VAX/VMS Command Definition Utility Reference Manual

Order Number: AA–Z408A–TE

September 1984

This document describes the Command Definition Utility. This utility lets you modify the DIGITAL command language (DCL) by adding commands to your process command table or to a specified command table file.

Revision/Update Information:	This is a new manual.
Software Version:	VAX/VMS Version 4.0

digital equipment corporation
maynard, massachusetts

September 1984

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

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

ZK–2305

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Command Definition Contents

TABLES

Preface

Intended Audience

This manual is intended for all system users who wish to define their own DCL commands.

Structure of This Document

This document is composed of five major sections.

The Format Section is an overview of the Command Definition Utility and is intended as a quick reference guide. The format summary contains the DCL command that invokes the Command Definition Utility, listing all command qualifiers and parameters. The usage summary describes how to invoke and exit from the the Command Definition Utility, how to direct output, and any restrictions you should be aware of. The statement and clause summary lists all statements and clauses that can be used within the Command Definition Utility.

The Description Section explains how to use the Command Definition Utility.

The Command Definition File Statements Section describes each statement that can be used in a command definition file. Statements appear in alphabetical order.

The Qualifier Section describes each DCL command qualifier. Qualifiers appear in alphabetical order.

The Examples Section contains examples of common operations that you perform with the Command Definition Utility.

Associated Documents

For related information about this utility, see the following documents:

- *VAX/VMS DCL Dictionary*
- *Guide to Programming on VAX/VMS*

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.

Convention	Meaning
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

Changes for the SET COMMAND Command

The following changes apply to the SET COMMAND command, which invokes the Command Definition Utility.

- A new qualifier, /REPLACE, has been added.
- The /DELETE qualifier has been modified; /NODELETE is no longer permitted.

Changes in Command Definition Files

The following changes affect how command definition files are written and processed:

- The VALUE(TYPE=type-name) clause has been added to allow the definition of value types. Value types define the type of value that can be used with parameters, qualifiers, or keywords. The Command Definition Utility allows built-in and user-defined value types.
- A new statement, DEFINE TYPE, can be used in conjunction with the VALUE(TYPE=type-name) clause to define keywords that can be used as values for parameters, qualifiers or other keywords.
- A new clause, DISALLOW, disallows combinations of parameters, qualifiers, or keywords in a command line. The DISALLOW clause can be used in DEFINE SYNTAX and DEFINE VERB statements.
- Forward references to DEFINE TYPE and DEFINE SYNTAX statements are allowed. Therefore, a DEFINE statement can reference another DEFINE statement that appears later in the command definition file.
- Parameter definitions must be copied to syntax changes if the syntax change defines additional parmeters.
- A new statement, IDENT, can be used to provide identifying information for object modules.
- Image names specified with the IMAGE clause should be surrounded in quotation marks.
- Values in command definition file statements can be specified as symbols or character strings. Character strings should be enclosed in quotation marks, while symbols should not. For example, the ROUTINE clause requires that the routine name be specified as a symbol. The descriptions for each clause indicate how values should be specified.

Changes That are Incompatible With Version 3.0

The following technical changes are incompatible with the V3.0 version of the Command Definition Utility.

- The way parameters are defined in DEFINE SYNTAX statements has changed. You must rewrite DEFINE SYNTAX statements if they contain parameter definitions and if parameters are defined before the syntax change (in the DEFINE statement that refers to the syntax change).

  To correct the DEFINE SYNTAX statement, repeat the initial parameter definitions and then define the new parameters.

- You should recompile object modules that were created with the Version 3.0 Command Definition Utility. Although the old object modules will still work, DCL will perform certain conversions every time the modules are used. Therefore, to improve performance, it is recommended that you recompile object modules.
- This version of the Command Definition Utility does not support undocumented functions from Version 3.0.

Command Definition

The Command Definition Utility (CDU) creates, deletes, or changes command definitions in a command table. As input, the CDU accepts a command table and/or a file that contains command definitions. The CDU processes this input to create a new command table. The new table can be either executable code or an object module.

FORMAT

SET COMMAND *[file-spec[,...]]*

Command Qualifiers	Defaults
/DELETE=(verb[,...])	*None.*
/[NO]LISTING[=file-spec]	*/NOLISTING*
/OBJECT[=file-spec]	*None.*
/[NO]OUTPUT[=file-spec]	*/OUTPUT*
/REPLACE	*See text.*
/TABLE[=file-spec]	*/TABLE*

Command Parameter

file-spec[,...]

Specifies the name of one or more command definition files. If you specify two or more file specifications, separate them with commas. The default file type is CLD.

Wildcard characters are allowed in the file specification.

usage summary

Invoking

Use the DCL command SET COMMAND to invoke the Command Definition Utility. 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 are mutually exclusive; you can use only one SET COMMAND mode in a command string.

Exiting

When the CDU has finished processing the command definition file and/or table, the DCL prompt reappears on your screen.

Directing Output

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

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

Privileges/Restrictions

To modify the system command table in SYS$LIBRARY:DCLTABLES.EXE you need SYSPRV privilege.

---

statements and clauses

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

Verb Clauses for DEFINE SYNTAX

DISALLOW expression
IMAGE image-string
NODISALLOWS
NOPARAMETERS
NOQUALIFIERS
PARAMETER param-name [,param-clause[,...]]
QUALIFIER qual-name [,qual-clause[,...]]
ROUTINE routine-name

Verb Clauses for DEFINE VERB

DISALLOW expression
IMAGE image-string
NODISALLOWS
NOPARAMETERS
NOQUALIFIERS
PARAMETER param-name [,param-clause[,...]]
QUALIFIER qual-name [,qual-clause[,...]]
ROUTINE routine-name
SYNONYM synonym-name

Type Clause

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

Parameter Clauses

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

Qualifier Clauses

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

Keyword Clauses

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

Parameter Value Clauses

CONCATENATE
DEFAULT = default-string
LIST
NOCONCATENATE
REQUIRED
TYPE = type-name

Qualifier and Keyword Value Clauses

DEFAULT = default-string
LIST
REQUIRED
TYPE = type-name

Placement Clauses

GLOBAL
LOCAL
POSITIONAL

DESCRIPTION

The CDU allows you to use DCL command processing techniques to check command syntax and execute commands. The following sections describe

- How DCL processes commands
- Ways to use the CDU
- How to write command definitions
- How to process command definitions
- How to use command language routines in programs

1 Command Processing

In order to write command definitions and modify command tables, you must understand how the DCL command interpreter processes commands. To process a command, DCL prompts for and accepts a command string. Then DCL parses the command string using definitions in your process command table. Your process command table contains a list of valid commands and their attributes. DCL parses the command string from left to right.

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 Parts of a Command String

A command string can contain the following entities:

Verb	Specifies the command to be executed.
Parameter	Specifies what the verb acts upon. The DCL command definitions describe the allowable parameter values for each command. 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.
Qualifier	Describes or modifies the action taken by the verb. The DCL command definitions indicate whether qualifiers can accept values. Command definitions also describe the types of values that can be specified. Examples of qualifier values include file specifications, integer values, keywords, and character strings.

The following command string illustrates the components of a DCL command line.

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

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

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

```
$ SHOW DEFAULT
```

In this example, SHOW is the verb and DEFAULT is a keyword that is used as a parameter.

1.2 The System and Process Command Tables

The system command table is contained in the file SYS$LIBRARY:DCLTABLES.EXE. By default, when you log in to VAX/VMS this command table is copied to your process. DCL uses your process command table to parse command strings.

Because you can change your process command table without affecting the permanent table in SYS$LIBRARY, you do not need any special privileges to modify your process command table. However, to modify the DCL command table in SYS$LIBRARY:DCLTABLES.EXE, you need SYSPRV 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. DIGITAL maintains the command definition files for DCL; they are not shipped with your system.

2 Using the CDU

To use the CDU:

- Determine what table you want to create or modify. In general, you will modify your process command table or the DCL table in SYS$LIBRARY, or you will create an object module for a new table.
- Choose a name and syntax for the command(s) you are defining. Use a text editor to create a command definition file that defines the command(s).
- Use the DCL command SET COMMAND to add your command definition file 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. Use the appropriate command language routines in this code.

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. See the *VAX/VMS DCL Dictionary* for information on how to define a foreign command.

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
- New tables that allow user-written programs to process commands

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. In the command definition, specify the name of an image for the command to invoke. Then use the SET COMMAND 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
```

Now you can enter the new command after the DCL prompt, and DCL will parse the command and invoke the appropriate image. Note that when you write the code to execute the new command, you must use the command language routines CLI$PRESENT and CLI$GET_VALUE to obtain information about the command string.

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.

The first example in the Examples Section shows how to add a new command to your process command table. This example also shows how to write a program that is invoked by the new command.

3.2 Adding a Command to the DCL Command Table

To add a command to the DCL command table in SYS$LIBRARY, define the command in a command definition file. In this definition, specify the name of an image for the command to invoke. Then use the SET COMMAND command to add the new definition to the DCL command table and copy the new table back to SYS$LIBRARY. (You must have SYSPRV to change the DCL command table.) For example:

```
$ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES -
$_ /OUTPUT=SYS$LIBRARY:DCLTABLES NEWCOMMAND
```

Now any user who logs in will have this modified DCL table copied to his /her process and will therefore be able to enter the new command after the DCL prompt.

3.3 Creating an Object Module

To create an object module for a new command table, define the commands in a command definition file. For each command, specify the name of a routine in a program that executes the command. Then use the SET COMMAND command to create an object module from this 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 a user-written program, the program must perform the functions of a command interpreter. That is, the program must obtain command strings and call the parsing routine CLI$DCL_PARSE to verify and create an internal representation of the command string. The program must also call CLI$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 Examples Section shows how to write and process command definitions for an object module. This example also shows how to write a program that parses commands and invokes routines.

4 Writing a CFILE

A command definition file contains information that defines a command, 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(s); you can use clauses to specify additional information for statements. The default file type for a command definition file is CLD.

Use the following conventions in command definition files:

- The exclamation point delimits comments in a command definition file; an exclamation point causes all following characters on a line to be treated as comments.
- Any statement and its clauses can be broken into multiple 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) 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 double quotation marks ("").

Note: To maintain compatibility with earlier releases, the CDU accepts character strings that are not enclosed in quotation marks. However, it is recommended that you surround character strings in quotation marks. If you do not enclose a string in quotation marks, alphabetic characters are converted to uppercase.

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

The Command Definition Utility Language includes the following statements:

- DEFINE SYNTAX syntax-name [verb-clause[,...]]
- DEFINE TYPE type-name [type-clause[,...]]
- DEFINE VERB verb-name [verb-clause[,...]]

- IDENT
- MODULE module-name

The following sections provide a general overview of each CDU statement; complete information is given in the Command Definition File Statements Section that follows this description.

4.1 Defining Command Verbs

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

The format for the DEFINE VERB statement is

```
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. Use of verb clauses is optional.

You can specify the following verb clauses:

DISALLOW	Restricts use of an entity or a combination of entities.
IMAGE	Specifies an image to be invoked by the verb.
NODISALLOWS	Indicates that no entities or combinations of entities are disallowed.
NOPARAMETERS	Indicates that no parameters are allowed.
NOQUALIFIERS	Indicates that no qualifiers are allowed.
PARAMETER	Defines a parameter that can be specified.
QUALIFIER	Defines a qualifier that can be specified.
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 on the command line. 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.

For complete information on the DEFINE VERB statement and its verb clauses, see the Command Definition File Statements Section.

4.2 Defining Syntax Changes With the DEFINE SYNTAX Statement

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

To indicate a syntax change, the DEFINE VERB statement that defines an entity (the primary DEFINE statement) must include a SYNTAX=syntax-name clause to point to the alternate syntax. The alternate syntax is the secondary DEFINE statement.

For example, you can write a command definition that uses a different syntax if a certain qualifier is present. When you specify this qualifier, the syntax defined in the secondary DEFINE statement applies to the command string.

The format for the DEFINE SYNTAX statement is

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

The syntax name 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 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 issue the command ERASE/LINE, the definitions in the DEFINE SYNTAX LINE statement override the definitions in the DEFINE VERB ERASE statement. However if you issue 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 issue 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.

For complete information on the DEFINE SYNTAX statement and its verb clauses, see the Command Definition File Statements Section.

4.3 Defining Values for Parameters, Qualifiers, and Keywords

To indicate that a parameter, qualifier, or keyword requires a value, 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 must be a built-in type (for example, the value must be a file specification) or you can specify that a value must be a user-defined keyword. Section 4.3.1 lists the built-in value types; Section 4.3.2 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 double quotation marks ("") in the place you want the quotation marks to appear.

4.3.1 Built-In Types

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

$ACL	The value must be an access control list.
$DATETIME	The value must be an absolute or a combination time. DCL converts truncated time values, combination time values, and keywords for time values (such as TODAY) to absolute time format. If the value is missing any date fields, DCL fills in the current date. If the value is missing any time fields, DCL fills these fields with zeros.
$DELTATIME	The value must be a delta time. If the value is missing any time fields, DCL fills these fields with zeros.
$FILE	The entity value must be a valid file specification.
$NUMBER	The entity value must be an integer. The command string can contain decimal, octal, or hexadecimal numbers. However, DCL converts all numbers to decimal.
$QUOTED_STRING	DCL uses the default method of processing the entity, with one exception. DCL does not remove quotation marks when processing the string.
$REST_OF_LINE	Everything until the end of the the line is equated to the entity value.

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)
```

4.3.2 User-Defined Keywords

The DEFINE TYPE statement defines keywords that can be used as values for parameters, qualifiers, or other keywords. The keywords listed in a DEFINE TYPE statement are the only values that can be used with the corresponding parameter, qualifier, or keyword.

To indicate that a parameter, qualifier, or keyword requires a keyword, use the VALUE(TYPE=type-name) clause when the entity is defined. The type name refers to the DEFINE TYPE statement that defines the allowable keywords. When you use VALUE(TYPE=type-name), this starts a keyword path.

The format for the DEFINE TYPE statement is

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

The type name is the name of the keyword list. The type clause specifies allowable keywords and provides information about each keyword.

You can use the following type clause:

KEYWORD Defines a keyword that can be used with the parameter, qualifier, or keyword that references the keyword list.

This example illustrates a DEFINE TYPE 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 accepts the qualifier SPEED. The verb SKIM invokes the image [TOOLS]SKIM.EXE and accepts the qualifier /SPEED.

❷ The VALUE clause indicates that the qualifer /SPEED accepts a list of keywords. These keywords are defined in the DEFINE TYPE SPEED_KEYWORDS statement.

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

For complete information on the DEFINE TYPE statement and the KEYWORD type clause, see the Command Definition File Statements Section.

4.4 Disallowing Combinations of Entities in Command Definitions

When you use DEFINE statements, you can indicate that an entity (a parameter, qualifier, or keyword) or a combination of entities is invalid in certain cases. To prohibit use of an entity or a combination of entities, use the DISALLOW verb clause.

The DISALLOW verb clause has the following format:

```
DISALLOW expression
```

The expression specifies an entity in a command string, or a group of entities connected by operators. 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 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.

For example, a command definition may contain a DEFINE VERB statement that defines the verb SPORTS with three qualifiers: /TENNIS, /BOWLING, and /BASEBALL. However, you may 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 line. Therefore, local qualifiers are treated as if they were global for the purposes of the DISALLOW processing. 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,POSITION=LOCAL
        QUALIFIER       QUAL3,POSITION=LOCAL
        DISALLOW        P1 AND QUAL1
        DISALLOW        QUAL2 AND QUAL3
```

Thus, the following two command lines 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.

4.4.1 Specifying Entities in an Expression

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 the keyword name or label is unique. (See the next section for more information on keyword paths.)

If the LABEL=label-name keyword is used to assign a label to an entity, you must use the label 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 parameter and qualifier names RED and BLUE in the DISALLOW verb clause because both names are used in the current definition. You can use the keyword name ALL because it is in a keyword path which 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 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 in the path that references the keyword's 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, you must use the label name in the keyword path. Otherwise, use the keyword name. You can omit names from the beginning of a keyword path that are not needed to resolve a keyword reference. However, you must include enough names to uniquely reference the keyword.

The following command string illustrates a situation where keyword paths are needed 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 type definition RED_AMOUNT; you must use the keyword path GREEN.ALL to refer to the ALL keyword when it is used in the type definition GREEN_AMOUNT.

Definition Paths

A definition path is used to refer to an entity that is defined in another DEFINE statement. Use a definition path in a syntax definition when parameters or qualifiers are inherited from a primary definition, but new disallow clauses are provided.

A definition path has the 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 spec can be an entity name, a label, or a keyword path. The angle brackets are required syntax.

For example:

```
DISALLOW <SKIP>FIRST
```

This clause disallows a command string if the entity FIRST (which is defined in a DEFINE statement named 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 starts in a DEFINE statement named FILE) is present.

The CDU does not check a definition path to determine that 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. The following example shows a definition path that is not valid in the syntax definition where it is used.

```
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. The clause DISALLOW <FILE> RECEIPTS would have been valid in the DEFINE SYNTAX statement, however. The DEFINE SYNTAX statement inherits the qualifier RECEIPTS from the primary DEFINE statement (FILE) because no qualifiers are specified. Therefore, the qualifier RECEIPTS can be disallowed. See the description of the DEFINE SYNTAX statement in the Command Definition File Statements Section for more information on how entities are inherited by DEFINE SYNTAX statements.

4.4.2 Operators

If an expression contains operators, the operators are evaluated after the entities are evaluated as present (true) or absent (false). If the result of the expression is true, then the syntax is disallowed. If the result of the expression is false, then the syntax is valid.

Table CDU–1 shows the operators you can use in expressions and the order in which these operators are evaluated. (Operators with a precedence of 1 are performed first.) Operations of the same precedence are performed from left to right, in the order they appear in the expression.

Table CDU–1 Summary of CDU Operators

Operator	Precedence	Meaning
ANY2(entity[,...])	1	Result is true if any two or more of the entities listed are present.
NEG entity	1	Result is true if the negated form of the entity is present.
NOT entity	1	Result is true if the entity is not present or if an entity is present by default.
exp AND exp	2	Result is true if both expressions are true.
exp OR exp	3	Result is true if either expression is true.

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.

4.5 Providing Identifying Information for Object Modules

Use the MODULE and IDENT statements to provide identifying information if your command definition file will be used 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 which you can link with a user-written program.)

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

```
MODULE module-name
```

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

The IDENT statement provides additional information to identify the module. The format for the IDENT statement is

```
IDENT ident-string
```

The following command definition file shows how to use the MODULE and IDENT statements:

```
MODULE TABLE      ❶
IDENT "Updated 4/15/84"      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 table that is created when an object module is created with the command SET COMMAND/OBJECT.

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 verbs that can be used by the user-written program that will be linked with the object module containing the command table. Each verb invokes a routine within the user-written program.

5 Processing the Command Definition File

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; thus you can use only one SET COMMAND mode on a command line. In addition to the qualifiers which 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 Qualifiers Section for complete information on using the SET COMMAND qualifiers.

5.1 Adding Command Definitions to a Command Table

Use the /REPLACE qualifier to add or replace verbs in the command table you are modifying. By default, SET COMMAND uses /REPLACE mode to add commands to your process command table and returns 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 table other than your process table, use the /TABLE qualifier to specify an input table; if you want to write the modified table to a file (instead of to your process), use the /OUTPUT qualifier.

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
```

5.3 Creating an Object Module

Use the /OBJECT qualifier to create an object module from a command definition file. The following example shows how to create an object module:

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

Entering SET COMMAND with the /OBJECT qualifier creates an object module containing a command table with the verb definitions in NEWCOMS.CLD. You can link this module with a program that parses commands using the linked command table module.

5.4 Creating a New Command Table

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 verbs that invoke images can be added.

To create an empty command table, create a command definition file that uses the MODULE statement to define a module name and optionally uses the IDENT statement. For example, the command definition file TEST_TABLE.CLD contains:

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

Create an object module from TEST_TABLE.CLD, and link it to create a shareable image:

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

Next, create a command definition file that defines verbs that invoke images. For example, the command definition file VERBS.CLD contains:

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

Then 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 resulting file will have a version number of one greater than the version number of the input table.) For example:

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

The /TABLE and /OUTPUT qualifiers specify the input and output table files. Be sure to use the /OUTPUT qualifier to specify the output file. Otherwise, the modified command table will be written to your process and will replace your process command table.

6 Using Command Language Routines

A user-written program invoked by a command that you have added to your process (or system) command table will need information about the command string that invoked it. Call DCL's command language interface routines from your program to retrieve this information.

There are two command language interface routines:

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.

If a program is using its own command table (that is, the command table has been linked with the program) the program can call DCL's command parsing routines to parse the command string and invoke the appropriate routine to execute the command. The routine then calls CLI$PRESENT and CLI$GET_VALUE to obtain information about the command string.

There are two command parsing routines:

CLI$DCL_PARSE	Parses a command string.
CLI$DISPATCH	Invokes the routine which corresponds to the verb most recently parsed.

The Examples Section shows two programs that call these routines. For complete information on the command language routines and their arguments, see the *VAX/VMS Utility Routines Reference Manual.*

COMMAND DEFINITION FILE STATEMENTS

This section provides complete information on the statements that can be used in a command definition file. The statements are

DEFINE SYNTAX
DEFINE TYPE
DEFINE VERB
IDENT
MODULE

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 a different syntax depending on the parameters, qualifiers, and keywords that are present in the command string.

The DEFINE statement that refers to a changed syntax is called a primary define statement. The DEFINE SYNTAX statement that defines the new syntax is called a secondary DEFINE statement.

When a command string is parsed, its components are scanned from left to right. The line is parsed according to the current definition until an entity occurs that specifies a syntax change. Then, the following entities are parsed using the new definition. DCL does not rescan the entities that appeared before the entity that specified the syntax change.

Table CDU–2 explains how the DEFINE SYNTAX statement modifies the current command definition if an entity specifies a syntax change. The command definition that exists after the last entitity in the command string has been parsed is saved by DCL. DCL uses the disallows in this definition to determine if any entities are not allowed. Then, DCL invokes the image or routine specified by the saved definition and uses this definition to process CLI$PRESENT and CLI$GET_VALUE calls.

Table CDU–2 How the DEFINE SYNTAX Statement Modifies the Primary DEFINE Statement

DEFINE SYNTAX Specifies	Result
An image	This image overrides the image in the primary DEFINE statement. DCL invokes the new image after it parses the command string.
A routine	This routine overrides the routine in the primary DEFINE statement. DCL invokes the new routine when CLI$DISPATCH is called.
One or more disallows	These disallows are used during command parsing; disallows in the primary DEFINE statement are ignored. 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.

Table CDU–2 (Cont.) How the DEFINE SYNTAX Statement Modifies the Primary DEFINE Statement

DEFINE SYNTAX Specifies	Result
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 will be 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 still refers to the first parameter in the command string. Therefore, if you want to define parameters in addition to parameters defined in the primary DEFINE statement, you must use the PARAMETER clause to redefine the original parameters in the secondary DEFINE statement. Define these parameters exactly as they appeared in the primary DEFINE statement. Then, define the new 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 that were already parsed are not reparsed according 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	Qualifiers that were already parsed are ignored. If the entity that specifies the syntax change is a qualifier, this qualifier is also ignored. Qualifiers that appear in the command line after the entity that specifies the new syntax will be parsed according to the new definitions. DCL uses the new qualifier definitions when processing CLI$PRESENT and CLI$GET_VALUE calls. When DCL parses a command line that contains qualifiers that are ignored (due to a syntax change), DCL issues a warning message.

Table CDU–2 (Cont.) How the DEFINE SYNTAX Statement Modifies the Primary DEFINE Statement

DEFINE SYNTAX Specifies	Result
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 that were already 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[,...]

Specify 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

These clauses are described below.

DISALLOW expression
NODISALLOWS

Disallows a command string if the result of the expression is true. The NODISALLOWS clause indicates that no entities or combinations of entities are disallowed.

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 is 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 4.4.1 for more information on these entities. You can specify the operators AND, ANY2, NEG, NOT, OR, or ANY2. See Section 4.4.2 for more information on these operators.

IMAGE image-string

Names an image to be invoked for the syntax change. The **image-string** is the file specification of the image that DCL invokes when you issue the command. If you do not provide a complete file specification, the command language interpreter supplies a default directory specification of SYS$SYSTEM: and a default file type of EXE. Specify the image string as a character string that does not exceed 63 characters.

If you do not specify an IMAGE verb clause (and you use SET COMMAND /REPLACE to process the command definition), the CDU uses the image name from the DEFINE statement that references the syntax change. If this DEFINE statement does not contain an image name, the CDU checks to see if the DEFINE statement is pointed to by another DEFINE statement. It keeps searching until it finds the name of an image to invoke. If no image is specified, then DCL searches, at run time, for an image whose file name is the same as the verb name and whose device name and file type are SYS$SYSTEM: and EXE, respectively.

PARAMETER param-name [,param-clause[,...]]
NOPARAMETERS

Specifies whether parameters can be included in the command string. You can use the PARAMETER clause up to eight times in a DEFINE SYNTAX statement. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** is the position of the parameter in the command line. The name must be in the form Pn, 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-clauses** specify additional chacteristics 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 indicate 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 for a parameter that is not entered in the command string. If you do not specify a prompt string and a required parameter is missing, DCL will use the parameter name as the prompt string. Specify the prompt string as a character string that does not exceed 31 characters.

If you define more than one parameter and the first parameter is required but the other parameters are optional, then prompting is done in the following way. If the user types the command without any parameters, DCL will prompt for the first (required) parameter until the user types a value, or aborts the command with a CTRL/Z.

After the user types a value for P1, DCL will prompt for subsequent parameters, even though these parameters are optional. After the prompt, if the user types a CTRL/Z, the command is aborted. If the user presses the return key without entering a value, the command is executed. If the user types a value, then DCL prompts for the next optional parameter.

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 in the absence of an explicit parameter value. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 95 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	Indicates that a list of parameters separated by commas or plus signs can be specified.
NOCONCATENATE	Indicates that the parameter cannot be concatenated to another parameter with a plus sign.
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 type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the type name as a symbol. See Section 4.3.1 for more information on built-in 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 used on the qualifier name to indicate that the qualifier is not present.

PLACEMENT=placement-clause indicates where the qualifier can appear on the command line. 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, the qualifier will be GLOBAL.
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 on the command line, but the meaning of the qualifier depends on the position which it is used in. If the qualifier is used after a parameter value, it applies only to that parameter. If it is used after the verb, the qualifier 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, surround the list in parentheses and separate items with commas. If you do not specify any qualifier value clauses, then DCL converts letters in qualifier values to uppercase.

VALUE accepts the following 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 95 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 qualifer 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 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 and the DEFAULT clauses are mutually exclusive.
TYPE=type-name	Gives either a built-in type or the name of a DEFINE TYPE statement that defines a list of keywords that can be used with the qualifier. Specify the type name as a symbol. See Section 4.3.1 for more information on built-in types.

ROUTINE routine-name

Names a routine in a user-written program to be invoked when the changed syntax is used. Use the ROUTINE clause if you will 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 the primary DEFINE statement does not specify a routine, no default is provided.

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 line. 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 command:

```
$ SET COMMAND WRITER
```

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

```
$ WRITER/LINE
```

you invoke the LINE editor instead of the default editor (WRITER). Syntax redefinition is done from left to right because parsing of the line 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 (since 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 qualifier /SAVE, then one parameter is required. This parameter indicates the name of the file to be displayed. If the DISPLAY command is used with the qualifier /SAVE, 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

Describes the syntax of the keywords that are 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[,...]]
Specifies a keyword that may be used as the value of the entity that referenced the DEFINE TYPE statement. Repeat the KEYWORD type clause for each keyword that may be used. You can specify up to 255 keywords in a DEFINE TYPE statement.

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

You can use the following keyword clause:

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

DEFAULT indicates that the keyword is present by default. In order for this keyword to be recognized as present by default, then 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. Specify the label name as a symbol. If you do not specify a label name, 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. Specify the syntax name as a symbol.

VALUE[(key-value-clause[,...])] specifies additional characteristics for the keyword. VALUE accepts the following 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 95 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 may be given. This list must be enclosed in parentheses and separated by commas. Note that plus signs may not 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	Gives either a built-in type or the name of a DEFINE TYPE statement that defines keywords that can be specified as values for the keyword. The TYPE clause cannot be specified if the DEFAULT clause is specified. Specify the type name as a symbol. See Section 4.3.1 for more information on built-in 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 for DISPLAY to your process command table, you can issue 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, 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 issue 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 in order 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. You must also specify DEFAULT when defining the NUMBER keyword in the DEFINE TYPE statement.

DEFINE VERB

Defines a new command and specifies information about the parameters and qualifiers that can be used with the command. The DEFINE VERB statement also specifies the image or routine that is invoked by the newly defined command.

FORMAT

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

verb-name

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

verb-clause[,...]

Specify 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

These clauses are described below.

DISALLOW expression
NODISALLOWS

Disallows a command string if the result of the expression is true. The NODISALLOWS clause indicates that no entities or combinations of entities are disallowed.

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 is 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 4.4.1 for more information on these entities. You can specify the operators AND, ANY2, NEG, NOT, OR, or ANY2. See Section 4.4.2 for more information on these operators.

IMAGE image-string

Names an image to be invoked by the command. The **image-string** is the file specification of the image that DCL invokes when you issue the command. If you do not provide a complete file specification, the command

language interpreter supplies a default device and directory specification of SYS$SYSTEM: and a default file type of EXE. Specify the image string as a character string that does not exceed 63 characters.

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

Specifies whether parameters can be included in the command string. You can use the PARAMETER clause up to eight times in a DEFINE VERB statement. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** is the position of the parameter in the command string. The position must be in the form Pn, 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 chacteristics 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 indicate 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 for a parameter that is not entered in the command string. If you do not specify a prompt string and a required parameter is missing, DCL will use the parameter name as the prompt string. Specify the prompt string as a character string that does not exceed 31 characters.

If you define more than one parameter and the first parameter is required but the other parameters are optional, then prompting is done in the following way. If the user types the command without any parameters, DCL will prompt for the first (required) parameter until the user types a value, or aborts the command with a CTRL/Z.

After the user types a value for P1, DCL will prompt for subsequent parameters, even though these parameters are optional. After the prompt, if the user types a CTRL/Z, the command is aborted. If the user presses the return key without entering a value, the command is executed. If the user types a value, then DCL prompts for the next optional parameter.

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 in the absence of an explicit parameter value. The DEFAULT clause and the REQUIRED clause are mutually exclusive. Specify the default string as a character string that does not exceed 95 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	Indicates that a list of parameters separated by commas or plus signs can be specified.
NOCONCATENATE	Indicates that the parameter cannot be concatenated to another parameter with a plus sign.
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 type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the type name as a symbol. See Section 4.3.1 for more information on built-in 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. 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 used on the qualifier name to indicate that the qualifier is not present.

PLACEMENT=placement-keyword indicates where the qualifier can appear on the command line. 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 behavior is the default; if you do not specify the PLACEMENT clause, the qualifier will be GLOBAL.
LOCAL	Indicates that the qualifier can appear only after a parameter, and applies only to that parameter.
POSITIONAL	Indicates that the qualifier can appear anywhere on the command line, but its meaning depends on the position it is used in. If the qualifier is used after a parameter, it applies only to that parameter. If it is used after the verb, the qualifier applies to all parameters.

SYNTAX=syntax-name specifies an alternate syntax definition to be invoked when the qualifier is present. This alternate syntax is useful for commands that invoke different images depending upon the particular qualifiers that are 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, surround the list in parentheses and separate items with commas. If you do not specify any qualifier value clauses, the DCL converts letters in a qualifier value to uppercase.

VALUE accepts the following 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 95 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 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 type or the name of a DEFINE TYPE statement that defines a list of keywords that can be used with the qualifier. Specify the type name as a symbol. See Section 4.3.1 for more information on built-in types.

ROUTINE routine-name

Names a routine in a user-written program to be invoked when the command is issued. Use the ROUTINE clause if you will 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. Specify the routine name as a symbol.

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

SYNONYM synonym-name

Defines a synonym that can be used in place of 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 the user does 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 which defines a new command, called SCATTER. The new command will invoke the image WRKD$:[MORRISON] SCATTER.EXE. It has two required parameters, an input file and an output file. It also has two qualifiers, /SLOW and /FAST. If you do not explicitly specify any qualifiers, /SLOW is present by default. You cannot specify both /SLOW and /FAST on the same command line.

IDENT

Provides identifying information for an object module that is created from a command definition file.

FORMAT

IDENT *ident-string*

ident-string

A string which contains identifying information. Specify the **ident-string** as a character string that does not exceed 31 characters.

EXAMPLE

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

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

MODULE

Creates a symbolic name for an object module that is created from a command definition file.

FORMAT

MODULE *module-name*

module-name

Symbolic name for the object module. This name refers to the address where the linker locates a command table module that is linked with a user-written program.

If you do not specify a module name, the CDU uses the object file name (specified with the /OBJECT command qualifier) as the default module name. If no object file is explicitly specified, then the 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
```

This example shows a command definition file, TEST.CLD, which defines two commands, SEND and SEARCH. These commands invoke routines in the program PROG.EXE. PROG.EXE is a user-written program that accepts command strings and uses DCL to parse these strings and execute routines.

The SET COMMAND command is used to create an object module which contains a command table for the SEND and SEARCH commands. When the object module for the program (PROG.OBJ) is linked with the object module for the command table (TEST.OBJ) the resulting image (PROG.EXE) contains the code for the program and command table. The symbolic name TEST_TABLE refers to the address in the image where the command table is located.

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

COMMAND QUALIFIERS

The following pages describe the qualifiers that can be used with the DCL command SET COMMAND to invoke the Command Definition Utility. The qualifiers are

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

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

/DELETE

Specifies /DELETE mode to delete verb or synonym names from the command table you are modifying. If a verb name has synonyms, the /DELETE 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 can 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 qualifiers 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 definition for the command 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 did not include the /OUTPUT qualifier, the modified table would have been written to your process, and would have overwritten the commands in your process command table.

/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 will have 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/[NO]LISTING *[=listing-file-spec] [file-spec[,...]]*

listing-file-spec

The file specification for the listing file. If no file name is specified, the name will default to the name of the first command definition file. The default file type is LIS.

file-spec

The name of the command definition file to be processed. The default file type is CLD. Wildcard characters are allowed in the file specification.

EXAMPLES

1
```
$ SET COMMAND/LISTING TEST
```

In this example, the command definition file TEST.CLD is processed by the 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 the 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 object module can be linked with a user-written program. A listing file named GAMES.LIS is created.

/OBJECT

Specifies /OBJECT mode to create 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 a user-written program. The program can then use its own command table to parse command strings and execute routines.

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 will have the same name as the input file and a file type of OBJ.

You cannot use the /DELETE, /OUTPUT, /REPLACE, or /TABLE qualifiers with /OBJECT.

FORMAT

SET COMMAND/OBJECT *[=object-file-spec] file-spec*

object-file-spec

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

file-spec

The name of the command definition file to be processed. The default file type is CLD. Wildcard characters are allowed in the file specification.

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. (The name of the object module, if not explicitly given, 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 file is processed and the command table is written as an object module to a file named A.OBJ.

/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-file-spec] [file-spec[,...]]*

SET COMMAND/NOOUTPUT

output-file-spec

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

You can specify an output file only when you also use the /TABLE=file-spec qualifier to provide the input table.

file-spec

The name of the command definition file to be processed. The default file type is CLD. Wildcard characters are allowed in the file specification.

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 the file TEST.CLD are added to the command table A.EXE. The modified table is written to a new file named A.EXE with a version number one greater than the number of the input table file.

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 will be written to your process and will replace 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

Specifies /REPLACE mode to add or replace verbs in the command table you are modifying.

You can use the /REPLACE qualifier to modify either the process command table or 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 modify 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 /OBJECT or /DELETE qualifiers in /REPLACE mode.

If you do not explicitly specify /DELETE, /OBJECT, or /REPLACE, the default is /REPLACE.

FORMAT

SET COMMAND/REPLACE *[file-spec [,...]]*

file-spec

The name of the command definition file to be processed. The default file type is CLD. Wildcard characters are allowed in the file specification.

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; the /OUTPUT 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
```

In this example the command definitions from the file TEST.CLD are added to the command table in the file COMMAND_TABLE.EXE. The modified command table is written to the file NEW_TABLE.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 will be written to your process and will 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; the DCL commands that were copied to your process command table when you logged in will be overwritten.

/TABLE

Specifies the command table that is to be modified. If you specify the /TABLE qualifier and omit the file specification, the current process command table is modified. If you include a file specification, the specified command table is modified. The default file type is EXE.

If you use the /TABLE qualifier to provide an input table file, you should also use the /OUTPUT qualifier to provide an output table file. Otherwise, the modified command table will be written to your process and will replace your process command table.

You can use the /TABLE qualifier in /DELETE or /REPLACE mode; the default is /TABLE with no input file specification. You cannot use the /TABLE qualifier in /OBJECT mode.

FORMAT

SET COMMAND/TABLE *[=input-file-spec] [file-spec [,...]]*

SET COMMAND/NOTABLE

input-file-spec

The file specification of the input file which contains the command table to be edited. The default file type is EXE.

file-spec

The name of the command definition file to be processed. The default file type is CLD. Wildcard characters are allowed in the file specification.

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
```

In this example the command definitions from the file TEST.CLD are added to the command table in the file A.EXE. The modified command table is written to the file 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 will be written to your process and will replace your process command table.

Command Definition
/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; the DCL commands that were copied to your process command table when you logged in will be overwritten.

COMMAND DEFINITION UTILITY 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 the CDU to process the command definition file (SAMPLE.CLD) and 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 have finished compiling and linking the source program you will have created a file called SAMPLE.EXE, which contains an executable image.

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

```
$ SAMPLE
```

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

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

```
$ SAMPLE MYFILE
```

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

```
FILE-SPEC = MYFILE
```

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

```
$ SAMPLE MYFILE/EDIT
```

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

```
/EDIT IS PRESENT
FILE-SPEC = MYFILE
```

If you include a qualifier that is not accepted by the command verb, you will 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 will also 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 a User-Written Program

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

When you write a command definition file from which you will create an object module, specify routines (not images) for each command verb. These routines will be called by your program when it processes command strings.

The following example illustrates a command definition file called TEST.CLD that contains the names of 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 an object module named TEST.OBJ. For example:

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

Design a program to invoke the routines listed in the command table in TEST.OBJ. You can then link TEST.OBJ with an object module that was created from your user source program.

The following program, entitled USEREXAMP.BAS, is written in BASIC. It uses the command language routines CLI$DCL_PARSE and CLI$DISPATCH to parse command strings and 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 'FILE_SPEC = ',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
    EXTERNAL INTEGER FUNCTION SYS$EXIT
    CALL SYS$EXIT(1%)
290 SUBEND
1   EXTERNAL INTEGER FUNCTION CLI$DCL_PARSE,CLI$DISPATCH,LIB$GET_INPUT
    EXTERNAL INTEGER FUNCTION SEND_COMMAND,SEARCH_COMMAND,EXIT_COMMAND
    EXTERNAL INTEGER TEST_TABLE
2   IF NOT CLI$DCL_PARSE(0,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 as shown below:

```
$ 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 issuing the DCL command

```
$ LINK USEREXAMP,TEST
```

You now have a file containing an executable image (USEREXAMP.EXE). To execute the image, use the DCL 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 on your screen:

```
SEND COMMAND
TEST>
```

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

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

The SEND_COMMAND routine will be invoked again and it will display the following text on your screen:

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

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

```
TEST> SEND/EDIT MESSAGE.TXT
SEND COMMAND
/EDIT is present
FILE-SPEC = MESSAGE.TXT
TEST>
```

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

```
TEST> SEARCH
```

The SEARCH command string invokes a different routine than the one defined by SEND. In this case, the following text will be displayed on your screen:

```
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 will signal 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
```

Index

B

C

D

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line