Prime Computer, Inc.

PRIME

CPL
Rev. 19.3

The Programmer's Companion

```
&args treename
como -ntty
&if [exists %treename%] &then &
   &data ed product>installati
         bottom
         insert [date -full] I
         file
         &end
  &if [exists product>source>[
     copy product>source>[ent
          product>arc>[entryn
     &data ed product>install
          bottom
```

FDR7811-193

CPL PROGRAMMER'S COMPANION

REVISION 19.3

FDR7811-193

by

Alice Landy

This document reflects the software as of Master Disk Revision 19.3.

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

The Programmer's Companion is a series of pocket-size, quick reference guides to Prime software products.

Published by Prime Computer, Inc.
Technical Publications Department
500 Old Connecticut Path
Framingham, Massachusetts 01701

Credits

Editor	Irene Rubin
Designer	Susan Windheim
Typesetter	Datatext
Printer	Winthrop Printing Company, Inc.

TABLE OF CONTENTS

Printing history:

March 1984, First Printing

Command Format Conventions

UPPERCASE: Commands or keywords that must be entered literally are shown in uppercase letters.

lowercase: Lowercase letters identify variable arguments. When entering the command, the user substitutes an appropriate numerical or text value.

Abbreviations: Abbreviations for commands are shown in rust-colored letters, as in: LOGOUT.

Braces { }: Braces indicate a choice of arguments. At least one choice must be selected.

Square brackets [ ]: Square brackets printed in brown indicate that the word or argument enclosed is optional. Square brackets printed in rust indicate a function call and must be entered literally.

Hyphen -: A hyphen identifies a command line option, as in: SPOOL -LIST. Hyphens must be entered literally.

Ellipsis ...: An ellipsis means that the preceding option or argument can be repeated.

Angle brackets < >: Angle brackets are used literally to separate the elements of a pathname, as in:

```
<FOREST>BEECH>BRANCH537>TWIG43>LEAF4
```

CPL TERMINOLOGY

CPL

CPL is Prime's **Command Procedure Language.** It provides the ability to direct and control the flow of command execution under PRIMOS, thus allowing you to create programs that conditionally execute sequences of commands. An interpretive language, CPL uses high-level language features, such as branching, looping, and argument passing, to facilitate the creation of structured command scripts.

Statements

CPL programs are made up of statements.

A statement may be either:

- A PRIMOS command
- A sequence of PRIMOS commands, separated by semicolons
- A CPL directive plus its arguments

The maximum length of a CPL statement is 1024 characters.

Commands

CPL programs can contain any PRIMOS commands with the following three exceptions:

- COMINPUT (in any form)
- CLOSE ALL
- DELSEG ALL

The use of any of these commands will abort the CPL program.

Directives

A CPL directive is an instruction to the CPL interpreter. Each directive begins with an ampersand (&). You can use directives to establish values for arguments and variables, to cause conditional execution of commands, to perform debugging, and so on.

Variables

A variable in CPL, as in other high-level programming languages, has a name and a value. The name of the variable remains constant, but its value can change.

When the name of a variable is given in a CPL program, it is given without any extra characters before or after it. When the value of a variable is to be referenced, the name of the variable is given with percent signs before and after it, as in %NAME%.

Variable References

When interpreting a statement that contains variable references, the CPL interpreter substitutes a character string representing the current value of the variable for the character string representing the reference before the statement is executed.

Values may be of three kinds: integer, logical, or character string. Values may be supplied as constants or as expressions.

Expressions

Expressions are composed of operands and operators.

Operands

Operands can be literal character strings, variable references, or function calls.

Operators

Operators can be arithmetic, logical, or relational operators. They are listed in Table 1. Note that there is no concatenation operator in CPL. To concatenate two strings in CPL, simply juxtapose them with no intervening spaces. For example, to concatenate the values of two variables, you would give the string %FIRST_HALF%%SECOND_HALF%.

Arguments

An argument is a variable whose value is passed to the CPL program by the program's caller. Arguments in CPL are defined by the &ARGS directive.

Callers

The caller of a CPL program is the process that invokes the program. The caller can be:

- A user, running the program interactively
- A Batch job or phantom
- Another CPL program

Functions

CPL also uses built-in functions. Function calls are enclosed in brackets, as in [DATE].

When the CPL interpreter encounters a function call in a CPL statement, it evaluates the call. Then, before the statement is executed, it replaces the character string representing the function call with a character string containing the result of the evaluation. Function calls are evaluated *after* variables have been evaluated; therefore, variable references may be used inside function calls.

Function calls may also be nested. In this case, evaluation is performed from the inside out; that is, the innermost function call is evaluated first, and so on until the outermost function call is reached.

Quoted Strings

Placing quotation marks around either variable references or function calls prevents their evaluation.

Return

When a CPL program completes execution, it returns to its caller.

CPL OPERATORS

TABLE 1
CPL OPERATORS

ARITHMETIC OPERATORS

Operator	*Meaning*
+	Addition, unary plus
−	Subtraction, unary minus
*	Multiplication
/	Integer division (Result is truncated to integer, fractional remainder is dropped.)

LOGICAL OPERATORS

Operator	*Meaning*
&	And
|	Or
^	Not

RELATIONAL OPERATORS

Operator	*Meaning*
=	Equal
<	Less than
>	Greater than
< =	Less than or equal
> =	Greater than or equal
^ =	Not equal

CPL FORMAT RULES

▶ *Rule 1:* Each statement in a CPL file must start on a separate line.

A statement is either a PRIMOS command, a sequence of PRIMOS commands separated by semicolons, or a CPL directive plus its arguments. An argument may be either a PRIMOS command or another CPL directive with its argument(s).

The maximum length of a physical line is 160 characters. (This is also the maximum length allowed for a PRIMOS command.)

Statements may be split over more than one line by the use of the tilde (~). See Rule 3 for details.

The maximum length of a CPL statement is 1024 characters. This is true before, during, and after variable evaluation and function expansion. Thus, if you have a statement of 600 characters, and variable evaluation adds 600 characters more, your statement will become 1200 characters long. CPL will produce an error message, and the statement will not execute.

▶ *Rule 2:* A statement may start anywhere on the line.

We suggest that you indent CPL programs for ease of reading, as you would indent any structured program. There are no rules governing indentation.

An exception to this rule is a line of text within an &DATA statement. Any blanks in these lines are passed as part of the input line to the program invoked by &DATA.

▶ *Rule 3:* To continue a statement over two or more lines, place a tilde (~) at the end of each incomplete line.

If there is a blank between the tilde and the word that precedes it, or if the beginning of the next line is indented by one or more spaces, the contents of the two lines are separated by one or more spaces when the statement is evaluated.

For example:
```
BREAK ~
HERE
```

is read as:
```
BREAK HERE
```

If no space precedes the tilde and the next line starts in column 1, the two lines are concatenated with no space between them.

For example:
```
NO BREAK~
HERE
```

is read as:
```
NO BREAKHERE
```

▶ *Rule 4:* Comments may be included in CPL programs by preceding each comment with a slash-asterisk (/*).

Comments end at the end of the physical line on which they appear. They are never continued onto the next line.

Comments are not evaluated or passed on to the command processor. If a tilde appears at the end of a line containing a comment, then any non-comment material from that line is joined with any non-comment material from the next line when forming the CPL statement.

For example:

```
&IF %VAR% = 1 /*Comment ~
&THEN /*more comment ~
SEG MYFILE.SEG /*more comment
```

is read as:

```
&IF %VAR% = 1 &THEN SEG MYFILE.SEG
```

▶ *Rule 5:* Every CPL file ends with an &RETURN directive. If the user omits the &RETURN directive, it is supplied automatically by the CPL interpreter.

▶ *Rule 6:* Filenames for CPL programs follow Prime's standard rules for filenames. They must end with the suffix, ".CPL".

Filenames must not exceed 32 characters, including the .CPL suffix. Allowable characters are:

A–Z, 0–9, _ # $ – . * &

The first character may not be numeric, nor may it be a hyphen. The name "*" is not a legal name.

▶ *Rule 7:* Variable names must also follow standard rules.

Variable names may not exceed 32 characters in length. They may contain only the characters A–Z, 0–9, underscore (_), and dot (.).

Names of local variables must begin with a letter. Names of global variables must begin with a dot.

The CPL interpreter translates lowercase letters to uppercase.

▶ *Rule 8:* Any arithmetic, relational, or Boolean operators in a CPL expression must be preceded and followed by one or more spaces. Parentheses must also be preceded and followed by one or more spaces.

For example:
```
( 3 + 5 ) * 4
&IF %THIS% > %THAT%
```

Table 1 lists the operators recognized by CPL.

▶ *Rule 9:* Any string containing blanks or special characters must be placed inside single quotes when the string is used as the value of a variable.

Special characters are:

- Single quotes ('), which must be doubled inside the string.
- Commas (,)
- Square brackets ([ ])
- Semicolons (;)
- Percent signs (%)
- Hyphens (–) at the beginning of strings, when the string is not a CPL option argument.
- CPL expressions, if you do not want them evaluated.

Whenever you use a quoted string in CPL, the quotes are considered part of the string.

CPL provides built-in QUOTE and UNQUOTE functions to place quotes around strings and to remove quotes from strings.

CPL VARIABLES

Local and Global Variables

Local variables are defined (or created) inside a CPL program. They are known only to the program that creates them, and they disappear when that program terminates.

Global variables may be defined inside CPL programs, inside high-level language programs, or at command level. They are stored in a global variable file in a user directory. When the file is active, the variables it contains can be referenced or modified in any of the following ways:

- Interactively, at command level
- By any of your CPL programs
- By high-level language programs

Global variables survive program termination and logouts. Once defined, they last until you delete them.

Defining Variables

Local variables may be defined in three ways:

- By the &ARGS directive
- By the &SET_VAR directive
- By the SET_VAR command

Global variables may not be defined by the &ARGS directive. They may be defined in three ways:

- Within a CPL program, by the &SET_VAR directive or the SET_VAR command
- At command level, by the SET_VAR command
- From a high-level program, by the GV$SET routine

Variable Names

Variable names must be 32 characters or less in length. They may contain only the characters A–Z (uppercase or lowercase), 0–9, underscore (_), and dot (.).

Names of local variables must begin with a letter. Names of global variables must begin with a dot.

Variable Values

Variables may have one of three types of values:

- Character string
- Logical value
- Integer value

The value of a variable may be given as one of the following:

- A character string (up to 1024 characters, quoted if necessary)
- An integer ($-2^{31} + 1$ to $2^{31} - 1$)
- A logical value (TRUE, true, T, or t for "true"; FALSE, false, F, or f for "false")
- An expression that evaluates to any of the above values

Evaluation of Variables

A variable is referenced by enclosing its name in percent signs, as in %VARIABLE_NAME%. When a statement contains variable references, all references are replaced by their values before the statement executes.

Note

An exception to this rule occurs when you use the &EXPAND ON directive. This directive forces subsequent statements to be handed to the abbreviation preprocessor before either variable or function evaluation is performed. In this case, function calls will be handled at command level; variables will not be evaluated unless you force their evaluation with the function call [ABBREV –EXPAND %var%].

Variable evaluation is performed only once per statement. If the evaluation of %VAR1% produces the string %VAR2%, then %VAR2% is the value of that variable; the variable is not re-evaluated.

Placing a variable reference inside quotes prevents its evaluation.

If you want to place the result of a variable reference inside quotes, use the QUOTE function. For example:

```
&SET_VAR ANSWER := [ RESPONSE ~
  [ QUOTE %PROMPT%]]
```

Comparison of Variables

The operators that can be used to compare variables are:

```
>  >=  =  <=  <  ^=
```

Comparisons are done by the following rules:

- If either operand is a character string, a string comparison is done.
- If both operands are integers, an arithmetic comparison is done.
- If both operands are Boolean (or logical) values, an arithmetic comparison is done. (TRUE = 1 and FALSE = 0.)

CPL DIRECTIVES

The following directives can be used within CPL programs only.

▶ &ARGS [object-args] [option-args] [rest-arg]

Format for **object-args** is:

name [:[type][=default]][;...]

Format for **option-args** is:

name: -control-list [name[:[type] [=default]][;...].

Format for **rest-arg** is:

name:REST [=default]

Defines names (plus types, default values, and keywords, if desired) for arguments passed to a CPL program from the command line that executes the program. Except for REST arguments, **type** can be any type shown in Table 2. If **type** is not specified, the argument defaults to type **char.** If **default** is not specified, system defaults are assigned as shown in Table 2.

Object Arguments: Object arguments are positional. The first item on the command line, after the name of the CPL program, is taken as the value of the first argument; the second item is taken as the value of the second argument, and so on.

If more object arguments exist than are defined on the command line, default values are assigned to the remaining arguments. If more items exist on the command line than there are arguments defined, then one of two things happens:

- If an argument of type REST or UNCL has been defined, then all further items on the command line are taken as the value of that argument. (This allows you to pass PRIMOS options as arguments to a CPL program without having to quote them.)
- Otherwise, the last argument takes only one item as its value. All further items on the command line are ignored.

Option Arguments: Option arguments are not positional. If one of the options specified by **-control-list** is present anywhere on the command line, the first option in the control-list is passed to the CPL program. (Each option-name in the control-list must begin with a hyphen and contain at least one alphabetic character. If more than one option-name is specified, the option-names must be separated by commas.)

As the format statement shows, the definition of an option argument can include one or more arguments that will follow the option argument on the command line. These arguments are positional, in that they must follow immediately after the option-name on the command line, and will be interpreted in the order in which they appear in the &ARGS directive.

If an undefined option argument appears on the command line, it is assigned all the arguments between itself and the next option argument, or (if there is no other option argument) between itself and the end of the line.

REST Arguments: Only one argument of type REST can be specified. It must be the last argument specified in the directive. Its evaluation will halt parsing of the command line; everything remaining on the line will be assigned as the value of the REST argument.

Examples:

```
&ARGS TRUTH; BEAUTY; CHARM

&ARGS TRUTH:DEC;

&ARGS CHARM:CHAR;~
TR_FLAG:-TR,-T TRUTH:DEC~
BE_FLAG:-BE BEAUTY:~
TREE=A_UFD>FILE
```

TABLE 2
CPL ARGUMENT TYPES AND DEFAULTS

Argument Type	*Explanation*	*Default Value*
char	Any character string up to 1024 characters long, mapped to uppercase. (Default)	"" (the null string)
charl	Any character string up to 1024 characters long, no case shifting.	""
tree	A filename, directory name, or pathname, up to 128 characters long. The last element of the pathname (that is, the final file or directory name) can contain wildcard characters.	""
entry	A filename up to 32 characters long; can contain wildcard characters.	""
dec	A decimal integer.[1]	0
oct	An octal integer.[1]	0
hex	A hexadecimal integer.[1]	0
ptr	Pointer; a virtual address in the format "octal/octal" (segno/offset).[2]	7777/0 (the null pointer)
date	Calendar date in the formats described in the DATE command.	""
rest	The remainder of the command line.	""
uncl	All tokens not accounted for in the &ARGS picture.	""

[1]Numeric arguments must be within the range $-2^{31} + 1 \ldots 2^{31} - 1$.
[2]User specified default values are not supported for this data type.

▶ &CALL routine-name

Transfers control to the internal routine designated by **routine-name.**

Example:

```
&CALL THIS_ROUTINE
      .
      .
      .
&ROUTINE THIS_ROUTINE
```

▶ &CHECK expression &ROUTINE routine-name

Defines an error condition **(expression)** and a routine **(routine-name)** to handle the condition. When this directive is present, the CPL interpreter evaluates **expression** after executing each PRIMOS command. If **expression** is true, control passes to **routine-name .**

Example:

```
&CHECK %THIS_VAR% > %THAT_VAR% ~
&ROUTINE DISASTER
```

▶ &DATA statement

statement-1

.
.
.

statement-n
[&TTY]
[&TTY_CONTINUE]
&END

Statements 1 through **n** are treated as data or subcommands for user programs or PRIMOS utilities. The statement immediately following

&DATA must invoke the program or utility. All other statements between &DATA and &END are evaluated, and the results written into a temporary file. The program (or utility) is then invoked and information from the file passed to it, a line at a time, when called for.

Either the &TTY_CONTINUE or the &TTY directive can be used as the last statement in the &DATA group, immediately preceding the &END statement. These directives can be used conditionally (as in &IF statements) or unconditionally.

The two directives differ in where they send control when they execute. The &TTY_CONTINUE directive takes input from the command input stream, whether that be an interactive user, a command input file, or an &DATA group in another CPL program. The &TTY directive always sends control to the terminal. Thus, it cannot be used in programs that may run as Batch jobs or phantoms.

In both cases, control returns to the CPL program when the caller or user exits from the program or utility called by the &DATA group.

Example:

```
&DATA SEG
    VL PROG.SEG
    &IF %DEBUGGER_USED%~
    &THEN LO~
    *>BIN>NEW_PROG.BIN.DBG
    &ELSE LO~
    *>BIN>NEW_PROG.BIN
&END
```

▶ &DEBUG [options]

Enables debugging for the CPL procedure containing the &DEBUG directive. If given without options, &DEBUG is equivalent to &DEBUG &NO_EXECUTE &ECHO ALL. Options are:

Option	*Meaning*
&OFF	Turns off all debugging options. Initially all options are off.
&EXECUTE	Enables execution of PRIMOS commands.
&NO_EXECUTE	Suppresses execution of PRIMOS commands, but interprets CPL directives.
&ECHO { **ALL** / **COM** / **DIR** }	If ALL is specified, echoes PRIMOS commands and CPL directives. If COM is specified, echoes only PRIMOS commands. If DIR is specified, echoes CPL directives. (Default is ALL.)
&NO_ECHO { **ALL** / **COM** / **DIR** }	ALL cancels all echoing. COM cancels echoing of PRIMOS commands. DIR cancels echoing of CPL directives. (Default is ALL.)

&WATCH
[var1 ... var16]

Adds the specified variables to the watchlist. When the value of a watched variable is changed using the &SET_VAR directive (not the SET_VAR command), CPL reports this fact and the new value of the variable. At most, 16 variables can be on the watchlist. If no variables are specified, all variables in the CPL programs are watched.

&NO_WATCH
[var1 ... var16]

Removes the specified variables from the watchlist. If no variables are specified, watching is turned off completely.

Example:

```
&DEBUG &ECHO ALL &WATCH BERSERK_VAR
```

▶ **&DO [iteration]**
statement-1
.
.
.
statement-n
&END

Allows a group of statements to be used anywhere a single statement can be used. If **iteration** is present, allows conditionally repeated execution of the statements contained between the &DO and the &END. **iteration** can take any of the following forms:

1. **null** (no iteration)
2. **[&WHILE while] [&UNTIL until]**

3. var := start [&TO to] [&BY by]
   [&WHILE while] [&UNTIL until]
4. var &LIST list [&WHILE while]
   [&UNTIL until]
5. var &ITEMS items [&WHILE while]
   [&UNTIL until]
6. var := start &REPEAT repeat
   [&WHILE while] [&UNTIL until]

Examples:

```
&DO I := 1 &TO 3
  FTN ABC%I%.FTN
&END

&DO &WHILE [ NULL %A% ]

&DO &UNTIL [ NULL %A% ]

&DO A := 5 &TO 10

&DO A := 5 &TO 10 &BY 2

&DO A := 5 &BY 2 &TO 10

&DO A := 5 &TO 10 &WHILE ~
[ NULL %A_STRING% ]

&DO A := 5 &TO 10 &UNTIL ~
[ NULL %A_STRING% ]

&DO A &LIST %LIST_OF_NAMES%

&SET_VAR UNIT := 0
&DO A &ITEMS [ WILD A_UFD>@@.PL1G ~
-SINGLE UNIT ]

&DO A := 6 &REPEAT %A% * %A_CONSTANT%

&DO A := 6 &TO -100 &BY -2

&DO A := -1 &REPEAT %A% * -1 ~
&UNTIL [ LENGTH %A_STRING% ] > 10
```

▶ &EXPAND {ON / OFF}

Turns statement expansion on or off. (The default is OFF).

When expansion is turned on, the CPL interpreter passes each command in the CPL program to the PRIMOS abbreviation pre-processor for abbreviation expansion. (Abbreviations must have been enabled with the ABBREV command for this to work.) The commands are passed before variable evaluation, function evaluation, or execution occurs.

Directives are not passed to the pre-processor. Therefore, user-defined abbreviations cannot be used in CPL directives.

Example:
```
&EXPAND ON
```

▶ &GOTO label-name

Transfers control to the statement following the &LABEL **label-name** directive. (See **&LABEL,** below.)

Example:
```
&GOTO A_LABEL
      .
      .
      .
&LABEL A_LABEL
```

▶ &IF expression &THEN true-statement [&ELSE false-statement]

Evaluates **expression.** If **expression** is true, **true-statement** is executed. If **expression** is false, then:

- If &ELSE is present, **false-statement** is executed.
- If &ELSE is not present, control passes to the next statement in the CPL program.

Example:

```
&IF %I% > 5 &THEN TYPE I = %I%
            &ELSE TYPE %I% TOO SMALL
```

▶ &LABEL label-name statement

Defines a label, **label-name,** to which a &GOTO can go. **label-name** must be a character string that is a valid variable name. It may not contain variable references or function calls.

When the &GOTO is reached, control passes to the statement following the &LABEL directive.

The execution of the &LABEL directive is never conditional. Therefore, the &LABEL directive should never be used in an &IF, &ELSE, or &SELECT statement.

Example:

```
&LABEL A_LABEL
ATTACH BEECH
```

▶ &ON condition &ROUTINE routine-label

Defines an internal routine to act as a condition handler (or on-unit) for the defined **condition.** (See the **Subroutines Reference Guide** for a list of PRIMOS-defined conditions and an explanation of PRIMOS's Condition Mechanism.)

Examples:

```
&ON QUIT$ &ROUTINE QUIT_HANDLER

&ON BAD_INPUT &ROUTINE ~
BAD_INPUT_HANDLER
```

▶ &RESULT expression

Used only in CPL programs invoked as function calls, either from other CPL programs or from PRIMOS command level.

When the &RESULT directive is reached, **expression** is evaluated. This value is returned as the result of the function call when the &RETURN directive is reached.

Sample program (FACTORIAL.CPL):

```
&ARGS N:DEC
&IF %N% <= 0 &THEN &RESULT 0
   &ELSE &IF %N% = 1 &THEN &RESULT 1
      &ELSE &RESULT~
      [CALC [R FACTORIAL~
         [CALC %N% - 1 ]] * %N% ]
&RETURN
```

▶ &RETURN [severity] [&MESSAGE text]

Halts execution of the procedure in which it occurs. Returns control to the procedure's caller.

If &MESSAGE is present, displays **text** on terminal when control returns. If **severity** is present, returns its value as a severity code to the procedure's caller. **severity** must evaluate to an integer.

Examples:
```
&RETURN

&RETURN 1

&RETURN %SEVERITY$%

&RETURN &MESSAGE HELLO!

&RETURN 1 &MESSAGE OOPS!
```

▶ &REVERT condition

Disables the latest condition handler defined (by an &ON directive) for the named **condition.** (All handlers defined within a program are automatically reverted when the program terminates.)

Example:
```
&REVERT QUIT$
```

▶ &ROUTINE routine-name

Names and defines the entry point for an internal routine.

Example:
```
&ROUTINE QUIT_HANDLER
```

▶ **&SELECT test-expression**
&WHEN expr-1 [,... ,expr-n]
statement
&WHEN expr-1 [,... ,expr-n]
statement
.
.
.
[&OTHERWISE
statement]
&END

test-expression is evaluated and tested against **expr-1, expr-2,** in turn. When a match for **test-expression** is found, **statement** following the matching expression is executed.

If no match is found, then:

- If an &OTHERWISE directive is present, the statement following it is executed.
- If no &OTHERWISE directive is present, control passes to the statement following the &END of the &SELECT group.

Example:

```
&SELECT %WHAT_TO_DO%
    &WHEN ABC
        ATTACH BEECH
    &WHEN 6, %ONE_VAR% + %TWO_VAR%
        &RETURN
    &OTHERWISE
        RESUME NOT_ONE_OF_THOSE.CPL
&END
```

▶ &SET_VAR var-1 [, var-2, ... , var-n] := value

Sets the value of the named variables to **value.** The variables need not exist already.

Examples:

```
&SET_VAR THIS_VAR := THIS_STRING
&S THIS_VAR := THIS_STRING
&S A,B,C := 0
```

▶ &SEVERITY [level [action]]

Checks for severity codes other than 0 (where codes > 0 indicate errors and codes < 0 indicate warnings) after execution of each PRIMOS command. If a code matching **level** is found, takes the specified **action.**

level can be:

&ERROR	Ignores warnings, takes action on errors.
&WARNING	Takes action on both warnings and errors.

action can be:

&FAIL	Halts execution, returns a positive severity code to the routine's caller.
&IGNORE	Continues execution.
&ROUTINE **routine-name**	Passes control to the designated routine.

If neither **level** nor **action** is given, all severity codes are ignored. If no &SEVERITY directive is given, warnings are ignored and errors halt execution.

Examples:

```
&SEVERITY &WARNING &IGNORE

&SEVERITY &ERROR &ROUTINE FIX_IT

&SEVERITY &ERROR &FAIL

&SEVERITY
```

▶ &SIGNAL condition [&NO_RETURN]

Raises the condition **condition** and causes the CPL mechanism to search for a handler for that condition. If &NO_RETURN is specified, execution of the error-causing procedure cannot be continued.

Example: `&SIGNAL BAD_BUG &NO_RETURN`

▶ &STOP [severity] [&MESSAGE text]

Halts execution of the procedure in which it occurs. If this procedure is a routine, &STOP also halts execution of the program containing the routine, and any other active routines the program has. Control returns to the caller of the main program.

If **severity** is present, the specified severity code is returned to program's caller. The code indicates the success or failure of the program. **severity** must be an integer.

If &MESSAGE is present, **text** is printed at the caller's terminal.

Example: `&STOP 1 &MESSAGE OH, NO!`

COMMAND FUNCTIONS

The four types of command functions described below are used primarily in CPL programs. However, they can be used in PRIMOS command lines also.

Throughout the following descriptions, any function marked with ** quotes its results when appropriate.

Arithmetic Functions

▶ [CALC expression]

Evaluates arithmetic or logical expressions. Returns the string that results from the evaluation. Accepts logical operators: & (and), | (or), ^ (not); arithmetic operators +, –, *, /, unary +, unary –; and relational operators =, ^=, >, <, >=, <= All operators *must* be delimited by blanks. Arithmetic values are integer only; their range is from $-2^{31} + 1$ to $2^{31} - 1$.

For example: `[CALC 254 * 19]`
Returns: `4826`

Similarly: `[CALC 9 > 3]`
Returns: `TRUE`

▶ [HEX number]

Returns the decimal equivalent of a hexadecimal **number.**

```
For example:  [ HEX A ]
Returns:      10
```

▶ [MOD string1 string2]

Divides **string1** by **string2** and returns the remainder.

```
For example:  [ MOD 360 23 ]
Returns:      15

Similarly:    [ MOD 10 123 ]
Returns:      10
```

▶ [OCTAL number]

Returns the decimal equivalent of an octal **number.**

```
For example:  [ OCTAL 10 ]
Returns:      8
```

▶ [TO_HEX number]

Returns the hexadecimal equivalent of a decimal **number.**

```
For example:  [ TO_HEX 15 ]
Returns:      F
```

▶ [TO_OCTAL number]

Returns the octal equivalent of a decimal **number.**

For example: `[ TO_OCTAL 8 ]`
Returns: `10`

String-Handling Functions

▶ [AFTER string find-string] **

Returns the part of **string** that appears after the first occurrence of **find-string.** Returns the null string if **find-string** is not in **string** or is at the end of **string.**

For example: `[ AFTER ABCDE D ]`
Returns: `E`

▶ [BEFORE string find-string] **

Returns the part of **string** that appears before the first occurrence of **find-string.** Returns **string** if **find-string** is not in string; returns the null string if **find-string** is at the beginning of **string.**

For example: `[ BEFORE ABCDE C ]`
Returns: `AB`

▶ [INDEX string find-string]

Returns an integer representing the starting position of a substring (**find-string**) within **string.**

For example: `[ INDEX ABCDE DE ]`
Returns: `4`

▶ [LENGTH string]

Returns the number of characters in **string.**

For example: `[ LENGTH This is a test ]`
Returns: `14`

▶ [NULL string]

Returns TRUE if **string** has no text characters, and FALSE otherwise.

```
For example:  [ NULL [ WILD @.CPL ] ]
Might return: TRUE
```

▶ [QUOTE string1 [string2 ...stringn]]

Adds an outer pair of quotes and doubles the quotes already inside the given strings. Prevents misinterpretation of special symbols.

```
For example:  [ QUOTE xy'|'z ]
Returns:      'xy''|''z'

Similarly:    [ QUOTE 'abc''de''fg' ]
Returns:      '''abc''''de''''fg'''
```

▶ [SEARCH string1 string2]

Returns the index of the first character in **string1** that appears in **string2.** Returns 0 if no character from **string1** appears in **string2.**

```
For example:  [ SEARCH abc9def 3692 ]
Returns:      4
```

▶ [SUBST string1 string2 string3] **

Substitutes **string3** for **string2** wherever **string2** occurs within **string1.** Returns the altered **string1.**

```
For example:  [ SUBST abccabccab cc @ ]
Returns:      ab@ab@ab
```

▶ [SUBSTR string start-position [num-chars]] **

Returns a substring of **string** that begins at position **start-position** and extends for length **num-chars.** If **num-chars** is omitted, the substring runs from **start-position** to the end of **string. Start-position** and **num-chars** (if given) must be positive integers.

For example: `[ SUBSTR ABCDE 3 2 ]`
Returns: `CD`

▶ [TRANSLATE string1 string2 string3]

Replaces characters in one string with characters from another. TRANSLATE looks for **string3** characters in **string1,** replaces them with characters from **string2,** then returns the altered **string1.** If **string2** and **string3** are omitted, TRANSLATE converts all **string1** characters to uppercase, then returns **string1.** If **string3** alone is omitted, the ASCII collating sequence is used for **string3.**

For example: `[ TRANSLATE abc 345 cba ]`
Returns: `543`

▶ [TRIM string [-LEFT / -RIGHT / -BOTH] [char]] **

Removes a given character from the left, right, or both sides of a given string. If you do not specify a side, TRIM assumes –BOTH. If you omit **char,** TRIM removes blanks.

For example: `[ TRIM BBBABCBBB -BOTH B ]`
Returns: `ABC`

▶ [UNQUOTE string]

Removes one outer pair of quotes from around **string** and changes all remaining pairs of quotes to single quotes.

```
For example:  [UNQUOTE '''xx''''yy''']
Returns:      'xx''yy'
```

▶ [VERIFY string1 string2]

Returns an integer representing the position of the first character in **string1** that does **not** appear in **string2.** Returns 0 if all characters in **string1** appear in **string2.**

```
For example:  [VERIFY 1298s8 ~
              0123456789]
Returns:      5
```

File System Functions

▶ [ATTRIB pathname att [-BRIEF]]

Returns information about the file in **pathname. att** must be specified and must be one of the following keywords:

-TYPE	Returns the file type of **pathname** (ACAT, SAM, DAM, SEGSAM, SEGDAM, UFD, or UNKNOWN)
-DTM	Returns the date and time when the object was last modified, in the form 83-05-31.13:24:29.Tue
-DTB	Returns the date and time the object was last backed up by the BACKUP utility
-LENGTH	Returns the file length (in halfwords)

The –BRIEF option suppresses most error messages.

For example: `[ATTRIB TWIG -TYPE]`
Might return: `SAM`

Similarly: `[ATTRIB TWIG -LENGTH]`
Might return: `86`

▶ [DIR pathname [–BRIEF]] **

Returns the directory portion of **pathname.** Returns "*" if **pathname** is a simple filename. The –BRIEF option suppresses most error messages.

For example: `[DIR BEECH>BRANCH1>TWIG]`
Returns: `BEECH>BRANCH1`

▶ [ENTRYNAME pathname]

Returns the entryname portion of **pathname** — that is, the portion following the final >. If the > character does not occur in **pathname,** returns the entire **pathname.**

For example:
`[ENTRYNAME>BEECH>BRANCH1>TWIG]`
Returns: `TWIG`

▶ [EXISTS pathname [type] [-BRIEF]]

Returns TRUE if **pathname** exists and matches **type** specified; otherwise, returns FALSE. **type** can be:

-ANY
Any type is acceptable.

-ACCESS_CATEGORY
Must be an access category.

-FILE
Must be a file.

-DIRECTORY
Must be a directory.

-SEGMENT_DIRECTORY
Must be a segment directory.

If **type** is omitted, -ANY is assumed. The -BRIEF option suppresses most error messages.

For example: `[EXISTS TWIG]`
Might return: `TRUE`

▶ [GVPATH]

Returns the pathname of your active global variable file. GVPATH returns -OFF if you have no active or defined global variable file.

For example: `[GVPATH]`
Might return: `<FOREST>BEECH>GVARS`

▶ [OPEN_FILE pathname status-var -MODE m]

Opens the file **pathname** for reading or writing on some available file unit, then returns the unit number. **m** can be:

R	Read only.
W	Write only.
WR	Read and write.

If –MODE **m** is omitted, the file is opened for reading. The variable whose name is **status-var** is set to 0 if the file is opened successfully, and to a positive nonzero value otherwise. It must be a global variable if the function is invoked at command level; it can be global or local if the function is invoked inside a CPL program.

▶ [PATHNAME path [–BRIEF]]

Returns the full pathname of **path.** [PATHNAME *] gives the full pathname of the current directory. If any of the intermediate directories do not exist, an error message appears. The –BRIEF option suppresses most error messages.

```
For example:    [ PATHNAME BRANCH1 ]
Might return:   <FOREST>BEECH>BRANCH1
While:          [ PATHNAME * ]
Would return:   <FOREST>BEECH>BRANCH1>*
```

▶ [READ_FILE unit status-var [–BRIEF]] **

Reads a line from the file opened on **unit** and returns the quoted line as its value. **status-var** is set to 0 if the operation is successful, to 1 if end of file is reached, or to some other positive nonzero value otherwise. The –BRIEF option suppresses most error messages.

▶ [WILD wild-name-1 [...wild-name-n] [options] [-SINGLE unit-var] [-BRIEF]]

Returns a list of all names within a directory that match one or more wildcard names. Without the -SINGLE option, returns a blank-separated list of file system objects that match the **wild-names** and option arguments. **wild-name-1** through **wild-name-n** are wildcard names. **wild-name-1** can be a pathname; the others cannot. **options** can be any combination of:

-BEFORE date
Matches only the objects last modified before **date.**

-MODIFIED_BEFORE date
Same as -BEFORE date.

-AFTER date
Matches only the objects last modified on or after **date.**

-MODIFIED_AFTER date
Same as -AFTER date.

-BACKEDUP_BEFORE date
Matches only objects saved by BACKUP before **date.**

-BACKEDUP_AFTER date
Matches only objects saved by BACKUP on or after **date.**

-FILE
Matches files only.

-DIRECTORY
Matches directories only.

-SEGMENT_DIRECTORY
Matches segment directories only.

–ACCESS_CATEGORY
Matches access categories only.

–RBF
Matches ROAM files only.

The –BRIEF option suppresses most error messages.

With the –SINGLE option, WILD returns names one at a time, rather than listing them. Use –SINGLE when you think WILD's list might overrun its limit of 1024 characters, or when it is more convenient to deal with filenames one at a time. Set **unit-var** to 0 before using the WILD function. WILD uses **unit-var** to store the number of the file unit on which it opens the directory for reading. The directory remains open until all matching names have been returned. WILD returns the true null string when no entries are matched, or when, in –SINGLE mode, the end of the directory is reached.

▶ [WRITE_FILE unit text]

Strips one layer of quotes from **text** and writes **text** (as a new line) to the file open on **unit.** Returns 0 if the operation is successful, a positive nonzero integer otherwise.

Miscellaneous Functions

▶ [ABBREV –EXPAND text]

Returns the expanded form of a current abbreviation, named in **text.**

▶ [CND_INFO flag]

Allows a CPL condition handler to examine the condition information of the most recent condition on the stack. Returns the information requested by **flag** as follows:

-NAME
Returns the name of the condition. Returns $NONE$ if no condition name is on the stack.

-CONTINUE_SWITCH
Returns the Boolean value of the continue-to-signal switch. Returns FALSE if no condition frame exists.

-RETURN_PERMIT
Returns the Boolean value of the return-permitted switch. Returns FALSE if no condition frame exists.

▶ [DATE [option]] **

Returns the current date/time in a variety of formats. If **option** is omitted, the date only is returned: 83-05-31. The other possibilities are:

-FULL	83-05-31.13:24:49.Tue
-USA	05/31/83
-UFULL	05/31/83.13:24:49.Tue
-DAY	31
-MONTH	May
-YEAR	1983
-TIME	13:24:49
-AMPM	1:24 PM
-DOW	Tuesday
-CAL	May 31, 1983

```
-TAG        830531
-FTAG       830531.132449
-VFULL
31 May 83 13:24:49 Tuesday
-VIS        31 May 83
```

▶ [GET_VAR expr]

Returns the value of the variable name given by **expr.** Returns $UNDEFINED$ if the variable named by **expr** has not been defined.

▶ [QUERY text [default] [-TTY]]

Prints **text,** followed by a question mark, on your terminal output stream. (If **text** is null, no text appears.) Use quotes around **text** and **default** if they contain special characters or embedded blanks; these quotes are stripped before printing. After **text** appears, answer by typing YES, Y, OK, NO, N, or null. (You can use uppercase or lowercase letters.) QUERY returns TRUE if the answer was YES or OK, and FALSE if it was NO. A null answer returns the **default** value. If this has not been specified, it is assumed to be NO.

The -TTY option forces QUERY to take input from the terminal. A CPL program with the -TTY option cannot be executed as a Batch job or phantom. If you omit -TTY, QUERY takes its response from the command input stream.

▶ [RESCAN string]

Returns a string produced by stripping one level of quotes from **string** and evaluating any function calls or variable references that no longer appear in quotes.

▶ [RESPONSE text [default] [-TTY]] **

Prints **text,** followed by a colon, on your terminal output stream. (If **text** is null, no text appears.) Use quotes around **text** and **default** if they contain special characters or embedded blanks; these quotes are stripped before printing. RESPONSE reads the command input stream for your reply, or takes it from the terminal if you use the -TTY option. (Do not use -TTY with Batch jobs or phantoms.) RESPONSE then returns your input as the value of the function. If a null reply is entered, **default** is returned. If **default** is omitted, the null string is assumed.

CPL-RELATED COMMANDS

Four PRIMOS commands handle global variables:

DEFINE_GVAR	Activates or deactivates a global variable file.
DELETE_VAR	Deletes global variables.
LIST_VAR	Lists names and values of global variables.
SET_VAR	Sets the value of a variable.

▶ **DEFINE_GVAR** { **pathname** [**–CREATE**] / **–OFF** }

Activating a File: The format, "DEFINE_GVAR pathname –CREATE" creates and activates a new global variable file. The format "DEFINE_GVAR pathname" activates an existing file. You must have LURW rights to open or create a global variable file. If the directory containing a global variable file is password-protected, you must provide the full pathname (including the diskname) within the

DEFINE_GVAR command. You must have an active global variable file in order to define or refer to global variables.

Deactivating a File: You can deactivate a global variable file in any of three ways:

- By giving the command "DEFINE_GVAR –OFF"
- By giving the command "DEFINE_GVAR pathname", which deactivates the current file and activates the new one
- By logging out

▶ DELETE_VAR id-1 [. . . id-n]

Removes the specified variables from an active global variable file. **id-1** through **id-n** may be:

- Names of global variables
- Wildcards
- Variable references or function calls that evaluate to the names of global variables.

▶ LIST_VAR [name-1 . . . name-n]

Lists the specified global variables, if they are contained in the active global variable file.

name-1 through **name-n** may be either variable names or wildcards. If no names are given, LIST_VAR lists all the variables in the active file.

▶ SET_VAR name [:=] value

Creates and/or sets the value of a global variable.

name is any legal variable name, up to 32 characters long. Names of global variables must begin with a dot (.).

value can be any one of the following:

- A character string, up to 1024 characters long. (Lowercase characters are not converted to uppercase.) If the string contains special characters, it must be enclosed in single quotes. These quotes are included in the character count.
- A numeric character string representing an integer between the values of $-2^{31} + 1$ to $2^{31} - 1$.
- A character string consisting of the logical value TRUE or FALSE (the forms TRUE, T, true, t, FALSE, F, false, and f are acceptable).

The assignment symbol (:=) is optional.

Note

The SET_VAR command may be used both interactively and within CPL programs. However, since the &SET_VAR directive is faster than the SET_VAR command and can produce debug information on watch-list variables, we recommend the use of the directive within CPL programs.

Two other PRIMOS commands, RSTERM and TYPE, are often used within CPL programs to perform terminal-related operations.

▶ RSTERM [-READ] [-WRITE]

Resets terminal input (READ) and/or output (WRITE) buffers. If no arguments are given, empties both buffers. Often used by condition handlers for the QUIT$ condition.

▶ TYPE text

Outputs **text** at the user terminal or into a command output file. **text** may contain embedded blanks, variables, or function calls.

CPL-RELATED SUBROUTINES

Two subroutines allow high-level programs to access global variables:

GV$SET	Creates and/or sets the value of a global variable.
GV$GET	Retrieves the value of a global variable.

Datatypes

These routines use the PL/I data types CHAR(*) VAR and FIXED BIN. To use them in COBOL and FORTRAN programs, use the following datatype conversions.

FORTRAN: The FORTRAN equivalent of CHAR(*) VAR is an INTEGER*2 array. The first element of the array stores the length of the string to be passed. The rest of the array contains the string, two characters per element.

The FORTRAN equivalent of FIXED BIN is INTEGER*2.

COBOL: The COBOL equivalent of CHAR(*) VAR is a record structure. The first element (datatype COMP) contains the length of the character string to be passed; the rest of the structure (datatype PIC X(n)) contains the string itself.

The COBOL equivalent to FIXED BIN is COMP.

Note

Before calling either the GV$SET or the GV$GET subroutines, make sure you have used the PRIMOS command DEFINE_GVAR to define your global variable file.

▶ GV$SET

GV$SET sets the value of a global variable. Its calling sequence is:

```
DCL GV$SET ENTRY (CHAR(*) VAR,
  CHAR(*) VAR, FIXED BIN);

CALL GV$SET (var-name, var-value, code);
```

var-name (input argument) is the name for the global variable to be set. The name must follow the rules for CPL global variables. All letters must be in uppercase.

var-value (input argument) is the new value for the variable **var-name.**

code (output argument) is a return error code. Codes returned include:

- E$BFTS if the specified value is too big.
- E$UNOP if the global variable area is bad or uninitialized.
- E$ROOM if an attempt to acquire more storage by the variable management routines fails.

▶ GV$GET

GV$GET retrieves the value of a global variable. Its calling sequence is:

```
DCL GV$GET ENTRY (CHAR(*) VAR,
  CHAR(*) VAR, FIXED BIN, FIXED BIN);

CALL GV$GET (var-name, var-value, value-size,
  code);
```

var-name (input argument) is the name of the global variable whose value is to be retrieved. It must be in uppercase.

var-value (output argument) is the returned value of **var-name.**

value-size (input argument) is the length of the user's buffer **var-value** in characters.

code (output argument) is a return error code. Codes returned include:

- **E$BFTS** if the user buffer **var-value** is too small to hold the current value of the variable.
- **E$UNOP** if the global variable storage is uninitialized or in bad format.
- **E$FNTF** if the variable is not found.