DataGeneral

Customer Documentation

Using the Command Processor

AVIION™
PRODUCT LINE

Using the Command Processor

093-000706-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000706

Printed in the United States of America
Rev. 00, December 1989

Notice

Using the Command Processor
093-000706-00
093-000707-00 (Japan only)

Revision History:

Original Release - December, 1989
Addendum 086-000167-00 - June, 1990
(086-000169-00 Japan only)

Effective with:

Mxdb, Rev. 1.10

A vertical bar in the margin of a replacement page indicates substantive technical change from the previous revision.

Preface

This manual describes how to use the Command Processor (CP), a utility that provides the user interface to interactive tools, such as Mxdb, that operate from a textual interface. The CP defines command rules, checks command arguments, and offers several predefined facilities. This manual contains both tutorial and reference information.

Using the Command Processor is intended for readers who are familiar with the AViiON™ DG/UX™ or 386/ix™ system, have programming experience, and use interactive tools such as Mxdb.

Manual Organization

Chapter 1 introduces the CP and describes how to create command lines.

Chapter 2 describes the CP utilities such as help, session logging, and execution control-flow.

Chapter 3 describes how to customize your environment by writing macros, creating and managing realms, and changing the values of arguments.

Chapters 4-7 contain the available on-line help messages in printed form. The messages include command descriptions and information about a variety of topics.

Reader, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. All DG/UX commands, pathnames, and names of files, directories, and manual pages also use this typeface.
`constant width/ monospace`	Represents a system response on your screen. Syntax lines also use this font.
italic	In format lines: Represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands. In text: Indicates a term that is defined in the manual.
[*optional*]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional. The brackets are in regular type and should not be confused with the boldface brackets shown below.
[ ]	In format lines: Indicates literal brackets that you should type. These brackets are in boldface type and should not be confused with the regular type brackets shown above.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.
$ and %	In command lines and other examples: Represent the system command prompt symbols used for the Bourne and C shells, respectively. Note that your system might use different symbols for the command prompts.
↵	In command lines and other examples: Represents the New Line key, which is the name of the key used to generate a new line. (Note that on some keyboards this key might be called Enter or Return instead of New Line.) Throughout this manual, a space precedes the New Line symbol; this space is used only to improve readability – you can ignore it.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as **<Ctrl-D>** and **<Esc>**) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, boldface versions of these symbols (described below) that you do type.
<, >, >>	In text, command lines, and other examples: These boldface symbols are redirection operators, used for redirecting input and output. When they appear in boldface type, they are literal characters that you should type.

Contacting Data General

Manuals

- To order any Data General manual, please use the enclosed TIPS order form (USA only) or contact your local Data General sales representative.
- If you have comments on this manual, please use the prepaid Comment Form that appears at the back. We want to know what you like and dislike about this manual.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, and you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS for toll-free telephone support. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

Free telephone assistance is available with your warranty and with most Data General service options. Lines are open from 8:30 a.m. to 8:30 p.m., Eastern Time, Monday through Friday.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Related Documents

This section lists the documents referred to in the text of this manual.

- *Using the Multi-extensible Debugger (Mxdb for DG/UX and 386/ix Systems)* (093-000710)

End of Preface

Contents

Chapter 1 Introduction to the Command Processor

Chapter 2 Using Command Processor Utilities

Chapter 3 Customizing Your Environment

Tables

Table

Chapter 1
Introduction to the Command Processor

The Command Processor (CP) is a command interpreter; it is a utility that provides a uniform user interface to interactive tools, such as Mxdb, that operate from a textual interface. With the CP, you can dynamically create variables and tailor your working environment by creating commands (macros), organizing commands into groups, and modifying commands.

The CP defines rules for the syntax of commands, checks the syntax and meaning of command arguments, and offers several predefined facilities, such as help, session logging, execution control-flow, and command set management.

This chapter discusses CP terms and concepts, and then tells how to do the following tasks:

- Create a command line
- Continue a line
- Capture command output by using backquotes
- Put special characters in a command line
- Balance character pairs
- Use backquotes within braces
- Evaluate a series of commands
- Insert a comment

For information about the help system, see "Getting Help" in Chapter 2.

Terms and Concepts

This section defines the terms character, word, phrase, and command; it also describes how to abbreviate commands and other words.

Character

A character is any ASCII character that you can enter from your keyboard. Table 1-1 shows the characters that have special syntactic meaning to the CP.

Table 1-1 Characters with Syntactic Meaning

Character	Symbol	Meaning
space		Separate words in a phrase
tab	<tab>	Separate words in a phrase
comma	,	Separate phrases in a command
semicolon	;	Separate commands in a line
New Line	↵	Separate commands on different lines
colon	:	Connect a realm name and a command name
double quotes	""	Enclose a string in quotation marks
single quotes	''	Enclose a string in quotation marks
parentheses	()	Group characters or words
brackets	[]	Group characters or words
braces	{}	Group characters or words
backquote	`	Capture command output

Word

A word is normally composed of one or more printable characters. The ordinary word characters are as follows: letters (A–Z and a–z), digits (0–9), and the characters !#$%&*+–./<=>?@\^_|~. However, a word can contain any ASCII character (including those shown in Table 1-1), as explained later in this chapter in the section "Putting Special Characters into a Command Line." Grouping characters ((), [], and {}) are considered part of the words they group together, as are any enclosed separators.

Multiple words are separated by whitespace (spaces or tabs), commas, or semicolons.

Phrase

A phrase consists of one or more words. Phrases are separated by commas.

Command

A command contains one or more phrases and is terminated by a New Line character or a semicolon. The name of a command is normally a word but may be a phrase (see "Putting Special Characters into a Command Line"). A command name cannot contain a colon.

Abbreviating Commands and Other Words

Commands are generally complete English words. To provide flexibility, the CP enables you to abbreviate the names of commands, arguments, certain variables, and some argument values. Since an abbreviation must be unique, the minimum abbreviation depends on the names against which the abbreviation is being compared. The minimum abbreviation is determined by what commands, variables, and macros are visible or by what arguments exist for a command or macro.

A name has one or more "syllables" separated by hyphens or underscores; for instance, the command **print-command** has two "syllables." Names are case-insensitive, and the hyphen (-) and underscore (_) are equivalent. The name you specify is resolved to a command (or other name) as follows:

1. The command and the name you specify are an exact match. For example, the specified name "evaluate" matches the command **evaluate** exactly.
2. The command has the same number of "syllables" as the specified name, and each syllable begins with the characters you specify. Thus, you could specify "eval" for the **evaluate** command or "pri-com" for the **print-command** command.
3. The command has more syllables than the specified name has, and begins with the characters you specify. As an example, you could use "pri" to indicate the **print-command** command.

Remember that any abbreviations must be unique. For instance, "i" is not a unique abbreviation for the **include** command; the **if** command begins with the same character. You must specify "in" for **include**.

Creating a Command Line

This section explains how to create and enter a command line. It also describes the three kinds of arguments that a command can take, the four ways in which an argument can receive its value, and the relationship between argument values and types.

Entering a Command

A command takes a series of arguments and performs the appropriate action. Each argument is classified as required, optional, or keyword, and can receive its value by position, by name, by default, or implicitly. The output is normally displayed on your screen.

The first phrase of a command starts with the command name as the first word; succeeding words are values for required or optional arguments of the command. The rest of the phrases each start with a comma followed by optional whitespace, then a keyword and, optionally, a value for that keyword.

To enter a command, type the command after the prompt on your screen and press the New Line key. A prompt indicates the realm in which you are working; a realm contains a group of commands that you can access.

Here are some sample commands that follow the default prompt for the c-p realm (c-p):

```
(c-p) write Here are some symbols: #$&*<>?\|~ ↵
Here are some symbols: #$&*<>?\|~
(c-p) include script_file, continue ↵
(c-p)
```

Above, **write** and **include** are commands, **Here are some symbols: #$&*<>?\|~** and **script_file** are required arguments, and **continue** is a keyword.

Required Arguments

If an argument is required, you must specify it. You can specify the value by position (usually the simpler method) or by keyword (if you remember the keyword but forget the order of arguments). See the "Values by Position and Name" section for more detail.

The following one-phrase command (**assign**) requires two arguments, one a variable (x, for example), and the other a phrase ("computer"):

```
(c-p) assign x computer ↵
(c-p)
```

Optional Arguments

You can specify an optional argument by position or keyword. If you omit an optional argument, the CP uses the default value associated with that argument.

Following is a one-phrase command (**prompt-string**) with an optional argument that represents a new prompt:

```
(c-p) prompt-string (Yes?) ↵
(Yes?)
```

Keyword Arguments

A keyword argument cannot receive a value by position; to specify a keyword argument, you must use the keyword. If you omit a keyword argument, the CP uses the argument's default value. If the argument has an implied value, you can specify the keyword and omit the value.

Following is a command (**realm-use-list**) with a keyword argument name (**realm**, which specifies which realm use list to display) and its value (c-p):

```
(c-p) realm-use-list, realm c-p ↵
{ { command-processor } { characters } }
(c-p)
```

In the next example, the **realm-use-list** command uses the **realm** keyword without specifying a value; the implied value is the current realm (c-p in this case):

```
(c-p) realm-use-list, realm ↵
{ { command-processor } { characters } }
(c-p)
```

Values by Position and Name

You can specify arguments by position or by name. A value by position is associated with a particular argument because of its position in the command line. A value by name follows a keyword.

In the following **assign** command, the required arguments (**variable** and **phrase**) receive their values (x and "computer") by position:

```
(c-p) assign x computer ↵
(c-p)
```

In the following equivalent examples the arguments receive their values by name:

```
(c-p) assign, variable x, phrase computer ↵
(c-p)
```

```
(c-p) assign, phrase computer, variable x ↵
(c-p)
```

Default and Implied Values

Every command argument is given a value when the command is executed. Arguments that are not given values by name or by position are given values by default. Arguments that are mentioned by name but are given no explicit value on the command line are given values implicitly. Implied values are often set up for keywords, so that just mentioning the keyword does something useful.

Command Arguments

Use the **help** command to find out what arguments a command accepts. To generate a one-line list of arguments for a command (**define-realm**, for example), specify the keyword **verbosity** followed by the phrases "text none" and "arguments short" in braces:

```
(c-p) help define-realm, verbosity { text none, arguments short } ↵
define-realm   name [use], prompt, doc
(c-p)
```

As shown above, the **define-realm** command accepts arguments in each of the three categories: required (**name**), optional (**use**), and keyword (**prompt** and **doc**). The following example shows argument values being specified by position:

```
(c-p) define-realm macros { macros command-processor } ↵
(c-p)
```

This example shows argument values being specified by name:

```
(c-p) define-realm, name macros, use { macros c-p }, prompt (m) ↵
(c-p)
```

This example shows argument values being specified by default:

```
(c-p) define-realm macros ↵
(c-p)
```

This example shows argument values being specified implicitly:

```
(c-p) define-realm macros, prompt ↵
(c-p)
```

Here is a summary of command argument rules:

- Any argument can be specified by name.
- Any argument can have an implied value.
- A keyword argument cannot receive its value by position; you must use the keyword or accept the argument's default value.
- A required argument cannot have a default value.

To reset default and implied values, use the **change-argument-value** command.

The rest of this section goes into more detail about command arguments.

The following tables show possible combinations of command **c** with required argument **a1**, optional argument **a2**, and keyword argument **a3**. Values assigned explicitly (by name or position) are indicated as **v1**, **v2**, and **v3**. Values assigned implicitly are indicated as **i1**, **i2**, and **i3**. Values assigned by default are indicated as **d2** and **d3**.

A help message for command **c** with arguments displayed at the "short" verbosity level (**help c, v {text none, arguments short}**) would show the following:

```
c a1 [a2], a3
```

Table 1-2 shows all the combinations of command **c** and its required argument (which cannot have a default value). In the example, "def-r" is the **define-realm** command.

Table 1-2 Combinations of a Command and a Required Argument

How Value Is Specified

	By Position	By Value	Default	Implied
Command	c v1	c, a1 v1	Cannot have a default value	c, a1
Resulting Values	v1, d2, d3	v1, d2, d3	No resulting values	i1, d2, d3
Example	def-r macros	def-r, name macros	Cannot have a default value	def-r, name

Table 1-3 shows the combinations of command **c**, its required argument (**v1**, with a value assigned by position), and its optional argument.

Table 1-3 Combinations of a Command, a Required Argument, and an Optional Argument

	How Value Is Specified			
	By Position	**By Value**	**Default**	**Implied**
Command	c v1 v2	c v1, a2 v2	c v1	c v1, a2
Resulting Values	v1, v2, d3	v1, v2, d3	v1, d2, d3	v1, i2, d3

Table 1-4 shows the combinations of command **c**, its required argument (**v1**, with a value assigned by position), and its keyword argument (which cannot receive a value by position).

Table 1-4 Combinations of a Command, a Required Argument, and a Keyword Argument

	How Value Is Specified			
	By Position	**By Value**	**Default**	**Implied**
Command	Cannot have value by position	c v1, a3 v3	c v1	c v1, a3
Resulting Values	No resulting values	v1, d2, v3	v1, d2, d3	v1, d2, i3

Argument Values and Types

A type is a category of argument values accepted by the CP; each argument of a command has a type. When you specify an argument value, that value is checked to see whether it conforms to the syntax of the particular type. If the argument value you specify is invalid, you will receive an error message and execution will abort instead of having the invalid value passed to the command.

For example, the first argument to Mxdb's debugger realm's **breakpoint** command is the **line** argument, which is of type line-number. If you specify a decimal integer, CURRENT (the current line number, plus or minus an optional value), LAST (the last line number, minus an optional value), or an abbreviation of CURRENT or LAST for this argument, the CP passes the value to the command. The line-number type accepts values matching this syntax; the command can then check whether a specified integer is within the range of the specified module. Other values are rejected. For instance, if you specify "**breakpoint a**," you receive an error message, because "a" is not a recognized value for a line number.

Continuing a Command Line

To continue a command onto the next line, type a backquote and press the New Line key. The backquote may be followed by blank space.

The CP then adds a backquote to the prompt on the continued line. Here is an example, where zoo is the variable:

```
(c-p) assign zoo lion tigers and ' ↵
(c-p)` bears ↵
(c-p)
```

Inserting Comments

You can insert comments after a command. To begin a comment, type two commas; to terminate a comment, type a semicolon or press the New Line key.

The following example shows a comment terminated by a New Line:

```
(c-p) assi pi 3.14159 ,, The value of pi ↵
(c-p) pi ↵
3.14159
(c-p)
```

The following example shows a comment terminated by a semicolon:

```
(c-p) wri Current realm: ,,show realm; realm ↵
Current realm:
command-processor
(c-p)
```

All input from the comma pair through the New Line or semicolon is ignored, including a line continuation character.

Capturing Command Output

The CP enables you to capture command output, and then insert it into a command line. To do this, put a backquote before the command whose output you want to capture. If this command has arguments, enclose the command and its arguments in a pair of braces.

A simple example follows:

```
(c-p) write The current realm is '{realm}. ↵
The current realm is command-processor.
(c-p)
```

Here is an example using an argument and braces:

```
(c-p) assign x '{realm-use-list, realm c-p} ↵
(c-p) x ↵
{ { command-processor } { characters } }
(c-p)
```

Note that if you type a variable name (such as x above) at the beginning of a line, that variable's value is displayed:

```
(c-p) assi name realm ↵
(c-p) name ↵
realm
(c-p)
```

If you precede such a variable name with a backquote, the CP resolves the variable's value and executes it as a command:

```
(c-p) 'name ↵
command-processor
(c-p)
```

You can also capture output from multiple commands:

```
(c-p) assi x '{realm; realm-use} ↵
(c-p) x ↵
command-processor
{ { command-processor } { characters } }
(c-p)
```

More involved instances using backquotes are covered later in this chapter in the section "Using Backquotes Within Braces."

Putting Special Characters into a Command Line

This section explains how to do these tasks:

- Put a syntactic character (such as a comma or space) into a command line without having the CP treat it specially.
- Put a control character other than a tab or New Line into a command line.

Table 1-1 lists the characters that the CP interprets as having syntactic meaning. If you try to create a CP variable containing one of these characters, you may have difficulty.

Control characters in general may pose difficulties. For example, trying to type a control character while you are using the debugger may produce an error message.

Four ways exist to put special characters into a command line:

1. Enclose ("group") the character in braces, brackets, or parentheses.
2. Enclose ("quote") the character in a pair of double or single quotation marks preceded by a backquote.
3. Use a command from the characters realm for a specific character (see Chapter 7, "Character Commands").
4. Use the **character-from-code** command (see Chapter 7).

Table 1-5 shows which of the first three methods apply to various special characters. Method 4 applies to any character if you know its ASCII value.

Table 1-5 Ways to Enter Various Special Characters

Character	Grouped[1]	Quoted	Character Command
space	Yes	Yes	Yes
tab	Yes	Yes	Yes
comma	Yes	Yes	Yes
semicolon	Yes	Yes	Yes
New Line	Yes	Yes	Yes
double quote	No	Yes	Yes
single quote	No	Yes	Yes
brace	No	Yes	Yes
bracket	No	Yes	Yes
parenthesis	No	Yes	Yes
backquote	No	No	Yes
carriage return	No	Yes	Yes
form feed	No	Yes	Yes
null	No	Yes	Yes

[1]Enclosed in braces, brackets, or parentheses

Restrictions for putting a character into a command line may depend on context. For example, it is easy to create a CP variable whose *value* contains spaces:

```
(c-p) assi x Now is the time. ↵
(c-p) x ↵
Now is the time.
(c-p)
```

However, you must use one of the methods from Table 1-5 to create a CP variable whose *name* contains a space. For example, you can use braces as grouping characters to create a variable whose name is the word '{ }':

```
(c-p) assi { } braces ↵
(c-p) { } ↵
braces
(c-p)
```

If you want to create a CP variable whose value contains a comma, you can enclose the comma in quotation marks and use a backquote:

```
(c-p) assi x '","phrase containing comma ↵
(c-p) x ↵
,phrase containing comma
(c-p)
```

You can put a literal backquote into a phrase by using the **backquote** command from the characters realm:

```
(c-p) assi y backquote '{char:backquote}phrase ↵
(c-p) y ↵
backquote `phrase
(c-p)
```

To put control characters such as the bell (Ctrl-G) into a phrase, you must use the **character-from-code** command (see Chapter 7). This example creates a CP variable that produces a beep on most display units:

```
(c-p) assi beep '{char:char 7}Beep! ↵
(c-p) beep ↵
Beep!
(c-p)
```

You can create a CP variable whose name is a phrase rather than a word, though this is not recommended (see the note below):

```
(c-p) define-realm test ↵
(c-p) realm test ↵
(test) assign '"do it" This is not wise., doc ' ↵
(test)' "CP variable whose name is a phrase" ↵
(test) '"do it" ↵
This is not wise.
(test) help, command ↵
                Command: do it          Realm: test

Summary         CP variable whose name is a phrase

   Arguments    <none>

(test)
```

NOTE: If you create a CP variable whose name contains a space, tab, or new line, you will not be able to use that variable in the **name-and-phrase** argument to a **do-sequence** command (described in Chapter 4); **do-sequence** would treat the name as multiple names.

You can put braces, brackets, and parentheses into a command with no difficulty if they are paired. However, to use one alone you must take special action, as previously indicated in Table 1-5. The next section discusses the rules for balancing character pairs.

Balancing Character Pairs

If a command line has a word containing a single quotation mark, double quotation mark, parenthesis, bracket, or brace, that word normally must contain a matching character to form a pair. To create a word containing one of these characters unpaired, you can use either of the following two techniques shown earlier in Table 1-5:

1. Enclose the character in a pair of double or single quotation marks preceded by a backquote; or,
2. Use a command from the characters realm for a specific character.

The relevant character commands are as follows:

single-quote
double-quote
left-parenthesis
right-parenthesis
left-square-bracket
right-square-bracket
left-curly-brace
right-curly-brace

The following example creates and executes a CP variable whose name contains parentheses:

```
(c-p) assi abc(1) xyz ↵
(c-p) abc(1) ↵
xyz
(c-p)
```

The following example writes a word containing an unpaired brace:

```
(c-p) write ab'"{"cd ↵
ab{cd
(c-p)
```

Here is an equivalent example using the left-curly-brace command:

```
(c-p) wri ab'{characters:left-curly}cd ↵
ab{cd
(c-p)
```

If you put an unpaired right brace, bracket, or parenthesis in a command line and do not use one of the above methods, the CP displays an error message. If you put an unpaired left brace, bracket, or parenthesis in a command line and do not use one of the above methods, the CP changes the prompt until you provide the matching character. For example:

```
(c-p) assi bracket-stuff [ ↵
(c-p)[ line of input ↵
(c-p)[ ] ↵
(c-p) bracket-stuff ↵
[
line of input
]
(c-p)
```

Using Backquotes Within Braces

As described earlier, you can capture command output by putting a backquote before the command whose output you want to capture.

However, if you use a single backquote within braces, that backquote has no special syntactic meaning. For example:

```
(c-p) assi name realm ↵
(c-p) write {`name} ↵
{`name}
(c-p)
```

To execute a command within braces, use one more backquote than the number of pairs of braces. To continue the above example:

```
(c-p) write {``name} ↵
{realm}
(c-p) write {{```name}} ↵
{{realm}}
(c-p)
```

Other paired characters, such as parentheses and square brackets, do not affect backquote resolution:

```
(c-p) write "([`name])" ↵
"([realm])"
(c-p)
```

An exception to the rule for using backquotes within braces occurs within the body of a macro definition. In this case, the CP resolves a command preceded by a backquote. For example:

```
(c-p) define-macro bang {phrase} {write ! `phrase !} ↵
(c-p) bang two words ↵
! two words !
(c-p)
```

For more information about macro definitions, see the "Writing Macros" section in Chapter 3.

Evaluating a Series of Commands

The CP **evaluate** command evaluates one or more commands and displays the output. Use **evaluate** to capture command output that contains characters you want the CP to interpret syntactically:

```
(c-p) assi x '","verbosity {text short, arg short} ↵
(c-p) eval { help shell ''x } ↵
shell      Execute a sub-shell or a shell command sequence.
   [command-line]
(c-p)
```

In the previous example, **evaluate** is used after the value of x is assigned. In the next example **evaluate** is used when a value is assigned to x:

```
(c-p) assi name realm ↵
(c-p) assi x '{eval { name }} ↵
(c-p) x ↵
realm
(c-p)
```

You can do the same thing by using backquote evaluation:

```
(c-p) assi name realm ↵
(c-p) assi x 'name ↵
(c-p) x ↵
realm
(c-p)
```

By combining **evaluate** with backquote evaluation, you can carry the command evaluation a step further:

```
(c-p) assi name realm ↵
(c-p) assi x '{eval { 'name }} ↵
(c-p) x ↵
command-processor
(c-p)
```

If the argument is a command containing no captured command output, the **evaluate** command has the same effect as if you omitted it:

```
(c-p) evaluate {realm} ↵
command-processor
(c-p) realm ↵
command-processor
(c-p)
```

End of Chapter

Chapter 2
Using Command Processor Utilities

This chapter describes various Command Processor (CP) utilities. It tells how to do the following tasks:

- Use the help facility, including command prompting
- Log a session
- Perform CP control flow
- Manipulate phrases as sequences

Getting Help

The CP offers two ways to use its **help** system: a help command and command prompting.

The help Command

The **help** command displays information about a command, argument, realm, or topic. To use this command, type **help** after invoking the tool you are using, such as Mxdb. Then, if you want general information, press the New Line key. If you want information about a specific command, argument, realm, or topic, type that name after **help** and press the New Line key. For example, if you type **help define-realm** and press the New Line key, you will see the summary portion of the **define-realm** command's help message, which defines the command and its arguments, and shows examples:

```
(c-p) help define-realm ↵
                Command: define_realm        Realm: command-processor

Summary         Create a new realm.

   Arguments    Required:
                  name          The name for the new realm
                Optional:
                  use           A list of realms grouped using braces
                Keyword:
                  prompt        The prompt string for this realm
                  doc           Up to three enquoted help text strings

  Examples      define-realm quick
                def-r myrealm ,use {{myrealm c-p}}

For further help, type "help define_realm <argument name>".
(c-p)
```

To get a more detailed message, add a **,verbosity** argument. For example, type this command:

```
(debug) help define-realm, v ↵
```

You will then see the entire **define-realm** help message, which also elaborates the definitions and examples.

Command Prompting

The command prompting facility helps you to enter commands interactively. Any command will prompt you for input if you type the command followed by a comma and no argument. Command prompting displays each argument name, one at a time, showing the default value in parentheses.

To use the default value, press the New Line key. To use another value, type the value and press New Line. If no default is shown, the argument is required and you must enter a value.

Invoking Command Prompting

To invoke the command prompting facility for a command, type the command followed by a comma; then press the New Line key. The comma may be preceded or followed by blank space.

For example, to get command prompting on the **write** command:

```
(c-p) write, ↵
      Type ",help" for help.
      text () =
```

At this point, the prompting facility is asking for a value for the **text** argument. To enter a value, type the value and press the New Line key. For example:

```
(c-p) write, ↵
      Type ",help" for help.
      text () = computer ↵
```

You are then prompted for the remainder of the arguments. To use the defaults, press New Line for each one.

```
(c-p) write, ↵
      Type ",help" for help.
      text () = computer ↵
      message (no) = ↵
      no-newline (no) = ↵
```

The final line asks whether you want to execute the selections you have just made. To answer yes, press New Line.

```
      Execute? (Yes) = ↵
computer
(c-p)
```

If you want to change one or more of your selections before you execute the command, type **No** and the query process repeats. Type your new selection(s):

```
                    Execute? (Yes) = No ↵
                    text (computer) = Computers are fun. ↵
                    message (no) = ↵
                    no-newline (no) = Yes ↵
                    Execute? (Yes) = ↵
Computers are fun.(c-p)
```

All arguments that have defaults are initialized to their default value unless you have explicitly supplied another value. In the example above, the **text** argument initially has no default. However, the default is set to "computer." Thus, when you go through the prompting a second time, that value is displayed.

Issuing Prompting Facility Commands

At any time during the prompting session you can issue a command, preceded by a comma, that will take a particular action. The **,help** command displays the available prompting facility commands (which you may abbreviate). Table 2-1 organizes these commands by topic and task.

Table 2-1 Prompting Facility Commands by Category

Topic	Task	Command
Information	Describe the current argument	,
	Display a help message	,help
	Refresh the screen	,refresh
Argument	Specify a value	value
	Select the default value	,default
	Select the implied value	,implied
Termination	Abort back to the top level	,abort
	Execute the command	,execute
Navigation	Move back one argument	,previous

Logging a Session

To create files containing records of command line input, output, or errors during the debugging session, use the **log** command.

This command line creates an input log file named **login**, an output log file named **logout**, and an error log file named **logerr**:

```
(c-p) log, input login, output logout, error logerr ↵
```

If the files **login, logout,** and **logerr** do not exist, the **log** command creates them. If the files do exist, output will be appended to them.

If you want one log file that includes input, output, and errors, type a command line like this:

```
(c-p) log logfile ↵
```

You can also specify an absolute (complete) pathname:

```
(c-p) log /usr/mark/mxdb/anotherlogfile ↵
```

To create a log file overwriting any existing file, use one of these arguments: **input-delete, output-delete,** or **error-delete**. This command line overwrites any existing input logfile named **login**:

```
(c-p) log, input login, input-delete ↵
```

To display the current log files, type **log** with no arguments:

```
(c-p) log ↵
input log files:
                /usr/chris/login
output log files:
                /usr/chris/logout
error log files:
                /usr/chris/logerr
```

To turn all logging off, use the **unlog** command with no arguments:

```
(c-p) unlog ↵
input log files:
                /usr/chris/login
output log files:
                /usr/chris/logout
error log files:
                /usr/chris/logerr
```

This command turns logging off and writes the names of the log files to the standard output. You can also specify a filename to turn off logging to a file:

```
(c-p) unlog logerr ↵
```

Performing CP Control Flow

This section compares CP control flow with Mxdb debugger control flow and describes how to do these tasks:

- Execute command(s) if a command writes a nonnull phrase
- Execute command(s) while a command writes a nonnull phrase
- Protect commands in case an error occurs
- Check whether two CP variables have the same value
- Negate a test
- Perform an AND test
- Perform an OR test

Comparison with Debugger Control Flow

This section discusses similarities and differences between Mxdb debugger control flow and Command Processor control flow.

The general semantics of Mxdb and CP control flow are similar. The debugger and the CP both provide **if** and **while** commands to control the flow of command execution. Each **if** command accepts three arguments: a **predicate**, a **then** phrase, and an **else** phrase. Each **while** command accepts two arguments: a **predicate** and a command **body**. The kind of values accepted by the **then**, **else**, and **body** arguments are the same in the debugger and the CP.

However, the value that the **predicate** argument accepts is not the same. In the debugger realm, the **predicate** argument accepts a language expression that evaluates to true or false as defined by the language being used. In the command-processor realm, the **predicate** argument accepts and evaluates a series of commands, each of which returns a phrase. If any of the phrases is nonnull, the predicate is considered true.

In the c-p realm, control-flow commands capture and discard the standard output from predicate commands. If you want to write output in a predicate command that is not discarded, used the **write** command's **message** argument; this writes to the error output. See "Terms and Concepts" in Chapter 3 for a discussion of standard output and error output.

Executing If Phrase Is Nonnull (c-p:if)

The Command Processor's **if** command conditionally executes one or more commands. **If** evaluates the predicate. If it returns a nonnull phrase, then it evaluates the **then**-part argument value; otherwise it evaluates the **else**-part value.

This example sets x to the value of abc, if abc is nonnull:

```
(c-p) assign abc xyz ↵
(c-p) if { abc } { assign x 'abc } ↵
(c-p) x ↵
xyz
(c-p)
```

To evaluate an empty variable, try this example:

```
(c-p) assign x "" ↵
(c-p) if { x } {wri x is not empty}, else {wri x is empty} ↵
x is empty
(c-p)
```

These commands evaluate a nonempty variable:

```
(c-p) assign x abc ↵
(c-p) if { x } {wri x is not empty}, else {wri x is empty} ↵
x is not empty
(c-p)
```

Executing While Phrase Is Nonnull (c-p:while)

The Command Processor's **while** command executes one or more commands while a predicate is nonnull. **While** evaluates the predicate; if the predicate writes a nonnull phrase, **while** evaluates the body and repeats.

The following example sets a CP variable, and then displays and shortens the value of the variable while it is nonnull:

```
(c-p) assign x a b c ↵
(c-p) while {x} {wri X is "'x"; assign x '{rest 'x}} ↵
x is "a b c"
x is "b c"
x is "c"
(c-p)
```

See Chapter 4 for a description of the **rest** command.

Protecting Commands from Errors (protect)

The **protect** command executes commands in a protected region and, optionally, commands specified as cleanup actions. This command is useful if you want to recover reliably from potential errors that may occur in the protected region. You can have cleanup actions execute unconditionally or only when an error occurs; the cleanup actions execute after the main body of commands.

An example of the **protect** command follows:

```
(c-p) assign var "" ↵
(c-p) protect {write 1; if {var} {error E}, else {write 2}; write 3} ' ↵
(c-p)' ,cleanup {write 4} ↵
1
2
3
4
(c-p) assign var test ↵
(c-p) protect {write 1; if {var} {error E}, else {write 2}; write 3}, cleanup {write 4} ↵
1
Error: E
4
(c-p)
```

If you specify the **errors-only** keyword, you can capture any error output in a CP variable. If you are writing a macro (see "Writing Macros" in Chapter 3), you can suppress error messages. In many situations, an error may occur that affects what actions the macro takes.

Following is an example of an error message captured in a CP variable:

```
(c-p) define-macro capture-error {obj} {assi an-error ' ↵
(c-p){' '{protect {eval 'obj}, errors-only }} ↵
(c-p) capture-error .z ↵
(c-p) an-error ↵
Error: '.z' is not a visible command, macro or variable.
(c-p)
```

If you rebind the error stream as above, errors in CP flow control commands will not be written to you in the context of error protection. Since the CP **if** and **while** commands capture and discard the standard output of their predicate phrase to determine whether the predicate is null or nonnull, error output is discarded, but any errors will affect the flow of control in the execution environment.

The following examples show error output being suppressed while an error controls the flow. The examples show what happens in three cases:

- The CP variable *junk* exists and is nonnull.

```
(c-p) assign *junk* stuff ↵
(c-p) protect {write *junk*, no-newline; if {*junk*} { ↵
(c-p){{ write '" is not", no-newline}, else {write '" is", no-newline}; write '" null."} ' ↵
(c-p)' ,clean { write '" does not exist." }, errors-only ↵
 *junk* is not null.
(c-p)
```

- The CP variable *junk* exists and is null.

```
(c-p) assign *junk* ""
(c-p) protect {write *junk*, no-newline; if {*junk*} {
(c-p){{ write " is not", no-newline}, else {write " is", no-newline}; write "  null."} '
(c-p)' ,clean { write " does not exist." }, errors-only
*junk* is null.
(c-p)
```

- The CP variable *junk* does not exist (see Chapter 4 for a description of **delete-command**).

```
(c-p) delete-command *junk*
(c-p) protect {write *junk*, no-newline; if {*junk*} {
(c-p){{ write " is not", no-newline}, else {write " is", no-newline}; write " null."} '
(c-p)' ,clean { write " does not exist." }, errors-only
*junk* does not exist.
(c-p)
```

Comparing Two CP Variables (equal)

The **equal** command determines whether two arguments are equal, and then writes a phrase to the standard output. If the arguments are equal, "true" is written. If the arguments are not equal, a null string ("") is written. **Equal** is useful as a predicate evaluator for the **c-p:if** command.

Comparisons are case insensitive unless **equal**'s **case-sensitive** argument has a "yes" value. Case insensitivity includes considering the hyphen (-) and underscore (_) to be equivalent.

The following example assigns a value to CP variables x and y, and then compares them:

```
(c-p) assi x foo
(c-p) assi y foo
(c-p) if { eq 'x 'y } { write same }
same
(c-p)
```

The next example resets the value of y and compares x and y again:

```
(c-p) assi y bar
(c-p) if { equ 'x 'y } { write equal }
(c-p)
```

The following example demonstrates case insensitivity:

```
(c-p) assi x foo-bar
(c-p) assi y Foo_Bar
(c-p) if { equ 'x 'y } { write yes }
yes
(c-p)
```

Here are two examples that use the **case-sensitive** argument:

```
(c-p) if { equ Foo_Bar foo-bar, cas } { wri yes } { wri no} ↵
no
(c-p)

(c-p) if { equal 'x 'X, cas } { wri Yes'","' indeed. } ↵
Yes, indeed.
(c-p)
```

Note that when a comparison involves the output of commands, case sensitivity applies to the values being output into the command line, not to the names of the commands producing the output. Command names (including CP variables) are always case insensitive.

Negating a Test (not)

The **not** command negates a value and writes the negated value to the standard output. **Not** converts '"" (the null string) into "true" and everything else into the null string. The following example negates a null string:

```
(c-p) not '"" ↵
true
(c-p)
```

The next example negates a nonnull string:

```
(c-p) not '{ not '"" } ↵
(c-p)
```

The following example uses the **not** command with other commands:

```
(c-p) if {not '{equal foo bar}} {write hello} ↵
hello
(c-p)
```

Doing an AND Test (and)

To do an AND test, use the **and** command.

```
(c-p) assi x one ↵
(c-p) assi y two ↵
(c-p) if { and {x} {y} }  { write x and y } ↵
x and y
(c-p)

(c-p) assi x one ↵
(c-p) assi y '"" ↵
(c-p) if { and {x} {y} } { write x and y } ↵
(c-p)
```

Doing an OR Test (or)

To do an OR test, use the **or** command. Two examples follow:

```
(c-p) assi x "" ↵
(c-p) assi y two ↵
(c-p) if { or {x} {y} } { write x or y } ↵
x or y
(c-p)

(c-p) assi x "" ↵
(c-p) assi y "" ↵
(c-p) if { or {x} {y} } { write x or y} ↵
(c-p)
```

You can do an exclusive OR test with the **if** command. Two examples follow:

```
(c-p) assi x "" ↵
(c-p) assi y two ↵
(c-p) if { if {x} {not 'y}; if {y} {not 'x}} {wr x xor y} ↵
x xor y
(c-p)

(c-p) assi x one ↵
(c-p) assi y two ↵
(c-p) if { if {x} {not 'y}; if {y} {not 'x}} {wr x xor y} ↵
(c-p)
```

Manipulating Phrases as Sequences

This section discusses commands that manipulate phrases as sequences: **do-sequence**, **first**, **rest**, **last**, **position**, **subphrase**, and **length**. The tasks you can perform with them are as follows:

- Execute a command repeatedly (**do-sequence**)
- Write the first word of a phrase (**first**)
- Write all but the first word of a phrase (**rest**)
- Write the last word of a phrase (**last**)
- Write the position of an expression in a phrase (**position**)
- Write a subphrase (**subphrase**)
- Write the length of a phrase (**length**)

Executing Commands Repeatedly (do-sequence)

The **do-sequence** command executes a command repeatedly. The command has two required arguments: **name-and-phrase** and **body**. **Do-sequence** executes the body once for each word in the phrase with the specified name bound to the *n*th word on the *n*th iteration. If the phrase is the null string, **do-sequence** does nothing.

The following examples show different uses of **do-sequence**:

```
(c-p) assign list all good boys ↵
(c-p) do-sequence {x list} {write .. 'x .. } ↵
.. list ..
(c-p) do-sequence {x 'list} {write .. 'x .. } ↵
.. all good boys ..
(c-p) do-sequence {x ''list} {write .. 'x .. } ↵
.. all ..
.. good ..
.. boys ..
(c-p)
```

If you want to eliminate the space after the x value, you must enclose x with braces:

```
(c-p) do-sequence {x 'list} {write ..'{x}.. } ↵
..all good boys..
```

The next example shows how to use **do-sequence** to set variables AA through JJ to 1 to 10:

```
(c-p) debug:define-variable j 0 ↵
(c-p) do { x AA BB CC DD EE FF GG HH II JJ } ' ↵
(c-p)' { debug:as j j+1; as 'x '{debug:eval j} } ↵
(c-p) AA ↵
1
(c-p)
```

Writing the First Word of a Phrase (first)

The **first** command writes the first word of a phrase. This is useful in macros (see "Writing Macros" in Chapter 3). If you use the **character** keyword, **first** writes the first character of a phrase.

Following is a simple example:

```
(c-p) first a b c ↵
a
(c-p)
```

This example uses the **character** keyword:

```
(c-p) first abc def, character ↵
a
(c-p)
```

In the next two examples the first word contains spaces:

```
(c-p) first { a b } c d ↵
{ a b }
(c-p)

(c-p) first foo( bar ) baz ↵
foo( bar )
(c-p)
```

The following two examples use **first** with other commands (including **rest**, described below):

```
(c-p) assign x now is the time ↵
(c-p) write "'{first 'x} '{rest 'x} ..." ↵
"now is the time ..."
(c-p)

(c-p) write "'{rest 'x} '{first 'x} ?" ↵
"is the time now ?"
(c-p)
```

Writing the Rest of a Phrase (rest)

The **rest** command writes all but the first word of a phrase. Following is a simple example:

```
(c-p) rest a b c ↵
b c
(c-p)
```

Here are two more examples:

```
(c-p) rest { a b } c d ↵
c d
(c-p)

(c-p) rest foo( bar ) baz ↵
baz
(c-p)
```

Writing the Last Word of a Phrase (last)

The **last** command writes the last word of a phrase. This is useful in macros (see "Writing Macros" in Chapter 3). If you use the **character** keyword, **last** writes the last character of a phrase.

Following is a simple example:

```
(c-p) last a b c ↵
c
(c-p)
```

This example uses the **character** keyword:

```
(c-p) last abc def, character ↵
f
(c-p)
```

In the next example the last word contains spaces:

```
(c-p) last a b { c d } ↵
{ c d }
(c-p)
```

Write the Position of an Expression in a Phrase (position)

The **position** command writes the numeric position (starting with position 0) of the first character in a phrase that matches a specified regular expression. See Chapter 6 for a discussion of regular expressions.

In this example, a CP variable x is assigned a pathname for a file, **my_inventory_file**. The **position** command then returns the numeric position of and number of characters in **my_inventory_file**:

```
(c-p) assign x /somedir/otherdir/my_inventory_file ↵
(c-p) position my_inventory_file 'x ↵
18 17
(c-p)
```

Write a Subphrase (subphrase)

Use the **subphrase** command to write part of a phrase. If you use the **character** keyword, **subphrase** writes the specified number of characters from a phrase.

The following example continues the example from the **position** command. If you just want the filename **my_inventory_file** instead of the entire pathname to be contained in a CP variable (here, file), use the **assign** and **subphrase** commands:

```
(c-p) assign file '{subphrase 18 17 'x, character} ↵
(c-p) file ↵
my_inventory_file
(c-p)
```

Write the Length of a Phrase (length)

The **length** command calculates the size of a phrase (in words, by default), which is useful when you are lining up formatted output. If you use the **character** keyword, **length** writes the length of the phrase in characters.

In the following example, the CP variables var1 and var2 receive values. Then, **length** writes the size (in characters) of the two variables:

```
(c-p) assign var1 987654321 ↵
(c-p) assign var2 32 ↵
(c-p) length 'var1, character; length 'var2, character ↵
9
2
(c-p)
```

End of Chapter

Chapter 3
Customizing Your Environment

This chapter describes how you can customize your environment. After defining terms and concepts, the chapter tells how to do the following tasks:

- Write a macro
- Create and manage realms
- Change an argument's default and implied values
- Create command aliases
- Save your customizations

Terms and Concepts

This section defines several terms that relate to customizing the environment.

Command

A command is a keyword that tells the CP what to do. Commands can occur at the beginning of a line or following a semicolon. The CP recognizes three kinds of entities as commands: built-in commands, macros, and CP variables.

When you execute a command, there is no visible difference among the various types of commands. This regularity lets you concentrate on the task you are doing instead of learning a different syntax for each kind of command. One exception to this regularity is that you cannot write a macro to permanently set the current realm; a macro is executed in the realm in which it is defined and then returns to the realm from which it was invoked. A macro can, however, set the current realm for the remainder of the macro's execution.

Built-in Commands

Built-in commands are part of the standard environment. Normally, for most common tasks you will invoke built-in commands directly. For more complicated tasks you can use the built-in commands as building blocks to create macros.

Macro

A macro is a collection of commands saved as a single unit for later invocation. Macros are especially useful if you have a complex invocation of a series of commands that you use repeatedly.

CP Variable

A CP variable is a Command Processor environment variable that is created by the **c-p:assign** command. When executed, a CP variable displays its value.

Realm

All commands are organized into groups called realms. Realms organize commands similar to the way directories organize files, except that a realm cannot contain another realm. As every file in a file system is in a directory, every command is in a realm. Realms also control command visibility.

Default Value

The default value is the value associated with an argument if you omit the argument in a command line.

Implied Value

The implied value is the value associated with an argument if you specify the argument name but omit the value in a command line.

Standard Output

The standard output is the file to which a command's normal output is written. The standard output is by default the display unit associated with your debugging process.

Error Output

The error output is the file to which a command's error output is written. The error output is by default the display unit associated with your debugging process.

Standard Input

The standard input is the input device currently associated with your debugging process. This is by default the keyboard of your terminal or workstation.

Include File

Normally, command input comes from the keyboard associated with your process. An include file is a file containing commands to be executed by the **include** command. When you execute the **include** command, the standard input is temporarily changed from the keyboard to the include file.

Writing Macros

This section tells how to do the following tasks:

- Create a macro
- Return from a macro
- View a macro definition
- Delete a macro
- Prompt for user input
- Write a message
- Write an error message

Creating a Macro (define-macro)

The **define-macro** command creates a new command with the name and interface you specify. This command has three required arguments:

name This is the macro's name. If you choose the name of an existing command, that command will be overwritten unless it is built in. To overwrite a built-in command, you must explicitly delete it first.

arguments This value (or values), which must be enclosed in braces, specifies the names of the new macro's arguments and whether they are required (the default), optional, or keyword. You can set default and implied values for CP variables and for each macro argument, and you can document each argument. If you omit the argument documentation, the help facility uses a short string from the documentation for the argument's type. You can also define macro variables with this argument.

body The body contains one or more commands enclosed in braces. A macro typically uses backquotes in the body to substitute the value of the specified arguments into the definition.

Define-macro also accepts two keyword arguments: **doc** and **invocation-realm**. The **doc** argument accepts up to three quoted help text strings. The first string is displayed by a help message of short, medium, or long text verbosity. The second is displayed by a help message of medium or long text verbosity. The third is displayed only by a help message of long text verbosity.

If you specify the **invocation-realm** argument, the defined macro will always execute in the realm in which it is invoked. By default, a macro executes in the realm in which it is defined. The following macro usually would not work, as shown below, because the realm changes only for the duration of the macro; this is due to the behavior of CP variables:

```
(debug) define-macro my-change-realm {,optional name} {c-p:realm 'name} ↵
(debug) my-change-realm c-p ↵
(debug)
```

However, using the **invocation-realm** argument enables you to define a macro that has a permanent realm change as a side effect:

```
(debug) define-macro my-change-realm {,optional name} { ↵
(debug){ c-p:realm 'name}, invocation-realm ↵
(debug) my-change-realm c-p ↵
(c-p)
```

With the **arguments** argument, you can specify arguments and local variables; the specifications can be fairly complex or very simple, depending upon the macro. For instance, the macro **write-two-words** accepts two required arguments, **word1** and **word2**; you don't need to specify that the arguments are required since that is the default.

```
(c-p) define-macro write-two-words {word1 word2} {write 'word1; write 'word2} ↵
(c-p) write-two-words Hi there! ↵
Hi
there!
(c-p)
```

The next macro, **write-more-words**, is defined with a required argument, an optional argument, a keyword argument, and a local variable (which must be explicitly defined); it also uses the **doc** keyword:

```
(c-p) define-macro write-more-words {reqword, optional optword, keyword kword, ↵
(c-p){ variable varword} {assign varword words ↵
(c-p){ write 'reqword 'optword 'kword '{eval varword} ↵
(c-p){ }, doc "This macro writes words." ↵
(c-p) write-more-words ↵
Error: No value supplied for the required argument 'reqword' of the 'write-
more-words' command/macro.
(c-p) write-more-words Here are, kword four ↵
Here are four words
(c-p) help write-more-words ↵
                Command: write-more-words        Realm: command-processor
Summary         This macro writes words.

   Arguments    Required:
                  reqword      Any value
                Optional:
                  optword      Any value
                Keyword:
                  kword        Any value

For further help, type "help write-more-words <argument name>"
(c-p)
```

The next example adds types, default and implied values, and more documentation to the write-more-words macro:

```
(c-p) define-macro write-more-words { ↵
(c-p){      {reqword, implied Godzilla, type anything, ↵
(c-p){{      doc "reqword accepts anything"}, ↵
(c-p){ optional ↵
(c-p){      {optword, default Meets, implied Eats, type anything, ↵
(c-p){{      doc "so does optword"}, ↵
(c-p){ keyword ↵
(c-p){      {kword, default The, implied The, type anything, ↵
(c-p){{      doc "ditto kword"}, ↵
(c-p){ variable ↵
(c-p){      {varword, default Blob, ↵
(c-p){{      doc "varword is a local variable to this macro"} ↵
(c-p){ } {write 'reqword 'optword 'kword '{eval varword} ↵
(c-p){ }, doc "This macro still writes words." ↵
(c-p) write-more-words ↵
Error: No value supplied for the required argument 'reqword' of the 'write-
more-words' command/macro.
(c-p) w-m-w King_Kong, optword ↵
King_Kong Eats The Blob
(c-p) w-m-w, reqword ↵
Godzilla Meets The Blob
(c-p) help write-more-words, v ↵
                Command: write-more-words        Realm: command-processor

Summary         This macro still writes words.

   Arguments    Required:
                  reqword     reqword accepts anything
                             Implied: Godzilla
                Optional:
                  optword     so does optword
                             Default: Meets
                             Implied: Eats
                Keyword:
                  kword       ditto kword
                             Default: The
                             Implied: The

For further help, type "help write-more-words <argument name>".
(c-p)
```

This defines an **up** macro for viewing source text:

```
(c-p) def-mac up ' ↵
(c-p)` {,optional {screens, default 1, type ordinal }} ' ↵
(c-p)` {debug:view, up 'screens } ↵
(c-p)
```

The next example defines a **down** macro for viewing source text:

```
(c-p) def-mac down ' ↵
(c-p)` {,optional {screens, default 1, type ordinal }} ' ↵
(c-p)` {debug:view, down 'screens } ↵
(c-p)
```

Since the **down** macro specifies the ordinal type for the screens variable, the CP provides type checking the same as for commands:

```
(c-p) down 0 ↵
Error: '0' is not a valid ordinal expression.
(c-p)
```

Returning from a Macro (return)

The **return** command writes a phrase to the standard output and terminates the execution of a macro. Following is an example of the **return** command:

```
(c-p) define-macro star {x} {return *'{x}*'characters:new-line} ↵
(c-p) star foo ↵
*foo*
(c-p)
```

Viewing a Macro (print-command)

The **print-command** command displays the definition of a macro or a CP variable. It displays a macro's definition as an invocation of the **define-macro** command and a CP variable's definition as an invocation of the **c-p:assign** command. Invoking **print-command** for a built-in command writes a null string to the standard output.

The following example sets the CP variable my_var to 32 with documentation, then displays the definition:

```
(c-p) assi my_var 32, doc "value of my_var" ↵
(c-p) print-com my_var ↵
command-processor:assign command-processor:my_var 32
,doc "value of my_var"
(c-p)
```

This defines and prints the definition of a macro named **say-hello:**

```
(c-p) define-macro say-hello {} { ↵
(c-p){ write hello } ↵
(c-p) print-command say-hello ↵
command-processor:define-macro debugger:say-hello {
  } {

write hello }
(c-p)
```

Print-command lets you save a macro or variable definition to a file if you use the **redirect-output** command (comments in the macro body are retained). You can then include the macro in another session. The names of printed variables or macros are displayed in their fully qualified form (with a realm prefix) to ensure that they will be defined in the same realm later in case you are using **redirect-output.**

For example, to save the macro **say-hello** to a file named **hellofile,** you could use this command:

```
(c-p) redirect-output {print-com say-hello} hellofile ↵
(c-p)
```

Deleting a Macro (delete-command)

The **delete-command** command deletes a command. The command can be any variable, macro, or built-in command. You cannot abbreviate when you specify the command name.

This example deletes the macro named **say-hello:**

```
(c-p) delete-command say-hello ↵
(c-p)
```

Prompting for User Input (query)

The **query** command writes a prompt to the standard output and reads a one-line user response from the standard input, as in this example:

```
(c-p) query How many? ↵
How many? 7 ↵
7
(c-p)
```

Here is **query** in a macro definition:

```
(c-p) define-macro ask { name } { assign x 'name; ↵
(c-p){ assign y '{query Number: }; ↵
(c-p){ write Name = 'x; write Number = 'y } ↵
(c-p) ask Fred Rogers ↵
Number: 12345 ↵
Name = Fred Rogers
Number =12345
(c-p)
```

Writing a Message (write)

The **write** command writes the value of its **text** argument, plus a New Line character, to the standard output. If the value of the **message** argument is "yes," the text is written to the error output instead of the standard output. If the value of the **no-newline** argument is "yes," the New Line is omitted.

To specify characters that are special to the CP (for example, braces, a comma, or a semicolon), use either backquote substitution or characters from the characters realm as specified in Chapter 1.

The following example writes "Hello everybody.":

```
(c-p) wri Hello everybody. ↵
Hello everybody.
(c-p)
```

This example writes "Hello, Mark.":

```
(c-p) write Hello'char:comma Mark. ↵
Hello, Mark.
(c-p)
```

This example does the same thing, but encloses the comma in quotation marks:

```
(c-p) write Hello'","  Mark. ↵
Hello, Mark.
(c-p)
```

This command specifies that both writes will be on the same line with no intervening space:

```
(c-p) evaluate {wr hello, no-new; wr hello} ↵
hellohello
(c-p)
```

If you want to write a message to the error output, type this command:

```
(c-p) write An error has occurred., message ↵
(c-p)
```

Writing Error Messages (error)

The **error** command writes a message to the error output and signals that an error has occurred. If this command occurs in a macro outside a protected region, the CP abandons execution of the macro at that point. If this command occurs inside a protected region, the CP executes the cleanup action for the statement. For information about protected regions, see "Protecting Commands from Errors" in Chapter 2.

Following is an example of the **error** command:

```
(c-p) error Something is wrong. ↵
Error: Something is wrong.
(c-p)
```

Here is an example in a macro; if the argument **arg** is null, you will receive the value of the **error** command's **message** argument:

```
(c-p) define-mac assert-not-null {arg message} { ↵
(c-p){ c-p:if {not 'arg} {error 'message}} ↵
(c-p) assert-not-null "" arg is null ↵
Error: arg is null
(c-p)
```

Creating and Managing Realms

All commands are organized into groups called realms. Realms control command visibility. If you use a specific group of commands often, you may want to create a realm that includes just those commands. This section tells how to do the following tasks:

- Display and set the name of the current realm
- Create a realm
- Display or set a realm's realm use list
- Display or set a realm's prompt string
- Delete a realm

Displaying and Setting the Current Realm (realm)

To display the name of the current realm, use the **realm** command with no arguments:

```
(c-p) realm ↵
command-processor
(c-p)
```

To set the current realm, use the **realm** command with an argument value:

```
(debug) rea c-p ↵
(c-p) rea ↵
command-processor
(c-p)
```

Creating a Realm (define-realm)

To create a realm, use the **define-realm** command. For example, this creates a realm named **macros**:

```
(debug) def-rea macros ↵
```

When a realm is created, it contains no commands. You can put commands into the realm using **copy-command** for built-in commands (see "Creating Command Aliases" later in this chapter), **define-macro** for macros (see "Creating a Macro" earlier in this chapter), or **c-p:assign** for CP variables (see Chapter 4).

Although a newly created realm contains no commands itself, many commands are immediately available in that realm through its realm use list; see the next section for more information.

Displaying and Setting the Realm Use List (realm-use-list)

The **realm-use-list** command displays or sets a realm use list. The **realm** keyword argument indicates the target realm whose use list is being displayed or set. The default is the current realm.

Without arguments, **realm-use-list** displays the realms that are used by the target realm. With a **used-realms** argument, **realm-use-list** replaces the use list of the target realm.

When you create a realm using the **define-realm** command, you can supply an explicit realm use list via the **use** argument. The default use list contains the new realm and the command-processor. For example:

```
(c-p) def-realm macros ↵
(c-p) realm macros ↵
(macros) realm-use-list ↵
{ { macros } { command-processor } }
```

Which realms are in the target realm use list affect how you can abbreviate command names, because each abbreviated command name must be unique among the commands in the realms on the current realm use list.

This command displays the current realm use list:

```
(c-p) realm-use-list ↵
{{command-processor foo}}
```

To set the realm use list for the **foo** realm so that the CP first looks in realms **foo** and **bar** to find commands, macros, and CP variables, and then looks in the **c-p** realm, type this command:

```
(c-p) realm-use-list {{foo bar} {c-p}}, realm foo ↵
(c-p)
```

Displaying and Setting the Prompt String (prompt-string)

The **prompt-string** command displays or sets the prompt string of the current realm.

To display the prompt, omit the **new-prompt** argument. To set the prompt string for a realm, specify a value for the **new-prompt** argument. The CP automatically adds a space after the phrase that you specify for the prompt.

This example displays the prompt string:

```
(c-p) prompt ↵
(c-p)
(c-p)
```

The next example sets the prompt string of the **macros** realm to (mac):

```
(c-p) rea macros ↵
(macros) prom (mac) ↵
(mac)
```

Deleting a Realm (delete-realm)

The **delete-realm** command deletes a user-created realm. The character, command-processor, graphical-interface, icobol, options, and debugger realms cannot be deleted.

To delete a realm, you must type the complete realm name; it cannot be abbreviated. This command deletes a realm named macros:

```
(debug) del-rea macros ↵
```

Changing an Argument's Default Value (change-argument-value)

The **change-argument-value** command sets the default or implied value for a command's argument. You can change a default or implied value, remove an existing one, or create one where none existed.

Change-argument-value has two required arguments: **command** and **argument**. These arguments specify which command argument is being changed to what value. **Change-argument-value** also accepts several keyword arguments. Table 3-1 shows tasks you can accomplish using these keyword arguments.

Table 3-1 Tasks and Keywords for change-argument-value

Task	Keyword
Set the default value to a value	default
Set the default value to the null string	empty-default
Set the implied value to a value	implied
Set the implied value to the null string	empty-implied
Take away an argument's implied behavior	no-implied

The following example changes the default value of **help**'s **command** argument to "realm":

```
(c-p) change-argument-value help command, default realm ↵
```

To reverse the default and implied values for the **instructions** argument of the **step** command, type this command:

```
(c-p) change-arg-value debug:step instruct, default yes ' ↵
(c-p)` ,implied no ↵
```

Creating Command Aliases (copy-command)

The **copy-command** command copies a command. You can use **copy-command** to make a copy of a built-in command, macro, or variable that has exactly the same interface and semantics as the original.

Changing a copy does not affect the original command, and assignment to a copied variable does not affect the original variable. Resetting the default or implied values of a copied command or macro does not affect the original command or macro.

A copy of a built-in command does not have the same permanence as the built-in command. The copy can be overwritten by a **define-macro** or **cp:assign** command.

This example copies variable var1 (which already exists) to var2:

```
(c-p) copy var1 var2 ↵
```

You could create an **exit** command identical to the **bye** command:

```
(c-p) copy-command bye exit ↵
```

Saving Your Customizations

This section tells how to save your customized environment. It explains how to write macros to a file and how to include such a file into a debugging session later.

Writing to a File (redirect-output)

The **redirect-output** command sends output to a file. You can redirect the standard output and the error output independently. You can also independently control whether the output appends to or overwrites existing data.

This command writes the help message for the **assign** command to a file called **help.messages**, deleting that file if it already exists:

```
(c-p) redirect-output {help assign} help.messages ↵
```

This command appends the help message for the **evaluate** command to **help.messages**; it uses the **standard-append** keyword:

```
(c-p) redir-o {help eval} help.messages, standard-append ↵
```

To save all of your macros to the file **savefile**, use this command:

```
(c-p) redir-o {do-seq {x ''{help, c ".", v {text no, arg no}}} {print-command 'x}} savefile ↵
```

Including a File (include)

The **include** command reads and executes the contents of a specified file.

Type the following to include a file named **savefile**:

```
(c-p) include savefile ↵
```

The following command includes a file named **crowd** and keeps going if errors are encountered:

```
(c-p) include crowd, continue ↵
```

End of Chapter

Chapter 4
Command Processor Commands

This chapter contains the on-line help messages for the command-processor (c-p) realm and for the commands in that realm. The realm help message is first, followed by the help messages for the individual commands, listed in alphabetical order. The c-p realm contains general commands that perform tasks such as getting help, manipulating the debugging environment, or controlling the flow of macros.

Help messages for commands use the following conventions:

Message format	Each command help message in this chapter has the following sections: "Summary," "Description," "Arguments" (if the command takes arguments), "Examples," and "See Also."
Command syntax	Each command follows the regular syntax described under "Creating a Command Line" in Chapter 1.
Arguments	Each argument is classified as required, optional, or keyword in the Arguments subsection of the Summary Section.
Argument keywords	Each argument, regardless of its classification, has a keyword identifying it.
Argument values	The kind of value the argument accepts is listed to the right of the keyword.
Argument semantics	This information and occasionally additional syntactic information is given under an entry's Arguments section.
<name>	The "To get" and "To do" subsections of the c-p realm help message use angle brackets to indicate a value that you supply.

Each command's help message is divided into two sections: the first part shows what you would see if you typed **help** <command-name>; the first and second parts together show what you would see if you typed **help** <command-name> **,verbosity {text long, arguments long}**.

Realm: command-processor

Summary

Introduction to the Command Processor (CP)

Here is how to perform some common tasks:

To get

A list of CP help topics:	**help, topic, r c-p**
A list of CP commands:	**help, command, r c-p**
Help on a specific topic:	**help** *<topic-name>*
Help on a specific command:	**help** *<command-name>*
More information about the CP, with CP commands categorized:	**help, v, r c-p**

To do

Insert input from a file:	**include** *<file>*
Redirect command output:	**redirect** *<commands>* *<stdout>*
Set a CP variable:	**assign** *<var-name>* *<value>*
Display CP variable's value:	*<var-name>*
Delete a CP variable:	**delete-command** *<var-name>*
Create a command alias:	**copy-command** *<old>* *<new>*
Exit from an interactive tool:	**quit**

Description

The Command Processor (CP) is the command interpreter. The CP lets you dynamically create CP variables and tailor your working environment by creating commands (macros), organizing commands into groups (realms), and modifying commands (resetting the default and implied values of arguments).

Here are some more tasks you can perform:

To get

A list of all help topics:	**help, topic, realm**
A list of all commands:	**help, command, realm**
A list of all realms:	**help, realm**
Help on a specific realm:	**help** *<realm-name>*

To do

TASK	CATEGORY
Manipulate the CP environment	Environment
Control the flow of CP commands	Flow
Display or create a help message	Help
Control command input and output	I/O
Manipulate realms	Realms
Manipulate phrases	Phrases
Perform another task	Misc.

Following is a summary of CP commands by category; capital letters indicate the shortest unique abbreviation:

Environment

ASsign*	Assign a value to a CP variable
CHange-argument-value	Change default or implied value
COpy-command	Copy a command or a variable
DEFine-Macro	Create a macro
DELete-Command	Delete a command or variable
Evaluate*	Evaluate a series of commands

Environment	EXpression	Evaluate an integer expression
	LET	Evaluate commands in a dynamic binding environment
	PRInt-command	Display a macro's definition
	PROMpt-string	Return or change the prompt string
	Trace-Commands	Trace execution of commands, variables, and macros
	Trace-Status	Display the status of traced objects
	Untrace-Commands	Stop tracing commands, variables, and macros
Flow	ANd	Test for logical AND
	DO-Sequence	Execute a series of commands
	EQual	Compare whether arguments are equal
	ERror	Signal an error in a macro
	Greater	Compare strings
	Greater-Equal	Compare strings
	IF*	Execute commands conditionally
	LESs	Compare strings
	LESs-Equal	Compare strings
	Not	Test for a null string
	Not-Equal	Compares whether arguments are unequal
	OR	Test for logical OR
	PROTect	Execute commands in protected region
	RETurn	Return from a macro
	WHile*	Execute while predicate nonnull
Help	DEFine-Topic	Create a topic help message
	DELete-Topic	Delete a topic
	Help	Display a help message
I/O	INclude	Read the contents of a file
	QUEry	Display a prompt and read user input
	REDirect-output	Make a file the default output
	WRite	Write arguments to standard output
Realms	DEFine-Realm	Create a new realm
	DELete-Realm	Delete a realm
	Realm	Display or set the current realm
	Realm-Use-list	Display or set the realm use list
Phrases	FIRst	Return the first word in a phrase
	LAst	Return the last word in a phrase
	LENgth	Return the length of a phrase
	RESt	Return all but first word of phrase
	SUbphrase	Write part of a phrase
Misc.	Bye	Exit from interactive tool
	DIrectory	Display or set working directory
	LOg	Start logging
	OPtion-status	Display or set global options
	PAge	Page through command output
	POsition	Return the position of a regular expression
	QUIt	Exit from interactive tool
	SHell	Execute a subshell or a shell command
	Unlog	Turn off logging

* A command with the same name but different action exists in the debugger (Mxdb) realm.

Command: and — Realm: command-processor

Summary

Write a phrase representing the logical AND of the arguments

Arguments

Required:

left	A string
right	A string

Examples

```
c-p:and `{some-variable} `{some-other-variable}
c-p:and `{sh cmp foo bar} `{sh cmp bar bletch}
```

Description

And writes "true" when both arguments are nonnull strings. Otherwise, it writes an empty string.

Arguments

left	A string
right	A string

Examples

This **and** command composes the OR operation:

```
(c-p) not {and {not '{a1}} {not '{as}}} ↵
```

See Also

Commands: **or, not, if**

Command: assign Realm: command-processor

Summary

Assign a value to a CP variable

Arguments

Required:

variable	A word
phrase	One or more words

Keyword:

doc	Up to three quoted help text strings

Examples

```
c-p:as x computer
c-p:assi s some words for "s"
c-p:assign jar box
c-p:assign jar `box, d "jar has box's value."
```

Description

Assign assigns a phrase to a CP variable (a function without arguments). If the variable does not exist, **assign** creates it. If a variable with the specified name already exists, **assign** overwrites it.

Arguments

variable — You must spell the variable name exactly. This is necessary to let you create a new variable that is a prefix of some other name. You can qualify the variable to a particular realm by preceding the variable name with a realm name and a colon.

phrase — For more information about words and phrases, see the syntax help topic.

doc — This text will be visible to the help command (see the documentation help topic).

Examples

To assign the word "computer" to x:

```
(c-p) as x computer ↵
```

To assign the phrase 'some words for "s"' to s:

```
(c-p) assi s some words for "s" ↵
```

To assign the word "box" to jar and display the value:

```
(c-p) assign jar box ↵
(c-p) write The value of jar is: 'jar ↵
The value of jar is: box
```

To assign a value to a CP variable named "box," and then assign the value of box to jar, document the jar variable, and display the value:

```
(c-p) assign box strawberries ↵
(c-p) assign jar 'box, doc "jar has box's value." " ↵
(c-p)" Description<tab>The variables are as follows: ↵
(c-p)" <tab><tab>box: the original variable ↵
(c-p)" <tab><tab>jar: the copied variable" ↵
(c-p) write jar = 'jar ↵
jar =strawberries
(c-p) evaluate { jar } ↵
strawberries
(c-p) help jar ↵
            Command: jar       Realm: command-processor

Summary     jar has box's value.

   Arguments <none>

Description The variables are as follows:
            box: the original variable
            jar: the copied variable
(c-p)
```

See Also

Commands: **c-p:evaluate, print-command, debug:assign**
Topics: substitution, syntax

Command: bye — Realm: command-processor

Summary Exit from an interactive tool

Examples bye

Description **Bye** exits from the current interactive tool (such as Mxdb).

Examples

```
(debug) bye ↵
```

```
(c-p) bye ↵
```

See Also Commands: **quit, terminate**

Note The **bye** and **quit** commands do exactly the same thing.

Command: change-argument-value Realm: c-p

Summary

Reset the default or implied value for a command argument

Arguments

Required:

command	Command name
argument	Argument name

Keyword:

default	New default value
empty-default	yes or no
implied	New implied value
empty-implied	yes or no
no-implied	yes or no

Examples

```
cha help command, default prompt-string
change-arg-value debug:step instruct, default yes
```

Description

Change-argument-value overrides the default and/or implied values for a command's argument. You can also give an implied value to an argument that did not have one, or take away its implied behavior.

Arguments

command	This name can be abbreviated.
argument	This name can be abbreviated.
default	Set the default value.
empty-default	Set the default value to the null string.
implied	Set the implied value.
empty-implied	Set the implied value to the null string.
no-implied	Take away an argument's implied behavior.

Examples

To set "**help**" equal to "**help, command realm**":

(c-p) **cha help command, default realm** ↵

To change the default value for the **step** command's **instructions** argument:

(debug) **change-arg-value step instruct, default yes** ↵

See Also

Commands: **copy-command, define-macro, print-command**

Command: copy-command Realm: command-processor

Summary

Copy a command, macro, or variable

Arguments

Required:

old-prefix	The name of a command, macro, or variable
new-name	The name of the new command, macro, or variable

Keyword:

doc	Up to three quoted strings of help message text

Examples

```
copy var1 var2
copy-command bye exit
```

Description

Copy-command makes a new command, macro, or CP variable that has exactly the same interface and semantics as the old one. Assignment to a copied variable does not affect the original variable. Changing the default or implied values of a copied command or macro does not affect the original command or macro.

Arguments

old-prefix — This name is case insensitive and can be abbreviated.

new-name — This name is case insensitive.

doc — This text is visible to the **help** command (see the documentation help topic).

Examples

To copy variable var1 to var2:

(debug) **copy var1 var2** ↵

To create an **exit** command identical to the **bye** command:

(debug) **copy-command bye exit** ↵

See Also

Commands: **define-realm, realm**

Command: define-macro Realm: command-processor

Summary

Create a new command

Arguments

Required:

name	The name of the macro being defined
arguments	An argument list enclosed in braces; each argument name can also have with it the following keywords: **optional** or **keyword**; **default**; **implied**; **type**; and **variable** (to create a CP variable)
body	One or more commands enclosed in braces

Keyword:

doc	Up to three strings of help message text enclosed in quotation marks
invocation-realm	Yes or no Default: no Implied: yes

Examples

```
def-mac bang {phrase} { write ! `phrase ! }
def-mac up {,optional { screens, default 1, type ordinal }
} { view, up `screens }
```

Description

Define-macro creates a new "command" with a name and interface you specify. The interface ranges from simple to complex.

A macro typically uses backquotes in the body to substitute the value of the specified arguments into the definition.

Arguments

name	This is a word (see the syntax help topic). You can qualify a macro to a particular realm by preceding the macro name with a realm name and a colon.
arguments	You can specify **required, optional,** and **keyword** arguments. The **optional** and **keyword** arguments may have default and implied values.
body	These commands compose the macro body.
doc	This text will be visible to the **help** command (see the documentation help topic).
invocation-realm	If you specify this argument, the defined macro will always execute in the realm in which it is invoked. By default, macros execute in the realm in which they are defined.

To define an up macro for viewing source text:

```
(debug) def-mac up { ↵
(debug){, optional { screens, default 1, type ordinal } ↵
(debug){ } { view, up 'screens } ↵
```

To define a down macro for viewing source text:

```
(debug) def-mac down { ↵
(debug){, optional { screens, default 1, type ordinal } ↵
(debug){ } { view, down 'screens } ↵
```

To define a frame macro for positioning to a frame:

```
(c-p) define-mac debug:frame {, optional level} { ↵
(c-p){ c-p:if {level} {position, f 'level} {position}}' ↵
(c-p)`, d "Display or set the current frame position." ↵
```

See Also

Commands: **define-realm, delete-command, help, realm, redirect-output**
Topics: substitution, syntax

Command: define-realm Realm: command-processor

Summary

Create a new realm

Arguments

Required:

name	The name for the new realm

Optional:

use	A list of realms grouped using braces

Keyword:

prompt	The prompt string for this realm
doc	Up to three strings of help message text enclosed in quotation marks

Examples

```
define-realm quick
def-realm macros, prompt (mac)
def-r myrealm, use {{myrealm c-p}}
```

Description

Define-realm creates a new realm. If a user-defined realm with that name already exists, **define-realm** overwrites it. To delete a realm explicitly, use the **delete-realm** command. The built-in realms (debugger, command-processor, characters, etc.) cannot be overwritten or deleted.

By default the realm use list of the newly created realm is { { new-realm command-processor } }. To override this, specify the **use** argument.

When you create a realm, it contains no commands, macros, or CP variables. You can create these with the **copy-command**, **define-macro**, and **cp:assign** commands, respectively.

Arguments

name	This is a word (see the syntax help topic).
use	Define the realm use list for the new realm. This determines the uniqueness of command abbreviations. See the **realm-use-list** command.
prompt	Set the new realm's prompt string to the specified value. The default is the name of the new realm in parentheses.
doc	This text will be visible to the **help** command (see the documentation help topic).

Examples

To create a realm named quick:

```
(debug) define-realm quick ↵
```

To create and document a realm named macros (<tab> indicates a tab character):

```
(debug) define-realm macros, prompt (mac) ' ↵
(debug) `, doc "This realm contains my macros." ' ↵
(debug) ` "These macros are defined automatically ↵
(debug)" <tab><tab>in my .mxdb_init file." ↵
```

See Also

Commands: **c-p:assign, copy-command, define-macro, delete-realm, include, realm, realm-use-list, redirect-output**
Topic: syntax

Command: define-topic Realm: command-processor

Summary

Create a new topic

Arguments

Required:

name	The name of the new topic
text	Up to three quoted strings of help message text

Examples

```
def-t quotes "The quotation marks are ´ and ""."
define-topic i/o "Input and output commands" `
"<tab><tab>Following are the I/O commands:
<tab><tab>include query redirect write"
```

Description

Define-topic creates a new topic accessible by the **help** command.

Arguments

name	This can be any word (see the syntax help topic). You can qualify a topic to a particular realm by preceding the topic name with a realm name and a colon.
text	See the help message for the documentation topic.

Examples

To create a simple quote topic:

```
(c-p) def-t quotes "The quote marks are ' and ""." ↵
```

To create a more complex quote topic:

```
(c-p) def-t quotes "The quote marks are ' and ""." ‘ ↵
(c-p)` "Examples<tab>write 'quoted stuff' ↵
(c-p)" <tab><tab>write ""another example""" ‘ ↵
(c-p)` "Description<tab>The CP recognizes two kinds of ↵
(c-p)" <tab><tab>quote marks: single and double." ↵
(c-p)
```

In the following example, <tab> represents the tab character:

```
(debug) define-topic c-p:i/o "Input and output commands" ‘ ↵
(debug)` "<tab><tab>Following are the I/O commands: ↵
(debug)" <tab><tab>include query redirect write" ↵
```

See Also

Commands: **define-realm, delete-topic, help**
Topics: documentation, syntax

Command: delete-command — Realm: c-p

Summary

Remove a command, macro, or variable

Arguments

Required:

name	The name of a command, macro, or variable

Examples

```
del-c var1
delete-command create-realm
```

Description

Delete-command removes a specified command, macro, or CP variable.

Arguments

name	This word cannot be abbreviated. You can qualify the name to a particular realm by preceding the name with a realm name and a colon.

Examples

To delete a variable named var1:

(debug) **del-c var1** ↵

To delete a macro named bang:

(debug) **delete-command bang** ↵

See Also

Commands: **c-p:assign, copy-command, define-macro**
Topic: abbreviation

Command: delete-realm Realm: command-processor

Summary

Delete a realm

Arguments

Required:

name	The name of a user-created realm

Examples

```
del-r macros
delete-realm myrealm
```

Description

Delete-realm deletes a user-created realm.

Arguments

name	This is the name of any user-created realm. You cannot abbreviate the name.

Examples

To delete a realm named macros:

(debug) **del-r macros** ↵

To delete a realm named myrealm:

(debug) **delete-realm myrealm** ↵

See Also

Commands: **define-realm, realm, realm-use-list**

Command: delete-topic Realm: command-processor

Summary

Remove a topic

Arguments

Required:

name	The name of the topic to be deleted

Examples

```
del-t quotes
delete-topic i/o
del-top debug:i/o
```

Description

Delete-topic removes a specified help topic. You can qualify a topic to a particular realm by preceding the topic name with a realm name and a colon.

Arguments

name	This word cannot be abbreviated.

Examples

To delete a topic named quotes:

(debug) **del-t quotes** ↵

To delete a topic named i/o:

(debug) **delete-topic i/o** ↵

To delete i/o from the debugger realm:

(c-p) **del-top debug:i/o** ↵

See Also

Commands: **define-topic, delete-command, delete-realm, help**

Command: directory Realm: command-processor

Summary

Set or display the current working directory

Arguments

Optional:

pathname Any syntactically valid file system pathname

Examples

```
dir
dir /tmp
```

Description

To display the current working directory, use the **directory** command with no arguments.

To set the current working directory, specify a **pathname** argument.

Arguments

pathname Any syntactically valid file system pathname

Examples

This **directory** command displays the current working directory:

```
(c-p) directory ↵
/usr/chris
(c-p)
```

The next example sets the current working directory to **/tmp**:

```
(c-p) directory /tmp↵
(c-p) directory ↵
/tmp
(c-p)
```

See Also

Commands: **debug:directory-list**

086-000167 updates
093-000706-00

This page intentionally left blank.

Command: do-sequence Realm: command-processor

Summary

Execute a command repeatedly

Arguments

Required:

name-and-phrase	A variable name and an associated phrase
body	Command(s) enclosed in braces

Examples

```
do-sequence {x list} { write ..`{x}.. }
do-sequence {x `list} { write ..`{x}.. }
do-sequence {x ``list} { write ..`{x}.. }
```

Description

Do-sequence executes the body once for each word in the phrase with the specified name bound to the *n*th word on the *n*th iteration. If the phrase is the null string (""), **do-sequence** does nothing.

Arguments

name-and-phrase — The name (word) must be separated from the phrase by one or more blanks (spaces or tabs). The variable-phrase pair must be enclosed in braces.

body — If you specify more than one command, the **body** argument must be enclosed in braces. Otherwise, braces are optional.

Examples

The following examples show different uses of **do-sequence**:

```
(c-p) assign list all good boys ↵
(c-p) do-sequence {x list} {write .. 'x .. } ↵
.. list ..
(c-p) do-sequence {x 'list} {write .. 'x .. } ↵
.. all good boys ..
(c-p) do-sequence {x ''list} {write .. 'x .. } ↵
.. all ..
.. good ..
.. boys ..
(c-p)
```

See Also

Commands: **assign**, **write**
Topic: substitution, syntax

086-000167 updates
093-000706-00

Command: equal Realm: command-processor

Summary

Compare whether two arguments are equal

Arguments

Required:

left	Any value
right	Any value

Keyword:

case-sensitive	yes or no

Examples

```
c-p:if { eq foo FOO } { write same }
c-p:if { equ foo bar } { write equal }
c-p:if { equ Foo_Bar foo-bar } { write yes }
c-p:if { equ Foo_Bar foo-bar, case-sens } { wri y }
c-p:if { equal `X bar} { write X equals bar. }
```

Description

Equal compares two arguments, writing "true" to the standard output if they are equal and a null string ('"") if they are not. **Equal** is useful as a predicate evaluator for the **cp:if** command.

Arguments

left — This is a word (see the syntax help topic).

right — This is a word (see the syntax help topic).

case-sensitive — If the value of the **case-sensitive** argument is "yes," equal checks for an exact match.

Examples

The following are valid:

```
(c-p) if { eq foo FOO } { write same } ↵
same
(c-p) if { equ foo bar} { write equal } ↵
(c-p) if { equ Foo_Bar foo-bar } { write yes } ↵
yes
(c-p) if { equ Foo_Bar foo-bar, cas } { wri y } ↵
(c-p) assign X foo ↵
(c-p) if { equal 'X bar } { wri X equals bar. } ↵
```

See Also

Commands: **cp:if, not**
Topic: syntax

Command: error Realm: command-processor

Summary

Display a message and signal an error

Arguments

Required:

message The text of the error message

Examples

```
error Something is wrong.
define-mac assert-not-null {arg message} {
c-p:if {not `arg} {error `message}}
```

Description

Error writes a message to the error output and signals that an error has occurred. This is a useful command for macros.

Arguments

message This will be displayed immediately after "Error: ".

Examples

Following are two examples:

```
(debug) error Something is wrong. ↵
Error: Something is wrong.
(debug)
```

```
(c-p) define-mac assert-not-null {arg message} { ↵
(c-p){ c-p:if {not 'arg} {error 'message}} ↵
(c-p) assign x something ↵
(c-p) assert-not-null 'x X is null ↵
(c-p) assign y '"" ↵
(c-p) assert-not-null 'y Y is null ↵
Error: Y is null
```

See Also

Command: **define-macro, protect, redirect-output**

Command: evaluate Realm: command-processor

Summary

Evaluate a series of commands

Arguments

Required:

commands Command(s) enclosed in braces

Examples

```
c-p:assign x help `",",realm
c-p:evaluate { ``x }
define-mac set {x} {c-p:eval {c-p:assign `x}}
```

Description

The **evaluate** command evaluates a series of commands and writes to the standard output the value returned by the last command evaluated. If the commands are enclosed in braces ({}), the braces are removed before evaluation.

Arguments

commands If you specify more than one command, you must separate them with semicolons or New Lines and enclose the entire series in braces.

Examples

To set x to a command string, and then execute the command in x:

```
(c-p) assign x help '",",realm ↵
(c-p) evaluate { ''x } ↵
```

To define a macro using **evaluate**:

```
(c-p) define-mac set {x} {c-p:eval {c-p:assign 'x}} ↵
```

See Also

Commands: **cp:assign**, **define-macro**, **debugger:evaluate**
Topic: substitution

Command: expression Realm: command-processor

Summary

Evaluate an integer expression and display the result

Arguments

Required:

expr	An integer

Keywords:

mode	A display format: decimal, octal, or hexadecimal Default: decimal Implied: hexadecimal
boolean	Yes or no Default: no Implied: yes

Examples

```
expression (3 * (4 ^ 3) - 1)
expr -(1), mode hex
expr ((1 + 0)), bool
```

Description

The **expression** command evaluates an integer expression and displays the result in octal, decimal, or hexadecimal format. Optionally, the result can be displayed as a CP-style boolean.

Arguments

expr	An integer expression
mode	A display format: decimal, octal, or hexadecimal. The default value is decimal and the implied value is hexadecimal.
boolean	Specify this argument if you want to display the integer expression as a CP-style boolean value.

Examples

To evaluate an integer expression and display the result in decimal format:

```
(c-p) expression (3 * (4 ^ 3) - 1) ↵
191
(c-p)
```

To evaluate an integer expression and display the result in hexadecimal format:

```
(c-p) expr -(1), mode hex ↵
ffffffff
(c-p)
```

To evaluate an integer expression and display the result as a CP-style boolean:

```
(c-p) expr ((1 + 0)), bool ↵
true
(c-p)
```

See Also

Commands: **debugger:evaluate**

Command: first — Realm: command-processor

Summary

Write the first word of a phrase

Arguments

Required:

phrase	One or more words

Keyword

character	yes or no

Examples

```
first a b c
first foo( bar ) baz
write "`{first `x} `{rest ´x}"
```

Description

First writes the first word or character of a phrase. This is useful in macros.

Arguments

phrase For more information about phrases, see the syntax help topic.

character Write the first character instead of the first word if the value is yes

Examples

Following are valid examples:

```
(c-p) first a b c ↵
a
(c-p) first { a b } c d ↵
{ a b }
(c-p) first foo( bar ) baz ↵
foo( bar )
(c-p) assign x now is the time ↵
(c-p) write "‘{first ‘x} ‘{rest ‘x} ..." ↵
"now is the time ..."
(c-p) write "‘{rest ‘x} ‘{first ‘x} ?.." ↵
"is the time now ?.."
```

See Also

Commands: **c-p:evaluate, last, rest**
Topic: syntax

086-000167 updates
093-000706-00

Command: greater Realm: command-processor

Summary

Determine if the left operand is greater than the right operand

Arguments

Required:

left	Left string
right	Right string

Keyword:

case-sensitive	yes or no

Examples

```
greater `{some variable} `{some-other-variable}
```

Description

Greater writes a nonnull string when the left operand is greater than the right operand when both are considered as strings.

Arguments

left	Left operand
right	Right operand
case-sensitive	Take the case of the operands into account for the comparison.

Examples

The following is valid:

```
(c-p) greater '{some-variable} '{some-other-variable} ↵
```

See Also

Commands: **equal, greater-equal, less, less-equal, not-equal**

Command: greater-equal Realm: command-processor

Summary

Determine if the left operand is greater than or equal to the right operand

Arguments

Required:

left	Left string
right	Right string

Keyword:

case-sensitive	yes or no

Examples

```
greater-equal `{some variable} `{some-other-variable}
```

Description

Greater-equal writes a nonnull string when the left operand is greater than or equal to the right operand when both are considered as strings.

Arguments

left	Left operand
right	Right operand
case-sensitive	Take the case of the operands into account for the comparison.

Examples

The following is valid:

```
(c-p) greater-equal '{some-variable} '{some-other-variable} ↵
```

See Also

Commands: **equal, greater, less, less-equal, not-equal**

Command: help — Realm: command-processor

Summary

Display information about a command, realm, or topic

Arguments

Optional:

item	A command name, realm name, type, or topic
argument	The name of a command argument

Keyword:

realm	A realm name or a realm use list
command	The name of a command
type	The name of a type
topic	The name of a topic
verbosity	{ text level, arguments level } level is none, short, medium, or long

Examples

```
help
help breakpoint
help find, v
```

Description

Help displays information about a command, realm, type, or topic. By default, **help** looks first for a command whose name or abbreviation is the value you specify. If no such command exists, **help** looks for a realm, then for a type, and then for a topic. Information on the first item found is displayed.

The help message for each item has the following sections: "Summary," "Description," "See Also," and, optionally, "Notes." Help messages for commands also have "Arguments" and "Examples" subsections and sections.

The "Arguments" subsection of the "Summary" section in on-line help messages for commands is generated dynamically from the current command interface. If you change default or implied values with the **change-argument-value** command, the information for those changed values will differ between the "Arguments" subsection under "Summary" and the "Arguments" section later in the help message.

To search for help on an item whose name matches a regular expression, enclose the regular expression in double quotation marks. Such an expression can be the value for an item, argument, command, realm, type, or topic argument.

Arguments

item — A command name, macro name, or topic may be preceded by a realm name and a colon. You can abbreviate the name of the item.

argument — If you specify this argument name, you must specify a command name as the value for the item.

realm — If you specify a realm name, look for help on only a realm with that name. If you specify a realm or realm use list and a command or topic with no value, list all commands or topics in the specified realm(s). If you specify a realm or realm use list and a command or topic with a value, look for that command or topic in the specified realm(s). The initial implied value is to list all the realms. Type **help, topic realm** for more information about realms.

command — If you specify a value, look for help on only a command with that name. If you also specify a realm, look for help on the command only in the specified realm. The initial implied value is all the commands in the current realm (or in the specified realm if the **realm** argument has a value).

type — If you specify a value, look for help on only a type with that name. If you also specify a realm, look for help on the type only in the specified realm. The initial implied value is to list all the types in the current realm (or in the specified realm if the **realm** argument has a value). Type **help, topic type** for more information about types.

topic — If you specify a value, look for help on only a topic with that name. If you also specify a realm, look for help on the topic only in the specified realm. The initial implied value is to list all the topics in the current realm (or in the specified realm if the **realm** argument has a value).

verbosity — This argument controls the amount of text and arguments information that **help** displays. The default and implied levels are initially medium and long, respectively. The levels have the following meanings:

- none — Omit the specified category (text or arguments). If both levels are none, display only command names.
- short — For text, display only the one-line summary. For arguments, display a one-line list of arguments.
- medium — For text, display the one-line summary and the Examples subsection of the Summary Section. For arguments, display each argument name on a separate line with a brief description of the value the argument accepts.
- long — For text, display all the help text available. For arguments, display the medium-level information plus the default and implied values for each argument.

Examples

To get help on the **breakpoint** command:

(debug) **help breakpoint** ↵

To see a verbose help message for the **find** command:

(debug) **help find, v** ↵

To get help on the **scope** argument of the **breakpoint** command:

(debug) **help breakpoint scope** ↵

To get help on the CP **if** command from the debugger realm:

(debug) **help c-p:if** ↵

To get a list of all commands in the current realm:

(debug) **help, com** ↵

To get a list of all realms:

(debug) **help, realm** ↵

To get a list of all topics in the current realm:

(debug) **help, topic** ↵

To get a list of all Command Processor commands:

(debug) **help, com, rea c-p** ↵

To get help on the debugger realm:

(debug) **help, realm debugger** ↵

To get a help on the c-builtin-types topic:

(debug) **help, topic c-builtin-types** ↵

To get only complete arguments information about the **find** command:

(debug) **help find, v { text none, arg long }** ↵

To get a list of all commands in the current realm with a one-line summary of each:

```
(debug) help, com, ver { text short, arg no } ↵
```

To get help on all commands in the current realm that contain the string "event" in their name:

```
(debug) help, command "event" ↵
```

See Also

Commands: **c-p:assign, change-argument-value, define-macro, define-topic, delete-topic, resume-prompting**
Topics: documentation, realm, regular-expression, type

Command: if — Realm: command-processor

Summary

Conditionally execute one or more commands

Arguments

Required:

predicate	One or more commands enclosed in braces
then-part	One or more commands enclosed in braces

Optional:

else-part	One or more commands enclosed in braces

Examples

```
c-p:assign abc xyz
c-p:if { abc } { c-p:assign x `abc }
if {first `x} {write x not empty} {write x empty!}
```

Description

If evaluates the predicate. If it returns a nonnull phrase, then **if** evaluates the then-part value; otherwise it evaluates the else-part value.

Arguments

predicate These commands return an empty or nonempty phrase.

then-part If the phrase is nonempty, these commands are executed.

else-part If the phrase is empty, these commands are executed.

Examples

This example evaluates the variable abc and sets the CP variable x to the value of abc if abc is nonnull:

```
(c-p) if { abc } { assign x 'abc } ↵
```

To evaluate an empty variable:

```
(c-p) assign x '"" ↵
(c-p) if {fir 'x} {wri x not empty} {wri x empty} ↵
x empty
(c-p)
```

To evaluate a nonempty variable:

```
(c-p) assign x abc ↵
(c-p) if {fir 'x} {wri x not empty} {wri x empty} ↵
x is not empty
(c-p)
```

See Also

Commands: **not**, **equal**, **c-p:while**, **debug:if**

Command: include Realm: command-processor

Summary

Read and execute the contents of a specified file

Arguments

Required:
- pathname — The pathname of a file

Keyword:
- continue — yes or no

Examples

```
include company
include crowd, cont
```

Description

Include reads the contents of a specified file and executes the file as a series of commands. This is useful for customizing your environment in a way other than that defined by your initialization file.

Arguments

pathname — The file should contain one or more commands.

continue — A yes value makes the **include** command keep going if any errors are encountered. A value of no makes the **include** command abort if errors occur.

Examples

To include a file named "company":

```
(c-p) include company↵
```

To include a file named "crowd" and keep going if errors are encountered:

```
(c-p) include crowd, continue ↵
```

See Also

Command: **c-p:assign, change-argument-value, define-macro, define-realm, define-topic, error, print-command, redirect-output**
Topic: initialization

Command: last — Realm: command-processor

Summary

Write the last word of a phrase

Arguments

Required:

phrase	One or more words

Keyword

character	yes or no

Examples

```
last a b c
last foo( bar ) baz
```

Description

Last writes the last word or character of a phrase. This is useful in macros.

Arguments

phrase	For more information about phrases, see the syntax help topic.
character	Write the last character instead of the last word if the value is yes

Examples

Following are valid examples:

```
(c-p) last a b c ↵
c
(c-p) last { a b } c d ↵
d
(c-p) last foo( bar ) baz ↵
baz
```

See Also

Commands: **c-p:evaluate, first, rest**
Topic: syntax

Command: length Realm: command-processor

Summary

Write the length of the given phrase

Arguments

Required:

phrase	One or more words

Keyword

character	yes or no

Examples

```
length `{some-phrase}
```

Description

Length writes the length of the given phrase in words (or characters).

Arguments

phrase	For more information about phrases, see the syntax help topic.
character	Write the length in characters instead of words if the value is yes

Examples

Following is a valid example:

```
(c-p) assign some-phrase This is a phrase ↵
(c-p) length '{some-phrase} ↵
4
(c-p) length '{some-phrase}, character ↵
16
```

See Also

Commands: **c-p:evaluate, position, subphrase**
Topic: syntax

Command: less Realm: command-processor

Summary

Determine if the left operand is less than the right operand

Arguments

Required:

left	Left string
right	Right string

Keyword:

case-sensitive	yes or no

Examples

```
less `{some variable} `{some-other-variable}
```

Description

Less writes a nonnull string when the left operand is less than the right operand when both are considered as strings.

Arguments

left	Left operand
right	Right operand
case-sensitive	Take the case of the operands into account for the comparison.

Examples

The following is valid:

```
(c-p) less '{some-variable} '{some-other-variable} ↵
```

See Also

Commands: **equal, greater, greater-equal, less-equal, not-equal**

Command: less-equal Realm: command-processor

Summary

Determine if the left operand is less than or equal to the right operand

Arguments

Required:

left	Left string
right	Right string

Keyword:

case-sensitive	yes or no

Examples

```
less-equal `{some variable} `{some-other-variable}
```

Description

Less-equal writes a nonnull string when the left operand is less than or equal to the right operand when both are considered as strings.

Arguments

left	Left operand
right	Right operand
case-sensitive	Take the case of the operands into account for the comparison.

Examples

The following is valid:

```
(c-p) less-equal '{some-variable} '{some-other-variable}
```

See Also

Commands: **equal, greater, greater-equal, less, not-equal**

Command: let — Realm: command-processor

Summary

Evaluate commands in a dynamic binding environment

Arguments

Required:

bindings	A value enclosed in braces
commands	Commands enclosed in braces

Examples

```
let {{b One line} {a Another line}} {a; b}
```

Description

Let evaluates commands in a dynamic binding environment. In this environment, the CP saves the current values, if any, of the bound variables before executing the commands enclosed in braces. After the CP executes the commands, it restores the values.

Arguments

bindings — A value enclosed in braces

commands — If you specify more than one command, you must separate them with semicolons or New Lines and enclose the entire series in braces.

Examples

Following is a valid example:

```
(c-p) assign a apple ↵
(c-p) ,, In this example, the CP saves the current value of variable a ↵
(c-p) let {{b One line} {a Another line}} {a; b} ↵
Another line
One line
(c-p) ,, The CP restores the value of variable a ↵
(c-p) a ↵
apple
(c-p)
```

See Also

Commands: **cp:assign**, **debugger:evaluate**

This page intentionally left blank.

Command: log Realm: command-processor

Summary

Send a record of input/output interaction to a file

Arguments

Optional:

input	Any syntactically valid file system pathname
output	Any syntactically valid file system pathname
error	Any syntactically valid file system pathname

Keyword:

input-delete	yes or no
output-delete	yes or no
error-delete	yes or no

Examples

```
log logfile
log, input login, output logout, error logerr
log
```

Description

Use the **log** command to create files containing records of command line input, output, or errors during a debugging session. You can have one file that contains all such records, or you can have separate files. Use the **input**, **output**, and **error** arguments to specify multiple files.

To display the current log files, use the **log** command with no arguments.

Arguments

input Specify a file to contain command line input; the pathname can be absolute or relative. Filename metacharacters ("wildcard" characters in Bourne shell terminology, "globbing" characters in C shell terminology) are not expanded.

output Specify a file to contain command line output; the pathname can be absolute or relative. Filename metacharacters ("wildcard" characters in Bourne shell terminology, "globbing" characters in C shell terminology) are not expanded.

error Specify a file to contain command line error messages; the pathname can be absolute or relative. Filename metacharacters ("wildcard" characters in Bourne shell terminology, "globbing" characters in C shell terminology) are not expanded.

input-delete Delete any existing command line input file with the specified pathname; the default is to append to the pathname.

output-delete Delete any existing command line output file with the specified pathname; the default is to append to the pathname.

error-delete Delete any existing command line error message file with the specified pathname; the default is to append to the pathname.

Examples

To start input, output, and error logging to the file **logfile**:

```
(debug) log logfile ↵
```

To create an input file named **login**, an output file named **logout**, and an error log file named **logerr**:

```
(debug) log, input login, output logout, error logerr ↵
```

To start logging into file **/usr/mark/debug/logfile**:

```
(debug) log /usr/mark/debug/logfile ↵
```

To create an input file named **login** that overwrites any existing input logfile with the same name:

```
(debug) log, input login, input-delete ↵
```

To display the current log files:

```
(debug) log ↵
input log files:
                /usr/chris/login
output log files:
                /usr/chris/logout
error log files:
                /usr/chris/logerr
(debug)
```

See Also

Command: **redirect-output, unlog**

This page intentionally left blank.

Command: not — Realm: command-processor

Summary

Negate a value

Arguments

Required:

value	Any value

Examples

```
not `""
not {not `""}
c-p:if {no `{equal foo bar}} {write hello}
```

Description

Not converts '"" (the null string) into '"true" and everything else into the null string. The value is written to the standard output.

Arguments

value	This is a phrase (see the syntax help topic).

Examples

Following are examples:

```
(c-p) not '"" ↵
true
(c-p) not '{ not '"" } ↵
(c-p) if {no '{equal foobar}} {write hello} ↵
hello
(c-p)
```

See Also

Commands: **equal, c-p:if**
Topic: syntax

Command: not-equal Realm: command-processor

Summary

Compare whether two arguments are not equal

Arguments

Required:

left	Any value
right	Any value

Keyword:

case-sensitive	yes or no

Examples

```
not-equal `{some-variable} `{some-other-variable}
```

Description

Not-equal writes a nonnull string if the **left** argument is not equal to the **right** argument when both are considered as strings.

Arguments

left	Left string
right	Right string
case-sensitive	Take the case of the operands into account for the comparison.

Examples

The following is valid:

```
(c-p) not-equal '{some-variable} '{some-other-variable} )
```

See Also

Commands: **equal, greater, greater-equal, less, less-equal**

Command: option-status — Realm: c-p

Summary

Display or set an option's status

Arguments

Optional:
- option-settings — Any value

Keyword:
- prompt — Yes or no

Examples

```
option-status pager 66
op lang
op
```

Description

To display the current global options and their values, type **option-status** without an **option-settings** argument.

Arguments

option-settings

This argument accepts one, two, or a list (enclosed in curly-braces) of tokens. When you supply one token, it must be the name of an options realm command; the command's value will be printed.

If you specify two tokens, the first name must be an options realm command and the second token is the command's new value; if the value is not valid, you will receive an error.

If you supply three or more tokens, they must be paired name–value bindings enclosed in curly-braces.

prompt

Specify this argument to invoke the prompting facility. This facility will prompt for each option using the standard prompting mechanism. See the prompting topic for more information.

086-000167 updates
093-000706-00

Examples

To display all currently set option values:

```
(c-p) op ↵
option-status {
  Pager_Lines                  23,
  Source_Lines                 10,
  Stop_Commands                ,
  Language                     c,
  Elide_Arrays                 yes,
  String_Display               yes,
  String_Display_Limit         100,
  Pointer_Dereference_Level    0,
  Convenience_Variables        no,
  Convenience_Variables_Limit  50,
  Bit_Format                   binary,
  Character_Format             ascii,
  Signed_Character_Format      ascii,
  Unsigned_Character_Format    ascii,
  Floating_Point_Format        ieee-float,
  Signed_Integer_Format        decimal,
  Unsigned_Integer_Format      unsigned-decimal,
  Unpacked_Decimal_Format      unpacked-decimal,
  Packed_Fixed_Decimal_Format  packed-decimal,
  Packed_Float_Decimal_Format  packed-decimal,
  Unknown_Type_Format          hexadecimal,
  Command_History              0,
  Message_History              0
}
(c-p)
```

To set the number of lines used by the pager to 66:

```
(c-p) option-status pager  66 ↵
```

To display the current option value for the expression evaluation language:

```
(c-p) op lang ↵
```

To set the number of source lines and set the signed integer format to hexadecimal:

```
(c-p) op {source 15, unsigned_integer_format hex} ↵
```

In this example, command prompting is invoked; since the prompting session is aborted, none of the options are actually changed:

```
(c-p) opt {language fortran}, prompt ↵
          Pager_Lines (23) =  20
          Source_Lines (15) =  ,abort
(c-p) opt lang; opt pager ↵
c
23
(c-p)
```

The next example shows how to you can create a customized command in the options realm:

```
(debug) c-p:assign options:my-vacation-location home ,, I'm broke
(debug) define-macro options:my-vacation {,optional location} {
(debug){ c-p:if {location} {,, remember the new location
(debug){{     c-p:assign my-vacation-location 'location
(debug){{ }, else {,, Report the current vacation location
(debug){{     my-vacation-location
(debug){{ }}
```

Now if you use the **option-status** command with no options, the global options will be listed first, and then your user-customization command (**my-vacation**, which contains exactly one optional argument, **location**) in the options realm will be listed:

```
(debug) op
option-status {
  Pager_Lines                 23,
  Source_Lines                10,
  Stop_Commands               ,
  Language                    c,
  Elide_Arrays                yes,
  String_Display              yes,
  String_Display_Limit        100,
  Pointer_Dereference_Level   0,
  Convenience_Variables       no,
  Convenience_Variables_Limit 50,
  Bit_Format                  binary,
  Character_Format            ascii,
  Signed_Character_Format     ascii,
  Unsigned_Character_Format   ascii,
  Floating_Point_Format       ieee-float,
  Signed_Integer_Format       decimal,
  Unsigned_Integer_Format     unsigned-decimal,
  Unpacked_Decimal_Format     unpacked-decimal,
  Packed_Fixed_Decimal_Format packed-decimal,
  Packed_Float_Decimal_Format packed-decimal,
  Unknown_Type_Format         hexadecimal,
  Command_History             0,
  Message_History             0,
  my-vacation                 home
}
```

If a group of tired developers decided to go to Hawaii, they would give the optional argument **location** a new value:

```
(debug) op my-va hawaii ,, I wish
(debug) op
option-status {
  Pager_Lines                  23,
  Source_Lines                 10,
  Stop_Commands                ,
  Language                     c,
  Elide_Arrays                 yes,
  String_Display               yes,
  String_Display_Limit         100,
  Pointer_Dereference_Level    0,
  Convenience_Variables        no,
  Convenience_Variables_Limit 50,
  Bit_Format                   binary,
  Character_Format             ascii,
  Signed_Character_Format      ascii,
  Unsigned_Character_Format    ascii,
  Floating_Point_Format        ieee-float,
  Signed_Integer_Format        decimal,
  Unsigned_Integer_Format      unsigned-decimal,
  Unpacked_Decimal_Format      unpacked-decimal,
  Packed_Fixed_Decimal_Format packed-decimal,
  Packed_Float_Decimal_Format packed-decimal,
  Unknown_Type_Format          hexadecimal,
  Command_History              0,
  Message_History              0,
  my-vacation                  Hawaii
}
```

See Also

Topic: c-p:prompting

This page intentionally left blank.

Command: or — Realm: command-processor

Summary

Write a phrase representing the logical OR of the arguments

Arguments

Required:

left	A string
right	A string

Examples

```
c-p:or `{some-variable} `{some-other-variable}
c-p:or `{sh cmp foo bar} `{sh cmp bar bletch}
```

Description

Or writes the empty string when both arguments are the empty string. Otherwise, it writes a nonnull string.

Arguments

left	A string
right	A string

Examples

This **or** command composes the AND operation:

```
(c-p) not {or {not '{a1}} {not '{as}}}
```

See Also

Commands: **and, not, if**

Command: page — Realm: command-processor

Summary

Page through the command output

Arguments

Required:
commands A command

Examples

```
page {sh ls -l *}
```

Description

Page allows you to page through command output in a manner identical to that of the **help** command. The **page** command is useful in conjunction with shell commands.

Arguments

command A command whose output will be paged.

Examples

Here is an example of the **page** command:

```
(c-p) page {sh ls -l *} ↵
```

Command: position Realm: command-processor

Summary

Write the numeric position of a regular expression in a phrase

Arguments

Required:

reg-expression	A regular expression
phrase	A phrase

Keyword:

from-end	yes or no

Examples

```
c-p:position foo somethingfoo somethingelse
```

Description

Position writes the numeric position (starting with 0) of the first character of the phrase which matches the regular expression; **position** also writes the number of characters in the matched phrase. If it does not find the regular expression, **position** writes a null string.

Arguments

reg-expression A regular expression

phrase A phrase

from-end Permits regular expression matching from the end of the phrase

Examples

Here is an example of the **position** command:

```
(c-p) position foo somethingfoo somethingelse ↵
9 3
(c-p) position foo foosomethingfoo, from-end ↵
12 3
```

See Also

Commands: **and, not, if, length, subphrase**
Topic: regular-expression

Command: print-command Realm: command-processor

Summary

Display a macro's definition

Arguments

Required:

name	The name of a macro or variable

Examples

```
pri frame
print-command say-hello
```

Description

Print-command displays a macro's definition as a well-formed invocation of the **define-macro** command or a CP variable's definition as a well-formed invocation of the **cp:assign** command. (Comments in the macro body are retained.) This lets you save a macro or variable definition to a file (using the **redirect-output** command). You can then include it in another session. Invoking **print-command** on a built-in command writes a null string to the standard output.

Arguments

name	This is a word (see the syntax help topic).

Examples

To set variable pi and its documentation string, and then display the variable and its documentation:

```
(c-p) assign pi 3.14159, doc "value of pi" ↵
(c-p) print-com pi ↵
command-processor:assign command-processor:pi 3.14159
,doc "value of pi"
```

To define and print the definition of a macro named say-hello:

```
(debug) define-macro say-hello {} { ↵
(debug){ write hello } ↵
(debug) print-command say-hello ↵
define-macro say-hello {} {
write hello }
```

See Also

Commands: **c-p:assign, define-macro, include, redirect-output**
Topic: syntax

Command: prompt-string Realm: command-processor

Summary

Display or set the prompt string

Arguments

Optional:
new-prompt A word

Examples

```
prom
prompt >
prompt-str (deb)
```

Description

Prompt-string displays or sets the current realm's prompt string. To set the prompt string, specify a **new-prompt** argument. To display the prompt string, omit the argument.

Arguments

new-prompt This is a word (see the syntax help topic).

Examples

To display the current prompt string:

```
(c-p) prom ↵
(c-p)
(c-p)
```

To set the prompt string to >:

```
(debug) prompt > ↵
>
```

To set the prompt string to (deb):

```
> prompt-str (deb) ↵
(deb)
```

See Also

Commands: **define-realm, realm, resume-prompting**
Topic: syntax

Command: protect Realm: command-processor

Summary

Execute commands in a protected region

Arguments

Required:
- protected-region — One or more commands enclosed in braces

Optional:
- cleanup-actions — One or more commands enclosed in braces

Keyword:
- errors-only — yes or no

Examples

```
pro { var x abc; eval x }, cleanup { var x, del }
protect {write 1st; error Something wrong; write 2nd},
cleanup { write Caught an error }, errors-only
```

Description

Protect executes the commands in the protected region, and then the commands specified as cleanup actions. This is useful in a macro for creating a variable in a protected region and deleting the variable as the cleanup action. It is also useful for catching errors and taking alternative actions.

Arguments

protected-region — This is a series of commands that might cause an error.

cleanup-actions — These commands are executed after the protected-region commands.

errors-only — Execute the cleanup-actions commands only if an error occurs.

Examples

To create and delete variable x:

```
(debug) pro { var x ab; eva x }, cl { var x, del } ↵
```

Another example:

```
(debug) prot {wr 1st; err Something wrong; wr 2nd}, ' ↵
(debug)` clean { wri Caught an error }, errors-only ↵
```

See Also

Commands: **error, write**

Command: query — Realm: command-processor

Summary

Write a prompt and read one line

Arguments

Required:

prompt	A text string for prompting

Examples

```
query How many?
c-p:assign x `{query Number of times:}
```

Description

Query writes a prompt to the standard output and reads a one-line user response from the standard input. This is useful for creating macros that the user can tailor dynamically.

Arguments

prompt	This is a phrase (see the syntax help topic).

Examples

Following are valid examples:

```
(c-p) query How many? ↵
How many? 4 ↵
4
(c-p) assign x '{query Number of times:} ↵
Number of times: 3 ↵
(c-p) x ↵
3
(c-p)
```

See Also

Command: **cp:assign, write**
Topic: syntax

Command: quit — Realm: command-processor

Summary Exit from an interactive tool

Examples

```
quit
```

Description **Quit** exits from the current interactive tool (such as Mxdb).

Examples

```
(debug) quit ↵
```

See Also Commands: **bye**, **terminate**

Note The **quit** and **bye** commands do exactly the same thing.

Command: realm Realm: command-processor

Summary

Display or set the current realm

Arguments

Optional:
realm-name The name of a realm

Examples

```
realm
realm c-p
```

Description

With no argument, **realm** displays the current realm. With a **realm-name** argument, it sets the current realm.

Arguments

realm-name This word can be abbreviated.

Examples

To display the current realm:

```
(c-p) realm ↵
command-processor
```

To change the current realm to debugger:

```
(c-p) realm deb ↵
(debug) realm ↵
debugger
```

See Also

Commands: **define-realm, prompt-string, realm-use-list**
Topics: abbreviation, syntax

Command: realm-use-list Realm: command-processor

Summary

Display or set realm use list

Arguments

Optional:

used-realm	A list of sets of realms (grouped by braces) that are used to search for commands. You must include the target realm in the list.

Keyword:

realm	The name of a realm; the default is the current realm

Examples

```
r-u
realm-use-list {{foo bar} {command-processor}}
```

Description

Without arguments, **realm-use-list** displays the realms that are used by the target realm indicated by the **realm** argument. With a **used-realms** argument, **realm-use-list** replaces the use-list of the target realm.

Which realms are in the target realm use list affects how you can abbreviate command names, because each abbreviated command name must be unique among the commands in the realms on the current realm use list.

Arguments

used-realm	Set rather than display the realm use list.
realm	Determine which realm to use. The initial default is the current realm.

Examples

To display the current realm use list:

```
(c-p) realm-use-list ↵
{{command-processor foo}}
```

To set the realm use list so that the CP first looks in realms foo and bar, and then in c-p, to find commands, macros, and CP variables:

```
(c-p) realm-use-list {{ foo bar } { c-p }} ↵
(c-p)
```

See Also

Commands: **define-realm**, **realm**
Topic: abbreviation

Command: redirect-output Realm: command-processor

Summary

Send output to a file

Arguments

Required:

body	One or more commands enclosed in braces

Optional:

standard-output	A pathname
error-output	A pathname

Keyword:

standard-append	yes or no
error-append	yes or no

Examples

```
red {help assign} help.messages
redirect-output {help evaluate} help.messages, standard-append
```

Description

Redirect-output changes where the output from a series of commands goes. By default, all output goes to the display unit. You can redirect standard output and error output independently.

Arguments

body	Redirect the output from these commands.
standard-output	This file is where the standard output goes.
error-output	This file is where any error messages go.
standard-append	Append to the standard output file if it already exists.
error-append	Append to the standard error file if it already exists.

Examples

To write a help message to **help.messages**, deleting that file if it already exists:

(debug) **red {help assign} help.messages** ↵

To append a help message to **help.messages**:

(debug) **redir-o {help eval} help.messages, standard-append** ↵

See Also

Commands: **include, write**

Command: rest — Realm: command-processor

Summary

Write all but the first word or character of a phrase

Arguments

Required:

phrase	One or more words

Keyword:

character	yes or no

Examples

```
rest a b c
rest { a b } c d
rest foo( bar ) baz, character
write "`{first `x} `{rest ´x}"
```

Description

Rest writes all but the first word or character of a phrase.

Arguments

phrase	For more information about phrases, see the syntax help topic.
character	Write all but the first character (instead of the first word) if the value is yes

Examples

Following are valid examples:

```
(c-p) rest a b c ↵
b c
(c-p) rest { a b } c d ↵
c d
(c-p) rest foo( bar ) baz, character ↵
oo( bar ) baz
(c-p) assign x now is the time ↵
(c-p) write "'{first 'x} '{rest 'x} ..." ↵
"now is the time ..."
(debug) write "'{rest 'x} '{first 'x} ?.." ↵
"is the time now ?.."
```

See Also

Commands: **do-sequence, first, last**
Topic: syntax

This page intentionally left blank.

Command: return — Realm: command-processor

Summary

Return from a macro

Arguments

Optional:

anything	One or more words

Examples

```
define-macro star {x} {return * `x *}
```

Description

Return returns from a macro passing back a phrase. For information about phrases, see the syntax help topic.

Arguments

anything	These words are displayed on the standard output (by default, your screen).

Examples

The following are valid:

```
(c-p) define-macro star {my_var} {return * 'my_var *} ↵
(c-p) star foo ↵
* foo *
(c-p)
```

See Also

Commands: **do-sequence, define-macro, redirect-output**

Command: shell — Realm: command-processor

Summary

Execute a subshell or a shell command sequence

Arguments

Optional:

command-line A shell command line

Examples

```
sh
sh date
shell ls -l
```

Description

Shell executes a subshell or a shell command sequence. If the environment variable SHELL is defined, it is used. Otherwise **/bin/sh** (the Bourne shell) is used.

Arguments

command-line If you specify a command line, that command line is executed in a subshell. If you omit the **command-line** argument, a shell is executed.

Examples

To enter the shell:

```
(debug) sh ↵
```

To display the current date and time:

```
(debug) sh date ↵
```

To list the files that are in your working directory:

```
(debug) shell ls -l ↵
```

Command: subphrase Realm: command-processor

Summary

Write a subphrase

Arguments

Required:

start	cardinal
length	cardinal
phrase	One or more words

Keyword:

character	yes or no

Examples

```
subphrase 3 1 This is a subphrase.
assign my_var  {subphrase 0 5 supercalifragilistic, character}
```

Description

Subphrase writes a specified part of a phrase.

Arguments

start	Specify from where to begin writing (starting at position 0).
length	Specify the length of the subphrase.
phrase	For more information about phrases, see the syntax help topic.
character	Write characters (instead of words) if the value is yes

Examples

This is an example using words:

```
(c-p) subphrase 3 1 This is a subphrase.↵
subphrase.
(c-p)
```

The following example uses the character keyword:

```
(c-p) assign my_var '{subphrase 0 5 supercalifragilistic, character} ↵
(c-p) my_var ↵
super
(c-p)
```

See Also

Commands: **length, position**
Topic: syntax

Command: trace-commands Realm: command-processor

Summary

Trace the execution of commands, variables, or macros

Arguments

Optional:

names	A whitespace-separated sequence of words

Keywords:

arguments	yes or no
body	yes or no

Examples

```
trace-commands write, arg
trace-commands c-p:assign, body
```

Description

Trace-commands traces the execution of one or more commands, variables, or macros. In each case, the CP will display the invocation of the traced object; optionally passed argument values, if any, can be displayed.

If you don't supply a value for the **names** argument, the CP outputs a list of the visible objects.

You can trace the commands that implement a macro with the **body** argument. Unless these commands are being explicitly traced themselves, they will be traced with the same options you specify for the macro.

If you assign a new value to a traced variable or redefine a traced macro, tracing (with whatever options you last selected) continues for the new command. If a command is explicitly deleted and then newly defined, the new command is not traced.

Tracing information is output to the CP's error stream.

Arguments

names	A valid command, variable, or macro name
arguments	Trace command arguments
body	Activate macro body tracing

Examples

This example uses the **trace-commands**, **trace-status**, and **untrace-commands** commands:

```
(c-p) trace-commands realm-use-list, arguments ↵
(c-p) realm-use-list, realm characters ↵
0: command-processor:realm-use-list {{}, characters}
{ { characters command-processor } }
(c-p) define-macro callee {arg} {return done'char:new-line} ↵
(c-p) define-macro caller {arg1 arg2} {callee 'arg1} ↵
(c-p) trace-commands caller, arguments, body ↵
(c-p) caller hello there now ↵
0: command-processor:caller {hello, there now}
   1: command-processor:callee {hello}
done
(c-p) trace-commands callee, arguments, body ↵
(c-p) caller hello there now ↵
0: command-processor:caller {hello, there now}
   1: command-processor:callee {hello}
      2: characters:new-line {}
      2: command-processor:return {done
}
done
(c-p) trace-commands callee, arguments ↵
(c-p) caller hello there now ↵
0: command-processor:caller {hello, there now}
   1: command-processor:callee {hello}
done
(c-p) trace-status ↵
c-p:trace-commands command-processor:caller ,arguments ,body
c-p:trace-commands command-processor:callee ,arguments
c-p:trace-commands command-processor:realm-use-list ,arguments
(c-p) untrace realm-use-list ↵
(c-p) t-s ↵
c-p:trace-commands command-processor:caller ,arguments ,body
c-p:trace-commands command-processor:callee ,arguments
(c-p) assign my_var red ↵
(c-p) trace-commands my_var ↵
(c-p) my_var ↵
0: command-processor:my_var
red
(c-p) if {my_var} {assign my_var blue} ↵
0: command-processor:my_var
(c-p) my_var ↵
0: command-processor:my_var
blue
(c-p)
```

See Also

Commands: **trace-status**, **untrace-commands**

Command: trace-status Realm: command-processor

Summary

Display the status of all traced objects

Arguments

none

Examples

```
trace-status
trace-s
```

Description

Trace-status displays, as invocations of the **trace-commands** command, the status of all traced commands, variables, and macros.

Examples

This example uses the **trace-commands**, **trace-status**, and **untrace-commands** commands:

```
(c-p) trace-commands write, arg ↵
(c-p) trace-c c-p:assign, body ↵
(c-p) trace-status ↵
c-p:trace-commands command-processor:assign
c-p:trace-commands command-processor:write ,arguments
(c-p) untrace write c-p:assign ↵
(c-p) trace-s ↵
(c-p)
```

See Also

Commands: **trace-commands**, **untrace-commands**

Command: unlog Realm: command-processor

Summary Turn logging off

Arguments Optional:

files	A whitespace-separated sequence of words

Examples

```
unlog
unlog logerr
```

Description Use the **unlog** command to turn logging off and write the names of any log files to the standard output. If you do not specify the **files** argument, **unlog** turns all logging off.

Arguments

files	The pathnames (absolute or relative) of log files.

Examples To stop all logging:

```
(debug) unlog ↵
```

To turn off logging to a specified file:

```
(debug) unlog logerr ↵
```

See Also Command: **log**

086-000167 updates
093-000706-00

Command: untrace-commands — Realm: c-p

Summary

Stop tracing the execution of commands, variables, or macros

Arguments

Optional:

names	A whitespace-separated sequence of words

Examples

```
untrace-commands write
untrace-commands c-p:assign
```

Description

Untrace-commands stops tracing the execution of one or more commands, variables, or macros.

Arguments

names	A valid command, variable, or macro name

Examples

This example uses the **trace-commands**, **trace-status**, and **untrace-commands** commands:

```
(c-p) trace-commands write, arg ↵
(c-p) trace-c c-p:assign, body ↵
(c-p) trace-status ↵
c-p:trace-commands command-processor:assign
c-p:trace-commands command-processor:write ,arguments
(c-p) untrace write c-p:assign ↵
(c-p) trace-s ↵
(c-p)
```

See Also

Commands: **trace-commands**, **trace-status**

This page intentionally left blank.

Command: while — Realm: command-processor

Summary

Execute commands conditionally while predicate is nonnull

Arguments

Required:

predicate	One or more commands enclosed in braces
body	One or more commands enclosed in braces

Examples

```
c-p:assign x a b c
c-p:whi {x} {write X is "`x"; assign x `{rest `x}}
```

Description

While evaluates the predicate. If it returns a nonnull phrase, then it evaluates the body and repeats. **While** returns the value returned by the last command to be evaluated.

Arguments

predicate	These commands return either an empty or nonnull phrase.
body	While the phrase is nonnull, these commands are executed.

Examples

Following are valid examples:

```
(c-p) assign x a b c ↵
(c-p) whi {x} {write X is "'x"; assign x '{rest 'x}} ↵
X is "a b c"
X is "b c"
X is "c"
(c-p)
```

See Also

Commands: **equal, c-p:if, not**

Command: write — Realm: command-processor

Summary

Display arguments

Arguments

Required:

text	One or more words

Keyword:

message	yes or no
no-newline	yes or no

Examples

```
wri Hello everybody
write Hello`"," Joe.
write Hello`char:comma Joe.
evaluate {wr hello, no-new; wr hello}
```

Description

Write writes the specified arguments to the standard output.

Arguments

text — To specify characters that are special to the CP (for example, braces, a comma, or a semicolon), use either backquote substitution or characters from the characters realm.

message — Write the arguments to the error output instead of standard output.

no-newline — Omit the New Line character at the end of the arguments being written.

Examples

To write "Hello everybody." to your screen:

```
(c-p) wri Hello everybody. ↵
Hello everybody.
```

To write "Hello, Joe." to your screen:

```
(c-p) write Hello`char:comma Joe. ↵
Hello, Joe.
```

To do the same thing with less effort:

```
(c-p) write Hello`"," Joe. ↵
Hello, Joe.
```

To write so that the next write will be on the same line with no intervening space:

```
(c-p) evaluate {wr hello, no-new; wr hello} ↵
hellohello
```

See Also

Commands: **query, redirect-output**
Realm: characters

End of Chapter

Chapter 5
Command Processor Types

This chapter contains the help messages for types in the command-processor (c-p) realm. A type is a category of argument values accepted by the Command Processor (CP). Each command argument has a type. The CP validates the argument value according to the type. If the argument value is of the appropriate type, the CP accepts it and passes it to the command. If the argument value is not of the appropriate type, the CP rejects it and displays an error message.

The help messages for these types are listed in alphabetical order, as follows:

anything
braces
cardinal
command-name
command-sequence
documentation
integer
list
ordinal
pathname
string
yes-no

This chapter uses the following notation conventions:

This typeface Indicates a literal value that you must type exactly as shown. (In some cases, you may be able to abbreviate or to interchange uppercase and lowercase letters.)

This typeface Describes a user-supplied value that you must insert. The value is usually described immediately after the syntax line.

A | B Indicates that you may choose either A or B.

[A] Indicates that A is optional.

A ... Indicates that you can repeat A as many times as necessary.

Type: anything Realm: command-processor

Summary

Accept any values and pass them on uninterpreted

Syntax

token

token	A string of characters delimited by spaces or tabs

Examples

```
foobar7
3.14159
1.414 * 23
```

Description

The **anything** type accepts any value and passes it on uninterpreted.

See Also

Topic: syntax

Type: braces Realm: command-processor

Summary

Accept a value enclosed in braces

Syntax

{}

Description

The **braces** type accepts a value enclosed in braces.

Type: cardinal — Realm: command-processor

Summary

Any integer expression greater than or equal to 0

Syntax

non-neg-int

non-neg-int	A nonnegative integer expression

Examples

```
(1 + (3 * 2))
07
1989
```

See Also

Types: integer, ordinal

086-000167 updates
093-000706-00

Type: command-name Realm: command-processor

Summary

A command, macro, or variable name

Syntax

prefix

prefix	A name or abbreviation that uniquely specifies a command, macro, or CP variable.

Examples

```
breakpoint
br
process-status
pro
p-s
```

See Also

Type: command-sequence

Type: command-sequence Realm: command-processor

Summary

One or more commands possibly surrounded by braces

Syntax

[{] *command* [; *command*] ... [}]

command A command followed by zero or more arguments

Examples

```
log
{ assign a 2 }
{ assign b i; write `b }
```

Description

The **command-sequence** type accepts one or more commands and removes any outer brace characters.

Syntax

command Braces are required if you specify command arguments or more than one command. If you use a brace, you must also use the matching brace to form a pair.

See Also

Commands: **if, while**
Types: anything, command-name
Topic: syntax

Type: documentation Realm: command-processor

Summary

A phrase in standard documentation format

Syntax

phrase

phrase	One or more words

Description

The **documentation** type accepts a phrase in standard documentation format. This type of phrase has an arbitrary number of quoted text strings enclosed in braces, followed by three character strings, at most. The brace-enclosed strings are either short descriptions of command arguments or a short description of a type to be included in the command text if a command specifies no descriptive string.

Type: integer — Realm: command-processor

Summary

An integer expression

Syntax

int

int	An integer expression (positive, negative, or zero)

Examples

```
0
(1 + (3 * 7))
+12
-6
```

See Also

Types: cardinal, ordinal

086-000167 updates
093-000706-00

Type: list — Realm: command-processor

Summary

Accept a phrase

Syntax

phrase

phrase	A whitespace-separated sequence of words

Description

The list type accepts a phrase.

Type: ordinal — Realm: command-processor

Summary

Any integer expression greater than or equal to 1

Syntax

pos-int

pos-int	A positive integer expression

Examples

```
(1 + (3 * 2))
07
1989
```

See Also

Type: cardinal, integer

086-000167 updates
093-000706-00

Type: pathname Realm: command-processor

Summary

Any syntactically valid file system pathname

Syntax

pathname

pathname	The pathname of a file; filename metacharacters (called "wildcard" characters in the Bourne shell and "globbing" characters in the C shell) are not expanded

Examples

```
abc7
foo.c
/usr/include/stdio.h
../test/file
```

See Also

Type: string

Type: string — Realm: command-processor

Summary

Any valid string

Syntax

[*quote*] *anything* [*quote*]

quote	A quote character (" or ')
anything	Any nonquote characters or two consecutive quote characters

Examples

```
"abc"
'abc'
""
"a'b"
abc
"a""b"
```

Description

The **string** type accepts any valid string; a valid string has optional quotation marks that enclose any nonquote characters or pairs of quote characters. If enclosing quotes are found, the string is transformed by removing the outer quotes and making any internal doubled quotes single.

Syntax

quote	If one quote is present, the matching quote must also be present.
anything	If enclosing quotes are present, this value cannot contain an unpaired quote of the same kind.

Examples

The following are legal strings and their transformations:

String	Becomes
"abc"	abc
'abc'	abc
""	
"a'b"	a'b
abc	abc
"a""b"	a"b

The following are invalid strings:

"	A matching double quote is not present.
'	A matching single quote is not present.
"abc	A matching double quote is not present.
'a'b'	The single quotation mark within the string is not paired.
"a"b	A matching, enclosing double quote is not present.

See Also

Type: string-quote

Type: yes-no Realm: command-processor

Summary

Yes or no

Syntax

yes | no

Examples

```
yes
no
y
```

Description

The **yes-no** type accepts a value of yes or no. The words "yes" and "no" can be abbreviated.

See Also

Topics: abbreviation, syntax

End of Chapter

Chapter 6
Command Processor Topics

This chapter contains the help messages for topics in the command-processor (c-p) realm. The messages are in alphabetical order.

The help topics in the c-p realm are as follows:

abbreviation
command-prompting
documentation
paging
prompting
realms
regular-expression
semantics
substitution
syntax
types

In this chapter, entries that show the syntax for performing a specific task use angle brackets (<>) to indicate a value that you supply.

Topic: abbreviation Realm: command-processor

Summary

How to abbreviate names

You can abbreviate the name of a command, argument, macro, variable, realm, type, or topic. The minimum abbreviation depends on the list of visible names in the realms on your realm use list.

A name has one or more syllables separated by hyphens or underscores. Names are resolved as follows:

1. An exact match
2. A name with the same number of syllables, each beginning with the characters you specify
3. A name with more syllables, beginning with the characters you specify

Description

The Command Processor lets you abbreviate the name of a command, argument, macro, variable, realm, type, or topic. You can also abbreviate an argument value if it is a literal rather than a user-supplied value. The minimum abbreviation for such a value depends on what values the command accepts for that argument.

When abbreviating a word, you can abbreviate the entire word or individual syllables within a word. The abbreviation is valid if it uniquely identifies the word.

Each syllable is composed of the following characters: letters (A-Z and a-z), digits (0-9), and #$%&*+./<=>?@|. Syllables are joined by a hyphen (-) or an underscore (_). The CP is case insensitive, and it treats an underscore as a hyphen.

Note that the CP resolves names enclosed in matching single or double quotation marks as regular expressions.

Examples

The following words represent **event-status, evaluate, machine-state,** and **realm-use-list** according to the criteria listed above:

1. evaluate	event-status	machine-state	realm-use-list
2. e	e-	m-	r-u-l
3.	eve	m	rea-

The following are equivalent:

```
event-status
event_status
EVENT-STATUS
EVENT_STATUS
Event-Status
```

Following are argument value abbreviations:

```
assign abc 100 ,mode oct
describe xyz ,meaning-kind ext
```

The CP resolves the following names enclosed in quotation marks as regular expressions:

```
(c-p) ,, This example lists commands that contain at least 2 ↵
(c-p) ,, occurrences of the letter "s" ↵
(c-p) help, c 's\{2\}' ↵
    assign expression less less-equal
(c-p) ,, This command lists commands that begin with the ↵
(c-p) ,, letter "a," "b," or "c" ↵
(c-p) help, c "^[a-c]" ↵
    and assign bye change-argument-value copy-command
```

See Also

Command: **realm-use-list**
Topic: regular-expression, syntax

Topic:command-prompting Realm:command-processor

Summary

Command prompting (interactive argument help)

To enter the command prompting facility, type a command followed by a comma but no argument. The CP then prompts you for input one argument at a time. The prompt appears in either of two forms (the first means a value is already assigned to the argument):

<argument-name> (<current-value>) =
<argument-name> =

Following are possible responses by category:

Category	Command	Task
Information	,	Describe the current argument
	,help	Display a help message
	,refresh	Refresh the screen
Argument	<value>	Specify a value
	<New Line>	Select the value in parentheses
	,default	Select the default value
	,implied	Select the implied value
Navigation	<New Line>	Skip to the next argument
	,previous	Move back one argument
Termination	,abort	Abort back to the top level
	,execute	Execute the command

Note that if you use the **include** command (from the Mxdb command line) to read and execute the contents of a file that contains a prompting request, the request will be ignored. Also, if you redirect Mxdb's input (from a shell prompt) to a file that contains a prompting request, the request will be ignored.

See Also

Command: **help**
Topic: prompting

086-000167 updates
093-000706-00

This page intentionally left blank.

Topic: documentation Realm: command-processor

Summary

User-defined documentation strings

When you define a macro, realm, or variable, you can associate with it up to three documentation strings. These documentation strings are displayed by the **help** command and normally contain the following information:

1. A one-line summary
2. Brief examples
3. A more verbose description

Each string must be enclosed in a pair of matching quotation marks (" or '). If you want to put one of the enclosed quote characters into the string, you must double the character.

Description

When each documentation string (described above) is displayed depends on the level of text verbosity to which the **help** command is set:

String	Displayed when Verbosity Is
1	short, medium, or long
2	medium or long
3	long

See Also

Commands: **c-p:assign, define-macro, define-realm, help**

Topic: paging Realm: command-processor

Summary

Help message paging

When you get a help message that is more than one screenful long, the first screenful of the message is displayed. You can display more text by pressing the following keys:

New Line	One line forward
d	Half a screenful forward
h or ?	Display a short help message on the page
z or space	One screenful forward
b	One screenful backward
Tab	To the beginning of the message
q, Q, or <intr>	Quit (exit from the help message)

<intr> indicates the interrupt key. This defaults to the Delete key but on DG/UX systems is often reset to Ctrl-C via the **stty**-command.

See Also

Command: **help**

Topic: prompting Realm: command-processor

Summary

Prompting

The CP prompts you for input one argument at a time. The prompt appears in either of two forms (the first means a value is already assigned to the argument):

```
<argument-name> (<current-value>) =
<argument-name> =
```

Following are possible responses by category:

Category	Command	Task
Information	,	Describe the current argument
	,help	Display a help message
	,refresh	Refresh the screen
Argument	<value>	Specify a value
	<New Line>	Select the value in parentheses
Navigation	<New Line>	Skip to the next argument
	,previous	Move back one argument
Termination	,abort	Abort back to the top level
	,execute	Execute the command

Note that if you use the **include** command (from the Mxdb command line) to read and execute the contents of a file that contains a prompting request, the request will be ignored. Also, if you redirect Mxdb's input (from a shell prompt) to a file that contains a prompting request, the request will be ignored.

See Also

Command: **help**
Topic: command-prompting

Topic: realms Realm: command-processor

Summary

Introduction to realms

All commands are organized into groups called realms. Realms organize commands in much the same way directories organize files, except that a realm cannot contain another realm. As every file in a file system is in a directory, every command is in a realm.

For example, when Mxdb begins running, you are working in the debugger (debug) realm. You can move to other realms, such as the command-processor (c-p) or characters (char) realm.

To access commands in another realm, you must precede a command with the desired realm name and a colon.

Description

Several realms exist, including the following:

debug This realm contains commands for Mxdb debugging programs.

c-p This realm contains Command Processor commands. The CP manages the syntax of commands and the set of visible commands. It also lets you define macros, get help, and control I/O and execution flow.

char This realm lets you put into your command line characters that would normally be special to the Command Processor.

icobol This realm contains commands for debugging Interactive COBOL programs.

g-i This realm contains commands for graphical interface users.

Each realm has a realm use list. The realm use list controls which realms' commands are visible from a particular realm. You can perform the following tasks with the following commands (<name> is the name of a realm; <list> is a realm use list):

Create a realm.	define-realm <name>
Delete a realm.	delete-realm <name>
Display the current realm.	realm
Set the current realm.	realm <name>
Display the realm use list.	realm-use-list
Set the realm use list.	realm-use-list { <list> }

When you create a new realm, it is empty. You can put commands, macros, topics, and CP variables into a realm as follows (each <name> is a word optionally preceded by a realm name and a colon):

Command	copy-definition <old-prefix> <name>
Macro	define-macro <name> {<arguments>} {<body>}
Topic	define-topic <name> "<text>" ...
Variable	c-p:assign <name> <value>

See Also

Commands: **define-realm**, **delete-realm**, **realm**, **realm-use-list**

Topic: regular-expression Realm: command-processor

Summary

Using regular expressions

A regular expression defines a set of one or more strings of characters; certain characters are interpreted to match patterns. These pattern-matching characters are called metacharacters because they represent something other than themselves. Regular expressions are used to quickly match strings. Here are the recognized metacharacters:

^	Force the match to the beginning of a line
$	Force the match to the end of a line
.	Match any single character
*	Match zero or more occurrences of a match of the preceding character
[*abc*]	Define a character class that matches *a*, *b*, or *c*
[^*abc*]	Define a character class that matches any character except *a*, *b*, or *c*
[*a-z*]	Define a character class that matches any character *a* through *z* inclusive
\	Denote a special character (^,$,.,*,–,[, or])
\(*abc*)\	Match what *abc* matches; a bracketed regular expression
n	Represent the *n*th bracketed regular expression matched
\{*n*\}	Match at least *n* occurrences of a match of the preceding character
\{*n*,\}	Match exactly *n* occurrences of a match of the preceding character
\{*n*,*m*\}	Match from *n* to *m* occurrences of a match of the preceding character

Description

Some characters are metacharacters only in a particular context. In the following contexts the characters listed above are not metacharacters:

^	Not at the beginning of a regular expression
$	Not at the end of a regular expression
–	Outside a pair of brackets or is the first or last character between a pair of brackets
.	Between a pair of brackets
*	Within brackets or as the first character in a regular expression not counting an initial ^
[	Between a pair of brackets
]	First character between a pair of brackets

Outside of a pair of brackets, you can make the period, asterisk, left bracket, or right bracket represent itself by preceding it with a backslash(\). The backslash is also an escape character for itself; you must use two backslashes to represent a literal backslash in a regular expression.

Note: If a user encloses a name on the command line in single or double quotation marks, the CP uses regular expression resolution instead of the default unique-prefix name resolution.

Examples

To get help on the commands in the command-processor realm whose names contain the word "realm":

```
(debug) help ,rea c-p ,com "realm" ↵
```

To search a source text file for the string "*char":

```
(debug) find *char ↵
```

To find an x followed by a a right bracket or a hyphen:

```
(debug) find x[]-] ↵
```

To list commands that contain at least two occurrences of the letter "s":

```
(c-p) help, c 's\{2\}' ↵
    assign expression less less-equal
```

See Also

Commands: **find, help**

Topic: semantics Realm: command-processor

Summary

Command Processor semantics

A command takes a series of arguments and performs a task. Each argument is required, optional, or keyword and can receive its value by position, by name, by default, or implicitly. Most commands display output on your screen.

The first phrase of a command starts with the command name as the first word; succeeding words are values for required or optional arguments of the command. The rest of the phrases each start with a keyword (the name of an argument to the command) and give a value to that argument.

You can abbreviate a command or argument name using standard CP abbreviation rules (see the abbreviation help topic).

Examples

```
realm c-p
write Here are some symbols: #$&*<>?\|~
include script_file ,continue
```

Description

In the Command Processor (CP), a command takes a series of arguments and produces textual output. The output text is normally displayed on your screen, but you can capture it (using a backquote) in a command line or or redirect it (using the **redirect-output** command) to a file.

You can specify arguments positionally (as in the first two examples above) or by name (as in the third example). The three kinds of arguments are as follows:

required	A positional argument that must be specified
optional	A positional argument that may be specified
keyword	An argument that cannot get values by position

When you give more values in the first phrase than there are positional arguments, the additional words are used as part of the value of the last positional argument. When you give more than one word as a value in a keyword phrase, the extra words are part of the keyword phrase.

Every command argument is given a value when the command is executed. Arguments that are not given values by name or by position are given values by default. Arguments that are mentioned by name but are given no explicit value on the command line are given values implicitly.

The following table shows the possible permutations of command **c** with required argument **a1**, optional argument **a2**, and keyword argument **a3**. Values assigned explicitly are indicated as **v1**, **v2**, and **v3**. Values assigned implicitly are indicated as **i1**, **i2**, and **i3**. Values assigned by default are indicated as **d1**, **d2**, and **d3**.

Value	Argument Type: required(a1)	optional(a2)	keyword(a3)
By position	c v1 v1 d2 d3	c v1 v2 v1 v2 d3	_____
By name	c, a1 v1 v1 d2 d3	c v1, a2 v2 v1 v2 d3	c v1, a3 v3 v1 d2 v3
Default	_____	c v1 v1 d2 d3	c v1 v1 d2 d3
Implied	c, a1 i1 d2 d3	c v1, a2 v1 i2 d3	c v1, a3 v1 d2 i3

A help message for command **c** with arguments displayed at the "short" verbosity level would show the following:

```
c a1 [a2] ,a3
```

The actual value assigned to an argument implicitly or by default is defined by the command. If the command does not assign an implicit or default value, then the relevant entries in the above table become illegal, in addition to the two dashed entries that are never legal.

A common use of implied and default values is with an argument whose only possible value is yes or no. For the debugger's built-in commands, the initial implied and default values are yes and no, respectively.

Another potential use for these rules is to skip over arguments and specify a trailing optional argument by keyword. Let's assume we have a command to set the time of day, with optional positional arguments set up so that they default appropriately if not given, but can be overridden. Let's say something like

```
set-time [ minutes hours day month year ]
```

Normally, one might just say "set-time 23" to set the minute or "set-time 23 08" to set the minute and hour. But let's assume somebody set the time correctly except for the year. To correct this, one might say "set-time, year 1989," specifying the year by keyword to skip over all the already-correct components.

In general, debug and c-p commands use positional arguments for values that often need to be specified and keyword arguments for values or options that seldom need to be specified. Implied values are often set up for these less-often-used keywords, so that just mentioning the keyword does some useful or obvious thing.

Examples

Following are one-phrase commands with required and optional arguments:

```
write Here are some symbols: #$&*<>?\|~
realm c-p
```

Following is a two-phrase command having a required argument with an explicit value and a keyword argument with an implied value:

```
include script_file, continue
```

The following are equivalent:

```
addr i
address i
address ,ref i
address ,reference i
```

See Also

Topics: abbreviation, syntax

Topic: substitution Realm: command-processor

Summary

Command and parameter substitution

The Command Processor lets you insert into a command line the output of a command, the value of a variable, or the value of a quoted string. To do this, precede the command, variable name, or string with a backquote character (`). To delimit a command with arguments, enclose the command and its arguments in braces. To substitute within braces, use two backquotes unless the backquotes are enquoted.

You can abbreviate the names of backquoted commands and variables. Following are some examples:

```
write The current realm is `realm
realm `NAME
prompt-string (`NAME)
prompt-string {``NAME}
assign x `{realm-use-list ,realm c-p}
write A single quote: `"'"
```

Description

Following is an expansion of the examples listed above:

```
(c-p) write The current realm is '{realm}. ↵
The current realm is command-processor.
(c-p) define-realm macros ↵
(c-p) assign NAME macros ↵
(c-p) assign PROMPT mac ↵
(c-p) realm 'NAME ↵
(macros) prompt-string {''PROMPT} ↵
{mac} prompt-string ('PROMPT) ↵
(mac) assign x '{realm-use-list ,realm c-p} ↵
(mac) x ↵
{ { command-processor } { characters } }
(mac) write A single quote: '"'" ↵
A single quote: '
(mac)
```

See Also

Topic: abbreviation, syntax

Topic: syntax Realm: command-processor

Summary

Command syntax

A command is composed of one or more comma-separated phrases terminated by a new line or semicolon. A typical command has one phrase.

A phrase consists of one or more words separated by blanks (spaces or tabs). A word contains one or more characters other than a blank, New Line, or semicolon.

For information about command semantics, see the semantics help topic.

Examples

Following are one-phrase commands:

```
write Here are some symbols: #$&*<>?\|~
assign x 23
include script_file
```

Description

A command is composed of one or more comma-separated phrases terminated by a New Line character or semicolon. A typical command has one phrase, with no comma.

A phrase consists of one or more words separated by blanks. A word contains one or more characters other than a blank, New Line, or semicolon, except that you can incorporate any characters into a word by enclosing them in matching pairs of double quotes (""), single quotes (''), parentheses (()), brackets ([]), or braces ({}).

The ordinary word characters are as follows: letters (A-Z and a-z), digits (0-9), and !#$%&*+-./<=>?@_|~.

The comma has three uses:

- To separate phrases.
- To begin a comment. A pair of commas not enclosed in quotes, parentheses, brackets, or braces begins a comment terminated by a New Line or semicolon.
- To invoke a help subsystem. If a command ends in a null phrase (a comma followed by a New Line or semicolon), the CP enters a help subsystem and prompts you for argument values. The null phrase may include blanks or a comment.

The backquote has two uses:

- To insert generated text into a command line. See the substitution help topic.
- To continue a command. To do this, put the backquote at the end of the command, optionally followed by blanks or a comment.

The only other character that has a special meaning is the colon. A colon between a realm name and the name of a command, CP variable, or topic indicates that the command, CP variable, or topic is located in the specified realm. This is useful when two commands have the same name and you want to indicate one in a specific realm.

Examples

Following are three two-word phrases and two three-word phrases:

```
foo bar
word "remove bletch"
"a, b, c" "d, e, f"
a + b
name value1 value2
```

Following are some multiple-phrase commands:

```
write The cursor will stay right here:, no-newline
assign x 23, doc "x contains # lines per screen."
include script_file, continue
```

Following is a comment after a command:

```
write This stuff gets written ,, but this does not
```

A null phrase invokes a help subsystem:

```
(c-p) write , )
     Type ",help" for help.
       text =
```

The following examples insert text:

Example	Value Inserted
'{first x y z}	x
'realm	Name of current realm
'arg1	Value of arg1 argument in macro
'abc	Value of variable abc
'"abc"	abc
''xyz'	xyz

Here is an example of line continuation:

```
(c-p) write This text is printed ' ↓
(c-p)` along with this stuff. ↓
This text is printed along with this stuff.
```

See Also

Topics: abbreviation, semantics, substitution

Topic: types — Realm: command-processor

Summary

Introduction to types

A type is a category of argument values accepted by the Command Processor (CP). Each command argument has a type associated with it that validates the value for that argument. The type has a checker function that validates a command argument.

The type checks for a particular kind of value, such as an address, a language expression, or an integer. If the argument value is of the specified kind, the CP accepts it and passes it to the command. If the argument value is not of the specified kind, the type checker rejects it and displays an error message.

See Also

Command: **help**

End of Chapter

Chapter 7
Character Commands

This chapter contains the help messages for the characters realm and for the commands in that realm. The realm help message is first, followed by the command help messages in alphabetical order.

The characters realm contains commands that let you insert into a command line characters that have special meaning to the Command Processor (CP).

Help messages in this chapter use the following conventions:

Message format	Each message has a Summary section. Some messages also have Description, Arguments (if the command takes arguments), Examples, and See Also sections.
Command syntax	Each command follows the regular syntax described under "Creating a Command Line" in Chapter 4.
Arguments	Each argument is classified as required, optional, or keyword in the Arguments subsection of the Summary Section.
Argument keywords	Each argument, regardless of its classification, has a keyword identifying it.
Argument values	The kind of value the argument accepts is listed to the right of the keyword.
<name>	The "To get" and "To do" subsections of the characters realm help message use angle brackets to indicate a value that you supply.

Realm: characters

Summary

Introduction to Mxdb character commands

Here is how to perform some common tasks:

To get

A list of characters help topics:	**help ,topic**
A list of character commands:	**help ,command ,r char**
Help on a specific topic:	**help** *<topic-name>*
Help on a specific command:	**help** *<command-name>*
More information about character commands, with a complete list:	**help ,v ,r char**

To do

Go back to the debugger realm	**realm debugger**

Examples

```
c:ch 123
c:code x
write Hello`char:comma all!
c-p:assi abc a`{c:left-sq-bracket}i
```

Description

The characters realm contains commands that let you insert into a command line characters that have special meaning to the Command Processor (CP). When you use a character from the characters realm, the CP does not interpret it specially; for example, the comma and the backquote do not have special syntactic meaning, and parentheses, brackets, braces, and quotation marks do not need to be paired.

You can abbreviate character names using standard CP abbreviation rules.

Here are some more tasks you can perform:

To get

A list of all help topics:	**help ,topic ,realm**
A list of all commands:	**help ,command ,realm**
A list of all realms:	**help ,realm**
Help on a specific realm:	**help** *<realm-name>*

To do

Display a backquote:	**char:backquote**
Display an apostrophe:	**char:single-quote**
Write a line feed:	**char:new-line**
Display character's ASCII code:	**char:code** *<character>*
Display any ASCII character:	**char:character** *<ascii-value>*

Following is a list of Mxdb character commands:

Command Name	Action
backquote	Display a backquote:
carriage-return	Write a carriage return (Ctrl-M)
character-from-code	Write an ASCII character
code-from-character	Display character's ASCII code
comma	Display a comma: ,
double-quote	Display a double quote: "
form-feed	Write a form feed (Ctrl-L)
left-curly-brace	Display a left brace: {
left-parenthesis	Display a left parenthesis: (
left-square-bracket	Display a left bracket: [
new-line	Write a new line (Ctrl-J)
null	Write a null character (Ctrl-@)
right-curly-brace	Display a right brace: }
right-square-bracket	Display a right bracket:]
right-parenthesis	Display a right parenthesis:)
semicolon	Display a semicolon: ;
single-quote	Display a single quote: '
space	Display a space character
tab	Write a horizontal tab (Ctrl-I)

Examples

The following command displays the character whose ASCII value is decimal 123:

```
(debug) c:ch 123 ↵
{
```

The **code-from-character** command displays the ASCII decimal value of the letter x:

```
(debug) c:code x ↵
120
```

The following command writes "Hello, all!" to your screen:

```
(debug) write Hello'char:comma all! ↵
Hello, all!
```

The following command assigns "a[i" to the CP variable abc:

```
(debug) c-p:assi abc a'{c:left-sq-bracket}i ↵
```

See Also

Commands: **write, cp:assign**
Topics: abbreviation, realms

Command: backquote — Realm: characters

Summary Display a backquote: ‘

See Also Topics: substitution, syntax

Command: carriage-return — Realm: characters

Summary Write a carriage return (Ctrl-M)

See Also Topic: syntax

Command: character-from-code — Realm: characters

Summary Write an ASCII character

Arguments Required:

code	A decimal integer

Examples

```
c:ch 80
char:char 114
characters:character-from-code 111
```

See Also Command: **char:code-from-character**

Command: code-from-character — Realm: characters

Summary Display character's ASCII code

Arguments Required:
character An ASCII character

Examples
```
c:co P
char:code r
characters:code-from-character o
```

See Also Command: **char:character-from-code**

Command: comma — Realm: characters

Summary Display a comma: ,

See Also Topic: syntax

Command: double-quote — Realm: characters

Summary Write a double-quote: "

See Also Topic: syntax

Command: form-feed — Realm: characters

Summary Write a form feed (Ctrl-L)

See Also Topic: syntax

Command: left-curly-brace — Realm: characters

Summary Display a left brace: {

See Also Topic: syntax

Command: left-parenthesis — Realm: characters

Summary Display a left parenthesis: (

See Also Topic: syntax

Command: left-square-bracket — Realm: characters

Summary Display a left bracket: [

See Also Topic: syntax

Command: new-line — Realm: characters

Summary Write a new line (Ctrl-J)

See Also Topic: syntax

Command: null — Realm: characters

Summary Write a null character (Ctrl-@)

See Also Topic: syntax

Command: right-curly-brace — Realm: characters

Summary Display a right brace: }

See Also Topic: syntax

Command: right-parenthesis — Realm: characters

Summary Display a right parenthesis:)

See Also Topic: syntax

Command: right-square-bracket — Realm: characters

Summary Display a right bracket:]

See Also Topic: syntax

Command: semicolon — Realm: characters

Summary Display a semicolon: ;

See Also Topic: syntax

Command: single-quote — Realm: characters

Summary Display a single-quote: '

See Also Topic: syntax

Command: space — Realm: characters

Summary Write a space character

See Also Topic: syntax

Command: tab — Realm: characters

Summary Write a horizontal tab (Ctrl-I)

See Also Topic: syntax

End of Chapter

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:
      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581-9973

   b) TELEPHONE – Call TIPS at (508) 870-1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1-4 Units	$5.00
5-10 Units	$8.00
11-40 Units	$10.00
41-200 Units	$30.00
Over 200 Units	$100.00

   If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1-$149.99	0%
$150-$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870-1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME _____	COMPANY NAME _____
ATTN: _____	ATTN: _____
ADDRESS _____	ADDRESS (NO PO BOXES) _____
CITY _____	CITY _____
STATE _____ ZIP _____	STATE _____ ZIP _____

Priority Code _____ (See label on back of catalog)

_____ _____ _____ _____ _____
Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (If applicable) _____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is _____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number _____ Expiration Date _____

_____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580

093-000706-00