displayed to confirm that the module was deleted.

/EXTRACT

/EXTRACT — Copies one or more modules from a library into a file.

Format

/EXTRACT=(module[,...])

module
The name of the module to be copied.

Description

If you specify more than one module, separate the module names with commas and enclose the list in parentheses.

Wildcard characters are allowed in the module specification.

If you specify the /OUTPUT qualifier with /EXTRACT, the LIBRARY command writes the output into the file specified by the /OUTPUT qualifier. If you do not specify a directory, the file is written to your current default directory. If you specify /EXTRACT and do not specify /OUTPUT, the LIBRARY command writes the file into a file that has the same file name as the library and a file type of .OBJ, .EXE, .MAR, .HLP, or .TXT, depending on the type of library.

Example

```
$ LIBRARY/EXTRACT=(ALLOCATE,APPEND)/OUTPUT=MYHELP SYS$HELP:HELPLIB.HLB
```

This LIBRARY command specifies that the modules ALLOCATE and APPEND are to be extracted from the help library HELPLIB.HLB and output to the file MYHELP.HLP in your current default directory.

/FULL

/FULL — Requests a full description of each module in the module name table.

Format

/FULL

Description

Use the /FULL qualifier with the /LIST qualifier to request a list of each library module in the following format, where `Ident nn` is the identification number of the module:

```
module-name [Ident  nn] Inserted dd-mmm-yyyy hh:mm:ss [n symbols]
```

The identification number and the number of symbols appear only in object libraries.

If an update history is maintained for the library, then /LIST/FULL/HISTORY lists the module name in the update history records.

Example

```
$ LIBRARY/LIST=MYMAC.LIS/FULL MYMAC.MLB
```

This LIBRARY command requests a full listing of the macro library MYMAC; the output is written to a file named MYMAC.LIS.

/GLOBALS

/GLOBALS — For object and shareable image module libraries, controls whether the names of global symbols in modules being inserted in the library, as well as Unix-style weak symbols and group symbols from ELF objects, are included in the global symbol table.

Format

/GLOBALS

/NOGLOBALS

Description

By default, the LIBRARY command places the module's symbol names in the global symbol table. Use /NOGLOBALS if you do not want the symbol names included in the global symbol table.

Example

```
$ LIBRARY/INSERT/NOGLOBALS TOOLS SPELL
```

This LIBRARY command inserts the modules in SPELL.OBJ into the object library TOOLS.OLB, but symbol names in the inserted modules are not included in the library's global symbol table.

/HELP

/HELP — Indicates that the library specified is a help library.

Format

/HELP

Description

When you use the /HELP qualifier, the library file type defaults to .HLB and the input file type defaults to .HLP.

Example

```
$ LIBRARY/HELP/CREATE ERRMSG EDITERRS
```

This LIBRARY command creates a help library called ERRMSG.HLB. Help text from the file EDITERRS.HLP is inserted into the library.

/HISTORY

/HISTORY — Requests that update history record headers be listed (for libraries that contain a history). The operation referred to in the header has one of three values: replaced, inserted, or deleted.

Format

username operation n modules on dd-mmm-yyy hh:mm:ss

Description

The /HISTORY qualifier is used with the /LIST qualifier. Use the /HISTORY qualifier with /LIST/FULL to request that the names of updated modules be listed in addition to the update history headers.

Example

```
$ LIBRARY/LIST/HISTORY/MACRO SETUP
```

This LIBRARY command lists the headers of the update history records in the macro library SETUP.MLB.

/INSERT

/INSERT — Requests the LIBRARY command to add the contents of one or more files to an existing library.

Format

/INSERT

Description

If an object module input file consists of concatenated object modules, the LIBRARY command creates a separate entry for each object module in the file; each module name table entry reflects an individual module name. If a macro or help file specified as input contains more than one definition, the LIBRARY command creates a separate entry for each one, naming the module name table entries according to the names specified in the .MACRO directives or in the key-1 name in the help format (see Section 2.1.5.1).

Unlike object, macro, and help libraries, the input file in text libraries contains data records of undefined contents. Therefore, LIBRARIAN catalogs the entire input file as a single module using the file name (not the directory or file type) as the module name. If you want to rename the inserted module, use the /MODULE qualifier.

When you use the /INSERT qualifier to insert modules into an existing library, the Librarian utility checks the module name table before inserting each module. If a module name or symbol name already exists in the library, an error message is issued and the module or symbol is not added to the library.

For OpenVMS VAX libraries, the maximum record size (established by LBR$C_MAXRECSIZ) that can be inserted into a library is 2048 bytes.

For OpenVMS Alpha and ELF libraries, the maximum record size (established by ELBR$C_MAXRECSIZ) that can be inserted into a library is 8192 bytes.

To insert or replace a module in a library, regardless of whether a current entry exists with the same name, use the /REPLACE qualifier (the default operation).

Example

```
$ LIBRARY/INSERT TESTLIB SCANLINE
$ LINK TERMTEST,TESTLIB/LIBRARY
```

This LIBRARY command adds the module SCANLINE.OBJ to the library TESTLIB.OLB. The library is specified as input to the linker by the /LIBRARY qualifier on the LINK command. If the module TERMTEST.OBJ refers to any routines or symbols not defined in TERMTEST, the linker searches the global symbol table of library TESTLIB.OLB to resolve the symbols.

/LIST

/LIST — Controls whether the LIBRARY command creates a listing that provides information about the contents of the library.

Format

/LIST[=file-spec]

/NOLIST

file-spec
The file specification of the file to be listed.

Description

By default, LIBRARY does not produce a listing. If you specify /LIST without a file specification, LIBRARY writes the output file to the current SYS$OUTPUT device. If you include a file specification that does not have a file type, LIBRARY uses the default file type .LIS.

If you specify /LIST with qualifiers that perform additional operations on the library, the LIBRARY command creates the listing after completing all other requests; thus, the listing reflects the status of the library after all changes have been made.

When you specify /LIST, the LIBRARY command provides, by default, the following information about the library:

```
Directory of OBJECT library _DBB0:[LIBRAR]LIBRAR.OLB;1 on 31-DEC-1993 -
  10:08:28
Creation date:  12-DEC-1992 19:40:36     Creator:  VAX-11 librarian V04-00
Revision date:  31-DEC-1992 16:04:58     Library format:  3.0
Number of modules:      15               Max. key length:  31
Other entries:        73                 Preallocated index blocks:   35
Recoverable deleted blocks:     15       Total index blocks used:     12
Max. update history records:    10       Update history records:      5
```

Examples

1. `$ LIBRARY/LIST=MYMAC.LIS/FULL MYMAC.MLB`

   This LIBRARY command requests a full listing of the macro library MYMAC; the output is written to a file named MYMAC.LIS.

2. `$ LIBRARY/LIST/NAMES/ONLY=$ONE/WIDTH=80 SYMBOLIB`

   This LIBRARY command requests a full listing of the module $ONE, contained in the object library SYMBOLIB.OLB. The /WIDTH qualifier requests that the global symbol display be limited to 80 characters per line.

3. `$ LIBRARY/INSERT/LIST ALLOBJECTS *`

   This LIBRARY command inserts into ALLOBJECTS.OLB all object modules from all object files in the current directory. If any of the modules to be inserted have the same name as an existing module in the library, the existing module is replaced. The LIBRARY command then lists the resulting library on SYS$OUTPUT.

/LOG

/LOG — Controls whether the LIBRARY command verifies each library operation.

Format

/LOG

/NOLOG

Description

If you specify /LOG, the LIBRARY command displays the module name, followed by the library operation performed, followed by the library file specification.

Other applications of the /LOG qualifier appear in the descriptions of /DELETE and /REPLACE.

Example

```
$ LIBRARY/REMOVE=(LIB_EXTRCT_MODS,LIB_INPUT_MAC)/LOG LIBRAR
```

This LIBRARY command requests the removal of the global symbols LIB_EXTRCT_MODS and LIB_INPUT_MAC from the object library LIBRAR.OLB. The /LOG qualifier requests that the removal of the symbols be confirmed by messages.

/MACRO

/MACRO — Indicates that the library specified is a macro library.

Format

/MACRO

Description

When you specify the /MACRO qualifier, the library file type defaults to .MLB and the input file type defaults to .MAR.

Example

```
$ LIBRARY/MACRO/INSERT MONTHS APRIL
```

This LIBRARY command inserts modules from APRIL.MAR into the macro library MONTHS.MLB.

/MODULE

/MODULE — Names a text module that you want to replace or insert into a text library. It also modifies the input file specification parameter.

Format

/MODULE=module-name

module-name
The name of the module to be inserted in the library.

Description

When you insert text modules into a library, the input file you specify is assumed to be a single module. Therefore, the file name of the input file specification becomes the module name. If you want the file

you are inserting to have a module name different from its input file name, use the /MODULE qualifier to name the added module.

You can also use the /MODULE qualifier to enter a text module interactively. If you specify the logical name SYS$INPUT as the input file and use the /MODULE qualifier, the LIBRARY command inserts the text you enter from the terminal into the specified library module. To terminate the terminal input, press Ctrl/Z.

Remember that the /MODULE qualifier is an input file qualifier; it assumes that you are either replacing or inserting a new text module. Therefore, the qualifiers that remove modules—/EXTRACT, /DELETE, /REMOVE—are incompatible with /MODULE.

Note that you must place the /MODULE qualifier after the input file you specify.

Example

```
$ LIBRARY/INSERT/TEXT TSTRING SYS$INPUT/MODULE=TEXT1
```

This LIBRARY command inserts a module named TEXT1 into the text library TSTRING.TLB. The input is taken from SYS$INPUT.

/NAMES

/NAMES — When /LIST is specified for an object module library, controls whether the LIBRARY command lists the names of all symbols in the global symbol table as well as the module names in the module name table.

Format

/NAMES (Alpha and VAX)

/NAMES [= option] (I64 only)

/NONAMES (default)

option
The option keywords pertain to I64 systems only and can be one of the following keywords:

Keyword	Explanation
BYMODULE	Lists each module followed by a list of the symbols in the global symbol table that are associated with that module. This is the default listing format when no keyword is given with the /NAMES qualifier.
BYSYMBOL	Lists each symbol in the global symbol table followed by a list of modules with which the symbol is associated, in precedence order.

Description

The /NAME qualifier lists the symbol names and the module names along with their association with each other. The default, /NONAMES, does not list the symbol names.

If you specify /NAME=BYSYMBOL, each symbol name is displayed followed by the modules with which the symbol is associated using the following format:

```
symbol "symbol name"
  Global:
```

```
    module-name
    .
    .
    .
  UxWk:
    module-name
    .
    .
    .
  Group Global:
    module-name
    .
    .
    .
  Group UxWk:
    module-name
    .
    .
    .
symbol "symbol name"
  UxWk:
    module-name
    .
    .
    .
```

Note

If you specify /NAMES and the library is a macro, help, or text library, no symbol names are displayed.

Example

```
$ LIBRARY/LIST/NAMES/ONLY=$ONE/WIDTH=80 SYMBOLIB
```

This LIBRARY command requests a full listing of the module $ONE, contained in the object library SYMBOLIB.OLB. The /WIDTH qualifier requests that the global symbol display be limited to 80 characters per line.

/OBJECT

/OBJECT — Indicates that the library specified is an object module library.

Format

/OBJECT

Description

Libraries are assumed to be object module libraries unless you specify the /SHARE, /MACRO, /TEXT, or /HELP qualifier. The library file type for object module libraries defaults to .OLB and the input file type defaults to .OBJ.

On OpenVMS VAX systems, the /OBJECT qualifier creates a VAX library by default. Note that you cannot have VAX modules and Alpha modules in the same library file. For more information, see the description of the /VAX (VAX and Alpha only) qualifier.

On OpenVMS Alpha systems, the /OBJECT qualifier creates an Alpha library by default. Note that you cannot have Alpha modules and VAX modules in the same library file. For more information, see the description of the /ALPHA qualifier.

On OpenVMS I64 systems, the /OBJECT qualifier creates only ELF object libraries.

Example

```
$ LIBRARY/OBJECT/INSERT MONTHS APRIL
```

This LIBRARY command inserts modules from APRIL.OBJ into the object library MONTHS.OLB. The /OBJECT qualifier is optional.

/ONLY

/ONLY — Specifies the individual modules on which the LIBRARY command can operate.

Format

/ONLY=(module-name[,...])

module-name
The module on which the LIBRARY command can operate.

Description

When you use the /ONLY qualifier, the LIBRARY command lists or cross-references only those modules specified.

If you specify more than one module, separate the module names with commas and enclose the list in parentheses.

The /ONLY qualifier must be used with the /LIST or /CROSS_REFERENCE qualifier.

Wildcard characters are allowed in the module name specification.

Example

```
$ LIBRARY/LIST/NAMES/ONLY=$ONE/WIDTH=80 SYMBOLIB
```

This LIBRARY command requests a full listing of the module $ONE, contained in the object library SYMBOLIB.OLB. The /WIDTH qualifier requests that the symbol display be limited to 80 characters per line.

/OUTPUT

/OUTPUT — When used with the /EXTRACT, /COMPRESS, /CROSS_REFERENCE, or /DATA qualifier, specifies the file specification of the output file.

Format

/OUTPUT=file-spec

file-spec

The file specification of the output file.

Description

For /EXTRACT, the output file contains the modules extracted from a library; for /COMPRESS, the output file contains the compressed library; for /CROSS_REFERENCE, the output file contains the cross-reference listing; for /DATA, the output file contains the data-reduced or data-expanded library.

No wildcard characters are allowed in the file specification.

If you omit the file type in the file specification, a default is used depending on the library function qualifier and, in some cases, the library type qualifier, as follows:

Qualifier	Library Type Qualifier	Default File Type
/COMPRESS or /DATA	/HELP /MACRO /OBJECT /TEXT /SHARE	.HLB .MLB .OLB .TLB .OLB
/CROSS_REFERENCE	—	.LIS
/EXTRACT	/HELP /MACRO /OBJECT /TEXT /SHARE	.HLP .MAR .OBJ .TXT .EXE

Examples

1. `$ LIBRARY/EXTRACT=(ALLOCATE,APPEND)/OUTPUT=MYHELP SYS$HELP:HELPLIB.HLB`

   This LIBRARY command specifies that the modules ALLOCATE and APPEND be extracted from the help library HELPLIB.HLB and output to the file MYHELP.HLP.

2. `$ LIBRARY/CROSS_REFERENCE=ALL/OUTPUT=SYS$OUTPUT LIBRAR`

   This LIBRARY command requests a cross-reference listing of the object library LIBRAR.OLB. The cross-reference listing is displayed at the terminal. The listing includes cross-references by symbol, by value, and by module.

/REMOVE

/REMOVE — Requests the LIBRARY command to delete one or more entries from the global symbol table in a library.

Format

Alpha and VAX:

`%LIBRAR-S-INSERTED, MODULE module-name INSERTED IN library-file-spec`

Example

```
$ LIBRARY/REMOVE=(LIB_EXTRCT_MODS,LIB_INPUT_MAC)/LOG LIBRAR
```

This LIBRARY command requests the removal of the symbols LIB_EXTRCT_MODS and LIB_INPUT_MAC from the object library LIBRAR.OLB. The /LOG qualifier requests that the removal of the symbols be confirmed by messages.

/REPLACE

/REPLACE — Requests the LIBRARY command to replace one or more existing library modules with the modules specified in the input files.

Format

/REPLACE

Description

The LIBRARY command first deletes any existing library modules with the same names as the modules in the input files. Then the new version of the module is inserted in the library. If a module contained in an input file does not have a corresponding module in the library, LIBRARY inserts the new module in the library.

The /REPLACE qualifier is the LIBRARY command's default operation. If you specify an input file parameter, the LIBRARY command either replaces or inserts the contents of the input file into the library. If you use the /LOG qualifier with the /REPLACE qualifier, the LIBRARY command displays, in the following format, the name of each module that it replaces or inserts:

```
%LIBRAR-S-REPLACED, MODULE module-name REPLACED IN library-file-spec
%LIBRAR-S-INSERTED, MODULE module-name INSERTED IN library-file-spec
```

Example

```
$ LIBRARY/REPLACE/HELP HELPLIB NEWTEXT
```

This LIBRARY command inserts into the help library HELPLIB.HLB the help modules from the file NEWTEXT.HLP. If a help module in NEWTEXT.HLP has the same name as an existing help module in the library, the module from NEWTEXT.HLP replaces the existing module.

/SELECTIVE_SEARCH

/SELECTIVE_SEARCH — Defines the input modules being inserted into a library as candidates for selective searches by the linker.

Format

/SELECTIVE_SEARCH

Description

If you specify /SELECTIVE_SEARCH and the library is specified as a linker input file, the linker selectively searches the modules; the linker takes from the library, for the symbol table of its output image file, only those symbols that have been referenced by other modules.

Note that the selective search operation applies only to those modules that were inserted in the library by a LIBRARY command that included the /SELECTIVE_SEARCH qualifier; it does not apply to the library itself.

Example

```
$ LIBRARY/SELECTIVE_SEARCH/INSERT MATHLIB TRIG
```

This LIBRARY command inserts the modules in TRIG.OBJ into the library MATHLIB.OLB. The inserted modules are selectively searched when MATHLIB.OLB is specified as an input file to the linker.

/SHARE

/SHARE — Indicates that the library specified is a shareable image library.

Format

/SHARE

Description

This LIBRARY command assumes a file type of .OLB for the shareable image library and .EXE for the input files. See Section 2.1.4 for additional information.

On OpenVMS VAX systems, the /SHARE qualifier creates a VAX shareable image library by default. Note that you cannot have VAX modules and Alpha modules in the same library. For more information, see the description of the /VAX (VAX and Alpha only) qualifier.

On OpenVMS Alpha systems, the /SHARE qualifier creates an Alpha shareable image library by default. Note that you cannot have Alpha modules and VAX modules in the same library. For more information, see the description of the /ALPHA qualifier.

On OpenVMS I64 systems, the /SHARE qualifier creates only ELF sharable image libraries.

SYS$LIBRARY:IMAGELIB.OLB is an example of a sharable image library.

Example

```
$ LIBRARY/SHARE/CREATE SHARELIB.OLB
```

This LIBRARY command creates a shareable image library called SHARELIB.OLB.

/SINCE

/SINCE — Specifies that only those modules inserted later than a particular time be listed.

Format

/SINCE [=time] 00

time
Limits the modules to be listed to those inserted in the library since a specified time.

You can specify an absolute time or a combination of absolute and delta times. For details about specifying times, see the *VSI OpenVMS DCL Dictionary*.

Description

This qualifier is used with the /LIST qualifier. If you omit the /SINCE qualifier, you obtain all the modules regardless of the date. If you specify /SINCE without a date or time, the default is to provide the modules inserted since today.

Example

```
$ LIBRARY/HELP/LIST/SINCE=:12 ERRMSG
```

This LIBRARY command displays information about help modules added to ERRMSG.HLB since noon today.

/SQUEEZE

/SQUEEZE — Controls whether the LIBRARY command compresses individual macros before adding them to a macro library.

Format

/SQUEEZE)

/NOSQUEEZE)

Description

When you specify /SQUEEZE, which is the default, trailing blanks, trailing tabs, and comments are deleted from each macro before its insertion in the library.

Use /SQUEEZE only with the /CREATE, /INSERT, and /REPLACE qualifiers to conserve space in a macro library. If you want to retain the full macro, specify /NOSQUEEZE.

Example

```
$ LIBRARY/MACRO/NOSQUEEZE/INSERT MYMACS MYMACS
```

This LIBRARY command inserts the macros in MYMACS.MAR into the library MYMACS.MLB. Trailing blanks, trailing tabs, and comments are not deleted from each macro before its insertion into the library.

/TEXT

/TEXT — Indicates that the library specified is a text library.

Format

/TEXT

Description

When you use the /TEXT qualifier, the library file type defaults to .TLB and the input file type defaults to .TXT.

Examples

1. `$ LIBRARY/INSERT/TEXT TSTRING SYS$INPUT/MODULE=TEXT1`

   This LIBRARY command inserts a module named TEXT1 into the text library TSTRING.TLB. The input is taken from SYS$INPUT.

2. `$ LIBRARY/INSERT/TEXT TSTRING TEXT2`

This LIBRARY command inserts the contents of the file TEXT2.TXT into the text library TSTRING.TLB. The name of the inserted module is TEXT2.

/VAX

/VAX (VAX and Alpha only) — Directs LIBRARIAN to work with an OpenVMS VAX object library when used with the /OBJECT qualifier or to work with an OpenVMS VAX shareable image library when used with the /SHARE qualifier. When used with the /CREATE qualifier, LIBRARIAN creates an OpenVMS VAX library of either an object or shareable image type depending whether /OBJECT or /SHARE is specified. The default is /ALPHA on OpenVMS Alpha systems and /VAX on OpenVMS VAX systems. The OpenVMS I64 Librarian does not accept this qualifier because the I64 Librarian works exclusively with I64 libraries.

Format

/VAX

Description

On OpenVMS Alpha systems, use the /VAX qualifier to create and manipulate OpenVMS VAX object and shareable image libraries. Because the formats of macro, help, and text libraries on OpenVMS Alpha systems are identical to those on OpenVMS VAX systems, using the /VAX qualifier with the /MACRO, /HELP, and /TEXT qualifiers has no effect.

Note

You cannot have both OpenVMS Alpha and OpenVMS VAX object modules in one object library, nor can you have OpenVMS Alpha and OpenVMS VAX shareable images in the same shareable image library.

Examples

1. `$ LIBRARY/VAX/CREATE TESTLIB ERRMSG.OBJ,STARTUP.OBJ`

   This LIBRARY command creates a VAX object module library named TESTLIB.OLB and places the files ERRMSG.OBJ and STARTUP.OBJ as modules in the library.

2. `$ LIBRARY/VAX/SHARE/CREATE SHARLIB`

   This LIBRARY command creates a VAX shareable image library called SHARLIB.OLB.

/WIDTH

/WIDTH — Controls the screen display width (in characters) for listing global symbol names.

Format

/WIDTH = n

n
The width of the screen display.

Description

Specify the /WIDTH qualifier with the /NAMES qualifier to limit the line length of the /NAMES display.

The default display width is the width of the listing device. The maximum width is 132.

Example

```
$ LIBRARY/LIST/NAMES/ONLY=$ONE/WIDTH=80 SYMBOLIB
```

This LIBRARY command requests a full listing of the module $ONE, contained in the object library SYMBOLIB.OLB. The /WIDTH qualifier requests that the global symbol display be limited to 80 characters per line.

Chapter 3. Message Utility

3.1. MESSAGE Description

The Message utility (MESSAGE) lets you supplement OpenVMS system messages with your own messages. Your messages can indicate that an error has occurred, or they can indicate other conditions; for example, that a routine has run successfully or that a default value has been assigned.

This section describes how to use the Message utility.

3.1.1. Message Format

Messages are displayed as a line of alphanumeric codes. The text of the message explains the condition that caused the message to be displayed. Messages are displayed in the following format:

```
%FACILITY-L-IDENT, message-text
```

FACILITY
Specifies the abbreviated name of the software component that issued the message.

L
Shows the severity level of the condition that caused the message. The five severity levels are represented by the following codes:

S	Success
I	Informational
W	Warning
E	Error
F	Fatal or severe

IDENT
Identifies a symbol of up to 15 characters that represents the message.

message-text
Explains the cause of the message. The message text can include up to 255 formatted-ASCII-output (FAO) arguments. For example, an FAO argument can be used to display the instruction where an error occurred or a value that you should be aware of.

% and ,
Included as delimiters if any of the first three fields—FACILITY, L, or IDENT—are present.

If you suppress FACILITY, L, and IDENT, the first character of the message text is capitalized by the Put Message ($PUTMSG) system service. The following example is a typical message:

```
%TYPE❶-W-❷OPENIN❸, error opening _DB0:[ROSE]STATS.FOR;❹ as input❺
```

❶ TYPE is the facility.
❷ W (Warning) is the severity level.
❸ OPENIN is the IDENT.
❹ _DBO:[ROSE]STATS.FOR is the FAO argument.

❺ “Error opening _DBO:[ROSE]STATS.FOR; as input” is the message text.

3.1.2. Constructing Messages

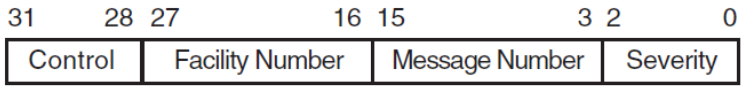
You construct messages by writing a message source file, compiling it using the Message utility, and linking the resulting object module with your facility object module. When you run your program, the Put Message ($PUTMSG) system service finds the information to use in the message by using a message argument vector.

The message argument vector includes the message code, a 32-bit value that uniquely identifies the message. The message code, which is created from information defined in the message source file, consists of the following:

- The severity level defined in the severity directive or message definition
- The message number assigned automatically by a message definition or specified with the base message number directive
- The facility number defined in the facility directive
- Internal control flags

Figure 3.1 shows the arrangement of the bits in the message code.

Figure 3.1. Message Code



You can refer to the message code in your programs by means of a global symbol called the message symbol, which also is defined by information from the message source file. The message symbol, which appears in the compiled message file, consists of the following:

- The symbol prefix defined in the facility directive
- The symbol name defined in the message definition

3.1.2.1. The Message Source File

The message source file consists of message definition statements and directives that define the message text, the message code values, and the message symbol. The various elements that can be included in a message source file are as follows:

- Facility directive
- Severity directive
- Base message number directive
- Message definition
- Literal directive
- Identification directive

- Listing directives
- End directive

Usually, the first statement in a message source file is a .TITLE directive, which lets you specify a module name for the compiled message file. You must specify a .FACILITY directive after the .TITLE directive. All the messages defined after a .FACILITY directive are associated with that facility. A .END directive or a new .FACILITY directive ends the list of messages associated with a particular facility.

You must define a severity level for each message either by specifying a .SEVERITY directive or by including a severity qualifier as part of the message definition.

Each message defined in the message source file must have a facility and a message definition associated with it. All other message source file statements are optional. See the Section 3.1.8 section for a detailed description of the format of each of these message source file statements.

The TESTMSG.MSG file that follows is a sample message source file. The messages for the associated FORTRAN program, TEST.FOR, are defined in TESTMSG.MSG with the following lines:

```
 .FACILITY       TEST,1 /PREFIX=MSG_
 .SEVERITY       ERROR
SYNTAX          <Syntax error in string '!AS'>/FAO=1
ERRORS          <Errors encountered during processing>
 .END
```

The FORTRAN program, TEST.FOR, contains the following lines:

```
EXTERNAL MSG_SYNTAX,MSG_ERRORS
CALL LIB$SIGNAL(MSG_SYNTAX,%VAL(1),'ABC')
CALL LIB$SIGNAL(MSG_ERRORS)
END
```

In addition to defining the message data, TESTMSG.MSG also defines the message symbols MSG_SYNTAX and MSG_ERRORS that are included as arguments in the procedure calls of TEST.FOR. The function %VAL is a required FORTRAN compile-time function. The first call also includes the string ABC as an FAO argument.

3.1.2.2. Compiling the Message Source File

You must compile message source files into object modules before you can use the messages defined in them. You compile your message source file by entering the MESSAGE command followed by the file specification of the message source file. For example:

```
$ MESSAGE TESTMSG
```

This command compiles the message source file TESTMSG.MSG and creates an object module file TESTMSG.OBJ.

For your convenience, you can put message object modules into object module libraries, which you can then link with facility object modules.

3.1.2.3. Linking the Message Object Module

After you compile the message file, you must link the message object module with the facility object module (created when the source file was compiled) to produce one executable image file.

For example, use the following command to link the message object module TESTMSG.OBJ to the FORTRAN object module TEST.OBJ to create the executable program TEST.EXE:

```
$ LINK/NOTRACE TEST+TESTMSG
```

At this point, you can execute the program, which contains both the message data and the facility code, with the following command:

```
$ RUN TEST
```

If an error occurs when you execute the program, the following messages are displayed:

```
%TEST-E-SYNTAX, Syntax error in string ABC
%TEST-E-ERRORS, Errors encountered during processing
```

3.1.3. Using Message Pointers

Message pointers are generally used when you need to provide different message texts for the same set of messages; for example, a multilingual situation. When you use message pointers, you do not link the message object module directly with the facility object module. Consequently, you do not have to relink the executable image file to change the message text included in it.

To use a pointer, you must create a non-executable message file that contains the message text and a pointer file that contains the symbols and pointer to the non-executable file.

You create the non-executable message file by compiling and linking a message source file. For example, to create the non-executable message file COBOLMF.EXE, you first create the object module by compiling the message source file, COBOLMSG.MSG, with the following command:

```
$ MESSAGE/NOSYMBOLS COBOLMSG
```

You link the resulting object module with the following command:

```
$ LINK/SHAREABLE=SYS$MESSAGE:COBOLMF COBOLMSG.OBJ
```

By default, the linker places newly created images in your default device and directory. In the preceding example, the non-executable image COBOLMF.EXE is placed in the system message library SYS$MESSAGE.

You create the pointer file by recompiling the message source file with the MESSAGE/FILE_NAME command. To avoid confusion, the pointer file should have a different file name from the non-executable file.

The object module resulting from the MESSAGE/FILE_NAME command contains only global symbols and the file specification of the non-executable message file.

For example, the following command creates the object module MESPNTR.OBJ, which contains a pointer to the non-executable message file COBOLMF.EXE:

```
$ MESSAGE/FILE_NAME=COBOLMF /OBJECT=MESPNTR COBOLMSG
```

In addition to the pointers, the object module MESPNTR.OBJ contains the global symbols defined in the message source file COBOLMSG.MSG. If the destination of the non-executable message file is not SYS$MESSAGE, you must specify the device and directory in the file specification for the /FILE_NAME qualifier.

After you create the pointer object module, you can then link it with the facility object module.

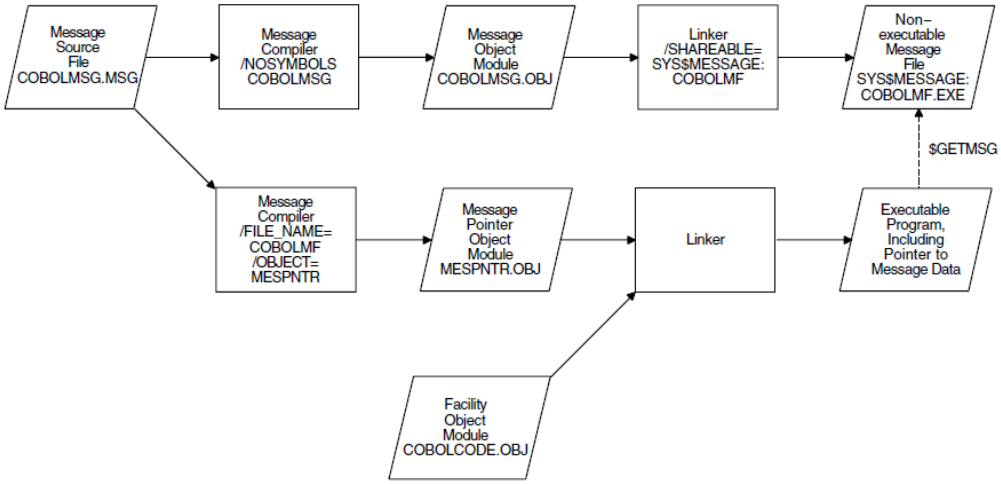
For example, the following command links the object module MESPNTR.OBJ to the COBOL program COBOLCODE:

```
$ LINK COBOLCODE,MESPNTR
```

When you run the resulting facility image file, the Get Message ($GETMSG) system service retrieves the message data from the message file COBOLMF.

Figure 3.2 illustrates the relationship of the files in this example.

Figure 3.2. Creating a Message Pointer



On Alpha systems, by default, the message compiler creates OpenVMS Alpha object modules from message source files. The /VAX qualifier causes the compiler to produce OpenVMS VAX object modules from those same message source files.

Note that you must convert messages to OpenVMS VAX object format if you want to link them against other OpenVMS VAX objects. For more information, see the *VSI OpenVMS Linker Utility Manual*.

3.1.4. The SET MESSAGE Command

The SET MESSAGE command allows you to do the following:

- Suppress or enable the various fields of the messages in your process
- Supplement the system message data with the message data in a non-executable message image for your process

For example, the following SET MESSAGE command specifies that the message information in MYMSG.EXE supplements the existing system messages:

```
$ SET MESSAGE MYMSG
```

In addition, the SET MESSAGE command used with one or more qualifiers suppresses or enables one or more fields in a message. For example, the following command suppresses the IDENT field in a message:

```
$ SET MESSAGE/NOIDENTIFICATION
```

For more information about the SET MESSAGE command, see the *VSI OpenVMS DCL Dictionary*.

3.1.5. Message Source Files

The message source file contains statements or directives and the information included in the message, the message code, and the message symbol.

Source File Statements

Message source file statements are embedded within a message source file. Generally, message source file statements help construct the message code and the message symbol and control output listings. The message source file statements are as follows:

- Facility directive (.FACILITY)
- Severity directive (.SEVERITY)
- Base message number directive (.BASE)
- Message definition message-name
- End directive (.END)
- Literal directive (.LITERAL)
- Identification directive (.IDENT)
- Listing directives
  - Title directive (.TITLE)
  - Page directive (.PAGE)

Many of these statements accept qualifiers and parameters. The specific format of each of the message source file statements is described in detail in the Section 3.1.8 section.

Any line in the message source file, except lines that contain the .TITLE directive, can include a comment delimited by an exclamation point. You can insert extra spaces and tabs in any line to improve readability.

The listing title specified with the .TITLE directive and the message text specified in the message definition must occupy only one line. All other statements in a message source file can occupy any number of lines; text that continues onto the next line must end with a hyphen.

Defining Symbols in the Message Source File

Symbols defined in the message source file can include any of the following characters:

- Uppercase and lowercase letters (A,a,B,b . . . Z,z)
- Digits (1,2,3 . . . 9)
- Dollar sign ($)
- Underscore (_)

Using Expressions in the Message Source File

Expressions used in the message source file can include any of the following radix operators:

^X	Hexadecimal
^O	Octal
^D	Decimal

Radix operators specify the radix of a numeric value. The default radix is decimal.

Expressions can include symbols and the plus sign (+), which assigns a positive value, and minus sign (–), which assigns a negative value. Expressions can include the following binary operators:

+	Addition
–	Subtraction
*	Multiplication
/	Division
@	Arithmetic shift

Expressions can also include parentheses as grouping operators. Expressions enclosed in parentheses are evaluated first; nested parenthetical expressions are evaluated from inside to outside.

3.1.6. MESSAGE Usage Summary

MESSAGE

MESSAGE — The Message utility (MESSAGE) lets you supplement system messages with your own messages. Your messages can indicate that an error has occurred, or they can indicate other conditions; for example, that a routine has run successfully or that a default value has been assigned.

Format

`MESSAGE file-spec[,...]`

Command Parameter

`file-spec`
Specifies the message source file to be compiled. If you do not specify a file type, the default is .MSG. Wildcard characters are allowed in file specifications.

If you specify more than one message source file, separated by either commas or plus signs, the files are concatenated and compiled as a single file.

If you specify SYS$INPUT, the message source files must immediately follow the MESSAGE command in the input stream, and both the object module name, identified by the /OBJECT qualifier, and the listing file name, identified by the /LIST qualifier, must be stated explicitly.

Usage Summary

The DCL command MESSAGE invokes the Message utility. After compiling the message source file, the Message utility returns you to DCL command level. For details about message statements and directives, qualifiers, and parameters in message source files, see the Section 3.1.8 section.

3.1.7. MESSAGE Qualifiers

MESSAGE command qualifiers let you specify the type and contents of output files produced. In addition, MESSAGE command qualifiers let you create non-executable message files that contain pointers to files that contain message data. Output files produced by command qualifiers are named according to the rules described in the *VSI OpenVMS DCL Dictionary*.

/ALPHA

/ALPHA — Directs MESSAGE to create an OpenVMS Alpha message object file. The default is to create OpenVMS Alpha message object files on OpenVMS Alpha systems and to create OpenVMS VAX message object files on OpenVMS VAX systems.

Format

/ALPHA

Description

Directs the message compiler to create an OpenVMS Alpha object modules from message source files.

Note that you must compile message source files using /ALPHA (default on OpenVMS Alpha systems) to link with other OpenVMS Alpha object modules and that you must compile using /VAX to link with OpenVMS VAX object modules. For more information, see the *VSI OpenVMS Linker Utility Manual*.

Example

```
$ MESSAGE/ALPHA TESTMSG
```

This MESSAGE command creates an OpenVMS Alpha message object module named TESTMSG.OBJ by compiling the message source file TESTMSG.MSG.

/FILE_NAME

/FILE_NAME — Specifies whether the object module contains a pointer to a file containing message data.

Format

/FILE_NAME=file-spec

/NOFILE_NAME

file-spec
Identifies a non-executable message file. The default device and directory for the file specification is SYS$MESSAGE and the default file type is .EXE. No wildcard characters are allowed in the file specification.

Description

The /[NO]FILE_NAME qualifier specifies whether the object module contains a pointer to a file containing message data. By default, the object module contains only compiled message information and no pointers.

The /FILE_NAME and /TEXT qualifiers cannot be used together because a message pointer file cannot contain message text. The message text is contained in the non-executable message file, specified by the /FILE_NAME qualifier.

1. `$ MESSAGE COBOLMSG`

   This MESSAGE command creates the message object module COBOLMSG.OBJ by compiling the message source file COBOLMSG.MSG. The default qualifier /NOFILE_NAME is implied.

2. `$ MESSAGE/FILE_NAME=COBOLMF COBOLMSG`

   This MESSAGE command creates a message pointer file COBOLMSG.OBJ, which contains a pointer to the non-executable message file SYS$MESSAGE:COBOLMF.EXE.

/LIST

/LIST — Controls whether an output listing is created and, optionally, provides an output file specification for the listing.

Format

`/LIST[=file-spec]`

`/NOLIST`

`file-spec`
Specifies an output file specification for the listing file. The default device and directory are the current device and directory. The default file type is .LIS. No wildcard characters are allowed in the file specification.

Description

The /LIST qualifier creates a listing file. If you do not specify a file specification, the listing file has the same name as the first message source file being compiled and a file type of .LIS. When you compile message source files in batch mode, the output listing is created by default; however, in interactive mode, the default is to produce no output listing.

Example

```
$ MESSAGE/LIST=MSGOUTPUT  COBOLMSG
```

This MESSAGE command compiles the message source file COBOLMSG.MSG and creates the output listing MSGOUTPUT.LIS in your current directory.

/OBJECT

/OBJECT — Controls whether an object module is created by the message compiler and, optionally, provides a file specification for the object module.

Format

`/OBJECT[=file-spec]`

`/NOOBJECT`

`file-spec`

Specifies a file specification for the object module. The default device and directory are the current device and directory. No wildcard characters are allowed in the file specification.

Description

By default, the message compiler creates an object module that contains the message data. If you do not specify a file specification, the object module has the same name as the first message source file being compiled and a file type of .OBJ.

Examples

1. `$ MESSAGE COBOLMSG`

   This MESSAGE command creates the message object module COBOLMSG.OBJ by compiling the message source file COBOLMSG.MSG. The default qualifier /OBJECT is implied.

2. `$ MESSAGE/FILE_NAME=COBOLMF /OBJECT=MESPNTR COBOLMSG`

   This MESSAGE command creates the object module MESPNTR.OBJ, which contains a pointer to the non-executable message file COBOLMF.EXE.

/SYMBOLS

/SYMBOLS — Controls whether global symbols are present in the object module. By default, object modules are created with global symbols.

Format

/SYMBOLS

/NOSYMBOLS

Description

By default, the message compiler creates an object module with global symbols. The /SYMBOLS qualifier requires that the /OBJECT qualifier be in effect, either explicitly or implicitly. If you are creating both a pointer object module and a non-executable message image, you can compile the object module, which becomes the non-executable image, with the /NOSYMBOLS qualifier. The symbols have to be in the pointer object module only.

Example

```
$ MESSAGE/FILE_NAME=COBOLMF /OBJECT=MESPNTR/SYMBOLS COBOLMSG
```

This MESSAGE command creates the object module MESPNTR.OBJ, which contains global symbols.

/TEXT

/TEXT — Controls whether the message text is present in the object module.

Format

/TEXT

/NOTEXT

Description

By default, the message compiler creates an object module that contains message text. The message text is obtained from the non-executable message file specified by the /FILE_NAME qualifier. The /TEXT and /FILE_NAME qualifiers cannot be used together because a message pointer file cannot contain message text.

The /TEXT qualifier requires that the /OBJECT qualifier be in effect, either explicitly or implicitly.

You can use the /NOTEXT qualifier with the /SYMBOLS qualifier to produce an object module containing only global symbols.

Example

```
$ MESSAGE/FILE_NAME=COBOLMF/NOTEXT /OBJECT=MESPNTR COBOLMSG
```

This MESSAGE command creates the object module MESPNTR.OBJ, which does not contain text; instead, it contains a pointer to the non-executable message file COBOLMF.EXE.

/VAX

/VAX — Directs MESSAGE to create an OpenVMS VAX message object file. The default is to create OpenVMS Alpha message object files on OpenVMS Alpha systems and to create OpenVMS VAX message object files on OpenVMS VAX systems.

Format

/VAX

Description

Directs the message compiler to create an OpenVMS VAX object modules from message source files.

Note that you must compile message source files using /ALPHA to link with other OpenVMS Alpha object modules and that you must compile using /VAX (default on OpenVMS VAX systems) to link with OpenVMS VAX object modules. For more information, see the *VSI OpenVMS Linker Utility Manual*.

Example

```
$ MESSAGE/VAX TESTMSG
```

This MESSAGE command creates an OpenVMS VAX message object module named TESTMSG.OBJ by compiling the message source file TESTMSG.MSG.

3.1.8. MESSAGE Commands

This section describes the message source file statements.

Base Message Number Directive

Base Message Number Directive — Defines the value used in constructing the message code.

Format

.BASE number

Parameter

number
Specifies a message number to be associated with the next message definition or an expression that is evaluated as the desired number.

Qualifiers

None.

Description

By default, all of the messages following a facility directive are numbered sequentially, beginning with 1.

If you need to supersede this default numbering system (for example, if you want to reserve some message numbers for future assignment), specify a message number of your choice using the base message number directive. The message number is used as a base for the sequential numbering of all messages that follow until either another .BASE directive or the end of the messages belonging to the facility is encountered. Note that the maximum number for any message cannot exceed 4095.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_  ❶
 .SEVERITY       ERROR

UNRECOG         < Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           < Ambiguous keyword>

 .SEVERITY       WARNING
 .BASE           10❷
SYNTAX          < Invalid syntax in keyword>

 .END
```

❶ The facility number (facnum) in the facility statement defines the first two message numbers as 1 and 2.

❷ The base message number directive supersedes this sequential numbering by assigning the message number 10 to the third message.

End Directive

End Directive — Terminates the entire list of messages for the facility.

Format

.END

Parameters

None.

Qualifiers

None.

Description

The .END directive terminates the entire list of messages for a facility. A .FACILITY directive also terminates a list of messages.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_
 .SEVERITY       ERROR

UNRECOG         < Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           < Ambiguous keyword>

 .SEVERITY       WARNING
 .BASE           10
SYNTAX          < Invalid syntax in keyword>

 .END ❶
```

❶ The .END directive terminates the list of messages for the SAMPLE facility.

Facility Directive

Facility Directive — Specifies the facility to which the messages apply.

Format

`.FACILITY [/qualifier,...] facnam[,]facnum [/qualifier,...]`

Parameters

`facnam`
Specifies the facility name used in the facility field of the message and in the symbol representing the facility number. The facility name can be up to 9 characters.

`facnum`
Specifies the facility number used to construct the 32-bit value of the message code. A decimal value in the range 1 to 2047, or an expression that evaluates to a value in that range, can be used. The system manager usually assigns facility numbers so that no two facilities have the same number.

Qualifiers

/PREFIX=prefix

Defines an alternate symbol prefix to be used in the message symbol for all messages referring to this facility. The default symbol prefix is the facility name followed by an underscore (_). If the /SYSTEM qualifier is also specified, the default prefix is the facility name followed by a dollar sign and an underscore ($_). The combined length of the prefix and the message symbol name cannot exceed 31 characters. The maximum length of an alternate symbol prefix created with the /PREFIX qualifier is 9 characters.

/SHARED

Inhibits the setting of the facility-specific bit in the message code. The /SHARED qualifier is used only for system services and shared messages and is reserved for use by VSI.

/SYSTEM

Inhibits the setting of the customer facility bit in the message code. This qualifier is reserved.

Description

The .FACILITY directive is the first directive in a message source file. All of the lines following a .FACILITY directive apply to that facility until a .END directive or another facility statement is reached. You must specify the facility name and the facility number in a .FACILITY directive. The facility name and facility number are separated by a comma or by any number of spaces or tabs.

The .FACILITY directive creates a global symbol of the following form:

```
facnam$_FACILITY
```

You can use this symbol to refer to the facility number assigned to the facility.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_      ❶
 .SEVERITY       ERROR

UNRECOG         < Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           < Ambiguous keyword>

 .SEVERITY       WARNING
 .BASE           10
SYNTAX          < Invalid syntax in keyword>

 .END
```

❶ The facility statement in this message source file defines the messages belonging to the facility (facnam) SAMPLE with a facility number (facnum) of 1. The message numbers begin with 1 and continue sequentially. The /PREFIX=ABC_ qualifier defines the message symbols ABC_UNRECOG, ABC_AMBIG, and ABC_SYNTAX.

Identification Directive

Identification Directive — Identifies the object module the Message utility produces.

Format

`.IDENT string`

Parameter

`string`
Identifies the object module; for example, a string that identifies a version number. If it is not delimited, the string is a 1- to 31-character string of alphanumeric characters, underscores, and dollar signs. If other characters are used, the string must be delimited with either apostrophes or quotation marks.

Qualifiers

None.

Description

The .IDENT directive is used in addition to the name you assign to the module with the .TITLE directive. You can label the object module by specifying a character string with the directive. If a message source file contains more than one identification directive, the last directive establishes the character string that forms part of the object module identification.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00' ❶
 .FACILITY       SAMPLE,1/PREFIX=ABC_
 .SEVERITY       ERROR

UNRECOG         <Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           <Ambiguous keyword>

 .SEVERITY       WARNING
 .BASE           10
SYNTAX          < Invalid syntax in keyword>

 .END
```

❶ This IDENT directive identifies the object module that the Message utility produces.

Literal Directive

Literal Directive — Defines global symbols in your message source file. You can either assign values to these symbols or use the default values the directive provides.

Format

`.LITERAL symbol[=value][,...]`

Parameters

`symbol`
Specifies a symbol name.

`value`
Specifies any valid expression. If you omit the value, a default value is assigned. The default value is 1 for the first symbol in the directive and 1 plus the last value assigned for subsequent symbols.

Qualifiers

None.

Description

You can use the .LITERAL directive to define a symbol as the value of another previously defined symbol, or as an expression that results from operations performed on previously defined symbols.

Examples

1. `.LITERAL    A,B,C`

The values of A, B, and C will be 1, 2, and 3.

2.
```
 .FACILITY       SAMPLE,1/PREFIX=MSG$_
 .SEVERITY       ERROR
FIRST           < first error>
    .
    .
    .
LAST            < last error>
 .LITERAL        LASTMSG=MSG$_LAST    ❶

 .LITERAL        NUMSG=(MSG$_LAST@-3)-(MSG$_FIRST@-3) !number of
 messages ❷
```

In this example, symbols defined in the facility and message definition statements are used to assign values to symbols created with the .LITERAL directives.

❶ The first .LITERAL directive defines a symbol that has the value of the last 32-bit message code defined.

❷ The second .LITERAL directive defines the total number of messages in the source file.

Message Definition

Message Definition — Defines the message symbol, the message text, and the number of FAO arguments that can be printed with the message.

Format

`name[/qualifier,...] <message-text>[/qualifier,...]`

Parameters

`name`

Specifies the name that is combined with the symbol prefix (defined in the .FACILITY directive) to form the message symbol. The combined length of the prefix and the message symbol name cannot exceed 31 characters.

The name is used in the IDENT field of the message unless you specify the /IDENTIFICATION qualifier in the message definition.

`message-text`

Defines the text explaining the condition that caused the message to be displayed. The message text can be delimited either by angle brackets or by quotation marks. The text can be up to 255 bytes long; however, you cannot continue the delimited text onto another line. The message text can include FAO directives that insert ASCII strings into the resulting message; the Formatted ASCII Output ($FAO) system service uses these directives. If you include an FAO directive, you must also use the /FAO_COUNT qualifier.

Qualifiers

/FAO_COUNT=n

Specifies the number of FAO arguments to be included in the message at execution time. The number specified must be a decimal number in the range 0 to 255. The Put Message ($PUTMSG)

system service, when constructing the final message text, uses `n` to determine how many arguments are to be given to the $FAO system service. The default value for `n` is 0.

/IDENTIFICATION=name

Specifies an alternate character string to be used as the IDENT field of the message. The name can include up to nine characters. If you do not specify this qualifier, the name defined in the message definition is used.

/USER_VALUE=n

Specifies an optional user value that can be associated with the message. The value must be a decimal number in the range of 0 to 255. The default is 0. The value can be retrieved by the Get Message ($GETMSG) system service for use in classifying messages by type or by action to be taken.

/SUCCESS

Specifies the level SUCCESS for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the SUCCESS level.

/INFORMATIONAL

Specifies the level INFORMATIONAL for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the INFORMATIONAL level.

/WARNING

Specifies the level WARNING for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the WARNING level.

/ERROR

Specifies the level ERROR for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the ERROR level.

/SEVERE

Specifies the level SEVERE for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the SEVERE level.

/FATAL

Specifies the level FATAL for a message. This qualifier overrides any .SEVERITY directive in effect. If no .SEVERITY directive is in effect, you must use this qualifier to specify the FATAL level.

Description

The message definition statement specifies the message text that is to be displayed and the name used in the IDENT field of the message. Additionally, you can use the message definition statement to specify

the number of FAO arguments to be included in the message text. Any number of message definitions can follow a .SEVERITY directive (or a .FACILITY directive if no .SEVERITY directive is included).

You can place qualifiers in any order before or after the message text.

You can use the severity level qualifiers either to override the severity level defined in a .SEVERITY directive or to replace .SEVERITY directives in your message source file. Only one message definition qualifier can be included per message definition.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_❶
 .SEVERITY       ERROR

UNRECOG         <Unrecognized keyword !AS>/FAO_COUNT=1❷
AMBIG           "Ambiguous keyword"     ❸

 .SEVERITY       WARNING
 .BASE           10
SYNTAX          <Invalid syntax in keyword>❹

 .END
```

This message source file contains a .FACILITY directive ❶ and three message definitions ❷ ❸ ❹. The symbol names—UNRECOG, AMBIG, and SYNTAX—specified in the message definitions are combined with a prefix, ABC_ (defined in the .FACILITY directive), to form the message symbols ABC_UNRECOG, ABC_AMBIG, and ABC_SYNTAX.

The message text of the UNRECOG ❷ and SYNTAX ❹ messages is delimited by angle brackets (<>); the message text of the AMBIG message ❸ is delimited by quotation marks ("").

In addition, the first message definition statement in this example includes the FAO directive !AS (which inserts an ASCII string at the end of the message text) and the corresponding qualifier /FAO_COUNT.

Page Directive

Page Directive — Forces page breaks in the output listing.

Format

.PAGE

Parameters

None.

Qualifiers

None.

Description

The .PAGE directive lets you specify page breaks in the output listing. You can specify only one page break per any one .PAGE directive; however, you can use the .PAGE directive as often as you like.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_
 .SEVERITY       ERROR

UNRECOG         <Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           <Ambiguous keyword>
 .PAGE ❶

 .SEVERITY       WARNING
 .BASE           10
SYNTAX          <Invalid syntax in keyword>

 .END
```

❶ This .PAGE directive forces a page break in the output listing after the AMBIG message definition.

Severity Directive

Severity Directive — Specifies the severity level to be associated with the messages that follow the .SEVERITY directive.

Format

`.SEVERITY level`

Parameter

`level`
Specifies the level of the condition that caused the message. The severity level codes are as follows:

SUCCESS	Produces an S code in a message.
INFORMATIONAL	Produces an I code in a message.
WARNING	Produces a W code in a message.
ERROR	Produces an E code in a message.
SEVERE	Produces an F code in a message.
FATAL	Produces an F code in a message.

The SEVERE parameter is equivalent to the FATAL parameter; they can be used interchangeably.

Qualifiers

None.

Description

Following the .FACILITY directive, the message source file generally contains a .SEVERITY directive. If you do not specify the severity on each message definition with one of the message definition severity qualifiers, you must include a .SEVERITY directive. If you attempt to define a message without specifying a severity level, an error results.

A new .FACILITY directive cancels the previous severity level in effect.

Example

```
 .TITLE          SAMPLE Error and Warning Messages
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_
 .SEVERITY       ERROR       ❶

UNRECOG         <Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           <Ambiguous keyword>

 .SEVERITY       WARNING     ❷
 .BASE           10
SYNTAX          <Invalid syntax in keyword>

 .END
```

❶ ❷ The two .SEVERITY directives included in this message source file define the severity levels for three messages. The first two messages have a severity level of E; the third message has a severity level of W.

Title Directive

Title Directive — Specifies the module name and title text that is to appear at the top of each page of the output listing file.

Format

`.TITLE modname [listing-title]`

Parameters

`modname`

Specifies a character string of up to 31 characters that is to appear in the object module as the module name.

`listing-title`

Defines the text to be used as the title of the listing. The title begins with the first nonblank character after the module name and continues through the next 28 characters. An exclamation point (!) within these 28 characters is treated as part of the title and not as a comment delimiter. The listing title has a maximum length of 28 characters and cannot be continued onto another line.

Qualifiers

None.

Example

```
 .TITLE          SAMPLE ❶ Error and Warning Messages ❷
 .IDENT          'VERSION 4.00'
 .FACILITY       SAMPLE,1/PREFIX=ABC_
 .SEVERITY       ERROR

UNRECOG         <Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG           <Ambiguous keyword>

 .SEVERITY       WARNING
```

```
 .BASE          10
SYNTAX         <Invalid syntax in keyword>

 .END
```

❶ The module name of the object module produced by this file is SAMPLE.

❷ The title of the output listing is defined as “Error and Warning Messages.”

3.1.9. MESSAGE Examples

The following examples demonstrate the use of message files and pointer files.

Creating an Executable Image Containing Message Data

The following example illustrates the steps involved in incorporating a message file within an executable image.

The message source file, TESTMSG.MSG, created with a text editor, contains the following lines:

```
.FACILITY       TEST,1 /PREFIX=MSG_
.SEVERITY       ERROR
SYNTAX          <Syntax error in string '!AS'>/FAO_COUNT=1
ERRORS          <Errors encountered during processing>
.END
```

You compile the message source file by entering the following command:

```
$ MESSAGE TESTMSG
```

You compile the FORTRAN source file by entering the following command:

```
$ FORTRAN TEST
```

You link the message object module TESTMSG.OBJ to the FORTRAN object module TEST.OBJ by entering the following command:

```
$ LINK/NOTRACE TEST+TESTMSG
```

You execute the image by entering the following command:

```
$ RUN TEST
```

If an error occurs when you execute the program, the system displays the following messages:

```
%TEST-E-SYNTAX, Syntax error in string ABC
%TEST-E-ERRORS, Errors encountered during processing
```

Creating an Executable Image Containing a Pointer

The following example demonstrates how to create an executable image that contains a pointer to a non-executable message file.

You compile the message source COBOLMSG by entering the following command:

```
$ MESSAGE/NOSYMBOLS COBOLMSG
```

You link the object module COBOLMSG.OBJ to create the non-executable message file by entering the following command:

```
$ LINK/SHAREABLE=SYS$MESSAGE:COBOLMF COBOLMSG.OBJ
```

You create the pointer object module MESPNTR.OBJ, which contains a pointer to the non-executable message file COBOLMF.EXE, by entering the following command:

```
$ MESSAGE/FILE_NAME=COBOLMF /OBJECT=MESPNTR COBOLMSG
```

You link the pointer object module MESPNTR.OBJ to the COBOL program object module COBOLCODE.OBJ by entering the following command:

```
$ LINK COBOLCODE,MESPNTR
```

You execute the program by entering the following command:

```
$ RUN COBOLCODE
```

The system then displays the messages defined in COBOLMSG.