VMS

digital

VMS Command Definition Utility Manual

Order Number AA–LA60A–TE

VMS Command Definition Utility Manual

Order Number: AA–LA60A–TE

April 1988

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 document supersedes the VMS Command Definition Utility Reference Manual Version 4.0
Software Version:	VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

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.

ZK4536

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

Contents

TABLES

Preface

Intended Audience

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

Document Structure

This document consists of the following five sections:

- Description—Provides a full description of the Command Definition Utility (CDU).
- Usage Summary—Outlines the following CDU information:

  -Invoking the utility
  -Exiting from the utility
  -Directing output
  -Restrictions or privileges required

- CDU File Statements—Describes the statements used in building command definition files including statement formats and parameters together with examples.
- CDU Qualifiers—Describes qualifiers including format, parameters and examples.
- CDU Examples—Provides additional CDU examples.

Associated Documents

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

- *VMS DCL Dictionary*
- *Guide to VMS Programming Resources*

Conventions

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (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

No enhancements have been made to the Command Definition Utility for VMS Version 5.0.

CDU Description

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

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

- How the DIGITAL Command Language processes commands
- How to write command definitions
- How to modify command tables
- How to process command definitions
- How to use command language routines in programs

1 Command Processing

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

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

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

1.1 Command String Components

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

The DCL command definitions describe the allowable parameter values for each command. The command definitions also indicate whether or not qualifiers can take values and the value types that can be specified. Examples of qualifier values include file specifications, integer values, keywords, and character strings. Some commands (SET and SHOW) accept keywords as

parameters. A keyword is a predefined string that can be used as a value for a parameter, qualifier, or another keyword.

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

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

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

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

```
$ SHOW DEFAULT
```

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

1.2 System and Process Command Tables

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

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

2 Using the CDU

To use the CDU:

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

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

3 Choosing a Table

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

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

3.1 Modifying Your Process Command Table

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

```
$ SET COMMAND NEWCOMMAND
```

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

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

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

3.2 Adding a System Command

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

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

To make the new command available to other users, use the Install Utility.

3.3 Creating an Object Module

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

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

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

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

4 Writing a Command Definition File

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

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

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

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

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

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

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

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, all alphabetic characters are converted to uppercase characters (capital letters).

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

The Command Definition Language includes the following statements:

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

The following subsections provide an overview of each CDU statement. See the CDU Files Statements section for more detailed descriptions of each type of statement.

4.1 Defining Syntax

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

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

For example, you can write a command definition that uses a different syntax for a command verb when a particular qualifier is present. When you include the specified qualifier in the command string, the syntax defined in the secondary DEFINE statement applies to the command verb described by the primary DEFINE statement.

This is the format for the DEFINE SYNTAX statement:

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

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

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

```
DEFINE VERB ERASE
        IMAGE "DISK1:[MYDIR]ERASE"
        QUALIFIER SCREEN
        QUALIFIER LINE, SYNTAX=LINE ❶

DEFINE SYNTAX LINE ❷
        IMAGE "DISK1:[MYDIR]LINE"
        QUALIFIER NUMBER, VALUE(REQUIRED)
```

❶ The DEFINE VERB statement defines the verb ERASE. This verb accepts two qualifiers, /SCREEN and /LINE. The qualifier /LINE uses an alternate syntax, specified with the SYNTAX=LINE clause. If you 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.

4.2 Defining Values

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

With the TYPE clause, you can specify that a value type must be a built-in type (for example, a file specification) or you can specify that a value must be a user-defined keyword. Section 4.2.1 lists the built-in value types; Section 4.2.2 describes how to specify a user-defined keyword.

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

4.2.1 Built-In Value Types

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

$ACL	The value must be an access control list.
$DATETIME	The value must be an absolute time or a combination time. DCL converts truncated time values, combination time values, and keywords for time values (such as TODAY) to absolute time format. DCL fills blank date fields from the current system date and fills omitted time fields with zeros.
$DELTATIME	The value must be a delta time. DCL fills missing fields with zeros.
$FILE	The value must be a valid file specification which may include wildcard characters.
$NUMBER	The value must be an integer represented by either decimal, octal, or hexadecimal numbers.
$QUOTED_STRING	The value must be a quoted string. Note that DCL does not remove the quotation marks.
$REST_OF_LINE	DCL treats the rest of the line literally as the specified value ignoring spaces or punctuation marks. DCL does not remove quotation marks when processing the string.

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

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

4.2.2 User-Defined Keywords

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

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

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

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

This is the format for the DEFINE TYPE statement:

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

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

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

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

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

DEFINE TYPE SPEED_KEYWORDS❸
        KEYWORD FAST, DEFAULT
        KEYWORD SLOW
```

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

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

❸ The DEFINE TYPE statement lists the keywords that can be used with the 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.

4.3 Defining Command Verbs

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

The format for the DEFINE VERB statement is as follows:

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

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

You can specify the following verb clauses:

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

The following example illustrates a DEFINE VERB statement:

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

❶ The DEFINE VERB statement names the verb "SEARCH."

❷ The IMAGE verb clause identifies the image to be invoked at run time.

❸ The PARAMETER verb clause defines the first parameter to appear after the verb in the command string. LABEL, PROMPT, and VALUE are parameter clauses that further define the parameter. LABEL defines a name that the image uses to refer to the parameter. PROMPT indicates the prompt string to be issued if you do not specify the parameter in the command string. VALUE uses the REQUIRED clause to indicate that the parameter must be present in the command string.

4.4 Disallowing Entity Combinations

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

The DISALLOW verb clause has the following format:

```
DISALLOW expression
```

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

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

For example, a command definition 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 string. 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 commands would be disallowed:

```
TEST P1 P2/QUAL1

TEST P1/QUAL2 P2/QUAL3
```

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

4.4.1 Specifying Expression Entities

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

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

Names and Labels

You can refer to a parameter or qualifier using its name or label if the entity is defined in the current definition. To refer to a keyword, you can use its name or label if the keyword is in a keyword path that starts from the current definition, and if the keyword name or label is unique. (See the next section for more information about keyword paths.)

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

The following example disallows combinations of entities:

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

DEFINE TYPE GREEN_AMOUNT
      KEYWORD ALL
      KEYWORD HALF
```

In this example, you can use the qualifier names RED and BLUE in the DISALLOW verb clause because both names are used in the current definition. You can use the keyword ALL because it is in a keyword path 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 that references the keyword's value type definition. A keyword path can contain up to eight names (the first parameter or qualifier definition, plus seven DEFINE TYPE keyword definitions).

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

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

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

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

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

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

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

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

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

DEFINE TYPE RED_AMOUNT
      KEYWORD ALL
      KEYWORD MIXED

DEFINE TYPE GREEN_AMOUNT
      KEYWORD ALL
      KEYWORD HALF
```

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

Definition Paths

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

A definition path has the following format:

<definition-name> entity-spec

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

For example:

DISALLOW <SKIP> FIRST

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

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

DISALLOW <FILE> BILLS.ELECT AND GAS

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

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. For example, if you try to disallow the qualifier NOTES in the DEFINE SYNTAX statement, the entity NOTES would not be recognized as valid because the path to BILL_TYPES is not established in the DEFINE VERB statement for the command verb READ.

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

DEFINE VERB READ
       QUALIFIER NOTES

DEFINE SYNTAX BILL_TYPES
       DISALLOW <FILE>RECEIPTS
```

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

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

4.4.2 Operators

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

```
DISALLOW SINCE AND BEFORE
```

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

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

Table CDU–1 Summary of CDU Operators

Operator	Precedence	Meaning
ANY2	1	True if any two or more of the entities listed are present.
NEG	1	True if the negated form of the entity is present.
NOT	1	True if the entity is not present or if an entity is present by default.
AND	2	True if both entities are present.
OR	3	True if either entity is present.

The following example shows how to use the AND operator:

```
DISALLOW TERMINAL AND PRINTER
```

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

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

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

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

4.5 Identifying Object Modules

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

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

```
MODULE module-name
```

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

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

```
IDENT ident-string
```

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

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

```
MODULE TABLE  ❶
IDENT "Updated 4/15/84"  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 command table that the CDU creates when you use the command SET COMMAND/OBJECT to develop an object module for the new command.

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

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

5 Processing Command Definition Files

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

The command SET COMMAND has the following modes:

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

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

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

See the SET COMMAND Qualifiers section for additional information.

5.1 Adding Command Definitions to a Command Table

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

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

```
$ SET COMMAND SKIP
```

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

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

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 Object Modules

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

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

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

5.4 Creating New Command Tables

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

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

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

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

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

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

   ```
   $ LINK/SHARE TEST_TABLE
   ```

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

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

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

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

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

6 Using Command Language Routines

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

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

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

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

The Examples section shows two programs that call these routines. For more information about the command language routines, see the *VMS Utility Routines Manual.*

CDU Usage Summary

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

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

FORMAT

SET COMMAND *[filespec[, . . .]]*

COMMAND PARAMETER

filespec[, . . .]

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

Wildcard characters are allowed in the file specification.

usage summary

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

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

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

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

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

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

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.

CDU FILE STATEMENTS

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

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

DEFINE SYNTAX

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

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

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

Table CDU–2 shows how the DEFINE SYNTAX statement modifies the current command definition if an entity specifies a syntax change. After parsing the command string, DCL saves the command definition to determine if any entities in the command string are not allowed. Then, DCL invokes the image or routine specified by the command definition and uses the definition to process CLI$PRESENT and CLI$GET_VALUE calls.

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

DEFINE SYNTAX Specifies	Result
An image	An image overrides the image in the primary DEFINE statement. DCL invokes the new image after it parses the command string.
A routine	A routine overrides the routine in the primary DEFINE statement. DCL invokes the new routine when CLI$DISPATCH is called.
One or more disallows	One or more disallows are used during command parsing and they override disallows in the primary DEFINE statement. This applies to all entities in the command that have not been invalidated by the new syntax definition.
No disallows	Disallows from the primary DEFINE statement are used during command parsing.
The NODISALLOWS clause	No disallows are permitted, regardless of definitions in the primary DEFINE statement.

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 are parsed according to the new definitions. DCL uses the new parameter definitions when processing CLI$PRESENT and CLI$GET_VALUE calls. Note that in the DEFINE SYNTAX statement, P1 refers to the first parameter in the command string. To define additional parameters, use the PARAMETER clause in a secondary DEFINE statement to first enter the definitons for the original parameters exactly as they appear in the primary DEFINE statement. Then, enter the definitions for the additional parameters.
No parameters	Parameter definitions from the primary DEFINE statement are used when DCL parses the remainder of the command string. DCL also uses these parameter definitions when processing CLI$PRESENT and CLI$GET_VALUE calls.
The NOPARAMETERS clause	Parameters previously parsed are not reparsed to the new definitions. However, no parameters are allowed when DCL parses entities to the right of the entity that specifies the new syntax. DCL uses the NOPARAMETERS definition when processing CLI$PRESENT and CLI$GET_VALUE calls.
One or more qualifiers	The qualifiers previously parsed and the qualifiers that specify the syntax change are not affected. Qualifiers that appear in the command string after the entity that specifies the new syntax are parsed according to the new definition. DCL uses the new qualifier definitions when processing CLI$PRESENT and CLI$GET_VALUE calls. When DCL parses a command string that contains qualifiers that are ignored because of 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 previously parsed are ignored. No qualifiers are allowed when DCL parses entities to the right of the entity that specifies the new syntax. DCL uses the NOQUALIFIERS definition when processing CLI$PRESENT and CLI$GET_VALUE calls.

FORMAT

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

syntax-name

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

verb-clause[, . . .]

Optional verb clauses that define attributes of the command string.

DEFINE SYNTAX accepts the following verb clauses:

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

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

DISALLOW expression
NODISALLOWS

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

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

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

You can specify entities in an expression using an entity name or label, a keyword path, or a definition path. See Section 4.4.1 for more information about entities. You can also specify the operators AND, ANY2, NEG, NOT, or OR. See Section 4.4.2 for more information about these operators.

IMAGE image-string

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

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

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

May be used to specify up to eight parameters in the command string. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** is the position of the parameter in the command string. 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 characteristics for the parameter. You can use the following parameter clauses:

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

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

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

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

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

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

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

VALUE accepts the following parameter value clauses:

CONCATENATE	Indicates that a parameter can be concatenated to another parameter with a plus sign.
DEFAULT=default-string	Specifies a default value to be used 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 clause.
LIST	Permits you to enter a list of parameters separated by commas or plus signs.
NOCONCATENATE	Indicates that the parameters cannot be concatenated.
REQUIRED	Indicates that the parameter is required. All required parameters must precede optional ones. If you use the REQUIRED clause, you should also specify a prompt string. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Gives either a built-in value type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the value type name as a symbol. See Section 4.2.1 for more information about built-in value types.

QUALIFIER qual-name [,qual-clause[, . . .]]
NOQUALIFIERS

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

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

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

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

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

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

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

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

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

GLOBAL	Indicates that the qualifier applies to the entire command and can be placed after the verb or after a parameter. This is the default if you do not specify the PLACEMENT clause.
LOCAL	Indicates that the qualifier can appear only after a parameter value and that it applies only to that parameter.
POSITIONAL	Indicates that the qualifier can appear anywhere in the command string, but the meaning of the qualifier depends on its position: if the qualifier is used after a parameter value, it applies only to that parameter; if it is used after the verb, 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, enclose 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 qualifier clause.
LIST	Indicates that a list of values separated by commas can be specified for the qualifier. This list must be enclosed in parentheses, and the items must be separated by commas. Note that plus signs cannot be used to separate items in a list of qualifier values.
REQUIRED	Indicates that the qualifier must have an explicit 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 value type or the name of a DEFINE TYPE statement that defines a list of keywords that can be specified for the parameter. Specify the value type name as a symbol. See Section 4.2.1 for more information about built-in value types.

ROUTINE routine-name

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

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

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

EXAMPLES

1

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

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

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

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

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

```
$ SET COMMAND WRITER
```

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

```
$ WRITER/LINE
```

Here you invoke the LINE editor instead of the default editor (WRITER). Syntax redefinition is done from left to right because parsing of the string is done from left to right. This order means that when you specify two qualifiers that invoke different syntax lists, the leftmost qualifier takes precedence (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 keywords referenced by the VALUE(TYPE=type-name) clause. You can use the VALUE clause in a DEFINE VERB, DEFINE SYNTAX, or DEFINE TYPE statement to indicate predefined values (keywords) for command parameters, qualifiers, or keywords.

FORMAT

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

name

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

type-clause[, . . .]

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

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

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

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

You can use the following keyword clauses:

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

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

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

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

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

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

VALUE accepts the following 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.

Note: 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
: Symbolically equates either a built-in value type or the name of a DEFINE TYPE statement that defines keywords that can be specified as the keyword value. The TYPE clause cannot be specified if the DEFAULT clause is specified.

  See Section 4.2.1 for more information about built-in value types.

EXAMPLES

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

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

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

```
DEFINE SYNTAX DISPLAY_FLOWERS
        IMAGE "USER:[JOHNSON]FLOWERS"
        PARAMETER P1, LABEL=OPTION, VALUE(REQUIRED)
        NOQUALIFIERS
```

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

After you add the command definition to your process command table, you can 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 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, its parameters, its qualifiers, and the image or routine it invokes.

FORMAT

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

verb-name

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

verb-clause[, . . .]

Specifies optional verb clauses that define command string attributes.

The DEFINE VERB statement 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 all entities and entity combinations are allowed.

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

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

You can specify entities in an expression using an entity name or label, a keyword path, or a definition path. See Section 4.4.1 for more information about entities. You can also specify the operators AND, ANY2, NEG, NOT or OR. See Section 4.4.2 for more information about these operators.

IMAGE image-string

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

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

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

May be used to specify up to eight parameters in the command string. The NOPARAMETERS clause indicates that no parameters are allowed.

The **param-name** defines the position of the parameter in the command string. The 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 characteristics for the parameter. You can use the following parameter clauses:

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

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

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

LABEL=label-name symbolically defines a label for referring to a parameter at run time. 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 (31 characters maximum) when a parameter is omitted from the command string. If you do not specify a prompt string and a required parameter is missing, DCL uses the parameter name as the prompt string.

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

VALUE[(param-value-clause[, . . .])] specifies additional parameter characteristics; multiple parameter characteristics must be enclosed in parentheses and separated with commas.

VALUE accepts the following parameter value clauses:

CONCATENATE	Indicates a parameter can be concatenated.
DEFAULT=default-string	Specifies a default value using a string (95 characters maximum); the DEFAULT and REQUIRED value clauses are mutually exclusive. Do not use this clause to specify a default if the value is a keyword. Specify keyword defaults in the DEFINE TYPE statement, and by using the DEFAULT parameter clause.
LIST	Permits you to enter a list of parameters separated by commas or plus signs.
NOCONCATENATE	Indicates that the parameter cannot be concatenated.
REQUIRED	Indicates that the parameter is required. All required parameters must precede optional ones. If you use the REQUIRED clause, you should also specify a prompt string. The REQUIRED clause and the DEFAULT clause are mutually exclusive.
TYPE=type-name	Symbolically defines either a built-in value type or the name of a DEFINE TYPE statement that lists keywords for the parameter. See Section 4.2.1 for more information about built-in value types.

QUALIFIER qual-name [,qual-clause[, . . .]] NOQUALIFIERS

Specifies a qualifier that can be included in the command string; NOQUALIFIERS indicates no qualifiers are allowed. You can use the QUALIFIER clause up to 255 times in a DEFINE VERB statement.

The **qual-name** specifies the qualifier name as a symbol (the first four characters 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 in a batch job.

DEFAULT indicates that the qualifier is present by default.

LABEL=label-name symbolically defines a label for requesting information about the qualifier at run time (the default is the qualifier name).

NEGATABLE and **NONNEGATABLE** indicate whether the qualifier can be negated by adding "NO" to the qualifier name (the default is NEGATABLE).

PLACEMENT=placement-keyword indicates where you can position the qualifier in the command string. PLACEMENT accepts the following placement keywords:

GLOBAL	This is the default if you do not specify the PLACEMENT clause. It indicates that the qualifier can be placed after the verb or after a parameter.
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 in the command string, but its function varies according to its position: if it is positioned as a qualifier to the verb, it qualifies all parameters; if it is positioned as a parameter qualifier, it qualifies only the parameter.

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

VALUE[(qual-value-clause[, . . .])] used to specify additional qualifier characteristics; must be in entered in parentheses and separated with commas. If you do not specify any qualifier value clauses, DCL converts letters in a qualifier value to uppercase.

VALUE accepts the following clauses:

DEFAULT=default-string	Specifies a default value using a character string limited to 95 characters; the DEFAULT clause and the REQUIRED clause are mutually exclusive. Do not use this clause to specify a default if the value is a keyword; specify keyword defaults in the DEFINE TYPE statement and by using the DEFAULT qualifier clause.
LIST	Indicates a list of qualifier values; 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	Symbolically identifies either a built-in value type or a DEFINE TYPE statement that lists qualifier keywords. See Section 4.2.1 for more information about built-in value types.

ROUTINE routine-name

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

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

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

SYNONYM synonym-name

Symbol that defines a synonym for the verb name.

EXAMPLES

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

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

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

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

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

IDENT

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

FORMAT

IDENT *ident-string*

ident-string
A string (31 characters maximum) containing identifying information.

EXAMPLE

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

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

MODULE

Symbolically locates a command table object module.

FORMAT

MODULE *module-name*

module-name

The **module-name** refers to the address where the linker locates a command table module linked with your program.

By default, the CDU uses the object file name specified with the /OBJECT command qualifier. 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
```

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

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

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

CDU QUALIFIERS

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

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

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

/DELETE

Used to delete verb names or synonym names from the command table. If a verb name has synonyms, this qualifier deletes the specified verb or synonym name. If any synonyms remain, or if you delete synonyms and the original verb name remains, the remaining names still reference the verb definition.

You can use the /DELETE qualifier to delete a verb in either your process command table or in a command table file specified with the /TABLE qualifier. If you do not use the /TABLE qualifier to specify an alternate command table, the default is to delete verbs from your process command table. If you do not use the /OUTPUT qualifier to specify an output file, the default is to return the modified command table to your process.

You cannot use the /LISTING, /OBJECT, or /REPLACE 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 verb DO from your process command table.

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

The commands PUSH and SHOVE are deleted from the command table TEST_TABLE.EXE. The /OUTPUT qualifier writes the modified table to the file NEW_TABLE.EXE. If you do not include the /OUTPUT qualifier, the CDU uses the modified table to overwrite 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 has the same name as the first command definition file and a file type of LIS.

You can use the /LISTING qualifier only in /OBJECT or /REPLACE mode; you cannot create a listing in /DELETE mode. In /OBJECT and /REPLACE modes, the default is /NOLISTING.

FORMAT

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

SET COMMAND/NOLISTING

listing-filespec

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

filespec

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

EXAMPLES

1 `$ SET COMMAND/LISTING TEST`

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

/OBJECT

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

An object module containing a command table can be linked with the object modules from your program. This enables the program to use its own command table for parsing command strings and executing routines.

You can specify only one command definition file when you use SET COMMAND/OBJECT.

If you specify the /OBJECT qualifier and omit the file specification, output is written to the default device and directory; the object file has the same name as the input file and a file type of OBJ.

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

FORMAT

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

object-filespec

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

filespec

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

EXAMPLES

1 `$ SET COMMAND/OBJECT TEST`

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

2 `$ SET COMMAND/OBJECT=A TEST`

In this example, the command definition file TEST.CLD 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-filespec] [filespec[, . . .]]*

SET COMMAND/NOOUTPUT

output-filespec

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

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

filespec

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

EXAMPLES

1 `$ SET COMMAND/OUTPUT TEST`

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

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

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

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

3 `$ SET COMMAND/NOOUTPUT TEST`

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

/REPLACE

Used to add or replace verbs in the command table.

You can use the /REPLACE qualifier to either modify the process command table or, with the /TABLE qualifier, to modify a command table file.

You cannot use the /REPLACE qualifier with the /OBJECT or the /DELETE qualifiers. If you do not explicitly specify /DELETE, /OBJECT, or /REPLACE, the default is /REPLACE.

FORMAT

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

filespec

The file to be processed (default file type CLD). Wildcard characters are allowed.

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 qualifier indicates that the modified command table is to be written to your process.

2 `$ SET COMMAND/TABLE/OUTPUT SCROLL`

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

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

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

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

4 `$ SET COMMAND/TABLE=TEST_TABLE MYCOMS`

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

/TABLE

Specifies the command table to be modified (default file type EXE). If you specify the /TABLE qualifier and omit the file specification, the current process command table is modified.

May be used with /DELETE or /REPLACE but not with /OBJECT; the default is /TABLE with no input file specification.

If you include a file specification, the specified command table is modified.

If you use the /TABLE qualifier without the /OUTPUT qualifier, the modified command table replaces your process command table.

FORMAT

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

input-filespec

The input file which contains the command table to be edited (default file type EXE).

filespec

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

EXAMPLES

1 `$ SET COMMAND/TABLE TEST`

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

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

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

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

3 `$ SET COMMAND/TABLE=A`

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

CDU EXAMPLES

Adding a Command to Your Process Command Table

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

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

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

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

```
$ SET COMMAND SAMPLE
```

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

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

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

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

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

30    END
```

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

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

```
$ SAMPLE
```

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

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

```
$ SAMPLE MYFILE
```

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

```
FILESPEC = MYFILE
```

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

```
$ SAMPLE MYFILE/EDIT
```

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

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

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

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

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

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

Creating an Object Module Table for Your Program

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

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

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

```
MODULE TEST_TABLE

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

DEFINE VERB SEARCH
        ROUTINE SEARCH_COMMAND
        PARAMETER P1, LABEL = SEARCH_STRING

DEFINE VERB EXIT
       ROUTINE EXIT_COMMAND
```

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

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

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

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

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

    PRINT 'SEND COMMAND'
    PRINT ''

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

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

90  SUBEND

100 SUB SEARCH_COMMAND
    EXTERNAL INTEGER FUNCTION CLI$PRESENT,CLI$GET_VALUE

    PRINT 'SEARCH COMMAND'
    PRINT ''

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

190 SUBEND

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

290 SUBEND

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

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

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

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

```
$ BASIC USEREXAMP
```

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

```
$ LINK USEREXAMP,TEST
```

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

```
$ RUN USEREXAMP
```

USEREXAMP.EXE displays the following prompt on your screen:

```
TEST>
```

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

```
TEST> SEND
```

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

```
SEND COMMAND

TEST>
```

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

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

DCL invokes the SEND_COMMAND routine which displays the following text:

```
SEND COMMAND

FILESPEC = MESSAGE.TXT

TEST>
```

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

```
TEST> SEND/EDIT MESSAGE.TXT

SEND COMMAND

/EDIT is present

FILESPEC = MESSAGE.TXT

TEST>
```

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

```
TEST> SEARCH
```

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

```
SEARCH COMMAND

TEST>
```

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

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

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

```
TEST> EXIT
```

Index

B

C

D

F

G

I

K

L

M

N

O

P

Q

R

S

T

V

Reader's Comments

VMS Command Definition Utility Manual
AA-LA60A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

— Do Not Tear - Fold Here and Tape ——————————————————————

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here ——————————————————————

Cut Along Dotted Line