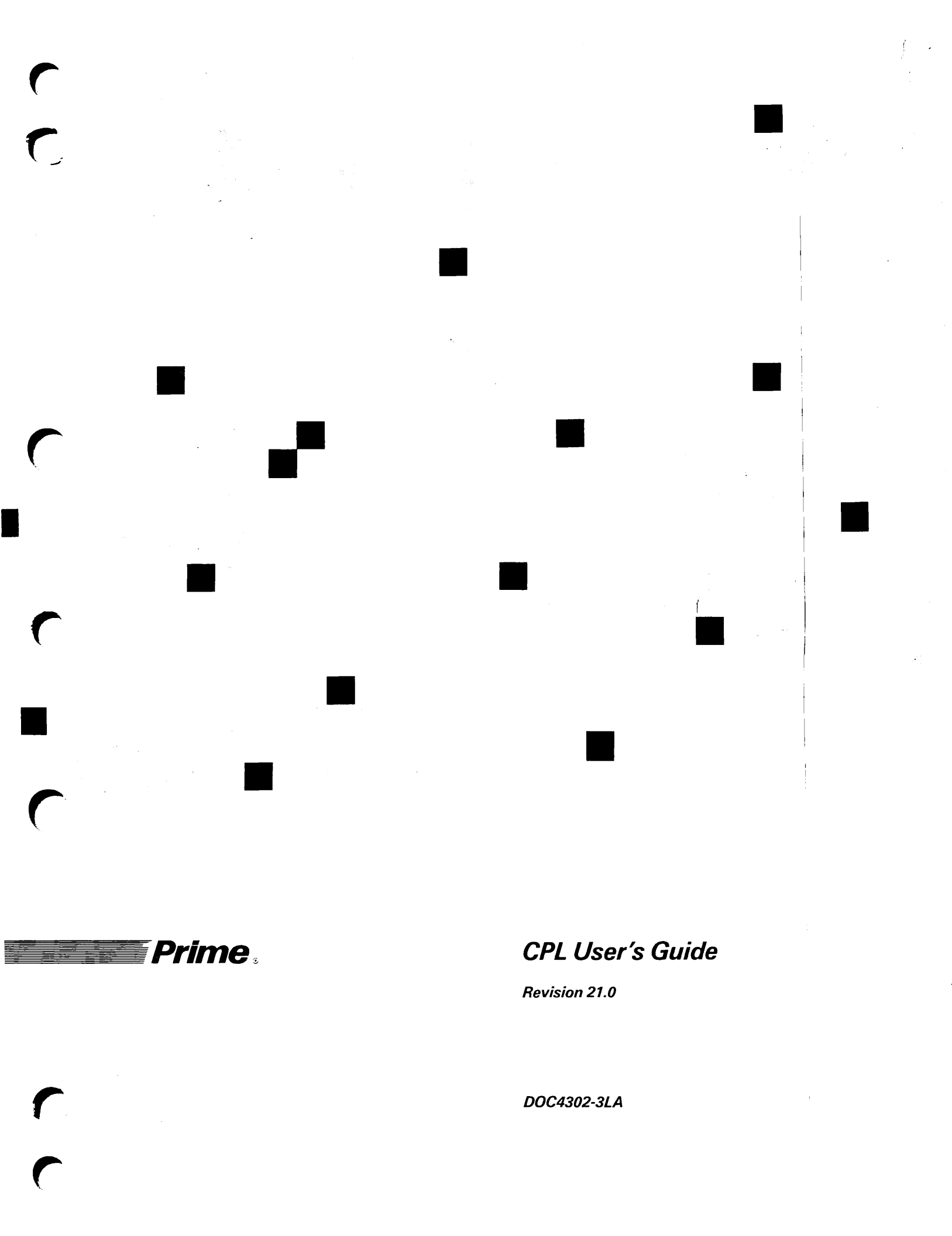


Prime®

CPL User's Guide

Revision 21.0

DOC4302-3LA

CPL User's Guide

Third Edition

Glenn Morrow

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 21.0. (Rev. 21.0).

Prime Computer, Inc., Prime Park, Natick, MA 01760

COPYRIGHT INFORMATION

PRINTING HISTORY

First Edition (IDR4302) January 1981 for Revision 18.1

Second Edition (DOC4302-190) July 1982 for Revision 19.0

Third Edition (DOC4302-3LA) July 1987 for Revision 21.0

CREDITS

Design: Carol Smith

Editorial: Barbara Fowlkes

Graphics Support: Mingling Chang

Illustration: Jerrie Kishpaugh

Illustration Support: Rosanne Dickey, Anna Spoerri

Document Preparation: Celeste Henry, Margaret Theriault

Production: Judy Gordon

Composition: Julie Cyphers, Anne Marie Fantasia

HOW TO ORDER TECHNICAL DOCUMENTS

Follow the instructions below to obtain a catalog, a price list, and information on placing orders.

United States Only: Call Prime Telemarketing, toll free, at 800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).

International: Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT CENTER

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)
1-800-541-8888 (within Alaska)
1-800-651-1313 (within Hawaii)
1-800-343-2320 (within other states)

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

Part III Full CPL

Appendices

About This Book

The CPL User's Guide provides both a tutorial and a reference guide for the Prime Command Procedure Language (CPL).

This book is divided into three parts.

- Part I introduces CPL and teaches the basics of CPL programming. These chapters describe how to create and run a CPL program, the basic statements used in virtually all CPL programs, and the formatting rules for all CPL statements. If you find your needs satisfied by the features provided in this basic subset of CPL, you need not read further.
- Part II presents an intermediate subset of CPL. Mastering this subset adds considerably to the power of the CPL programs you can write, while not introducing any great complexity. Many users will want to work with this subset.
- Part III presents the additional features that make up full CPL. In addition, it contains a fuller explanation of how CPL evaluates expressions, and a reference section on CPL's command functions. Although any user may want to refer to some part of this material, Part III as a whole is of most use to experienced programmers.

This manual assumes that you are familiar with general programming concepts, the PRIMOS® operating system, and the ED text editor. If you are not familiar with PRIMOS or ED, you should read

- *Prime User's Guide*
- *New User's Guide to EDITOR and RUNOFF*

Related Documentation

The following Prime publications are referred to in this manual:

Prime User's Guide (DOC4130-4LA)
New User's Guide to EDITOR and RUNOFF (FDR3104-101B)
PRIMOS Commands Reference Guide (DOC3108-6LA)
Subroutines Reference Guide, Volume II (DOC10081-1LA) and its update (UPD10081-11A)
Advanced Programmer's Guide, Volume II (DOC10056-2LA)

Some familiarity with structured programming concepts (such as DO loops and IF...THEN...ELSE constructs) is also helpful. If you haven't done structured programming before, you may want to refer to one of the many structured programming texts on the market. Two useful texts are

- Conway and Gries. *An Introduction to Programming: A Structured Approach.* Cambridge, MA: Winthrop, 1973
- Xenakis. *Structured PL/I Programming.* Boston, MA: Duxbury Press, 1979

Prime Documentation Conventions

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications.

Convention	*Explanation*	*Example*
UPPERCASE	In command formats, words in uppercase indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	`&ARGS`
lowercase	In command formats, words in lowercase indicate variables for which you must substitute a suitable value.	`&ARGS arg1; arg2`
Abbreviations in format statements	If an uppercase word in a command format has an abbreviation, the abbreviation is underscored.	ABBREV
Brackets [ ]	Brackets indicate a CPL function call. They must be entered literally.	`[EXISTS object]`
Single-line braces { }	Single-line braces indicate that the enclosed item is optional.	`DATE {option}`
Multiline braces	Multiline braces indicate that one of the enclosed items is required.	`&EXPAND {ON / OFF}`
Braces within brackets [{ }]	In CPL function calls, braces within brackets indicate an optional argument.	`[EXISTS pathname {type}]`

Convention	*Explanation*	*Example*
Ellipsis ...	An ellipsis indicates that the preceding item may be entered more than once.	`&ARGS name-1...; name-n`
Parentheses ( )	In command or statement formats, you must enter parentheses exactly as shown.	`DIM array (row, col)`
Hyphen –	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	`[DATE -USA]`
Italics	In text, italics indicate variables.	JEFF is the value of *filename*.

Bold italics in examples	In examples, user input is in bold italics but system prompts and output are not.	`OK, RESUME MY_PROG`
		`This is the output`
		`of MY_PROG.CPL`
		`OK,`

Bold type in examples	In CPL program examples, all statements are in bold typeface.	**`&DO`** **`  RESUME FIRSTPROG`** **`  RESUME SECONDPROG`** **`&END`**
Angle brackets in messages < >	In messages, text enclosed within angle brackets indicates a variable for which the program substitutes the appropriate value.	`The value <text>`

Part I
The Basic Subset

1
Introduction to CPL

This chapter describes what CPL is and what it can be used for. It tells you how to create and run a CPL program. It also provides a brief explanation of how the system executes a CPL program.

What Is CPL?

CPL is the Prime Command Procedure Language. A procedure language is used to invoke command operations. One example of a command operation is the command to execute a program; for example,

RESUME MYPROG.RUN

One common use of procedure language is to automate frequently-used sequences of command operations. Rather than invoke each command operation from your terminal, you specify the sequence of command operations once in a procedure language program. When you execute the procedure language program, it invokes all of these command operations in the sequence you specified.

CPL is an extremely sophisticated procedure language. Using CPL, you can execute user-written programs, invoke Prime system software, and issue PRIMOS commands. The CPL language includes directives that you can use to control decision making, branching, and looping. It has sophisticated features for the transfer of argument values. CPL also provides many function calls that perform specific operations.

CPL is easy to use. Non-programmers can create working CPL programs with minimal training. Programmers can draw upon powerful and flexible features of CPL to perform a wide variety of control and computational tasks.

How Might You Use CPL?

Suppose that you frequently compile three FORTRAN 77 programs. The commands that do this might be

```
F77 JEFF -B RICHS>BIN>JEFF.BIN
F77 DICK -B RICHS>BIN>DICK.BIN
F77 BARRY -B RICHS>BIN>BARRY.BIN
```

This is an annoying amount to type many times a day. But you can type it once into a text file (using Editor or EMACS) to create a CPL program (named, say, COMP.CPL). Then you can run the CPL program with the simple command

```
R COMP.CPL
```

to compile all three programs.

The convenience gained by creating programs composed exclusively of PRIMOS commands is just the beginning of what CPL offers. CPL is modeled on high-level algorithmic languages (such as PL/I and PASCAL). Thus, it also offers you the convenience of

- Variables
- Function calls
- Flow of control directives (such as &IF...&THEN...&ELSE, &GOTO, &SELECT)
- Error handling

By using these features, you can create sophisticated CPL programs. But before learning the details of the CPL language, you should understand how to create and run a CPL program.

Naming CPL Programs

You can write a CPL program using any text editor, such as ED or EMACS. CPL program files must have names ending in .CPL (for example, TEST.CPL, COMP.CPL). The .CPL suffix identifies the file as a CPL program to the RESUME, JOB, and PHANTOM commands. However, you usually do not have to specify the .CPL suffix when you run the program. (You may specify the suffix if you wish, as shown by the examples in this book.)

Running CPL Programs

CPL is an interpreted language. This means that each line of a CPL program is interpreted (put in machine-readable form) each time the program is run. Therefore, you do not have to compile, link, or load a CPL program; you simply write your CPL program into a file, then execute that file to run the program.

PRIMOS includes a special command, CPL, for running CPL programs. You can also use the other PRIMOS run commands, RESUME, PHANTOM, and JOB to run a CPL program. You can run CPL programs interactively using the RESUME or CPL commands. You can run a CPL program as a phantom using the PHANTOM command, or as a batch job using the JOB command. Thus, our sample program, COMP.CPL, can be run by any of the following commands:

- CPL COMP
- RESUME COMP.CPL

- PHANTOM COMP.CPL
- JOB COMP.CPL

For each command, you can supply the full filename (COMP.CPL), or the filename without its suffix (COMP). If you supply just the filename COMP, the four run commands assume a filename suffix. The CPL, PHANTOM, and JOB commands locate the file COMP.CPL and execute it as a CPL program. When you supply the filename COMP to the RESUME command, it looks for the file COMP with the .RUN, .SAVE, or .CPL suffix. RESUME first looks for an executable code file with the .RUN suffix. RESUME next looks for an executable code file with the .SAVE suffix. If neither COMP.RUN nor COMP.SAVE exist in the current directory, RESUME looks for the file COMP.CPL and executes it as a CPL program.

If the file COMP.CPL doesn't exist, the four run commands look for a file named COMP with no suffix. If COMP exists, the CPL command runs it as a CPL program. RESUME runs COMP as a runfile. PHANTOM and JOB run COMP as a command input file.

Running CPL Programs as Commands

If you use a CPL program frequently, you may wish to create a command that runs that CPL program. A command can be easily invoked from any attach point. You invoke a command by typing its name alone, with no run command, directory name, or suffix. You can establish a CPL program as a command in three ways:

- Place the CPL program in the system's command directory.
- Establish the CPL program's directory as a command directory.
- Establish an abbreviation that runs the CPL program.

CPL Programs in the System's Command Directory: The programs that carry out most PRIMOS commands are located in the system's command directory, CMDNC0. When you issue a command, one of these programs is run. You issue a command by just typing the name of the command. For example, when you type the command LD, you are actually running the program CMDNC0>LD.RUN.

You can also run user-written CPL programs that are in directory CMDNC0 by simply typing the name of the program. For example, you could execute CMDNC0>MYPROG.CPL by typing MYPROG.

On most systems, access to CMDNC0 is restricted to the System Administrator. Request that commonly-used CPL programs be placed in CMDNC0, especially those CPL programs that are run by many users. A CPL program in CMDNC0 can be executed as a command by any user. A CPL program in CMDNC0 should not have the same name as a .RUN or .SAVE file in CMDNC0.

CPL Programs in a User's Command Directory: You can execute a CPL program as a command if the name of its directory is listed in COMMAND$. COMMAND$ is a search list that contains a list of directories; PRIMOS automatically searches those listed directories for the desired program.

For example, if you set MYDIR in your COMMAND$ search list, you can execute MYDIR>MYPROG.CPL by typing MYPROG. PRIMOS searches each directory listed in COMMAND$ until it finds a program named MYPROG. It then executes that program. Therefore, no other program named MYPROG should be located in MYDIR or in any of the directories listed above MYDIR in the COMMAND$ search list. If MYPROG.RUN or MYPROG.SAVE exists in MYDIR, one of those programs is executed rather than MYPROG.CPL.

A CPL program in your directory can only be executed as a command by those users that have set that directory in their individual COMMAND$ search lists. A search list must be reset after each login or process initialization. For further information on setting the COMMAND$ search list, refer to the *Advanced Programmer's Guide, Volume II.*

CPL Program Abbreviations: If you use a CPL program frequently, you may wish to create an abbreviation to run the CPL program. An abbreviation, like a command, can be executed by typing its name alone, with no run command, directory name, or suffix.

You can use an abbreviation to execute several PRIMOS commands. For example, you could create the abbreviation RUNCPL that expands into the commands RESUME MYDIR>MYPROG.CPL and RESUME YOURDIR>TESTPROG.CPL. Typing RUNCPL executes both commands.

A user abbreviation is unique to that user. If both a command and an abbreviation exist with the same name, PRIMOS executes the abbreviation. Abbreviations are further described in the *Prime User's Guide.*

How Does CPL Work?

CPL has two parts: the language, and the interpreter. The CPL language allows you to write CPL programs that contain either a sequence of PRIMOS commands or a combination of PRIMOS commands and CPL directives. The commands give instructions to PRIMOS, or to one of its subsystems. The directives give instructions to the CPL interpreter itself. (PRIMOS never sees these directives; it sees only the commands that the interpreter passes to it.)

Instructions to the CPL interpreter are identified by special characters. In CPL, directives are preceded by ampersands (for example, &IF, &GOTO). Variable references are enclosed in percent signs (for example, %VAR%). Function calls are enclosed in brackets (for example, [NULL A]).

When you execute a CPL program, the CPL interpreter first evaluates variables and function calls and replaces them with their correct values. It then interprets and acts upon CPL directives. Finally, it passes the resulting PRIMOS commands to PRIMOS for execution. This means that not only can CPL execute a long sequence of PRIMOS commands pre-specified in the CPL program, but it also can modify the execution of those PRIMOS commands based on the current values of variables at the time you execute the CPL program. Let's take a closer look at how the interpreter accomplishes this.

The CPL Interpreter

When you run a CPL program, PRIMOS hands each line in turn to the CPL interpreter. If the line consists of a PRIMOS command (for example, F77 JEFF), the interpreter hands it to the PRIMOS command processor for execution. This is diagrammed in Figure 1-1.

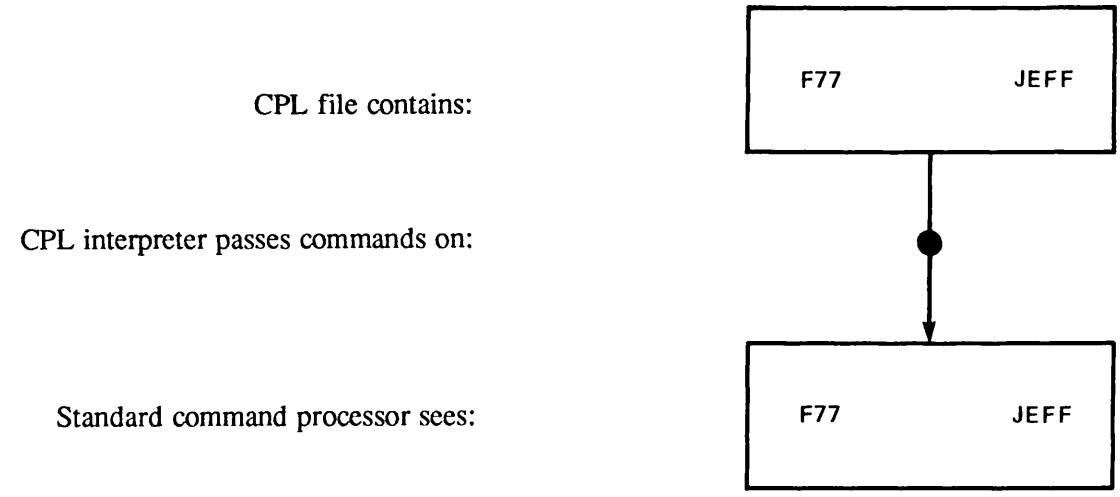


Figure 1-1
Command Execution via CPL

If a CPL command contains either variables or function calls, the CPL interpreter evaluates these operations before passing the command to PRIMOS. The CPL interpreter replaces each variable or function call with a literal value (a character string) before executing the CPL command.

Variables: The CPL interpreter replaces each variable in a CPL command with its current value before executing the command. (You set the current value of each variable elsewhere in the program.) For example, if JEFF is the current value of a variable called *FILENAME*, the CPL interpreter performs the substitution shown in Figure 1-2. When the CPL interpreter encounters this line, it identifies *FILENAME* as a variable reference by the percent signs that surround it. The CPL interpreter replaces this variable reference with its current value. In this case, the CPL interpreter removes the characters %FILENAME% from the command and replaces them with the characters JEFF. It then hands the modified command to the command processor for execution.

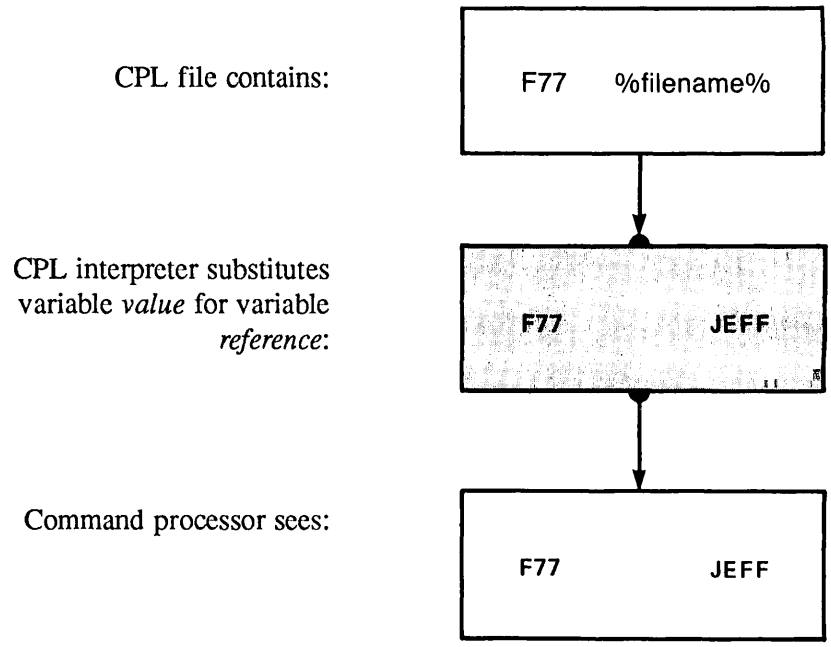


Figure 1-2
Execution of Command Containing a Variable

Function Calls: The CPL interpreter replaces each function call in a CPL command with a literal value before executing the command. First, the CPL interpreter executes the function call. When the function call completes, it returns a literal value (a character string). The CPL interpreter replaces the function call with this literal value, then hands the modified command to the command processor for execution.

Figure 1-3 shows an example of a function call in a CPL command. This CPL command spools a report to a printer. It uses the CPL function DATE to supply the current date as part of the name of the report.

When the CPL interpreter sees the square brackets that mark the function call, it evaluates the function. In this example, the CPL interpreter calls the DATE function. The DATE function determines the current date, converts the date to the desired format (in this case, the –TAG format: YYMMDD), and returns this value as a character string. The CPL interpreter substitutes the character string representing this date, 870623 (that is, June 23, 1987), for the character string [DATE –TAG]. The CPL interpreter then hands this command to PRIMOS for execution.

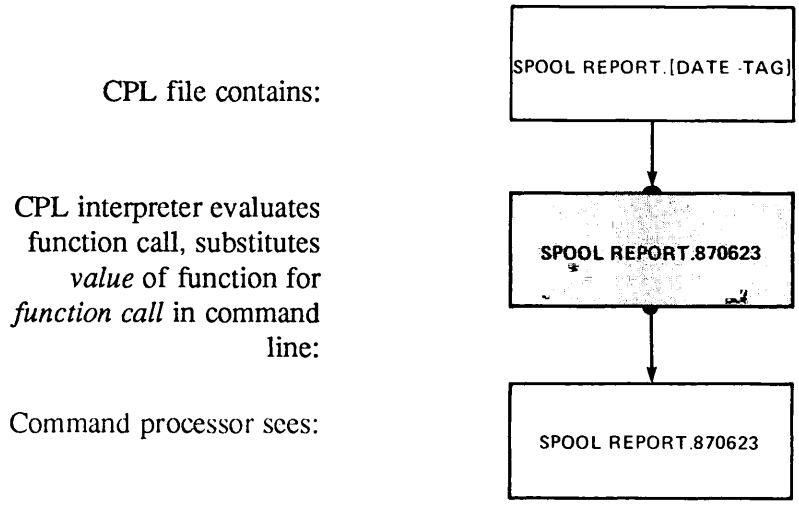


Figure 1-3
Execution of Command Containing a Function Call

Directives: CPL interprets a word beginning with an ampersand (&) as a CPL directive. If a line begins with a CPL directive, the interpreter recognizes the line as a CPL directive statement. For example,

```
&IF %A% > %B% &THEN F77 %FILENAME%
```

In this example, the CPL interpreter replaces the variable references %A%, %B%, and %FILENAME%, with their current values. The current values of these variables (3, 1, and JEFF) are established by other statements in the CPL program. The CPL interpreter then executes the &IF directive. The &IF directive tests to see if 3 is greater than 1. Since 3 is greater, the CPL interpreter executes the &THEN directive, passing the command F77 JEFF to the command processor for execution. This sequence of actions is diagrammed in Figure 1-4.

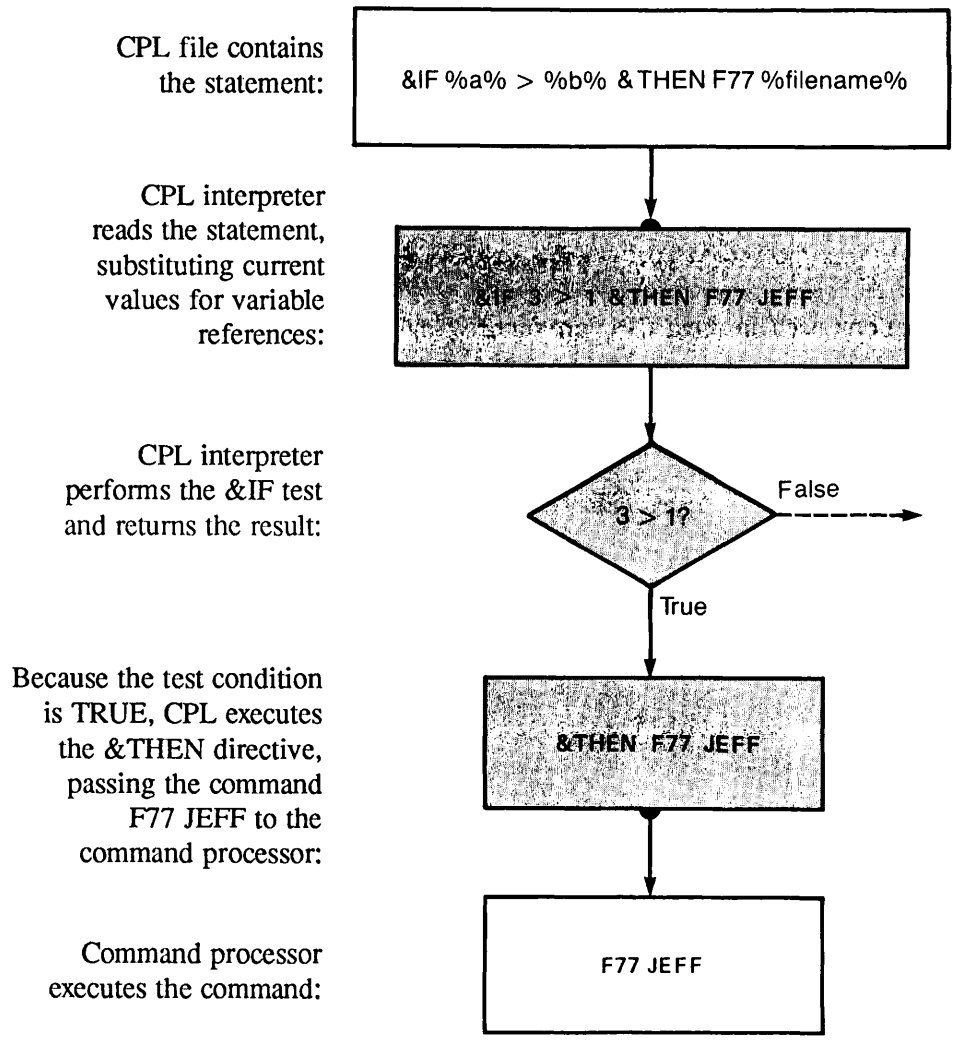


Figure 1-4
Execution of a Sample CPL Directive

Now that you are familiar with how the CPL interpreter operates, proceed to the chapters that follow, which describe how to write CPL programs. These chapters describe in much greater detail the features of the CPL language briefly mentioned in this chapter, as well as many other features of CPL.

As described in the preface, material in this book is divided into three levels of difficulty: basic CPL (for all users, including non-programmers); intermediate CPL (for users familiar with programming concepts); and full CPL (advanced concepts primarily of use to experienced programmers).

2
The Basics of CPL

This chapter describes the essential features of CPL. It describes the basic variables, directives, and functions used in most CPL programs. This chapter and the formatting rules found in the next chapter provide you with all the information necessary for writing most CPL programs.

PRIMOS Commands in CPL Programs

The simplest CPL programs are those composed entirely of PRIMOS commands. This type of CPL program consists of a sequence of PRIMOS commands, one per line. The example that follows is a complete CPL program that issues PRIMOS commands to open a COMOUTPUT file, display the current date, compile three FORTRAN 77 programs, then close the COMOUTPUT file.

```
COMO COMPILE.COMO
DATE
F77 THISFILE -XREF
F77 THATFILE -XREF -32I
F77 TOTHERFILE -DEBUG
COMO -E
```

The above example is a complete CPL program. CPL programs do not require any special statements identifying the file as a CPL program. The basic format of a CPL program is one statement per line. Formatting rules for CPL programs are further discussed in Chapter 3.

Which PRIMOS Commands Can You Use?

CPL programs can contain any of the following PRIMOS commands:

- All compiler commands: CBL, CC, F77, PASCAL, PL1G, PMA, RPG, and the like.
- All commands that execute programs. For example,

  R THISPROG.RUN
  R THATPROG.SAVE
  CPL ONEPROG.CPL
  PH OTHERPROG
  BASICV MYPROG

- Commands that do not invoke a subsystem or initiate a dialog. For example,

  ATTACH
  LISTF
  CREATE
  DELETE
  CNAME
  SET_ACCESS
  SET_DELETE
  SET_SEARCH_RULES
  SIZE

- Commands that invoke interactive subsystems. For example,

  ED
  BIND
  MAGNET
  SORT

  For commands of this type, you must either supply additional subcommands from the terminal each time you run the CPL program or specify these subcommands within the CPL program itself. If you want the CPL program to supply these subcommands, you must use CPL's &DATA directive, explained later in this chapter.

Which Commands Can't You Use?

Do *not* use the following commands in a CPL program:

- COMINPUT (in any form)
- CLOSE ALL
- DELSEG ALL
- ICE
- LOGIN or LOGOUT

Any of these commands abort execution of the program.

If you have existing COMINPUT files, you can easily convert them to CPL programs. For instructions on how to do so, see Appendix D.

Using Variables in CPL Programs

Although CPL programs composed entirely of PRIMOS commands can be extremely useful, most users want the flexibility that comes from using variable data in their commands. Variables are easily established in CPL. In their simplest form, a variable is established with the &ARGS directive, and references to the variable are indicated by percent signs. For example, the following CPL program (named COMP77.CPL) compiles any F77 source file:

```
&ARGS FILENAME
COMO %FILENAME%.COMO
DATE
F77  %FILENAME% -DEBUG
COMO -E
```

In this example, the &ARGS directive defines one variable, *FILENAME*. Each time you run the CPL program, you supply an argument value to be substituted for the variable *FILENAME* throughout the CPL program. You specify this argument value as part of the CPL program run command, following the name of the CPL program that you wish to execute. For example,

```
R COMP77.CPL JEFF
```

The &ARGS directive takes the character string JEFF and assigns it to the variable *FILENAME*. JEFF is now the value of *FILENAME*.

Therefore, each time the variable reference, %FILENAME%, is found in this program, the CPL interpreter substitutes the character string JEFF for the character string %FILENAME%. Thus, the command

```
COMO %FILENAME%.COMO
```

becomes

```
COMO JEFF.COMO
```

while the command

```
F77 %FILENAME% -DEBUG
```

becomes

```
F77 JEFF -DEBUG
```

Note that the variable, *FILENAME*, is not enclosed in percent signs when it is being defined in the &ARGS directive, but is enclosed in percent signs whenever it is referenced — that is, whenever its value, rather than its name, is wanted.

Notes

When you supply an argument value that contains letters to the &ARGS directive, the CPL program receives that value as all uppercase letters. For example, if you run the program with ***R COMP77.CPL Jeff***, the value of the %FILENAME% argument is JEFF.

When a variable reference is juxtaposed to another character string, with no blanks between them (as in %FILENAME%.COMO), the value of the variable is concatenated with the other string (as in JEFF.COMO). Two or more variable references may also be juxtaposed (as in %FILENAME%%FILENAME%). Again, a single string results (JEFFJEFF).

Multiple Arguments

A CPL program can contain multiple arguments. However, a CPL program should normally only contain one &ARGS directive. To specify multiple arguments for this single &ARGS directive, you list the variable names on the &ARGS directive line, separating the variable names with semicolons. For example,

```
&ARGS FILENAME; COMPILER
```

Now you can write a more general CPL file, called COMPILE_ALL.CPL, that can compile FTN, F77, or PL1G source files. It reads

```
&ARGS FILENAME; COMPILER
COMO  %FILENAME%.COMO
DATE
%COMPILER% %FILENAME% -64V -DEBUG
COMO -E
```

You can invoke this CPL program by typing

```
R COMPILE_ALL.CPL JEFF F77
```

which creates the command

F77 JEFF –64V –DEBUG

In general, arguments are defined by their position in the command line used to run the CPL program. In the above example, the first argument, JEFF, is assigned to *FILENAME*, the first variable in the &ARGS directive line. The second argument, F77, is assigned to the second variable, *COMPILER*. Giving the arguments in reverse order,

```
R COMPILE_ALL.CPL F77 JEFF
```

assigns F77 to *FILENAME* and JEFF to *COMPILER*. It is, therefore, extremely important that the sequence of arguments you supply when you invoke a CPL program is the same as the sequence of the variable names that follow the &ARGS directive within that CPL program.

Omitted Arguments

If you omit an argument from the command line used to invoke the CPL program, the CPL interpreter sets its value to the explicit null string. A null string is indicated by two single quotation marks (''). The PRIMOS command processor then removes the null string before executing the command.

In the above example, the command

```
R COMPILE_ALL.CPL TESTFILE
```

assigns the value TESTFILE to the first variable, *FILENAME*, and assigns the null string to the second variable, *COMPILER*. The resulting PRIMOS command first becomes

```
'' TESTFILE -64V -DEBUG
```

and then becomes

```
TESTFILE -64V -DEBUG
```

Since TESTFILE is not a legal command, PRIMOS returns you to command level with an error message.

Note that because arguments are positional, you can only omit the last argument(s) from a command line list of arguments. For this reason, list all essential arguments in the &ARGS directive before listing arguments that you might wish to omit when running the CPL program. CPL offers several ways to deal with null arguments. Some simple ones are explained later in this chapter. Others are described in Chapter 5, and in Chapter 13.

CPL's &ARGS directive can be expanded to

- Check the type of each supplied argument for accuracy
- Supply default values for omitted arguments
- Make arguments position-independent

The first two of these facilities are explained in Chapter 6. The third is explained in Chapter 13.

Decision Making in CPL Programs

When a CPL program contains only PRIMOS commands (or PRIMOS commands plus variables and the &ARGS directive), it is executed sequentially; that is, each command (each line of the file) is executed in turn.

Sometimes, however, you may want to alter the sequence in which the commands are executed. To alter the flow of control in this way, you use CPL's flow of control directives. The simplest and most important of these is the &IF directive.

The &IF Directive

The format of the &IF directive is

&IF test &THEN statement

test is a logical expression that can be determined to be TRUE or FALSE (for example, &IF A = B). *statement* is either a PRIMOS command or a CPL directive (for example, RESUME MYPROG, or &GOTO MAINROUTINE).

test may test variables, constants, functions, or expressions against each other. For example,

```
&IF %A% = 10                  (variable and constant)
&IF %A% = JEFF                (variable and constant)
&IF %A% > %B%                 (two variables)
&IF %A% < %B% + %C%           (variable and expression)
&IF %A% + %B% = %D% + 30      (two expressions)
&IF [LENGTH %A%] < 100        (function and constant)
```

The arithmetic and logical operators that can be used are shown in Table 2-1. They are explained in detail in the discussion of the CALC function in Chapter 12. Note that operators must be separated by at least one space from their operands.

test may also test the truth or falsity of logical functions (for example, &IF [NULL %A%]). This feature is explained later in this chapter.

How the &IF Directive Works: When the CPL interpreter reads an &IF directive, it substitutes current values for any variable references, expressions, or function calls it finds. Then it tests to see if *test* is true or false. If *test* is true, the interpreter executes the &THEN statement. If *test* is false, the interpreter skips over the &THEN statement and proceeds to the next line of the CPL program.

An Example: Suppose you compile a program frequently, but only occasionally want to spool the listing file. You can use an argument and the &IF directive to tell the CPL program whether or not to spool the listing file, as shown in the following program (called COMPSPOOL.CPL):

```
    /*This program compiles and optionally spools a F77 program.
    /*You type the value YES for the second argument
    /*to spool the listing file.
&ARGS FILENAME; SP
COMO %FILENAME%.COMO
DATE
F77 %FILENAME% -L %FILENAME%.LIST -XREF
    /*If desired, spool the program listing.
&IF YES = %SP%  &THEN SPOOL %FILENAME%.LIST -AT MS3
COMO -E
```

Note

As this program shows, you can use /* to place comments in CPL programs. For full rules governing comments, see Chapter 3.

If you give the command

```
R COMPSPOOL JEFF YES
```

then the test, YES = YES, is true, and the listing file, JEFF.LIST, is spooled. If you give the command

```
R COMPSPOOL JEFF
```

the test, YES = '', is false (the omitted argument is set to the null string). In this case, the listing file is not spooled. Instead, the CPL interpreter ignores the &THEN statement, and passes on to the next line in the program (in this case, COMO –E). Figure 2-1 shows the flow chart for these statements.

Table 2-1
CPL Operators

Operator	*Meaning*
Arithmetic Operators	
+	Addition, unary plus
-	Subtraction, unary minus
*	Multiplication
/	Integer division (result is truncated to integer, fractional remainder is dropped)
Logical Operators	
^	Not
&	And
|	Or
Relational Operators	
=	Equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
^=	Not equal

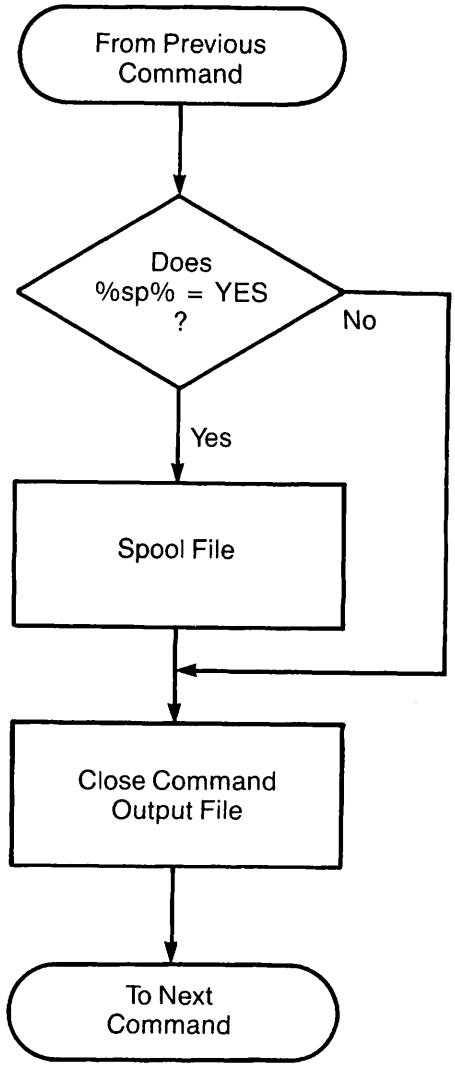


Figure 2-1
Sample &IF Statement

The &ELSE Directive

The &IF directive may be used by itself, as in the previous example, or it may be followed by the &ELSE directive. When used by itself, the &IF test tells the interpreter either to execute the &THEN clause and proceed to the next line of the program or to skip the &THEN clause and proceed to the next line of the program. In either event, the interpreter always executes the line after the &IF directive (unless the &THEN clause is executed and performs a jump to somewhere else in the program).

When the &IF and &ELSE directives are used together, they tell the interpreter to choose between two courses of action. Either the &IF test is true and the &THEN clause is executed, or the &IF test is false and the &ELSE clause is executed. In either case (barring a jump statement) the next statement executed is the statement after the &ELSE clause.

The format of the &ELSE directive is

&IF test &THEN statement-1
&ELSE statement-2

If *test* is TRUE, *statement-1* is executed. If *test* is FALSE, *statement-2* is executed. For example, suppose you compile many FTN programs and a few F77 programs. You may want a program (called COMPILE2.CPL) that looks like this:

```
&ARGS FILENAME; COMPILER
&IF %COMPILER% = F77 &THEN F77 %FILENAME% -DEBUG -32I
&ELSE FTN %FILENAME% -64V
TYPE 'Just ran a compilation'
```

If you give the command

```
R COMPILE2.CPL THISFILE F77
```

the &IF test (F77 = F77) is true, and THISFILE is compiled by the F77 compiler. If you give any other value for the COMPILER argument — or if you omit that argument altogether — the &IF test is false, and THISFILE is compiled by the FTN compiler. In either case, flow of control continues with the next statement after the &ELSE clause in the program. Figure 2-2 shows the flow chart for these statements.

Nested &IFs

&IF directives may be nested; that is, a &THEN or &ELSE statement may invoke another &IF directive. Each &IF directive must have its own &THEN clause. Nested &IF directives are discussed in Chapter 8.

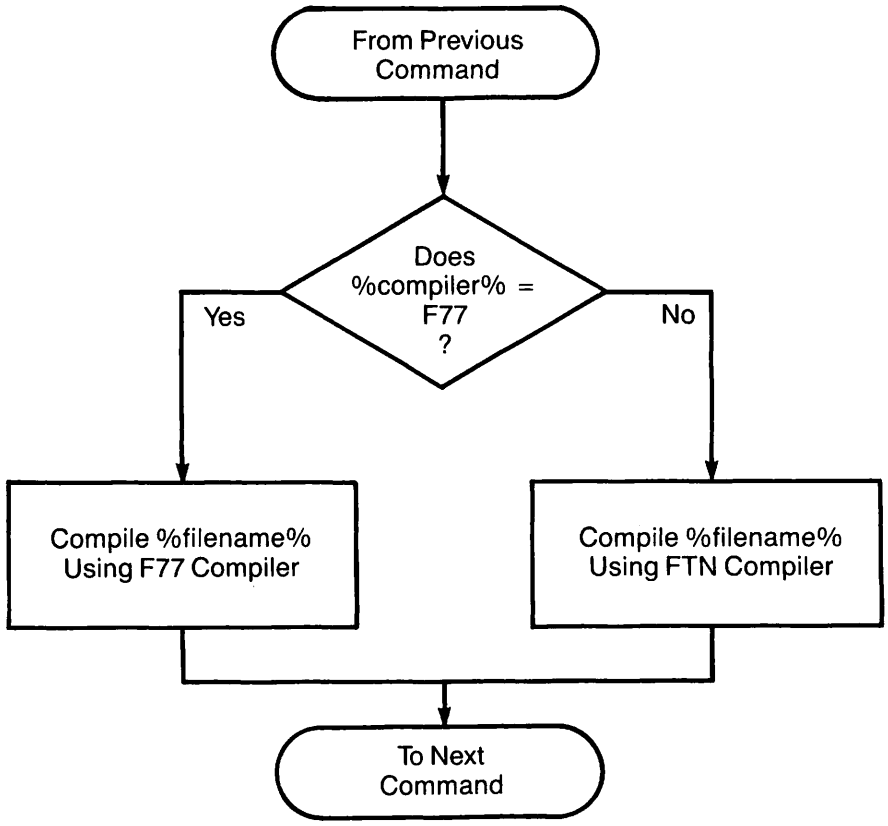


Figure 2-2
Sample &IF...&THEN...&ELSE Statement

&DO Groups

In the previous examples, the &THEN and &ELSE directives execute single commands. These directives may also execute groups of commands, by using the &DO and &END directives to mark the beginning and end of a &DO group of commands. The format for &DO groups is as follows:

```
&DO
   statement 1
   statement 2
   .
   .
   .
   statement n
   &END
```

Normally, each statement in a CPL program represents one action. In a &DO group, however, all the statements between the &DO and the &END represent a single action to the interpreter. Thus, instead of typing

```
&IF test &THEN statement-1
     -   &ELSE statement-2
```

you can type

```
&IF test &THEN &DO
               first-group-of-statements
               &END
         &ELSE &DO
               second-group-of-statements
               &END
```

For example, you can use &DO groups to modify an earlier sample program, COMPILE2.CPL, so that it compiles three modules instead of one:

```
&ARGS FILENAME; COMPILER
&IF %COMPILER% = CBL &THEN &DO
   CBL %FILENAME%1
   CBL %FILENAME%2
   CBL %FILENAME%3
   &END
 &ELSE &DO
   %COMPILER% %FILENAME%1 -64V
   %COMPILER% %FILENAME%2 -64V
   %COMPILER% %FILENAME%3 -64V
   &END
```

The statements inside the &DO group are indented for ease of reading. CPL allows indentation wherever you wish it; it never demands it.

The &GOTO Directive

The &GOTO directive permits you to jump from one part of a CPL program to another part of the program (either forward or backwards). CPL lends itself so well to structured programming that you may never need the &GOTO directive. However, if you do need or want it, here's how to do it:

1. Use the &LABEL directive to establish a label; for example, &LABEL HERE. A label is a line in your CPL program that identifies the destination of the jump performed by the &GOTO directive. The &LABEL directive must be on a line by itself, immediately preceding the statement or statements to be executed.
2. Use the &GOTO directive to transfer control from the current location to the specified &LABEL directive, for example, &GOTO HERE.

The format is

&GOTO label-name

.
.
.

&LABEL label-name

.
.

Once control has passed to the labeled statement, it continues sequentially until redirected by some other flow of control directive or halted by the end of the program. Here is an example of &GOTOs used with the &IF directive:

```
&ARGS FILENAME; COMPILER
COMO COMPILE.COMO
DATE
  /* Test for F77 compiler
&IF %COMPILER% = F77  &THEN &GOTO COMP_F77
  &ELSE &GOTO COMP_ANY
  /*
&LABEL COMP_F77 /* First alternative
  F77 %FILENAME% -B *>F77>%FILENAME%.BIN -L %FILENAME%.LIST -64V
  &GOTO WRAPUP
  /*
&LABEL COMP_ANY /*Second alternative
  %COMPILER% %FILENAME% -L %FILENAME%.LIST -64V
  /*
  /*Both alternatives finish off the same way
&LABEL WRAPUP
  SPOOL %FILENAME%.LIST
  COMO -E
```

When One CPL Program Runs Another

By using the RESUME or CPL command, one CPL program can run another. For example, a CPL program called ACCTS_UPDATE.CPL might contain the following commands:

```
COMO ACCTS_UPDATE.COMO
DATE
CPL NEW_ACCTS
CPL ACCTS_CLOSED
CPL ADDRESS_CHANGES
COMO -E
SPOOL ACCTS_UPDATE.COMO
&RETURN
```

The transfer of control that occurs when one program runs another is much the same as the transfer of control when a user runs a program. Each program begins with a run statement and ends with a return to the place from which the program was invoked (the &RETURN statement mentioned here is described later in this chapter).

For example, when you run ACCTS_UPDATE.CPL, the following actions occur:

1. You give the command

   CPL ACCTS_UPDATE

2. PRIMOS opens the file ACCTS_UPDATE.CPL on some available file unit, and accepts commands from it.
3. ACCTS_UPDATE finishes with a &RETURN directive.
4. PRIMOS closes the file and returns control to you.
5. You give the next command.

Similarly, when ACCTS_UPDATE.CPL invokes NEW_ACCTS.CPL, the following actions occur:

1. ACCTS_UPDATE passes the command CPL NEW_ACCTS to PRIMOS.
2. PRIMOS opens the file NEW_ACCTS.CPL on some available file unit, and accepts commands from it.
3. NEW_ACCTS ends with a &RETURN directive.
4. PRIMOS closes the file and returns control to ACCTS_UPDATE.
5. ACCTS_UPDATE passes its next command to PRIMOS.

When one CPL program runs another, each has (or may have) its own set of arguments and variables. If NEW_ACCTS needs any arguments, ACCTS_UPDATES must pass them to it, as in the following example:

```
CPL NEW_ACCTS WEST_BRANCH
```

In this case, the NEW_ACCTS program includes an &ARGS directive to receive a value for the BRANCH variable.

By using CPL programs to run other CPL programs, you can construct large, complex CPL programs from smaller independent modules. This type of structured, modular program design is much easier to test and maintain than a CPL program that relies upon numerous &GOTO directives.

Function Calls

Like other high-level languages, CPL provides built-in functions to simplify frequently made tests and computations. Functions are tools for quickly performing complex operations; for example, CPL provides a function that determines the current date.

The CPL interpreter locates and performs the appropriate function when you issue a **function call**. Function calls in a CPL program are easily identified; they are always enclosed in square brackets. Within the square brackets, a function call consists of the name of the function followed by its arguments (that is, [FUNCTION arg1 arg2]).

When a function call appears in a command or directive, the CPL interpreter first performs the function operation and substitutes the resulting character string for the function call. The interpreter then executes the command or directive.

If both variables and function calls are present in a statement, the variables are evaluated first and the function calls next. This allows the use of variables within function calls.

The following sections describe the NULL and EXISTS functions, two of the most commonly used CPL functions. CPL provides many other functions as well; the available CPL functions are described in Chapter 12.

The NULL Function

One of the most useful CPL functions is the NULL function. Its format is

[NULL var]

where *var* is any CPL variable.

The NULL function tests *var* to determine if it is a null character string. If *var* is null, the NULL function returns the character string TRUE. If *var* is not null, the NULL function returns the character string FALSE. Since the value of an omitted argument is the null string, the NULL function can be used in &IF directives to test for an omitted argument.

An Example: You can test for a null argument to set the home directory for some procedure. For example, a CPL program that begins

```
&ARGS DIR
&IF [NULL %DIR%] &THEN ATTACH MYDIR
                 &ELSE ATTACH %DIR%
```

allows you to make any desired ATTACH by specifying DIR; omitting DIR attaches you to your default choice (MYDIR).

Note

Remember that the &ARGS directive assigns values in positional order; that is, the first argument given is assigned to the first variable specified, and so on. Therefore, if you omit any one argument from a list of two or more, the last variable in the &ARGS directive is the one that gets set to the null string. If you omit two arguments, the last two variables are set to the null string, and so on. Therefore, when you use the NULL function to test for omitted arguments, always test first for the last argument in line. If it is not null, none of the others can have been omitted accidentally.

The EXISTS Function

The EXISTS function is a Boolean function that determines

- If a file system object exists
- If the file system object matches a specified type (file, directory, access category, or segment directory)

The format of the EXISTS function call is

[EXISTS pathname {type}]

pathname is the name or pathname of a file or directory.

type is an optional argument. You can omit *type* or specify one of the following values:

-ANY
-ACCESS_CATEGORY or -ACAT
-DIRECTORY or -DIR
-FILE
-SEGMENT_DIRECTORY or -SEGDIR

If *type* is present, then the EXISTS function returns the value TRUE if *pathname* exists and is of the right type. It returns the value FALSE if *pathname* does not exist or if it is of the wrong type.

For example, assume a directory that contains three files: PAYROLL.CBL, COMPILE_ALL.CPL, and PHONE_LIST. Assume that it also contains two subdirectories: WORKFILES and MEMOS. If you are attached to this directory, the function call

[EXISTS PHONE_LIST -FILE]

returns the value

TRUE

because PHONE_LIST is a file in the current directory. The function call

[EXISTS MEMOS -SEGDIR]

returns the value

FALSE

because MEMOS is not a segment directory.

If *type* is not present, the EXISTS function merely reports on the existence or non-existence of *pathname*. For example,

[EXISTS MEMOS]

returns the value TRUE, while

[EXISTS PAYROLL.F77]

returns the value FALSE.

Note

The EXISTS function does not use the PRIMOS search rules facility. Therefore, *pathname* must either be a full pathname or a filename. If *pathname* is a filename, EXISTS attempts to locate that file (or SEGDIR, ACAT, or subdirectory) within the currently attached directory. The EXPAND_SEARCH_RULES function, described in Chapter 12, permits you to invoke the search rules facility from a CPL program.

Examples: The first example checks to see if a file with the suffix .NEW has been written. If it has, the program calls ED to allow its user to edit this new file. If the new file does not exist, the program requests the older version:

```
&IF [EXISTS MEMO.NEW]  &THEN ED MEMO.NEW
    &ELSE ED MEMO
```

The second example uses the NOT symbol (^) to reverse the value returned by EXISTS. This program attaches to a specific directory. If the directory does not exist, the program creates it before doing the ATTACH:

```
&IF ^ [EXISTS SUBDIR] &THEN CREATE SUBDIR
ATTACH *>SUBDIR
```

Using CPL With Subsystems: &DATA Groups

Many Prime utilities, such as ED (a text editor) and BIND (a program linker), require subcommands to accomplish their function. Similarly, many user programs require that data be typed in from the terminal. CPL's &DATA directive allows CPL programs to supply the data or subcommands needed by these programs and utilities. A program run within a &DATA group is referred to as a subsystem.

&DATA groups resemble &DO groups in that both are groups of statements set off by an opening directive (&DO or &DATA), and a closing &END. In each case, the statements within the group are treated as a unit.

The format of the &DATA group is

&DATA command
statement-1
statement-2
.
.
.
statement-n
&END

command is the PRIMOS command that invokes the subsystem or user program; for example, BIND *filename*.

statement-1 through *statement-n* represent the commands or data to be passed to the subsystem or user program. As with all CPL statements, they may include variables, function calls, and directives.

The &END statement, on a line by itself, ends the &DATA group.

Note

> Commands within a &DATA group differ from other CPL commands in that blank lines and blank spaces used for indentation are not ignored. These blanks are passed on to the subsystem or user program. This permits you to enter a carriage return when one is required by the subsystem. Comments within a &DATA group are not passed on to the subsystem, unless the comment is prefaced by a tilde (~).

Here is an example of a CPL program that compiles, loads, and executes a PL/I program:

```
&ARGS FILENAME
PL1 %FILENAME% -DEBUG -B %FILENAME%.BIN  /*Compile program
/*
&DATA BIND                               /*Invoke BIND
  LOAD %FILENAME%.BIN                    /*Provide BIND commands
  LI PL1LIB                              /*via &data directives
  LI
```

```
    DYNT -ALL
    FILE
&END                                        /*end of &DATA group
R %FILENAME%.RUN                            /*execute run-file
```

Terminal Input in &DATA Groups

Sometimes you may want a CPL file to invoke a subsystem or user program, give a few subcommands from within the CPL file, and then allow you to give further commands from your terminal. You do this by including a &TTY directive inside the &DATA group.

It doesn't matter where inside the group the &TTY directive is. However, when the &DATA group is executed, the &TTY directive is always executed last, after all other statements within the group. For this reason, it is best to place the &TTY directive at the end of the &DATA group, just before the &END statement.

This placement is shown in the following format:

```
&DATA
 statement-1
 .
 .
 .
 statement-n
&TTY
&END
```

When execution reaches the &TTY directive, control returns to the user at the terminal. At this point you can enter additional subcommands for the &DATA group subsystem. The commands that you issue from the terminal are all interpreted as commands for the subsystem, until you issue the command required to leave that subsystem.

When you issue a command from the terminal to leave the subsystem, control returns to the CPL program. The command you issue to leave a subsystem depends on the subsystem; for example,

- Type ***QUIT*** in SEG or CONCAT.
- Type ***QUIT***, ***FILE***, or ***FILE filename*** in ED or BIND.
- RUNOFF or SORT finish their work and return control to command level automatically.

Within a subsystem, you use the &TTY directive to supply subsystem commands. However, the &TTY directive can be used in other contexts for any type of input from the terminal, not just commands. The section that follows describes some other uses for the &TTY directive.

Conditional Use of the &TTY Directive: You can use the &TTY directive as part of an &IF...&THEN or &IF...&THEN...&ELSE directive within a &DATA group. For example,

```
&IF test &THEN &TTY
   &ELSE another_statement
```

In this example, the &TTY directive executes only if *test* is true. If *test* is false, then *another_statement* is executed.

A Sample Program Using the &TTY Directive: One possible use of the &TTY directive is to customize the Editor by writing a CPL program that does the following:

1. Invokes the Editor.
2. Issues a set of commands that set Editor modes and symbols as you want them.
3. Gives you control at the terminal.
4. Returns you to PRIMOS command level when the edit session is finished.

Such a file (called EDD.CPL) is shown below.

```
/* Usage: R EDD {filename}
/* Use filename to edit existing file
/* EDD sets edit symbols at terminal,
/* then returns you to interactive mode
/* inside the editor.
/* Leave the editor by typing Quit, File,
/* or File filename, as usual.
/* EDD will then return you to PRIMOS command level.
&ARGS FILENAME
   /* Create variable EMPTYLINE to hold a null string
&SET_VAR EMPTYLINE :=
   /*  Enter editor
&DATA ED %filename%
&IF [NULL %filename%] &THEN %emptyline%      /* Go into edit mode
    SYMBOL SEMICO }
    MODE COLUMN
&IF [NULL %filename%] &THEN %emptyline%  /*Back to input mode
&TTY  /*  Give user control of editor
&END   /*  End &DATA group
&RETURN
```

Note

This program uses CPL's &SET_VAR directive followed by a carriage return to define a variable, EMPTYLINE, and to set its value to the true null string. This null string is then passed to the editor, if necessary, to force it from input to edit mode and back again. The &SET_VAR directive is discussed fully in Chapter 4.

Two terminal sessions using this program are shown below:

```
OK, R EDD.CPL
INPUT

EDIT
    SYMBOL SEMICO }
    MODE COLUMN

INPUT
         1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
This is a sample file
This is the second line of the file

EDIT
FILE SAMPLE
OK,


OK, R EDD.CPL SAMPLE
EDIT
    SYMBOL SEMICO }
    MODE COLUMN
p23
.NULL.
This is a sample file
This is the second line of the file
BOTTOM
n-1
This is the second line of the file
c/second/last
This is the last line of the file
file
SAMPLE
OK,
```

Another Example of &TTY: This example shows how the &TTY directive can be used with a user program. Assume a program (named PURCHASE) that asks for five items of information about a customer purchase:

Dept. name:
Dept. number:
Customer name:
Acct. number:
Amount of purchase:

A given department (for instance, the hardware department) uses a CPL program (named PURCH.CPL) to invoke the PURCHASE program and supply it with its first two items of information:

```
&DATA R PURCHASE
 HDWR
 38
&TTY
&END
```

The example as shown could be a complete CPL program or part of a larger CPL program.

A terminal session using this program is shown below:

```
OK, R PURCH.CPL
dept. name: HDWR
dept. number: 38
customer name: H.L. Smith
acct. number: 35684
amount of purchase:  536.89
OK,
```

Notes

By using a loop and the RESPONSE function, you can write a CPL program that passes information for any number of purchases to program PURCHASE. Chapter 5 explains the RESPONSE function. Chapter 9 explains loops.

Closely related to the &TTY directive is the &TTY_CONTINUE directive. This directive can receive input for a &DATA group from the terminal, just as &TTY does. But, it can also receive input for a &DATA group from a command input file. For information on this directive, see Chapter 5.

How CPL Programs End: The &RETURN Directive

Every CPL program ends with the directive &RETURN. If you do not supply this directive as the last line of the CPL program, the CPL interpreter automatically adds it at the end of the program.

You may also use the &RETURN directive to stop the program before the end of the file. For example,

```
&ARGS A
   .
   .
   .
&IF %A% > 20 &THEN &RETURN
&ELSE &DO
   .
   .
   .
   .
   .
   .
&END
&RETURN
```

When Errors Occur

Two types of errors can occur in CPL programs: CPL errors, which prevent the CPL interpreter from executing its directives; and PRIMOS command errors, which prevent execution of the commands contained in the file.

When a CPL error is encountered, the CPL interpreter halts execution of the CPL file and returns you to PRIMOS command level with an explanatory error message. For example, misspelling &ARGS produces the following message:

```
OK, R BAD_EXAMPLE

CPL ERROR 52 ON LINE 1.
"&ARGGS" is not a directive (statement) recognized by CPL.

SOURCE: &arggs foo

Execution of procedure terminated. BAD_EXAMPLE (cpl)
ER!
```

A list of CPL error messages is provided in Appendix B.

PRIMOS errors may represent one of two levels of severity: warning or error. By default PRIMOS errors are handled as follows:

- If a warning occurs, the CPL file continues operation.
- If an error occurs, PRIMOS halts program execution and returns the user to PRIMOS command level, usually with an error message and the ER! prompt. The error message generally includes the name of the command or subsystem that generated it.

You can override this default and establish a user-written message or handling routine that is invoked when a PRIMOS error occurs. Chapters 10 and 15 show how to do this. You cannot change the handling of CPL errors.

Displaying CPL Command Execution

When debugging a CPL program, it may be useful to display each CPL command as it is executed. The commands contained in CPL programs are not normally displayed during execution. Thus, when you run COMPILE.CPL, this is what you see at your terminal:

```
OK, r compile
24 Dec 86 11:41:52 Wednesday

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]
```

The same information is stored in the COMOUTPUT file created by this CPL program.

If you want the commands to be displayed, you can preface the CPL file with the &DEBUG &ECHO COM directive. This directive tells the CPL interpreter to display all commands at the terminal and record them in the COMOUTPUT file.

Note

> The &DEBUG directive, which controls all CPL debugging facilities, is discussed in full in Chapter 10.

With the &DEBUG &ECHO directive included, the COMPILE.CPL file looks like this:

```
&DEBUG &ECHO COM
COMO COMPILE.COMO
DATE
F77 THISFILE -XREF
F77 THATFILE -XREF -32I
F77 TOTHERFILE -DEBUG
COMO -E
```

When this version of COMPILE.CPL is run, the terminal session looks like this:

```
OK, r compile
COMO COMPILE.COMO
DATE
24 Dec 86 11:45:44 Wednesday
F77 THISFILE -XREF

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]
F77 THATFILE -XREF -32I

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]
F77 TOTHERFILE -DEBUG

[F77 Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [<.MAIN.> F77 Rev. 20.2]
COMO -E
```

The same information is also recorded in the COMOUTPUT file.

3
CPL Format

This chapter describes the formatting rules for CPL programs and the use of character strings in CPL.

CPL Format Rules

The format of CPL programs is simple; nine rules presented in this chapter cover all general cases. (Any specific rules that apply to a single advanced feature are presented within the discussion of that feature.) As these rules demonstrate, the format of CPL is similar to that of most high-level programming languages. Moreover, CPL supports PRIMOS command syntax. This means that

- The PRIMOS commands that you write in a CPL program are identical to the PRIMOS commands that you issue interactively at your terminal.
- CPL, like PRIMOS, uses the semicolon (;) as a command delimiter. This allows you to write two or more PRIMOS commands (separated by semicolons) on a single line of your CPL program.
- CPL supports other PRIMOS special characters, including the syntax suppressor (~) and the use of parentheses for iteration. However, use of these special characters is discouraged because CPL syntax uses the same characters for other purposes. Refer to Chapter 11 for further details on the use of PRIMOS special characters.

Further information on the format rules for PRIMOS commands and special characters is found in the *PRIMOS Commands Reference Guide*.

▶ RULE 1: Each statement in a CPL file must appear 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 in turn may be either a PRIMOS command or another CPL directive, with its argument(s). (See RULE 3 for handling of very long statements.) For example, the following statement shows a single command on a line by itself.

```
A MYDIR
```

The following statement shows two commands separated by a semicolon.

```
CR SUBDIR1; A *>SUBDIR1
```

The following statement shows a CPL directive and its arguments.

```
&IF %VAR% = 1 &THEN RESUME FIRSTPROG
```

The &THEN directive is the argument for the &IF directive. The command RESUME FIRSTPROG is the argument for the &THEN directive. Thus, this line represents one directive plus arguments.

The &ELSE directive is not an argument for the &IF directive. Therefore, it, with its arguments, goes on a new line.

```
&IF %VAR% = 1 &THEN &GOTO LABEL1
&ELSE &IF %VAR% = 3 &THEN &GOTO LABEL3
```

The directives &DO and &END must appear on lines by themselves. Each statement in the &DO group has a line to itself.

```
&DO
  RESUME FIRSTPROG
  RESUME SECONDPROG
&END
```

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

Indent CPL programs for ease of reading, as you would indent any structured program. In almost all cases, indentation has no effect on CPL directives or PRIMOS commands.

There are two exceptions to this rule. The first concerns statements within &DATA groups. CPL does not ignore blanks used for indenting statements within a &DATA group. Instead, CPL passes these leading blanks to the subsystem as part of the subsystem statement. Most subsystems (BIND, for example) ignore leading blanks before command statements.

The second exception concerns quoted strings. Within a quoted string, all blanks are retained as part of the string. This includes blanks used for indentation when a quoted string is continued onto additional lines. Line continuation is described in the next rule.

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

By using tildes to continue lines, you can create statements longer than a single line and can format statements with whatever indentations you like. For example,

```
&IF %VAR% = 1 ~
  &THEN RESUME FIRSTPROG
  &ELSE ~
    &IF %VAR% = 2 ~
      &THEN RESUME SECONDPROG
      &ELSE RESUME LASTPROG
```

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 space. 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 the same as

```
NO BREAKHERE
```

The CPL interpreter evaluates multiple blanks as a single blank. The CPL interpreter does not evaluate multiple blanks that are within a quoted string.

A single line of a CPL program may have a maximum of 161 characters (counting multiple blanks). By using tildes, you can continue a statement on additional lines. A multiline CPL statement may have a maximum of 1024 characters (including the tildes, but counting each string of blanks as a single character).

Note

The line that you type in your CPL program cannot exceed these character counts. The value that CPL returns when it evaluates the line also cannot exceed these character counts. For example, the CPL program line TYPE %ENCYCLOPEDIA% is an invalid line if the current value of %ENCYCLOPEDIA% exceeds the allowed character counts.

The maximum length for a CPL command line is 161 characters. You cannot use tildes to extend the command line that invokes a CPL program.

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

A comment can appear on its own line, or on the same line as a CPL statement. Comments end at the end of the physical line on which they appear. They are not continued onto the next line, even when a tilde is used to mark an incomplete statement. A tilde to continue a program line should appear at the end of the line, following any intervening comment.

Thus, the statement

```
&IF %VAR% = 1          /*Comment~
&THEN                  /*more comment~
RESUME MYPROG         /*more comment
```

is read as &IF %VAR% = 1 &THEN RESUME MYPROG. The comments are ignored (that is, not evaluated or passed to the command processor).

For example,

```
RESUME FIRSTPROG                    /*FIRSTPROG does such-and-so

&IF %VAR% = 1 &THEN RESUME FIRSTPROG    /*Test for case 1
```

▶ **RULE 5: CPL programs can be written in uppercase or lowercase letters.**

PRIMOS commands, CPL directives, function names, and variable names can be written in uppercase letters, lowercase letters, or a mixture of uppercase and lowercase. CPL considers variable names that are spelled the same to be identical, regardless of the case of the letters in the names.

However, variable values are case-sensitive. That is, CPL considers two variable values to be different if they are written in different cases. Variable values supplied from the command line to the &ARGS directive are, by default, automatically converted to uppercase letters. Chapter 5 describes how to override this default.

Logical values, such as TRUE and FALSE, are case-insensitive. That is, CPL considers two logical values to be identical even if they are written in different cases. Logical values generated by CPL are always in uppercase letters.

▶ **RULE 6: Filenames for CPL programs follow PRIMOS file-naming conventions and end with .CPL.**

Filenames must not exceed 32 characters. Allowable characters are A-Z, 0-9, / _ # $ - . * &. The first character cannot be a number or a hyphen (-). The CPL interpreter translates lowercase letters to uppercase. The .CPL suffix is included in the 32-character limit, even though you do not need to specify the suffix when you invoke the file.

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

Variable names must not exceed 32 characters in length. They can contain only the characters A-Z (uppercase and lowercase), 0-9, underscore (_), and dot (.). (The CPL interpreter translates lowercase letters to uppercase.) Names of local variables (such as those defined by the &ARGS directive) must begin with a letter. Names of global variables (explained in Chapter 4) must begin with a dot.

▶ **RULE 8: Operators in a CPL expression must be preceded and followed by one or more spaces.**

CPL uses the arithmetic operators +, -, *, /, unary +, and unary -; the logical operators & (and), | (or), and ^ (not); and the relational operators =, ^=, <, >, <=, and >=. Parentheses must also be preceded and followed by blanks. For example,

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

This spacing rule prevents confusion between operations and text strings. For example, B > A is a logical statement that means "B is greater than A". B>A is a pathname. The required spaces in the logical expression keep the distinction clear, both for users and for the CPL interpreter.

▶ **RULE 9. Any special character, or string containing blanks or special characters, must be placed inside single quotation marks when used as the value of a variable.**

The special characters are as follows:

- **Single quotation marks** (') When used as a value, a single quotation mark must be doubled and the string that contains it must be enclosed in single quotation marks. The first example shows a string containing a single quotation mark. The second example shows a single quotation mark by itself as a value.

  ```
  'I''m a quoted string'
  ''''
  ```

- Commas (,)

  ```
  'I''m quoted, too'
  ```

- Square brackets ([ ])

  ```
  'Don''t evaluate this [function call]'
  ```

- Semicolons (;)

  ```
  'This; isn''t; a; list; of; arguments'
  ```

- Percent signs (%)

```
'Don''t use the value for this %variable%'
```

- Hyphens (-) at the beginning of strings.

```
'-64V'
```

- Parentheses (()) at the beginning of strings, or within a string if not paired.

```
'(this string is in parentheses)'
'this is a left parenthesis ('
```

- Ampersands (&) at the beginning of strings.

```
'&Not a CPL directive'
```

- Tildes (~) at the end of strings.

```
'Do not continue to the next line~'
```

- Comment characters (/*)

```
'This is a slash and asterisk /* not a comment indicator'
```

- CPL operators (+−*/=<>) when you specify an individual operator that is not part of a string.

```
'+'
'>='
```

- CPL expressions if you don't want them evaluated. An expression must always be quoted when supplied as a command line argument.

```
'2 + 3'
'%A% > %B%'
```

The CPL interpreter normally evaluates variable references, function calls, and expressions. This evaluation replaces each variable reference with its current value, executes each function and replaces it with its return value, and calculates expressions and replaces each arithmetic expression with an integer value and each logical expression with TRUE or FALSE. Variable references, function calls, and expressions inside quoted strings are not evaluated. Thus, 2 + 3 is an expression but '2 + 3' (quoted) is merely a string. Hence,

```
2 + 3 = 5
```

is TRUE, because 2 plus 3 equals 5; but

```
'2 + 3' = 5
```

is FALSE, because the string '2 + 3' and the string 5 are not identical.

Note

If you do not set off operator characters with blanks, CPL does not read the string as an expression. Thus, CPL reads the string A > B as an expression. You must enclose this string in single quotation marks if it is to remain the character string A > B, rather than an evaluated expression. The string A>B is not an expression, and therefore does not need to be quoted.

Using Quoted Strings

In CPL, any material enclosed within single quotation marks is considered a character string. The actual length of the character string is the number of characters and blanks within the string; the enclosing single quotation marks are not counted, and multiple blanks are not ignored. Doubled quotation marks that are embedded in a quoted string (for example, 'DON"T') are counted as a single character.

A character string can be as long as a CPL statement (1024 characters). To extend a string beyond a single line, place a tilde at the end of the line. Even though the tilde is within the quoted string, it is evaluated as a line-continuation character because it is the last character on that line. This point is demonstrated in the following two examples:

```
'In this string the tilde is not evaluated ~'

'In this string the tilde ~
is evaluated as a line-continuation character'
```

Concatenating Quoted Strings

Two strings are concatenated if they are placed next to each other without an intervening blank space. Concatenating two quoted strings produces a single quoted string. For example, if

```
%A% = 'I''m a quo'
```

and

```
%B% = 'ted string'
```

then

```
%A%%B% = 'I''m a quoted string'
```

You can concatenate quoted strings returned by variable references and function calls as well as literal quoted strings.

Evaluation of Quoted Strings

Whenever you use a quoted string in CPL, that string remains quoted wherever it is used. This means that a quoted string is always viewed as a single, indivisible item. For example, a quoted string containing three words separated by blanks cannot be substituted for three variables. The string is a single value, and can only be substituted for a single variable.

CPL preserves multiple blanks within a quoted string. For example, the string 'A B C' retains the multiple blanks between A and B. This means that in the following example

```
'This statement continues ~
      on the next line.'
```

the blanks used to indent the line continuation are considered part of the quoted string.

Single quotation marks prevent the evaluation of special characters. Not only are quoted special characters not evaluated by the CPL interpreter, they are also not evaluated by other software invoked from the CPL program. For example, consider a CPL program SHUT.CPL that closes open files:

```
&ARGS pathname
CLOSE %pathname%
```

If you specify the pathname of an open file on the command line as follows,

```
CPL SHUT MYDIR>OPENFILE.F77
```

SHUT.CPL closes the specified file.

If you specify the quoted pathname of an open file on the command line,

```
CPL SHUT 'MYDIR>OPENFILE.F77'
```

SHUT.CPL still closes the specified file. The quotation marks simply indicate to the CLOSE command that the pathname is, indeed, a string.

However, if you attempt to close all open files by specifying the –ALL option,

```
CPL SHUT '-ALL'
```

the CLOSE operation fails. This is because the CLOSE command cannot evaluate the hyphen as a special character indicating an option. It reads '-ALL' as a pathname string, and fails to find a file of that name. This problem can be resolved by unquoting the character string.

The QUOTE and UNQUOTE Functions

CPL provides built-in QUOTE and UNQUOTE functions to place quotation marks around strings and to remove quotation marks from strings. The UNQUOTE function is particularly useful, as it allows you to use a quoted string as an argument for a CPL program, then remove the quotation marks from the string inside the program. For example, to pass PRIMOS command options (which begin with hyphens) as arguments, you can write a CPL file (MYPROG.CPL) like this:

```
&ARGS filename; options
F77 %filename% -64V -L %filename%.LIST [UNQUOTE %options%]
```

With this program, the command line

```
R MYPROG.CPL  FOO '-XREF -EXPLIST'
```

produces the CPL statement

```
F77 FOO -64V -L FOO.LIST -XREF -EXPLIST
```

The UNQUOTE function removes the single quotation marks from the string '-XREF -EXPLIST'. The CPL interpreter replaces the UNQUOTE function call with the unquoted string, and passes the finished command to the command processor.

MYPROG.CPL can also be invoked by the command line

```
R MYPROG.CPL  FOO
```

This invocation produces the CPL statement

```
F77 FOO -64V -L FOO.LIST
```

Because this invocation supplies no value to the options argument, the reference to [UNQUOTE %options%] first becomes [UNQUOTE ''], and then becomes the unquoted null string (that is, a string of length 0, containing no characters), which is ignored by PRIMOS.

For more information on quoted strings, see Chapter 12. For a better way to pass command options as arguments, see Chapter 6.

Part II
The Intermediate Subset

4
Variables

This chapter discusses

- How to define variables with the &SET_VAR directive
- The three types of values — string, integer, and logical — that variables can possess, and the operations that can be performed on these three types of values
- Local and global variables
- The four PRIMOS commands that govern global variables

The &SET_VAR Directive

You use the &SET_VAR directive to create a CPL variable and assign it a value within your program. A &SET_VAR directive can define a new variable or change the value of an existing variable, such as a variable established by the &ARGS directive. Like the &ARGS directive, you can place a &SET_VAR directive anywhere in your CPL program prior to the first statement that uses its assigned value. By convention, &ARGS and &SET_VAR directives for all variables are placed at the beginning of each CPL program module. You can place additional &SET_VAR statements in the CPL program wherever it is necessary to change the assigned value of a variable.

The &SET_VAR directive has the following format:

&SET_VAR name-1 {,name-2...,name-n} := value

You can abbreviate the name of the &SET_VAR directive to &S.

name-1 through *name-n* permit you to define multiple variables. Each of these can be either

- A valid variable name (naming conventions are described in Chapter 3, Rule 7).
- An expression that evaluates to a valid variable name. An expression can contain function calls and references to other variables.

In the following example, the first &SET_VAR creates a variable named *MONTH* with a value of JANUARY. The second &SET_VAR directive uses this variable to create two variables named *JANUARY1* and *JANUARY15* and assigns the value MONDAY to those variables:

```
&SET_VAR MONTH := JANUARY
     &SET_VAR %MONTH%1, %MONTH%15 := MONDAY
```

This method of creating variable names allows you to simulate array variables. Refer to Chapter 11 for details.

value can be

- A character string
- An integer
- A logical value (some form of TRUE or FALSE)
- An expression that evaluates to any of the above

The following sections describe the use of the three types of variable values: string, integer, and logical.

String Values for Variables

All CPL variable values are character strings. CPL can also process some of these values as integers or logical values, as described later in this chapter. Any value that is not a valid integer or logical value can only be processed as a character string. Any value enclosed in single quotation marks can only be processed as a character string.

A character string can contain characters of any type. Unlike the &ARGS directive (which converts all letters to uppercase), &SET_VAR permits you to set character strings that contain both uppercase and lowercase letters.

The following example shows one use of variables with string values. Note that this example uses &S, which is a shorter name for the &SET_VAR directive:

```
  &ARGS DISTRICT
&IF %DISTRICT% = E ~
   &THEN &S DISTRICT := ACCTS>RECEIVED>EAST
   &ELSE &IF %DISTRICT% = W ~
      &THEN &S DISTRICT := ACCTS>RECEIVED>WEST
      &ELSE &S DISTRICT := ACCTS>RECEIVED>CENTRAL
```

In this example, the &SET_VAR directive allows lengthy arguments to be entered in abbreviated form (as E or W), then expands those arguments to their full values.

Additional information on using character strings can be found in Chapter 3.

Integer Values for Variables

All CPL variable values are character strings. However, some character strings (such as 3, 0, 259, -6847) can be interpreted as integer values. CPL allows the following arithmetic operations on these integers: addition (+), subtraction (-), multiplication (*), and division (/).

Integer values are positive whole numbers, negative whole numbers, and zero. They do not include real numbers (numbers containing a decimal point) or numbers expressed in exponential notation. Valid integers can range in value from $-2^{31} + 1$ to $2^{31} - 1$. Do not enclose variable values to be interpreted as integers in single quotation marks or include commas in them. In some cases, negative integers must be initially supplied as quoted strings, then unquoted (using the UNQUOTE function call) when they are used as integers.

The following examples are all valid statements:

```
&SET_VAR A := 4                          (A = 4)
&SET_VAR B := 5                          (B = 5)
&SET_VAR C := %B% + 1                    (C = 6)
&SET_VAR D := %C% - %B%                  (D = 1)
&SET_VAR E := ( %A% + 2 ) * %C%          (E = 36)
&SET_VAR F := %E% / %B%                  (F = 7, remainder discarded)
```

Note

Remember to leave at least one blank space before and after arithmetic and logical operators — including parentheses, and before the minus signs in negative numbers.

Since integers, in CPL, are actually character strings that evaluate to integer values, integers can be concatenated exactly like character strings. For example,

```
&SET_VAR A := 5                          (A = 5)
&SET_VAR B := 6                          (B = 6)
&SET_VAR C := %A%%B%                     (C = 56)
&SET_VAR D := %A% + %C%                  (D = 61)
```

Logical Values for Variables

CPL variables can also take the logical values, TRUE and FALSE. Users may use the strings TRUE, true, T, and t to represent logical (or Boolean) true, and FALSE, false, F, or f for Boolean false. CPL itself uses the spellings TRUE and FALSE. You can set a logical value yourself:

```
&S A := TRUE
```

Or, you can have CPL do calculations that produce logical results. For example,

```
&SET_VAR A := 6
&SET_VAR B := 12
&SET_VAR C := %A% > %B%
```

When these three directives have been executed, *C* has the value FALSE.

Notes

If you enclose a logical value in single quotation marks (for example, 'TRUE'), CPL cannot evaluate the string as a logical value. It instead treats the string as an ordinary character string.

The logical operators >, >=, =, ^=, <=, and < perform string comparisons if either operand is a character string. Strings are compared based on the standard sorting sequence (that is, 1 < A < B < a < b). If both operands are integers or Boolean values, an arithmetic comparison is done. (Boolean TRUE = 1, and Boolean FALSE = 0.) Thus, the following expressions are all true:

128 > 40	because 128 > 40
'40' > '128'	because '4' > '1'
BARREL > APPLE	because 'B' > 'A'
TRUE > FALSE	because 1 > 0
34 > FALSE	because 34 > 0
'FALSE' > 34	because 'F' > '3'

Local and Global Variables

CPL supports two kinds of variables: local variables and global variables. Local variables and global variables can have the same types of values: string, integer, and logical. All variable values follow the same rules. What distinguishes local variables from global variables is the persistence and scope of the variable, not the nature of its value.

Local Variables

All variables shown so far have been local variables. Local variables have the following attributes:

- They are defined inside a running CPL program.
- They are defined by either
  - The &ARGS directive (explained in Chapter 2).
  - The &SET_VAR directive (explained earlier in this chapter).
  - The SET_VAR command (explained later in this chapter).
- They are known only to the program that creates them.
- They disappear when the program that creates them returns or terminates.

Precisely because they are local — that is, defined within one activation of one program — local variables from one program never interfere with those of any other program.

Global Variables

Sometimes you want to define variables that can be known to, and possibly modified by, a group of programs, rather than a single program. At these times, you can use global variables. Each user creates and activates a unique, personal set of global variables. Global variables have the following attributes:

- They are stored in one or more files in your directory, independent of any CPL program file.
- They can be used by many different CPL programs.
- They can be used by PRIMOS commands and programs written in high-level languages.
- They survive program termination and logouts. Once defined, global variables persist until you delete them.

To use global variables, you must create a global variable file, place global variables in that file, and activate access to that file from your CPL program. The PRIMOS commands governing global variables are shown in Table 4-1. They are explained in greater detail later in this chapter.

Table 4-1
Variable-handling Commands

Command	*Function*
DEFINE_GVAR	Creates or activates a global variable file.
SET_VAR	Defines a new variable or changes the value of an existing variable. If the variable is a global variable, SET_VAR places it in the active global variable file.
LIST_VAR	Lists the variables contained in an active global variable file.
DELETE_VAR	Deletes variables from an active global variable file.

Global variables are particularly useful for establishing variable values for use by programs of different types, as they may be set and referenced

- At command level
- By any of your CPL programs
- By high-level language programs

Note

Global variables are not designed for interprocess communication. Do not try to use global variables to pass messages between concurrently executing programs. Attempts to use global variables for this purpose are not guaranteed to work.

Global variables must have names that begin with dots (.). For example,

```
.SIZE
.MYDIR
```

At command level, global variables are defined by the SET_VAR command. Within a CPL program, they are defined by the &SET_VAR directive or the SET_VAR command. (They cannot be defined by the &ARGS directive.) High-level programs can define global variables using the GV$SET subroutine and can reference global variables using the GV$GET subroutine. These subroutines are described in the *Subroutines Reference Guide, Volume II*.

PRIMOS Commands

PRIMOS provides four commands for handling global variables: DEFINE_GVAR, SET_VAR, LIST_VAR, and DELETE_VAR. Like most PRIMOS commands, these can be executed from within a CPL program or interactively from the terminal. Within a CPL program, the SET_VAR command can be used for both global variables and local variables.

The DEFINE_GVAR Command

Each user's global variables reside in a file that is created and activated by the DEFINE_GVAR command (abbreviation: DEFGV). The command

DEFINE_GVAR pathname -CREATE

creates and activates a new global variable file. If the file named by *pathname* already exists, the command simply activates it. The command

DEFINE_GVAR pathname

activates an existing global variable file. The DEFINE_GVAR command may be used at command level or inside a CPL program. You must create a global variable file before you can define global variables. You must activate your global variable file before using the variables it contains, or adding or deleting global variables.

For example, to create an empty global variable file named MY_VARS, give the command

```
DEFINE_GVAR MY_VARS -CREATE
```

To use the file again in a later session, use the command

```
DEFINE_GVAR MY_VARS
```

Note

If you supply a filename to DEFINE_GVAR, it creates or activates a file in your current directory. To create or activate a global variable file elsewhere, supply the full pathname. DEFINE_GVAR cannot locate files using the PRIMOS search rules facility. If the directory containing the global variable is protected by a password, you *must* provide the full pathname of the file within the DEFINE_GVAR command. For example,

```
DEFINE_GVAR '<DISK>MY_DIR SECRET>MY_VARS'
```

where SECRET is the password for directory MY_DIR. Password protected directories are described in the *Prime User's Guide.*

Once activated, a global variable file remains active until one of the following occurs:

- You log out or ICE your process.
- You explicitly deactivate the file.
- You activate another global variable file.

You can create more than one global variable file, but you can have only one global variable file active at any time. Therefore, when you issue a DEFINE_GVAR command, it activates the named file and deactivates any global variable file already active.

You can also deactivate a global variable file by issuing the command

```
DEFINE_GVAR -OFF
```

No pathname is required for this form of the command.

Whenever a global variable file is active, you may add to, delete, list, and make use of any variables it contains. If your CPL program refers to a global variable when no global variable file is active, the program aborts with an error message. It is best to explicitly activate the global variable file in your CPL program, rather than assume that the user running the CPL program has that global variable file active.

You can delete a global variable file from your directory using the standard DELETE command. Make sure the file is inactive (by issuing the command DEFINE_GVAR -OFF) before you delete it. (If you fail to do this, you create a confusing situation in which you can list variables from your deleted file, but cannot add or modify any variables.)

The SET_VAR Command

SET_VAR is a PRIMOS command that you can use to set variables. It both creates a variable and assigns a value to the variable. You can use SET_VAR within a CPL program to set both local variables and global variables. The SET_VAR command has the format

SET_VAR name {:=} value

name is any legal variable name with a maximum of 32 characters. Naming conventions for variables are described in Chapter 3, Rule 7. Names of global variables must begin with a dot (.). *name* can define a new variable or refer to an existing variable.

value can be any of the following:

- A character string with a maximum of 1024 characters. Lowercase characters are not converted to uppercase. If the string contains special characters (as explained in Chapter 3), it must be enclosed in single quotation marks.
- An integer between the values of $-2^{31} + 1$ to $2^{31} - 1$. Positive numbers, negative numbers, and zero are permitted. Real numbers (containing a decimal point) and numbers expressed in scientific notation are not permitted.
- A logical (Boolean) value. Logical values include the character strings TRUE and FALSE. The forms TRUE, T; true, t, FALSE, F, false, and f are acceptable.

The assignment symbol (:=) is optional. If specified, it must be set off with blank spaces. For example,

```
SET_VAR .A ALPHA
```

and

```
SET_VAR .A := ALPHA
```

both define the global variable .A and assign it the value ALPHA.

You can use the SET_VAR command interactively, at command level, to define global variables. Or, you can use it inside a CPL program to define either global or local variables. However, since the &SET_VAR directive is faster than the SET_VAR command, use the SET_VAR command at command level only, and use the &SET_VAR directive inside CPL programs.

The following example demonstrates the use of both the SET_VAR command and the &SET_VAR directive. You start by setting a global variable, then invoke the DAILY.CPL program from your terminal:

```
DEFINE_GVAR WEEKLY
SET_VAR .DAY := FRIDAY
RESUME DAILY.CPL
```

These commands run the following CPL program:

```
    /* Program DAILY.CPL, which is run 5 days a week
    /* and invokes a weekly totals program on Fridays.
DEFINE_GVAR WEEKLY
&IF [NULL %.DAY%] &THEN ~
    &DO
    TYPE 'Specify a day and rerun'
    &RETURN
    &END
```

```
&ELSE  RESUME DAILYCALC.RUN
&IF %.DAY% = FRIDAY &THEN ~
   RESUME TOTALS.CPL
&ELSE TYPE 'Today is not a Friday'
&RETURN


   /* Program TOTALS.CPL, which performs weekly totals
   /* on Fridays, then resets .DAY for the next daily run.
&IF %.DAY% = FRIDAY &THEN ~
&DO
   RESUME WEEKLYCALC.RUN
   &SET_VAR .DAY := MONDAY
&END
&RETURN
```

The DELETE_VAR Command

The DELETE_VAR command removes one or more global variables from an active global variable file. Its format is

DELETE_VAR name-1 {...name-n}

name-1 through *name-n* are names of global variables. They can also be wildcards, variable references, or function calls that evaluate to the names of global variables. Names of global variables in the DELETE_VAR command are separated by blanks. All global variables listed in DELETE_VAR are deleted from the file. For example,

```
DEFINE_GVAR MY_VARS
DELETE_VAR .ABC
```

deletes the variable .ABC from the file MY_VARS. The command

```
DELETE_VAR .A .B .C
```

deletes three variables, .A, .B, and .C.

```
DELETE_VAR .AB@@
```

deletes all global variables in the file whose names begin with .AB.

If you specify the name of a variable to DELETE_VAR and that variable does not exist, DELETE_VAR skips over that variable name and deletes whichever listed variables do exist in your global variable file.

The LIST_VAR Command

The command LIST_VAR lists the global variables in your currently active global variable file. You can list all or some of the global variables contained in your active global variable file, with their values. The format of LIST_VAR is

LIST_VAR {name-1 ... name-n}

name-1 through *name-n* are the names of the global variables that you wish to list. *name-1* through *name-n* may be either global variable names or wildcard names. If no *names* are given, the LIST_VAR command lists all the variables in the file.

The following terminal session uses LIST_VAR to list all global variables in the active global variable file:

```
OK, list_var
.ERR_MESSAGE                    Sorry, try again!
.MYDIR                          glenn
.DIGITS                         0123456789
.AL                             ABCDEFGHIJKLMNOPQRSTUVWXYZ
.ERR_REPORT
OK,
```

In this example, the value of .ERR_REPORT is the null string.

If names are given, LIST_VAR lists only those names (or groups of names) and their values, as shown in the following terminal session:

```
OK, list_var .err@
.ERR_MESSAGE                    Sorry, try again!
.ERR_REPORT
OK, list_var .al
.AL                             ABCDEFGHIJKLMNOPQRSTUVWXYZ
OK,
```

5
Terminal Input and Output

This chapter describes directives and functions that a running CPL program can use to receive input from the user terminal. It describes how non-interactive CPL programs can use these same facilities to receive input from a COMINPUT file. Finally, this chapter describes two methods to output information from a CPL program to the user terminal or COMOUTPUT file.

Input Overview

CPL provides three facilities for runtime input:

- The &TTY directive (or the &TTY_CONTINUE directive) receives data of any type and supplies it to a &DATA group subsystem. These directives can only be used within a &DATA group. They allow the user to enter one or more lines of information to a utility or a user program run within a &DATA group. This information can be supplied interactively or from a COMINPUT file.
- The QUERY function displays a question at the user's terminal and accepts a YES or NO answer. The QUERY function is a logical (Boolean) function; it interprets "YES" answers as TRUE and "NO" answers as FALSE. If other types of answers are given, it issues a prompt requesting a YES or NO answer.
- The RESPONSE function displays a request for information at the user's terminal and accepts a character string answer. If this string contains blanks or special characters, the CPL interpreter encloses the string in single quotation marks. The CPL interpreter then returns this string as the value of the function; that is, the string replaces the function call.

Output Overview

CPL provides two facilities for output to the terminal or to a COMOUTPUT file:

- The TYPE command displays any message. This PRIMOS command can be placed almost anywhere within a CPL file.

- The &MESSAGE clause of the &RETURN directive sends a message when the CPL program returns. &MESSAGE can be used to announce the completion of a CPL program or subprogram. Another common use of &MESSAGE is to announce a fatal error in a program, such as the failure of an &IF test. The &MESSAGE clause may also be used with the error-handling &STOP directive. See Chapter 15, Error and Condition Handling, for details on &STOP.

Terminal Input

The &TTY and &TTY_CONTINUE Directives

The format of the &TTY directive is

&DATA {subsystem}
.
.
.
&TTY
&END

The &TTY directive enables you to input multiple lines of information from a user terminal to a subsystem invoked by a &DATA group. &TTY can only be issued from within a &DATA group; it is always the last statement executed in a &DATA group, immediately prior to the &END directive. The &TTY directive gives control of the subsystem to the terminal until the terminal user issues the QUIT command appropriate for that subsystem. Because the &TTY directive gives control to the user terminal, a CPL program containing an &TTY directive must be run interactively. Further details and examples of the use of the &TTY directive are provided in Chapter 2.

The &TTY_CONTINUE directive is similar to the &TTY directive, except that &TTY_CONTINUE permits you to input information from either a COMINPUT file or a user terminal. A CPL program containing &TTY_CONTINUE directives can be run non-interactively. &TTY_CONTINUE is further described in the section entitled Command Input Stream Error Handling, later in this chapter.

The QUERY Function

The format of the QUERY function is

[QUERY {prompt} {default} {-TTY}]

For example,

```
[QUERY 'Et tu, Brute' TRUE -TTY]
```

When the QUERY function is executed, it displays the value of *prompt* on the user's terminal, follows it with a question mark, and then waits for the user to type an answer. The user responds to the prompt with a YES or NO answer. The QUERY function returns either logical TRUE or FALSE, depending on the user's response.

prompt: The *prompt* text is displayed on the user's terminal. Because *prompt* is usually a question, CPL automatically adds a question mark to the end of the prompt. *prompt* may be any character string with a maximum of 1024 characters. If *prompt* contains blanks or special characters, it must be placed inside single quotation marks.

If *prompt* is omitted or is the null string (''), no prompt is displayed. You may wish to omit the prompt if your CPL program requests instructions by some other means, such as the TYPE command or output from a user program.

Remember, variables and function calls cannot be evaluated within a quoted string. Therefore, if you write [QUERY 'SPOOL %FILE%'], the user sees the prompt

```
SPOOL %FILE%?
```

at the terminal. If you write [QUERY 'SPOOL '%FILE%], the user sees the actual filename in the prompt.

You respond to a QUERY prompt by typing a YES or a NO response, followed by a carriage return. The QUERY function evaluates your response to produce a logical TRUE or FALSE value. QUERY accepts YES, yes, Y, y, OK, and ok as TRUE answers. It accepts NO, no, N, n, QUIT, and quit as FALSE answers.

default: The QUERY function returns its *default* value if the terminal user does not supply a value when prompted. Establishing a *default* value is optional. If you establish a *default* value, it must follow the *prompt* value; you can use the null string ('') to establish an omitted *prompt*. Permitted values for *default* are TRUE, T, FALSE, or F (uppercase or lowercase).

If you have specified a *default* value, then a null response from the user terminal (that is, a carriage return or empty line) is taken as the function's *default* response. If you do not specify a *default* value, a carriage return is interpreted as FALSE.

–TTY: The –TTY option forces the QUERY function to take input from the terminal. If this option is present, the CPL program containing the query cannot be executed as a phantom or batch job.

If the –TTY option is not used, the QUERY function returns one step up the command input stream to get its input. This can be the terminal, a &DATA block inside another CPL program, or a command input file. Receiving input from sources other than the user terminal is further discussed in the section entitled COMINPUT File Input, later in this chapter.

Examples: Here are some examples of the QUERY function in use.

```
&DATA ED %NAME%
          T
          .
      (Editor Commands)
          .
         FILE
&END
&IF [QUERY 'Spool file']~
    &THEN SPOOL %NAME% -AT DOC -FORM WHITE
```

A YES answer to the query spools the file. A NO, or a carriage return, does not spool it. Any other answer produces the message,

```
Please answer "YES", "OK", "NO", or "QUIT":
```

For example, if the above program is named TEST.CPL, the following terminal session might occur:

```
OK, R TEST.CPL BOOK
SPOOL FILE? SURE
Please answer "YES", "OK", "NO", or "QUIT": Y
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 14 added to queue,  2 records : <MAIN1>GLENN>BOOK
OK,
```

The following example demonstrates the QUERY function's default option:

```
&DATA ED %NAME%
          T
          .
      (Editor Commands)
          .
         FILE
&END
&IF [QUERY 'Spool file' TRUE] ~
    &THEN SPOOL %NAME% -AT DOC -FORM WHITE
```

Again, the user chooses whether or not to spool the file. This time, however, *default* has been given as TRUE. Therefore, a carriage return as answer spools the file, as shown below:

```
OK, R TEST.CPL BOOK
SPOOL FILE?
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 14 added to queue,  2 records : <MAIN1>GLENN>BOOK
OK,
```

The RESPONSE Function

The format of the RESPONSE function is

[RESPONSE {prompt} {default} {-TTY}]

For example,

```
[RESPONSE 'The flavor I want is' VANILLA -TTY]
```

When the RESPONSE function is executed, it displays the value of *prompt* on the user's terminal, follows it with a colon, and then waits for the user to type an answer. The user can respond to the prompt with any character string. The RESPONSE function returns this text string, in quotation marks if necessary.

prompt: The *prompt* text is displayed on the user's terminal. CPL automatically adds a colon to the end of the prompt. *prompt* may be any character string with a maximum of 1024 characters. If *prompt* contains blanks or special characters, enclose it in single quotation marks.

If *prompt* is omitted, or is the null string, no prompt is displayed. You may wish to omit the prompt if your CPL program requests a response by some other means, such as the TYPE command or output from a user program.

default: The RESPONSE function returns the *default* value if the terminal user does not supply a value. Specifying a *default* value is optional. The *default* value can be any character string with a maximum of 1024 characters. If *default* contains blanks or special characters, enclose it in single quotation marks.

If you have specified a *default* value, a null response from the user terminal (that is, a carriage return or empty line) is taken as the function's *default* response. If you did not specify a *default* value, a carriage return is interpreted as a null string. A null string is an acceptable value.

-TTY: The -TTY option forces the RESPONSE function to take input from the terminal. If this option is present, the CPL program containing this RESPONSE function cannot be executed as a phantom or batch job.

Example: The following example demonstrates the RESPONSE function.

```
&ARGS DIR
&IF [NULL %DIR%]~
     &THEN &SET_VAR DIR := [RESPONSE 'Which directory?']
ATTACH %DIR%
```

This example checks to make sure that you input an argument value when you run the CPL program. It tests for a null argument value. If the argument is null, it invokes the RESPONSE function. RESPONSE asks the user explicitly for the argument value. When the user supplies the value, the program sets the variable with that value.

COMINPUT File Input

As stated earlier, the &TTY directive, and the QUERY and RESPONSE functions with the –TTY option, all insist on input from the terminal. CPL programs employing these statements cannot be invoked as phantoms or batch jobs; these requests for terminal input would abort their execution.

In contrast, the &TTY_CONTINUE directive, and the QUERY and RESPONSE functions without the –TTY option, seek their input from the command input stream. Therefore, they can accept input from any of three sources:

- The terminal
- A COMINPUT file
- A &DATA group in a CPL program

If the CPL program that requests the input is invoked from the terminal, it takes its input from the terminal. If the CPL program is invoked from a COMINPUT file, it seeks its input there. If it is invoked by a &DATA directive, it gets its input from the &DATA group. The following examples invoke a CPL program using these three methods.

From the User Terminal

The following CPL program contains a &TTY_CONTINUE directive. The program invokes the EDITOR to edit a specified file, goes to the bottom of the file, goes into input mode, and waits for input.

```
&DATA ED TESTFILE
B
;
&TTY_CONTINUE
&END
&RETURN
```

This program (named LENGTHEN_FILE.CPL) can be invoked from the terminal. A sample session looks like this:

```
OK, R LENGTHEN_FILE.CPL
EDIT
B
;
INPUT
We can add lines
To this file.

EDIT
FILE
```

From a COMINPUT File

The following example shows a command input file that invokes LENGTHEN_FILE.CPL and inputs data to its &TTY_CONTINUE directive:

```
R LENGTHEN_FILE.CPL
Add this line
And this one
And this one.

FILE
CO -TTY
```

The first line of this COMINPUT file invokes the CPL program shown above. The second, third, and fourth lines contain input to be added to TESTFILE. The fifth line is a blank line; it returns the EDITOR to EDIT mode. The sixth line commands EDITOR to file TESTFILE and return. The CO –TTY line closes the COMINPUT file and returns control to the terminal. First, control returns to LENGTHEN_FILE, which returns to its caller, the command input file, which then returns to the terminal.

A terminal session that runs this COMINPUT file looks like this:

```
OK, CO TTY_CONT.COMI
OK, R LENGTHEN_FILE.CPL
EDIT
B
;
INPUT
Add this line
And this one
And this one.

EDIT
FILE
TESTFILE
OK, CO -TTY
```

From an &DATA Group

The following example is an &DATA group in a CPL program that invokes another CPL program (LENGTHEN_FILE.CPL) and inputs data to the &TTY_CONTINUE directive in that program:

```
&DATA R LENGTHEN_FILE.CPL
If we keep adding lines
This file will get very long.

FILE
&END
```

Again, the first line invokes LENGTHEN_FILE.CPL; the next three lengthen it; and the fourth and fifth close the file and leave the EDITOR.

A terminal session looks like this:

```
OK, R TTY_CONT.CPL
EDIT
B
;
INPUT
If we keep adding lines
This file will get very long.

EDIT
FILE
TESTFILE
OK,
```

Command Input Stream Error Handling

What happens if you forget the blank line before the FILE statement in the COMINPUT file or the CPL program?

The COMINPUT file adds every line in its file (including the CO –TTY, which should terminate the file) to TESTFILE. Then it returns to the terminal with an error message and a request for input. The user then has to leave the EDITOR interactively in order to return to PRIMOS command level. This sequence of events looks like this:

```
OK, CO TTY_CONT.COMI
OK, R LENGTHEN_FILE.CPL
EDIT
B
;
INPUT
Add this line
And this one
And this one.
FILE
CO -TTY

End of file. Cominput. (Input from terminal.)
;
EDIT
FILE
TESTFILE
OK,
```

The CPL program, on the other hand, recognizes that an error has occurred when it comes to the &END statement in the &DATA group. It simply terminates with an error message, like this:

```
OK, R TTY_CONT
EDIT
B
;
INPUT
If we keep adding lines
This file will get very long.
FILE

CPL ERROR 35 ON LINE 5. LAST TOKEN WAS: "&END".
The Primos command invoked by this &DATA block has read all
supplied input data and is requesting more.  To suppress this
message and continue execution using terminal input, use the
&TTY directive.

SOURCE: &END

ER!
```

Note that either program would abort if it were being run as a Batch job or a phantom, since such programs cannot seek help from the terminal.

Output

The TYPE Command

The format of the TYPE command is

TYPE text

text is a character string with a maximum of 1024 characters. The first line can contain 251 characters; additional lines of text can be added using the line continuation character (~). Each additional line can contain as many as 261 characters, including the line continuation character. When the TYPE command is executed, *text* is typed at the user's terminal.

The *text* string following the TYPE command does not have to be quoted, even if it contains blanks. Variables and functions within the *text* string are evaluated. To prevent evaluation of variables and functions, or to use special characters, enclose *text* in single quotation marks. TYPE removes one set of quotation marks from around *text* before it displays it.

For example,

```
TYPE Can not find %BOOK%        (prints: Can not find SAMPLE)
TYPE 'Can not find %BOOK%'      (prints: Can not find %BOOK%)
TYPE 'Can''t find %BOOK%'       (prints: Can't find %BOOK%)
TYPE 'Can''t find '%BOOK%       (prints: Can't find SAMPLE)
```

Since TYPE is an internal command, you can use it whenever a PRIMOS command can be used within a CPL file. For example, you can write a program, called EDTEST.CPL, as follows:

```
&ARGS BOOK
  /*  Check for null argument
&IF [NULL %BOOK%] &THEN ~
   &SET_VAR BOOK := [RESPONSE 'Please specify book']
ED %BOOK%
TYPE Do you want %BOOK% spooled?
&IF [QUERY '' TRUE] ~
   &THEN SPOOL %BOOK%
TYPE Thank you.
TYPE Good-bye.
```

A terminal session using this program looks like this:

```
OK, R EDTEST.CPL SAMPLE
EDIT
p23
.NULL.
This is a sample file.
This is the second line of the file.
BOTTOM
;
INPUT
Here is a third line for the file.
;
EDIT
file
SAMPLE
Do you want SAMPLE spooled?
yes
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 22 added to queue,  2 records : <MAIN1>GLENN>SAMPLE
Thank you.
Good-bye.
OK,
```

The &MESSAGE Clause

The format of the &MESSAGE clause is

&RETURN &MESSAGE text

Including the &MESSAGE clause in the &RETURN directive causes a CPL program to display a message when it returns to its caller. Thus, it is useful for announcing the success or failure of a program.

text may be any character string with a maximum of 1024 characters. You can continue messages longer than a single line on additional lines by using tildes (~). *text* can contain function calls and variable references. You do not need to quote *text* if it contains blanks. You should quote a message text that contains special characters that are to be displayed literally. For example,

```
&IF %LEFTOVERS% = 0 &THEN~
    &RETURN &MESSAGE It worked!
&ELSE~
    &RETURN &MESSAGE %LEFTOVERS% left undone. ~
Go back and run it again.
```

Because the second &MESSAGE clause is not quoted, the CPL interpreter evaluates %LEFTOVERS% before displaying the message.

Another application of the &MESSAGE clause informs the user that the command line for the CPL program was entered incorrectly. For example,

```
&ARGS DIR
&IF [NULL %DIR%] &THEN &RETURN &MESSAGE 'You forgot to ~
input the DIR argument; rerun this program.'
```

6
Arguments With Type-checking and Default Values

This chapter describes how to use the &ARGS directive to establish default values and data type checking for CPL variables. It also describes the REST argument for the &ARGS directive. The basic features of the &ARGS directive are described in Chapter 2.

Overview

Previous chapters of this guide include examples of programs that check for the existence of needed arguments and take action if they do not find them.

The methods shown include the following:

Method	*Example*

Setting up a default action (shown in Chapter 2)	`&ARGS DIR`
	`&IF [NULL %DIR%] ~`
	`  &THEN ATTACH MYDIR`
	`  &ELSE ATTACH %DIR%`

Using CPL's RESPONSE function to demand the argument from the user (shown in Chapter 5)	`&ARGS DIR`
	`&IF [NULL %DIR%] ~`
	`  &THEN &SET_VAR DIR := ~`
	`  [RESPONSE 'Which directory?']`
	`ATTACH %DIR%`

Using CPL's &RETURN &MESSAGE directive to terminate the CPL program and tell the user the appropriate command format (shown in Chapter 5)	`&ARGS DIR`
	`&IF [NULL %DIR%] &THEN ~`
	`  &RETURN &MESSAGE ~`
	`  'You forgot to input ~`
	`  the DIR argument; ~`
	`  rerun this program.'`

This chapter introduces

- A method of establishing a default value for each argument in the &ARGS directive. A default value is a value that is used when the user does not supply a value. With this method, when an argument is omitted from the command line, CPL automatically assigns the argument its designated default value, rather than setting the argument to the null value.
- A method for setting a type specification for each argument in the &ARGS directive. A type specification indicates the type of data that is permissible; for example, character string data, integer data, and the like. When this is done, each argument value given on the command line is checked against the argument's specified type. If the types do not match, the CPL program terminates with an explanatory error message.
- A special type of &ARGS directive argument, REST, which assigns the rest of the command line to a single variable.

Specifying Default Values for Arguments

The format for specifying a default value for an argument is

&ARGS name-1:=default-1{; name-n:=default-n}

For example,

```
&ARGS DIR:=MYDIR; STRING:='This is the default'
```

The example establishes two arguments, DIR and STRING, and assigns a default value to each. If you assign a default value to an argument, CPL uses the default value if the user does not specify a value for that argument on the command line used to execute the CPL program. For example, if you establish the two arguments without defaults,

```
&ARGS DIR; STRING
```

and the command line does not contain argument values,

```
R MYPROG.CPL
```

then running this program sets both arguments to the null value. If you specify default values in the &ARGS directive, CPL sets these arguments to their default values rather than to null.

The default values established in &ARGS are substituted for the corresponding variable references throughout your CPL program. A default value can be overridden by setting the variable, either through a command line argument value or a &SET_VAR directive. Setting a variable to null overrides the default value.

The default value can be a constant or a variable reference. It must be quoted if it contains a blank or a special character. It may not be an expression or a function call.

Note

You can set a default value, a type specification, or both a default value and a type specification for each argument. If you set a type specification and no default value, the argument defaults to either a null value or zero, depending on the type specification. If you set a type specification, the default value you specify must be a legal value for that data type. Refer to Table 6-1 for further details.

Table 6-1
Data Types for CPL Arguments

Data Type	*Explanation*	*Default Value*
CHAR	Any character string of as many as 1024 characters, lowercase letters converted to uppercase letters (default).	"
CHARL	Any character string of as many 1024 characters, lowercase letters not converted.	"
TREE	A filename, directory name, or pathname, of as many as 128 characters. The last element of the pathname (that is, the final file or directory name) may contain wildcard characters[1].	"
ENTRY	A filename of as many as 32 characters; may contain wildcard characters[1].	"
DEC	A decimal integer[2].	0
OCT	An octal integer[2].	0
HEX	A hexadecimal integer[2].	0
PTR	Pointer; a virtual address in the format octal/octal (segno/wordno)[3].	7777/0 (the null pointer)
DATE	Calendar date in the format mm/dd/yy.hh:mm:ss or yy-mm-dd.hh:mm:ss.	"
REST	The remainder of the command line.	"
UNCL	All unclaimed items on the command line. (Unclaimed arguments are discussed in Chapter 13.)	"

[1] The &ARGS directive does not perform a wildcard search. Wildcard characters are permitted values for these data types. Using wildcards, you can supply a pathname from the command line to a WILD function within your CPL program. See Chapter 7.
[2] Numeric arguments must be within the range $-2^{31} + 1 \ldots 2^{31} - 1$.
[3] User specified default values are not supported for this data type.

Specifying Data Types for Arguments

The format for specifying the data type for an argument is

&ARGS name-1 : type-1 {; name-n : type-n}

The format for specifying both the data type and default value for an argument is

&ARGS name-1 : type-1 = default-1 {; name-n : type-n = default-n}

The following are examples of these two formats:

```
&ARGS DIR:TREE ; STRING:CHARL

&ARGS DIR:TREE=MYDIR ; STRING:CHARL='This is the default'
```

You can omit either *type* or *default* (or both) for any argument. Spaces may precede or follow the equal sign, colon, and semicolon; they are not required.

The argument *type* specification establishes what type of data can be stored in a variable. The types of data discussed thus far have been character strings and integers. You can specify these data types, and others, in the &ARGS directive.

If you do not specify a data type for an argument, that argument defaults to the character string data type (CHAR). This data type includes all typeable characters as legal values. The default value for CHAR is the null string ('').

Table 6-1 provides a complete list of the available data types and their default values. In most cases, this default value is the null value. You can, of course, override this default value and establish another value as the default value for a particular argument, as described earlier in this chapter. Note, however, that the default value you establish must be a legal value for the data type of the argument.

Examples of Data Type Specification

```
&ARGS DIR:TREE=MYDIR
```

This &ARGS directive declares a variable named DIR. DIR must be a valid treename (that is, a pathname or directory name). Its default value is MYDIR.

```
&ARGS NAME:=XXXXX; NUMBER:DEC
```

This &ARGS directive declares two variables: NAME and NUMBER. NAME is of type CHAR (by default); its default value is XXXXX. NUMBER is of type DEC; any value given for NUMBER must be a decimal integer. Its default value is the system default value, 0.

```
&ARGS DIR:TREE=%.DIR%
```

This &ARGS directive declares a local variable named DIR. The value given must be a valid treename. The default value is the current value of the global variable, .DIR. The global variable file containing .DIR must be active for this default to function correctly. Otherwise, an invocation without arguments produces the following error message:

```
OK, R MYPROG.CPL

CPL ERROR 1017 ON LINE 1. LAST TOKEN WAS: "&ARGS".
In this &ARGS directive, a default value expression contains an
undefined variable reference, or a syntax error in a variable
reference.

SOURCE: &ARGS DIR:TREE=%.DIR%

Execution of procedure terminated. MYPROG (cpl)
ER!
```

```
&ARGS HEXNUM:HEX = 4AB
```

This &ARGS directive declares a variable named HEXNUM, specifies its data type as HEX, and gives it a default value of 4AB (1195 decimal). An argument of data type HEX can only accept values that look like hexadecimal numbers. That is, it accepts strings that contain only the digits 0-9 and the letters A-F (or a-f), and that evaluate to a hexadecimal number between the limits of $-2^{31} + 1$ and $2^{31} - 1$. It cannot distinguish between decimal, octal, and hexadecimal numerals; it accepts all three and interprets them as hexadecimal. For example, it interprets the decimal number 20 as hexadecimal 20 (decimal 32).

```
&ARGS EIGHTBALL:OCT
```

This &ARGS directive declares an octal variable named EIGHTBALL. Octal numbers can contain only the digits 0-7; therefore, a value for EIGHTBALL containing any other digits or characters is rejected with the message

```
Object "9" is not a valid octal integer. (cpl) ER!
```

How Type and Default Checking Works

When you use the &ARGS directive to specify *type* and *default* values, CPL takes the following actions:

1. It reads the command line and assigns the argument values to the variables declared in the &ARGS directive.
2. It checks whether the first argument *(name-1)* was omitted. If the argument was omitted, CPL assigns it its default value, *(default-1)* as specified in the &ARGS directive. If the argument has no specified default, CPL assigns it the system default value, as shown in Table 6-1.

   Note

   Since these are **positional arguments**, the first argument is seen as omitted only when all arguments are omitted. Otherwise, whatever comes first on the command line (after the name of the CPL program itself) is taken as the value of the first argument. (For position independent arguments in CPL, see the section entitled Option Arguments in Chapter 13.)

3. If the first argument was assigned a value in the command line, CPL checks to see if the given value is of the right type. (Acceptable types are defined in Table 6-1.)
4. If the value is not of the right type, CPL displays an explanatory message and returns the user to command level with an `ER!` prompt. For example,

```
OK, R EXAMPLE.CPL 5
Argument "5" is not a valid treename. (CPL)
ER!
```

5. If the value is of the right type, CPL accepts it and moves on to check the next argument (or, if all arguments have been checked and accepted, to execute the next directive or command).

An Example

Assume that X.CPL contains the following directive:

```
&ARGS WHO:ENTRY=JONES; HOWMANY:DEC=10
```

The following table shows some invocations of X.CPL and their results:

Invocation	*Argument Values*
R X.CPL SMITH 20	WHO = SMITH HOWMANY = 20
R X.CPL CLARK	WHO = CLARK HOWMANY = 10 (default)
R X.CPL	WHO = JONES (default) HOWMANY = 10 (default)
R X.CPL 50	Error generated; 50 is not a valid filename.

Specifying the REST Data Type for Arguments

REST is a special argument data type that allows you to pass the remainder of the command line (after all other arguments have been read) to a single variable. Using a REST argument, you can pass PRIMOS option arguments as positional arguments, without quoting them. The rules for REST arguments are as follows:

- Only one REST argument is permitted in an &ARGS directive.
- The REST argument must be the last argument in the directive.

For example,

```
&ARGS FILENAME:TREE; OTHER_ARGS:REST
```

In this example, the first argument on the command line must be a filename (or pathname). Everything that follows the first argument on the command line becomes the value of OTHER_ARGS. No quotation marks are required.

A Sample Program

A sample program, using the &ARGS directive, spools a file to a printer and permits you to specify additional printing options at runtime:

```
/* Usage R SPL.CPL filename other_args
&ARGS FILENAME:TREE; OTHER_ARGS:REST
SPOOL %FILENAME% -AT CAROUSEL %OTHER_ARGS%
```

Here are two sample terminal sessions. The first does not use the REST argument. The second assigns the value –FORM NOW –LIST to the REST argument. Note that this argument value does not have to be quoted or unquoted.

```
OK, R SPL.CPL MYFILE
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 3 added to queue, 2 records : <MAIN1>GLENN>MYFILE

OK, R SPL.CPL MYFILE -FORM NOW -LIST
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 4 added to queue, 2 records : <MAIN1>GLENN>MYFILE

System XXX
Request Time User           File                 No Size  State
------- ---- -------------- -------------------  -- -----  -------
7 Apr 87
1       1241 SMITH          MEMO.41              1   14
2       1235 JONES          CL-DEPT.O            2   6     Defer          QA.TST
3       1242 BROWN          MYFILE               1   2     Print
OK,
```

Default Values for REST Arguments

Like any other type of argument, a REST argument can be given a default value. For example,

```
&ARGS FILENAME:TREE;  OTHER_ARGS:REST= -LIST
SPOOL %FILENAME% -AT CAROUSEL %OTHER_ARGS%
```

A sample terminal session with this program looks like this:

```
OK, R SPL2.CPL MYFILE
[SPOOL Rev. 21.0 Copyright (c) 1987, Prime Computer, Inc.]
Request 3 added to queue, 2 records : <MAIN1>GLENN>MYFILE

System XXX
Request Time User           File                 No Size  State
------- ---- -------------- -------------------  -- -----  -------
7 Apr 87
1       1241 SMITH          MEMO.41              1   14
2       1235 JONES          CL-DEPT.O            2   6     Defer          QA.TST
3       1242 BROWN          MYFILE               1   2     Print
OK,
```

Although the command line in this example does not specify the –LIST option, the program lists the spool queue. This is because the default value for the REST argument is –LIST.

7
Processing Groups of Files

This chapter describes CPL features that you can use to select multiple file system objects. It explains the PRIMOS conventions for filename suffixes and wildcarding, and how to use these with the BEFORE, AFTER, and WILD functions.

Selecting Multiple Files and Directories

PRIMOS file-naming conventions help you set up your directory so that you can see easily what types of files it contains. By convention, files with similar uses are given the same file suffix.

The Prime wildcard facility lets you access groups of similarly named files (or subdirectories) within a directory. To access such a group of files, you specify the part of the filename that is the same in all of the files in the group, and substitute a wildcard character for the part of the filename that is different for each file.

CPL provides functions that allow you to take advantage of suffix naming conventions and wildcards to perform operations on selected groups of files or directories.

Using Suffixes: The BEFORE and AFTER Functions

A filename can consist of several components. Each component of a filename (except the first component) begins with a dot. The final component of a filename is the suffix. The other components of the filename are known as the base name. Filenames with more than three components are not recommended.

PRIMOS file-naming conventions use suffixes to identify various sorts of files. Suffixes, such as .RUN and .BIN, identify files with specific properties; you can also create your own suffixes to identify files that you consider similar.

CPL's BEFORE and AFTER functions make it easy to break a filename into its separate components. For example, you can use these functions to separate the name of a source file into the base name and the compiler name suffix, dropping the dot in the process.

The BEFORE Function

The format of the BEFORE function is

[BEFORE string-1 string-2]

The BEFORE function returns the part of *string-1* that occurs before *string-2*. For example,

```
[BEFORE ABCD C]
```

returns

AB

Hence

```
&S FILE := [BEFORE SOURCE.F77    .]
```

sets the value of FILE to SOURCE.

If *string-2* is not part of *string-1*, the BEFORE function returns the entire *string-1*. For example,

```
[BEFORE SOURCE.F77    ,]
```

returns

SOURCE.F77

If *string-2* represents the leftmost characters in *string-1*, the BEFORE function returns the null string.

The AFTER Function

The format of the AFTER function is

[AFTER string-1 string-2]

The AFTER function returns as its value the portion of *string-1* that occurs after *string-2*. For example,

```
[AFTER ABCD C]
```

returns

D

Hence,

```
&S COMPILER := [AFTER SOURCE.F77    .]
```

sets the value of COMPILER to F77.

If *string-2* is not part of *string-1*, or if *string-2* represents the rightmost characters in *string-1*, the AFTER function returns the null string. For example,

```
[AFTER SOURCE    .]
```

returns

```
,,
```

An Example

Here is an example of these functions in action. The CPL program shown below compiles, links, and runs any V-mode or I-mode program, using the filename as its argument. This program, named CLR_ALL.CPL, is a revision of the "compile, load, and run" program shown in Chapter 2.

```
/* CPL program to compile, BIND and
/* execute a program in any language
/* Usage: R CLR_ALL filename
   /*
   &ARGS FILENAME; OPTION_LIST:REST
   &S COMPILER := [AFTER %FILENAME%    .]
   &S SOURCE := [BEFORE %FILENAME%    .]
   /*
   /* Check for compiler suffix
   &IF [NULL %COMPILER%] ~
   &THEN &SET_VAR COMPILER := [RESPONSE 'Please specify compiler']
   /* compile the program
   %COMPILER%  %FILENAME%  -64V  -B %SOURCE%.BIN  %OPTION_LIST%
   /* set up language library name for BIND
   &SELECT %COMPILER%
       &WHEN PL1, CBL
          &S LIBFILE := %COMPILER%LIB
       &WHEN PL1G, VRPG
          &S LIBFILE := %COMPILER%LB
       &WHEN CC
          &S LIBFILE := C_LIB
       &OTHERWISE
          &S LIBFILE := PASLIB
   &END
```

```
/*  run the BIND linker
&DATA BIND
   LOAD %SOURCE%        /* BIND finds file source.BIN
   LI %LIBFILE%
   LI
   DYNT -ALL
   FILE  %SOURCE%.RUN  /* BIND names output file source.RUN
&END
/*  execute the program
R %SOURCE%    /* execute runfile
&RETURN
```

Wildcards

Wildcards allow you to specify groups of files within a directory using a single wildcard name. A wildcard name is a file or subdirectory name in which one or more characters have been replaced by one or more wild characters. A wild character may represent any other character (or characters), according to the rules shown in Table 7-1. A number of examples follow.

Table 7-1
Wild Characters

Character	*Function*
@	Replaces any number of characters within one component of a filename or directory name. Stops matching at the dot (.) that separates a name and its suffix.
@@	Replaces any number of characters in any number of components within a file or directory name.
+	Replaces a single character.
^	Negation character. The negation character must be the first character in the wildcard name. A wildcard name that begins with ^ matches all names that *do not* match the rest of the wildcard name.

Some Examples

Assume a directory, MYDIR, that contains the following files:

FOO.CBL	BARR1.CBL	BARR1.RUN
BARR2.CBL	BARR2.RUN	FOO.RUN
CLR.CPL	EDD.CPL	SCROLL
EDD.COMO	EDD.COMO.OLD	

The wildcard name FOO.@ matches all two-component names within MYDIR that begin with FOO.:

```
FOO.CBL          FOO.RUN
```

The wildcard name @.RUN matches all two-component names that end with .RUN:

```
BARR1.RUN        BARR2.RUN        FOO.RUN
```

The wildcard name BARR+.CBL matches

```
BARR1.CBL        BARR2.CBL
```

The wildcard name BARR+.@ matches

```
BARR1.CBL        BARR2.CBL
BARR1.RUN        BARR2.RUN
```

The wildcard name EDD.@ matches

```
EDD.CPL          EDD.COMO
```

EDD.@ does not match EDD.COMO.OLD, because the single @ cannot cross the dot (.) to match the suffix, OLD.

The wildcard name ED@@ matches

```
EDD.CPL          EDD.COMO         EDD.COMO.OLD
```

The wildcard name @@L matches all names that end with L:

```
FOO.CBL          BARR1.CBL        BARR2.CBL
CLR.CPL          EDD.CPL          SCROLL
```

The wildcard name @L matches all one-component names that end in L:

```
SCROLL
```

The wildcard name ^@.CPL matches all files in the directory that do *not* end with .CPL, or that do not have two components:

```
FOO.CBL          BARR1.CBL        BARR1.RUN
FOO.RUN          BARR2.CBL        BARR2.RUN
SCROLL           EDD.COMO         EDD.COMO.OLD
```

The wildcard name @@ matches all names in the directory, regardless of the number of components they contain:

FOO.CBL	BARR1.CBL	BARR1.RUN
BARR2.CBL	BARR2.RUN	FOO.RUN
CLR.CPL	EDD.CPL	SCROLL
EDD.COMO	EDD.COMO.OLD	

Using Wildcards: The WILD Function

CPL's WILD function produces a list of all names within a directory that match one or more wildcard names. It has two forms, discussed below. The first form returns all matching names at once, in a single list. Names within the list are separated by blanks. The second form, which uses the –SINGLE argument, returns one matching name per invocation until the list of names is exhausted.

The reason for the two forms of the WILD function is that the list produced by the WILD function is limited to 1024 characters. If a longer list is produced, an error occurs that aborts the CPL program. Because the WILD function with the –SINGLE argument returns one name at a time, it can handle cases that would produce over-long lists.

The Basic WILD Function

The basic format of the WILD function is

[WILD wild-name-1 {...wild-name-n} {options}]

The WILD function matches the wildcard names *wild-name-1* through *wild-name-n*. *wild-name-1* can be either a wildcard filename or a wildcard pathname. If *wild-name-1* is a full pathname, all the wildcard names are searched for in the directory that *wild-name-1* specifies. If *wild-name-1* is a filename, all the wildcard names are searched for in the current directory. The WILD function cannot use the search rules facility to search multiple directories. *wild-name-2* through *wild-name-n* specify additional wildcard names within the directory specified in *wild-name-1*; they may not be pathnames. For example,

```
ATTACH MYDIR
&SET_VAR SOURCES := [WILD @.CBL @.PMA]
```

This example creates a list of all CBL and PMA source files in the currently attached directory (MYDIR), and stores the list in the variable, *SOURCES*.

```
ATTACH JONES
&SET_VAR SOURCES := [WILD SMITH>@.CBL @.PMA]
```

This example creates a list of all CBL and PMA source files in directory SMITH, and stores the list in the variable, *SOURCES*.

options are optional arguments for the WILD function that place limits on the matching of objects by the wildcard names. You can specify one or more *options* in any sequence. The available *options* are as follows:

Option	*Meaning*
-ACL	Selects only ACLs.
-AFTER date	Selects only objects created or modified after the date specified by *date*. The *date* specified here is compared with each file's DTM (Date and Time Modified) attribute. The format for *date* is MM/DD/YY.
-BEFORE date	Selects only objects created or last modified before the specified date. The format for *date* is MM/DD/YY.
-DIRECTORY	Selects only directories.
-FILE	Selects only files.
-SEGMENT DIRECTORY	Selects only segment directories.

Some examples using *options* are as follows:

```
SET_VAR .OBJ := [WILD @@ -SEGDIR]
```

This example creates a list containing the names of all segment directories in the current directory, for example, FOO.SEG.

```
SET_VAR .OBJ := [WILD @.CBL -BF 05/30/86]
```

This example lists all CBL source files created or last modified before May 30, 1986, for example, FOO.CBL and BARR1.CBL.

```
SET_VAR .OBJ := [WILD MYDIR>@@ -DIR]
```

This example lists all subdirectories in the directory MYDIR, for example, REPORTS MEMOS OTHER_STUFF.

The WILD Function With the -SINGLE Argument

The -SINGLE argument causes the WILD function to return object names one at a time, rather than writing them into a list. Use it when you think that WILD's list may overrun its limit of 1024 characters or when it is more convenient to deal with the filenames one at a time.

The format of this version of the WILD function is

[WILD wild-name-1 {...wild-name-n} {options} -SINGLE unit-var]

You specify the *wild-names* and *options* arguments in the same way as those for the basic WILD function.

Following the –SINGLE argument, you specify a *unit-var* variable name. The first invocation of WILD automatically assigns this *unit-var* variable the number of the file unit that WILD used to open the directory. Subsequent invocations of WILD use this file unit number to locate this open directory, so that WILD can continue searching.

You must set *unit-var* to zero before invoking WILD for the first time. Setting *unit-var* to zero distinguishes that first call (in which WILD opens the file unit and returns the first matching name) from subsequent calls (in which WILD takes the next name from the open file unit). An example of the use of the WILD function with the –SINGLE option follows:

```
&SET_VAR UN := 0
&SET_VAR ONE_NAME := [WILD @.LIST -SINGLE UN]
```

The first &SET_VAR directive defines the variable *UN* and sets it to zero. The second &SET_VAR directive causes CPL to perform the following steps:

1. Open the current directory on some available unit.
2. Change the value of *UN* to the number of the file unit used.
3. Find the first listing file in the directory.
4. Set the value of the variable *ONE_NAME* to the name of the first listing file in the directory.

Subsequent invocations of the same function call return the second listing file, the third listing file, and so on, until there are no more listing files to be found. Then WILD returns a true null string, and closes the directory file unit.

Using the WILD Function in Loops

Why would you want to produce a list of file or directory names? One reason is that you want to do something with each of the files or directories on the list. For example, you might want to spool all your RUNOFF files, obtain a listing of the contents of each of your subdirectories, or update a group of reports or data files.

You can easily perform these tasks by using the WILD function to control a loop, thus performing the desired process once for each item on the list.

CPL offers a variety of loops, which are discussed in detail in Chapter 9. Among these loops are two that work most efficiently with the WILD function: the &DO &LIST loop and the &DO &ITEMS loop. Use the &DO &LIST loop with the basic WILD function to get the entire list of file or directory names at one time. Use the &DO &ITEMS loop with WILD's –SINGLE argument. An example of each of these types of loop is shown here. Full explanations of &DO &LIST and &DO &ITEMS are given in Chapter 9.

Example of &DO &LIST Loop: The following program spools all the RUNOFF files (ending in .RUNO) that are located in the user-specified directory PATH:

```
&ARGS PATH                    /* Specify directory
&DO X &LIST [WILD %PATH%>@.RUNO -FILES]
   SPOOL %PATH%>%X%    /* Spool each file in turn
&END                           /* End loop
&RETURN                        /* End program
```

Example of &DO &ITEMS Loop: If you have many RUNOFF files in your directory, you could write the same program with a &DO &ITEMS loop, as follows:

```
&ARGS PATH                  /* Specify directory
&SET_VAR UNIT := 0          /* Initialize variable for file unit
&DO X &ITEMS [WILD %PATH%>@.RUNO -FILES -SINGLE UNIT]
   SPOOL %X%               /* Spool each item
&END                         /* End loop
&RETURN                      /* End program
```

For further examples of loops using the WILD function, see Chapter 9.

8
Decision Making

This chapter describes various forms of the &IF and &SELECT directives. The &IF and &SELECT directives are among the most powerful features of CPL. When the CPL interpreter encounters one of these directives, it tests a value, then uses the results of that test to decide what line of the program to execute next. The &IF directive performs a test to select one of two possible statements for execution. The &SELECT directive performs a test to select one of many possible statements for execution.

The basic &IF directive is explained in Chapter 2. This chapter assumes an understanding of the material in that chapter.

&IF and &SELECT are only two of the control directives that CPL provides. Table 8-1 shows the complete list of available CPL control directives.[1]

Table 8-1
Control Directives

Directive	*Action*
&IF...&THEN...&ELSE	Chooses between two alternatives. &IF statements may be nested to allow further decisions to be made on the basis of the former decisions.
&SELECT	Chooses among any number of alternatives.
&DO group	Allows a group of statements to be treated logically as if it were a single statement.
&DO loop	Allows a group of statements to be executed • *n* times, with *n* as a pre-set number • *n* times, with *n* computed at runtime • While some logical expression is true (or false) • Until some logical expression becomes true (or false) • Until a list of items is exhausted
&GOTO	Allows arbitrary transfer of control from one place within a program to another.

[1] This chapter does not describe control directives that do not perform a test, such as the &GOTO directive and the &DO group. Those directives are described in Chapter 2. This chapter also does not describe directives that perform loop operations. &DO loops are discussed in Chapter 9.

&IF Directives Using Logical Operators

A single &IF...&THEN...&ELSE directive can choose between any two alternatives. For example,

```
&IF %A% > 10 &THEN R BIGNUM
  &ELSE R SMALLNUM
```

You can combine multiple expressions into a single test by the use of the logical AND (&) and inclusive OR (|) operators. When logical AND is used, both expressions must be true for the test to be true. When inclusive OR is used, if either expression (or both) is true, the test is true.

For example,

```
&IF %A% > 10 & %B% > 10 ~
  &THEN R BIGNUM
  &ELSE R SMALLNUM
```

In the previous example, BIGNUM is executed if the values of both *A* and *B* are greater than 10.

```
&IF %A% > 10 | %B% > 10 ~
  &THEN R BIGNUM
  &ELSE R SMALLNUM
```

In this example, BIGNUM is executed if the value of either *A* or *B* is greater than 10.

Note that each operator must be preceded and followed by at least one space. You can control the sequence in which multiple expressions are evaluated by enclosing expressions in parentheses; the innermost nested expression is always evaluated first. Expressions without parentheses or expressions on the same level of nested parentheses are evaluated from left to right.

A Sample Program

The following CPL program uses logical ANDs and ORs to decide which payroll program to run. If the program is run on March 31, June 30, September 30, or December 31, it generates a quarterly report. If the program is run on December 31, it also generates the annual report. It always runs a standard payroll program. (For details on the DATE function, used by this program, see Chapter 12.)

```
&SET_VAR MONTH := [DATE -MONTH]
&SET_VAR DAY := [DATE -DAY]
/* If this is the end of the quarter,
/* then generate the Quarterly Report
&IF ( ( ( %DAY% = 31 ) ~
      & ( ( %MONTH% = MARCH ) | ( %MONTH% = DECEMBER ) ) ) ~
    | ( ( %DAY% = 30 ) ~
      & ( ( %MONTH% = JUNE ) | ( %MONTH% = SEPTEMBER ) ) ) ) ~
   &THEN R QUARTERLY.RUN
/* If this is the end of the year,
/*  then generate the Annual Report
```

```
&IF ( ( %DAY% = 31 ) & ( %MONTH% = DECEMBER ) ) ~
   &THEN R ANNUAL.RUN
/* Always run the payroll program
R PAYROLL.RUN
&RETURN
```

Nested &IF Directives

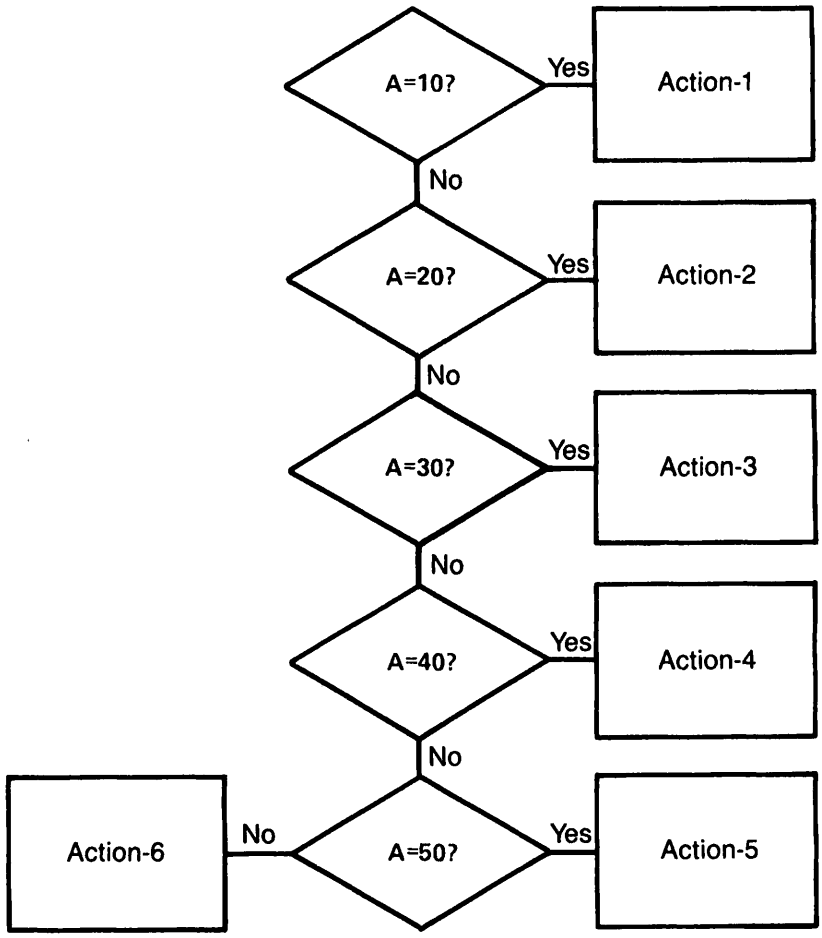
If you need to choose among three or more alternatives, you may use either a &SELECT directive or nested &IF directives. Nested &IF directives use another &IF directive as the argument to the &THEN clause, the &ELSE directive, or both. For example,

```
&IF %A% > 10 &THEN R BIGNUM
  &ELSE &IF %A% = 10 &THEN R TENPROG
    &ELSE R SMALLNUM
```

In this example, BIGNUM is executed if the value of *A* is greater than 10; TENPROG is executed if the value of *A* is equal to 10; and SMALLNUM is executed if the value of *A* is less than 10. (Note that each &ELSE directive matches, or depends on, the &THEN clause immediately preceding it.) There is no limit to the number of &IF directives that can be nested in this manner. Here is another example, from the field of education:

```
&IF %AVERAGE% > 89 &THEN &S GRADE := A
  &ELSE &IF %AVERAGE% > 79 &THEN &S GRADE := B
    &ELSE &IF %AVERAGE% > 69 &THEN &S GRADE := C
      &ELSE &IF %AVERAGE% > 59 &THEN &S GRADE := D
        &ELSE &S GRADE := F
```

Figure 8-1 diagrams nested &IF directives.




```
&IF %A% = 10 &THEN action-1
             &ELSE &IF %A% = 20 &THEN action-2
                   &ELSE &IF %A% = 30 &THEN action-3
                         &ELSE &IF %A% = 40 &THEN action-4
                               &ELSE &IF %A% = 50 &THEN action-5
                                     &ELSE action-6
```

Figure 8-1
Nested &IF Directives

Nested &IF and &ELSE Directives

A more complex form of nested &IF directive is one in which both &IF and &ELSE directives are nested. With this construction, use the following rule for matching &THEN and &ELSE directives: An &ELSE directive matches the last &THEN directive preceding it that is not already matched by an &ELSE directive. Examples of such matching are shown in Figure 8-2.

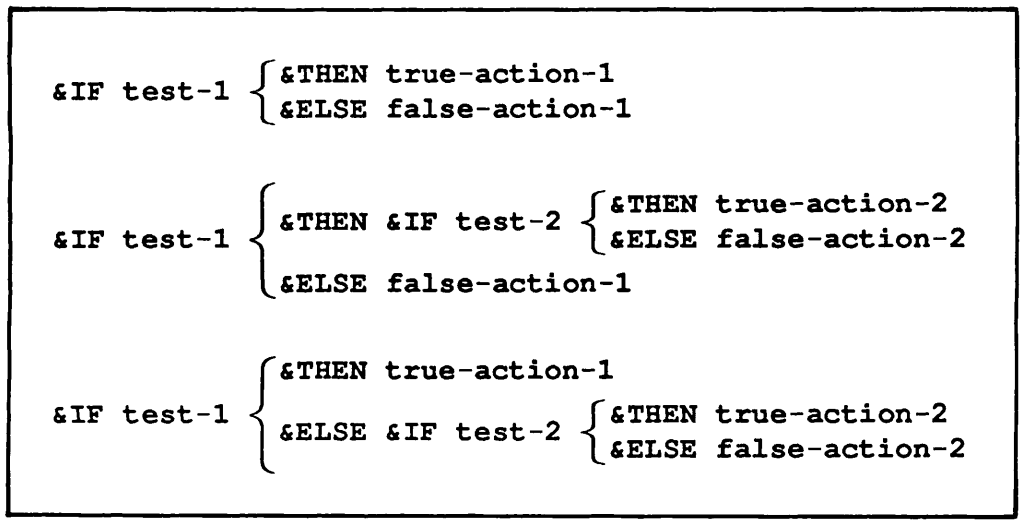


Figure 8-2
Matching of &THEN and &ELSE Statements

Here is an example of nested &IF and &ELSE directives:

```
&IF %A% > 50                    /*1st &IF tests value of A              ~
  &THEN                         /*take this path if A > 50              ~
     &IF %B% > 50               /*nested &IF tests value of B            ~
       &THEN RESUME MAXIMUM     /*A and B both > 50
       &ELSE RESUME MAJOR       /*A > 50, B <= 50
&ELSE                           /*take this path if A <= 50              ~
   &IF %C% > 10                 /*another 2nd level test                 ~
       &THEN RESUME MINOR       /*A <= 50, C > 10
       &ELSE RESUME MINIMUM     /*A <= 50, C <= 10
```

The decisions made by this example are diagrammed in Figure 8-3. Notice how the decision levels shown in this figure are reflected in the indentation of the example. Such indentations help you remember which &THEN and &ELSE pair goes with each &IF. Also note where line continuation characters (~) are required in this example.

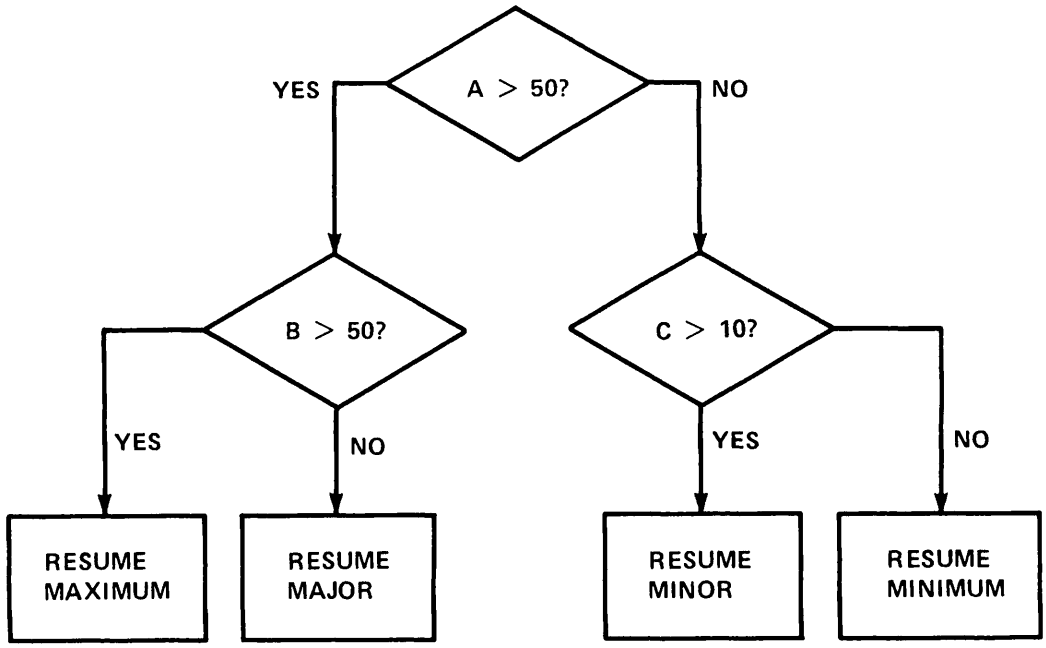


Figure 8-3
Nested &IF and &ELSE Directives

The &SELECT Directive

&IF directives can handle any situation in which you perform a test and then take action based on the result of that test. However, deeply nested &IF directives and &IF directives that contain many logical ORs are difficult to read. Therefore, when you want to choose between many alternatives, use the &SELECT directive, which provides a clear presentation of the test condition, its possible results, and the action to be taken in each case. Figure 8-4 diagrams the &SELECT directive.

&SELECT Directive Format

The format of the &SELECT directive is as follows:

```
&SELECT key-expression
  &WHEN expression-1a  {,expression-1b, expression-1c...}
     action-1
  &WHEN expression-2a  {,expression-2b, expression-2c...}
     action-2
  .
  .
  .
  {&OTHERWISE
     action-n}
&END
```

key-expression is the value that is used to select an option. In a &SELECT directive, each option represents one possible value of *key-expression*. *key-expression* may be a variable reference, a function call, or a string, arithmetic, or logical (Boolean) expression. For example,

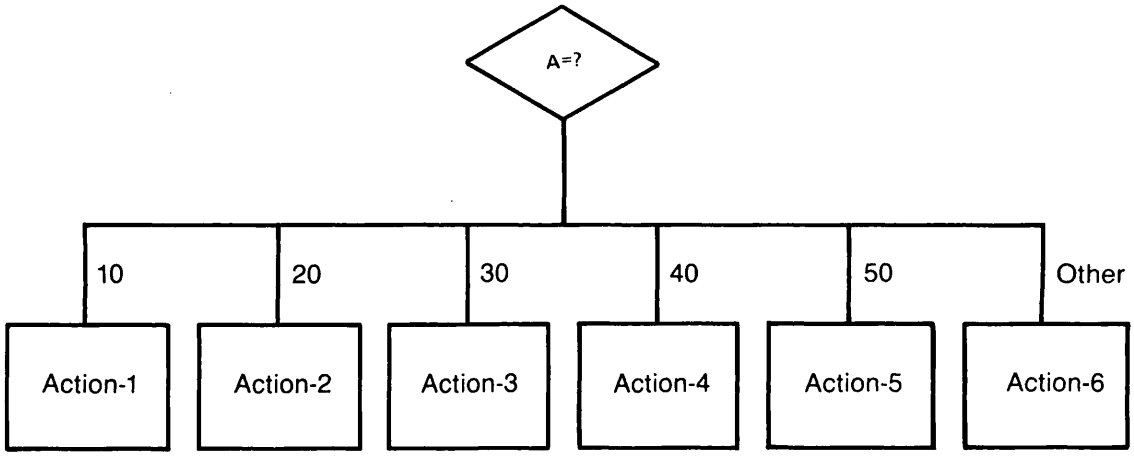
```
&SELECT %COMPILER%

&SELECT %A% + %B%

&SELECT [DATE -DOW]
```

expression-1 through *expression-n* represent possible values of *key-expression*. An *expression* can be a literal, a variable reference, a function call, or a string, arithmetic, or logical expression. A &SELECT directive can include *expressions* of different types; for example, one &WHEN clause can test for a Boolean value while others test for integer values. *expression* cannot be a CPL directive, such as a &DO group.

In the example &SELECT %COMPILER%, each *expression* represents a possible value of the variable, %COMPILER%. In the example &SELECT [DATE –DOW], each *expression* represents a possible result returned by the DATE function. In the example &SELECT %A% + %B%, each *expression* is an integer or an arithmetic expression representing a possible value for the sum of the current values of *A* and *B*.




```
&SELECT %A%
   &WHEN 10
         action-1
   &WHEN 20
         action-2
   &WHEN 30
         action-3
   &WHEN 40
         action-4
   &WHEN 50
         action-5
   &OTHERWISE
         action-6
   &END
```

Figure 8-4
The &SELECT Directive

action-1 through *action-n* are CPL statements. One of these statements is executed if the *key-expression* value matches the *expression* value for that *action*. Only one *action* is executed in each &SELECT directive. *action-1* through *action-n* may be any type of CPL statement; for example, a PRIMOS command, a CPL directive, a &DO group, or a &DATA group.

The optional &OTHERWISE clause executes its *action* if the *key-expression* value matches none of the *expression* values.

How a &SELECT Directive Works

When the CPL interpreter reads a &SELECT directive, it takes the following actions:

1. It evaluates *key-expression.*
2. It searches through the &WHEN directives until it finds an *expression* that is equal to the current value of *key-expression.*
3. When it finds a match, it executes the *action* statement immediately following that &WHEN directive.
4. As soon as it finds that first match and executes the accompanying statement, it drops to the end of the &SELECT group. It continues processing the CPL file by executing the statement that follows the &SELECT group &END directive.
5. If it finds no match, but does find an &OTHERWISE directive, it executes the *action* statement immediately following the &OTHERWISE directive.
6. If it finds neither a match nor an &OTHERWISE directive, it executes none of the &SELECT group's statements. It continues processing the CPL file by executing the statement that follows the &SELECT group &END directive.

Multiple Expressions in a &WHEN Clause

Following a &WHEN directive, you can list more than one *expression.* Multiple expressions are separated by commas. If any one of the listed *expressions* matches the *key-expression*, the *action* for that &WHEN clause is executed. For example,

```
&SELECT %A%
&WHEN 10, 20, 30
     R MYACTION
```

Note that a variable reference used in a &WHEN clause can only evaluate to a single expression. For example, assume that variable *B* has the value '5, 10, 15', and that it is used in a &SELECT directive beginning

```
&SELECT %A%
&WHEN %B%, 20, 25
```

This example tests the value of *A* three times: once against the quoted character string '5, 10, 15', once against the integer value 20, and once against the integer 25. It does *not* test for the integers 5, 10, or 15. If the value of *A* is 20, the &WHEN test is TRUE; if the value of *A* is 10, the &WHEN test is FALSE.

Logical Expressions in a &WHEN Clause

Both the *key-expression* and the &WHEN clause *expressions* can represent logical (Boolean) values. A logical value can be TRUE, FALSE, T, or F (uppercase or lowercase). The null value ('') is evaluated as FALSE. The following are two examples of &SELECT directives that use logical values:

```
&SELECT TRUE
    &WHEN %A% < 5
        R SMALLNUM
    &WHEN %A% >= 5
        R BIGNUM
&END
```

In this first example, each &WHEN clause *expression* is an arithmetic expression that either evaluates TRUE or FALSE. If %A% = 4, this program runs SMALLNUM; if %A% = 6, it runs BIGNUM.

You can write the same program another way:

```
&SELECT %A% < 5
    &WHEN TRUE
        R SMALLNUM
    &WHEN FALSE
        R BIGNUM
&END
```

In this second example, it is the *key-expression* that is evaluated as TRUE or FALSE. If %A% = 4, this program runs SMALLNUM; if %A% = 6, it runs BIGNUM.

&SELECT Examples

The first example demonstrates the use of multiple expressions in &WHEN clauses. In this example, the &SELECT directive adds the values of *A* and *B*, then matches the sum against the specified integers.

```
&ARGS A:DEC;  B:DEC
&SELECT %A% + %B%
  &WHEN 10, 20, 30, 40, 50
    RESUME RAND1
  &WHEN 5, 15, 25, 35, 45
    RESUME RAND2
```

```
  &WHEN 60, 70, 80, 90, 100
    RESUME RAND3
  &WHEN 55, 65, 75, 85, 95
    RESUME RAND4
  &OTHERWISE
    RESUME RAND5
  &END
```

The second example applies the &SELECT directive to the academic problem of turning students' numeric averages into letter grades. It uses Boolean expressions for its tests. Each Boolean expression produces a value of either TRUE or FALSE. The first TRUE expression thus equals the *key-expression* (&SELECT TRUE) and ends the search.

```
&ARGS AV
&SELECT TRUE
  &WHEN %AV% <= 60
   &S GRADE := F
  &WHEN %AV% <= 70
   &S GRADE := D
  &WHEN %AV% <= 80
   &S GRADE := C
  &WHEN %AV% <= 90
   &S GRADE := B
  &OTHERWISE
   &S GRADE := A
&END
```

The final example takes as input the name of a month of the year and responds with the number of days in that month.

```
&ARGS MONTH
&SELECT %MONTH%
   &WHEN ''
     &RETURN &MESSAGE 'No month specified'
   &WHEN FEBRUARY
     &DO
     &IF [QUERY 'leap year'] &THEN ~
        &S DAYS := 29
     &ELSE &S DAYS := 28
     &END
   &WHEN SEPTEMBER, APRIL, JUNE, NOVEMBER
     &S DAYS := 30
   &OTHERWISE
     &S DAYS := 31
&END
TYPE Number of days is %DAYS%
```

9
Loops

This chapter describes various forms of the &DO directive that are used for performing loop operations. Loops are useful when you want some operation (or operations) to be carried out repeatedly, with (or without) minor variations; for example, when you want many source files compiled or spooled, or many lines in a data file updated.

CPL provides a wide variety of loops. This chapter contains

- An overview of the sorts of loops CPL provides, the format of loops in general, and the behavior of loops in general
- A detailed explanation of how to use each kind of loop CPL provides

Overview

CPL provides the following sorts of loops:

- The counted &DO loop. For example, **&DO I := 1 &TO 100 &BY 5**
- The &DO &WHILE loop. For example, **&DO &WHILE %J% <= 100**
- The &DO &UNTIL loop. For example, **&DO &UNTIL %J% > 100**
- The counted &DO loop combined with a &WHILE or &UNTIL test. For example, **&DO I := 1 &TO 100 &WHILE %J% > 20**
- The &DO &REPEAT loop, which is usually combined with a &WHILE or &UNTIL test. For example, **&DO I := 50 &REPEAT %I% *** %I% &WHILE %I% <= 100000
- The &DO &LIST loop. For example, **&DO I &LIST %var_list%** or **&DO I &LIST 5 36 489**
- The &DO &ITEMS loop, a variant of the &DO &LIST loop. For example, **&DO I &ITEMS [WILD @.F77 -SINGLE UNIT]**

Loop Formats

All loops have the same basic format:

```
&DO {index-var} loop-instructions
   statement-1
   statement-2
      .
      .
      .
   statement-n
&END
```

index-var is a counter that is incremented each time the loop is performed. *index-var* can be any valid variable name. It may not be an expression. The use of an *index-var* is required in all loops except the &DO &WHILE and &DO &UNTIL loops.

loop-instructions contain

- A starting value for *index-var* (if *index-var* is used)
- A method for incrementing *index-var* (if *index-var* is used)
- One or more tests for loop completion

The presence of *index-var* and *loop-instructions* distinguish the iterative &DO loop from the simple &DO group. When the CPL interpreter reads the word &DO, it checks for *index-var* and *loop-instructions*. If it finds neither, it executes the &DO group once. If it finds *index-var* alone, or if it finds incorrect instructions, it displays an error message. If it finds syntactically correct *loop-instructions*, it prepares to execute the loop from zero to an infinite number of times, according to the instructions.

Loop Execution

When CPL encounters a loop statement during program execution, it performs the following actions. (Figure 9-1 contains the corresponding flow chart.)

1. If *index-var* is present, CPL sets it to its initial value. CPL tests this value for loop completion. If the loop is complete, control passes to the next statement after the loop.
2. If a &WHILE clause is present, CPL tests it for loop completion. If the loop is complete, control passes to the next statement after the loop.
3. If the loop has not completed, CPL executes *statement-1* through *statement-n*.
4. When execution reaches the &END statement that closes the loop, CPL tests the &UNTIL clause (if there is one). If it tests out TRUE, the loop is complete. Execution continues with the next statement after the loop.
5. If no &UNTIL clause is TRUE, execution returns to the top of the loop.

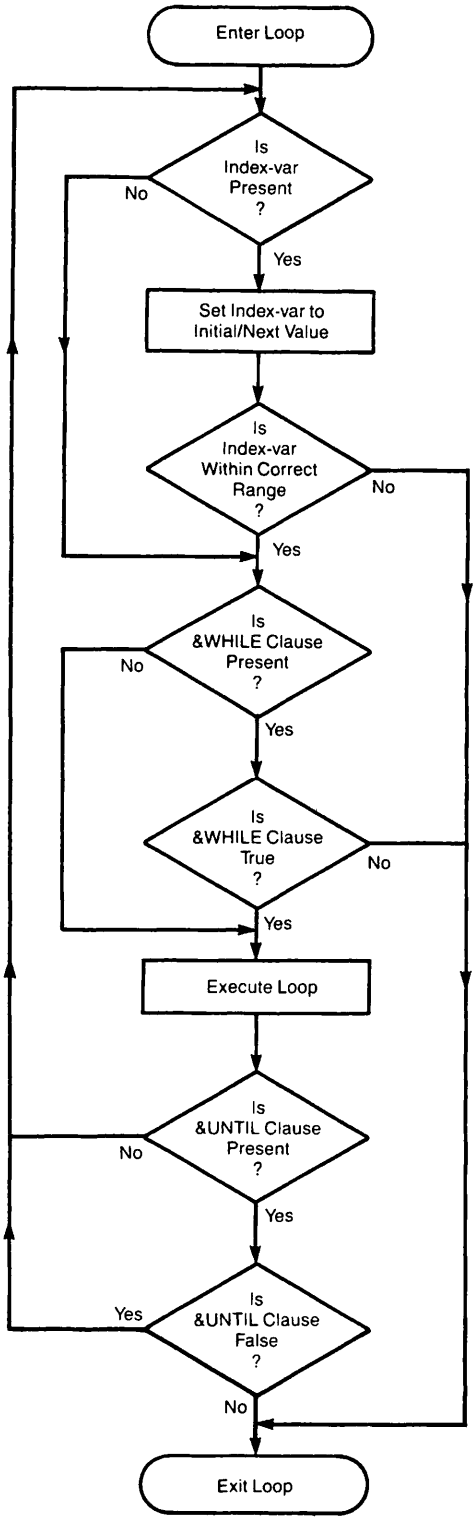


Figure 9-1
Flow of Control in CPL Loops

6. At the top of the loop, CPL sets *index-var* to its next value, then tests the *index-var* and/or &WHILE clause for loop completion.
7. If these tests are not TRUE, CPL executes *statement-1* through *statement-n* again.
8. And so on, until some test for completion (or some &GOTO or &RETURN statement inside the loop) stops execution of the loop. If no test (or directive) ever stops the loop, the loop executes "forever" — that is, until the user presses CONTROL-P or the BREAK key, or until someone forcibly terminates or logs out the CPL process.

When a loop terminates, *index-var* retains the last value it reached during execution of the loop. In a counted loop, this is the first out-of-range value reached. For example, for the loop &DO I := 1 &TO 10, the value of I at normal termination is 11. When &DO &LIST and &DO &ITEMS loops terminate, their *index-vars* are null.

If a loop is halted by execution of a &RETURN or &GOTO, *index-var* retains whatever value it had when the &RETURN or &GOTO was executed.

Note

You may write a &GOTO that exits from a loop, going from a point inside the loop to a point outside it. You may *not* use a &GOTO to enter a loop; that is, you may not &GOTO a point inside a loop from any point outside the loop. (If you write such a &GOTO into a CPL program, you get an error message from the interpreter when you try to execute the program.) Figures 9-2 and 9-3 show examples of legal and illegal uses of &GOTO.

```
&DO I : = 1   &TO 100000 &BY 2
     .
     .
     .
     &GOTO EXIT
     &END
     .
&LABEL EXIT
     .
     .
     .
```

Figure 9-2
Legal Use of &GOTO

```
&GOTO THERE
   &DO I : = 1   &TO 100000 &BY 2
   .
   .
   .
   & LABEL THERE
   .
   .
   .
   &END
```

Figure 9-3
Illegal Use of &GOTO

Nested Loops

Loops in CPL may be nested; that is, one loop may be written inside another. Nested loops are shown in the following sample program, called NEST.CPL:

```
&DO A := 10 &TO 30 &BY 10  /* Start outer loop
  TYPE %A%
  &DO B := 1 &TO 3         /*  Start inner loop
      TYPE %B%
      &END                 /*  End inner loop
  &END                     /*  End outer loop
```

When loops are nested, the outer loop begins executing first. When it reaches the inner loop, the inner loop executes until it is completed. Then the outer loop continues executing. The inner loop *always* ends first. Loops cannot overlap; the inner loop is always completely enclosed in the outer loop.

Each time the outer loop executes, the inner loop is re-initialized, and executes from start to completion again. When the outer loop does not execute, the inner loop cannot execute.

Here is what happens when you run NEST.CPL:

```
OK, RESUME NEST.CPL
10
1
2
3
20
1
2
3
30
1
2
3
OK,
```

Loops may be nested as deeply as you can keep track of them.

Counted Loops

Counted loops have the format

&DO index-var := start-value &TO stop-value {&BY increment}
{&WHILE test} {&UNTIL test}

index-var is any valid variable name. *start-value* and *stop-value* may be integers, expressions, variable references, or function calls. They must evaluate to integers. For example,

```
&DO A := 1 &TO 10
&DO B := 3 &TO %TOTAL%
&DO C := 5 &TO [LENGTH %A%]
```

If you specify a &BY clause, *increment* must evaluate to an integer. If you do not specify a &BY clause, *increment* defaults to 1. Negative increments, start-values or stop-values may be used; for example, `&DO I := 10 &TO -10 &BY -1`.

Execution of Counted Loops

When a counted loop executes, CPL sets *index-var* to *start-value*. It then tests *start-value* to see if it is less than or equal to *stop-value*. If it is, CPL executes the loop. When control returns to the top of the loop, the value of *index-var* is incremented by *increment*, and retested. When the value of *index-var* exceeds *stop-value*, the loop operation ends; execution passes to the statement that follows the loop's concluding &END statement.

The flow chart for the counted &DO loop is shown in Figure 9-4. As it shows, counted &DO loops are **zero-trip loops**; that is, if the initial value of *index-var* is out of range, the loop is not executed.

The following example demonstrates a counted loop:

```
&DO I := 1 &TO 3
    F77 MODULE%I% -64V -XREF
&END
```

This loop executes three times, compiling the programs MODULE1, MODULE2, and MODULE3.

Omitted &BY and &TO Clauses

The &BY clause specifies the value that CPL uses to increment *index-var* each time through the loop. The &TO clause value specifies the limit value for *index-var*. If you omit the &BY clause in a counted loop, incrementing defaults to &BY 1. If you omit the &TO clause, *index-var* has no stop-value, and thus can increment an infinite number of times. (For example, the directive &DO I := 1 &BY 1 produces this type of infinite loop.) Do not omit the &TO clause in a counted loop without providing some other test for loop termination.

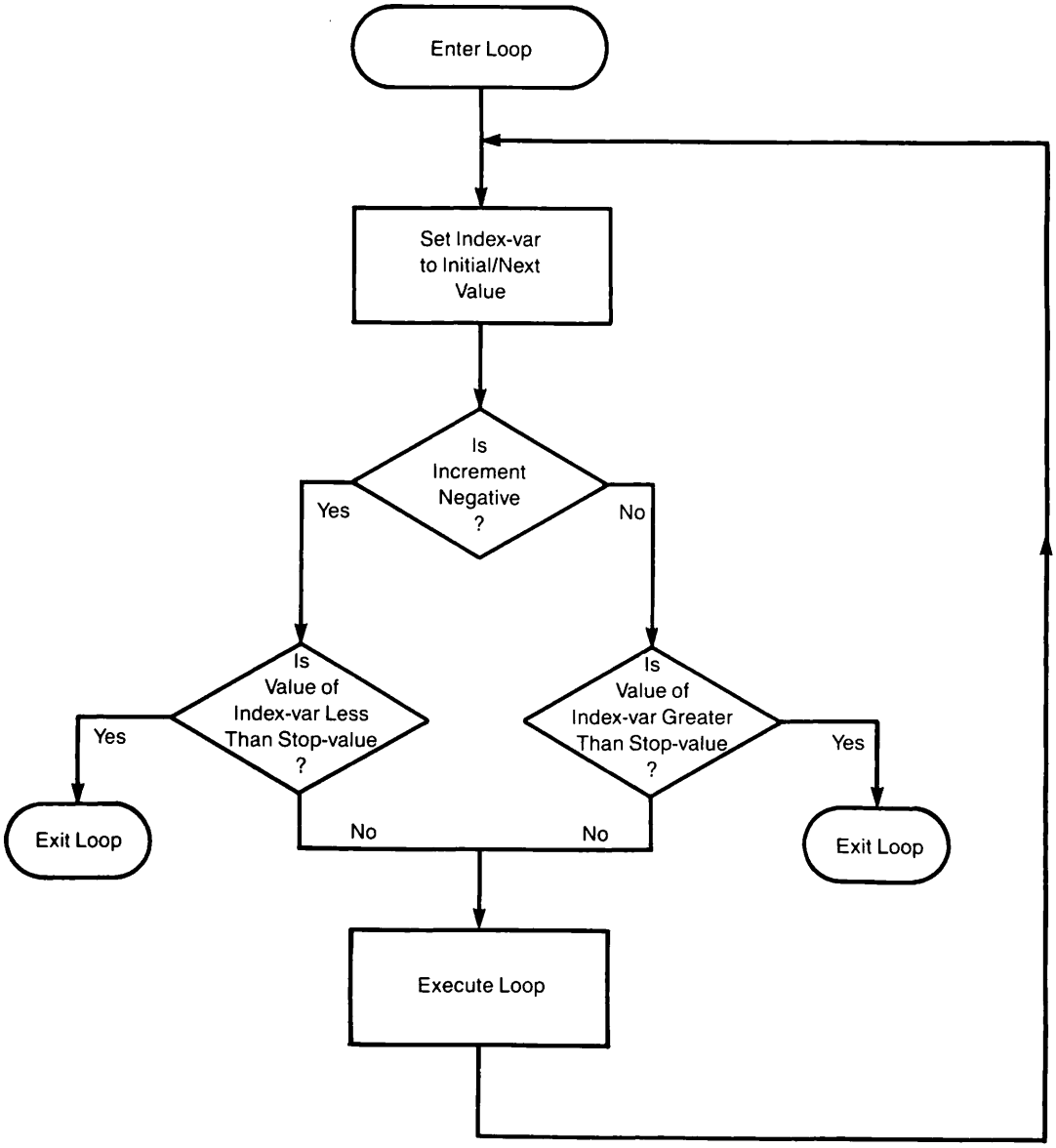


Figure 9-4
Action of Counted &DO Loop

You can use the following to test for loop termination:

- A &WHILE clause
- An &UNTIL clause
- A &RETURN directive inside the loop
- A &GOTO from some point inside the loop

A counted &DO loop with neither a &TO nor a &BY clause executes once and once only. The statement

```
&DO I := 5
```

initiates such a loop. More efficient code to do the same thing is

```
&DO
&SET_VAR I := 5
   .
   .
   .
&END
```

&DO &WHILE Loops

The format of the &DO &WHILE directive is

&DO &WHILE test

test can be any expression that evaluates to TRUE or FALSE. A TRUE result allows the loop to execute. A FALSE result prevents execution of the loop.

Some examples of &DO &WHILE statements are

```
&DO &WHILE [LENGTH %STRING%] > 0
&DO &WHILE %B% > 5 & ^ [NULL %A%]
```

&DO &WHILE loops, like counted loops, are **zero-trip loops**; that is, they are tested for completion at the top of the loop, and do not execute at all if the first test shows that the loop has completed.

Since &DO &WHILE loops are tested at the top of the loop, they require that *test* have some value assigned to it when you execute the &DO statement. In the examples above, you must assign values to %A%, %B%, and %STRING% before the &DO statement is executed.

The following example of a &DO &WHILE loop edits a file that contains a list of names, adding new names to the end of the file. The loop executes as long as you type a name after each prompt. It ends when you type in a carriage return, and thus set LINE to the null string.

```
&DATA ED NAME_LIST
BOTTOM    /* go to bottom of file
          /* get first name
&S LINE := [RESPONSE 'Please enter name to be added']
&DO &WHILE ^ [NULL %LINE%]
  INSERT %LINE%    /*  insert new line in file
          /* get next name
  &S LINE := [RESPONSE 'Please enter name to be added']
  &END    /* end loop
FILE      /*  file amended list of names
&END      /*  end &data group
```

&DO &UNTIL Loops

The format of the &DO &UNTIL loop is

&DO &UNTIL test

For example,

```
&DO &UNTIL %A% > 50
&DO &UNTIL [LENGTH %STRING%] = 0
```

test is any expression that evaluates to TRUE or FALSE. The loop executes as long as *test* remains FALSE.

&DO &UNTIL loops test at the bottom of the loop. Hence, they are **one-trip** loops; they always execute at least one time. The following example uses a simple &DO &UNTIL loop:

```
&ARGS STRING
&DO &UNTIL [NULL %STRING%]
   /*Isolate first letter in string
   &SET_VAR LETTER := [SUBSTR %STRING% 1 1]
   TYPE %LETTER%
   /*  Remove letter from string
   &SET_VAR STRING := [SUBSTR %STRING% 2]
   &END    /* End loop
&RETURN
```

This loop goes through a string letter by letter, removing and displaying one character on each pass. When the last character has been removed, the string becomes a null string, and the loop is complete. (For more information on the SUBSTR function, see Chapter 12.)

Loops That Combine Counting, &WHILE, and &UNTIL Tests

A &DO loop statement can contain more than one test. A counted loop can include a &WHILE test, an &UNTIL test, or both. A &DO &WHILE loop can include an &UNTIL test. Three examples of these combination loops are

```
&DO DAY := 1 &TO 31 &UNTIL [NULL %RECORDS%]

&DO I := 50 &TO 1 &BY -5 &WHILE %J% > 3

&DO &WHILE %A% > 100 &UNTIL %B% > 50
```

Each of these loops executes until one of its tests signals completion. See Figure 9-1 for the test points they can contain.

&DO &REPEAT Loops

The format of the &DO &REPEAT loop is

&DO index-var := start-value
&REPEAT expression {&WHILE test} {&UNTIL test}

A &DO &REPEAT loop is a counted loop. The loop counter (*index-var*) is incremented by the &REPEAT *expression*. The &REPEAT clause permits you to increment the loop counter based upon an expression calculated each time through the loop.

The *index-var* counter variable can be any valid variable name. *start-value*, which initializes the counter, may be any string or arithmetic expression. The &REPEAT *expression* can be any string or arithmetic expression that indicates how the value *index-var* is to be modified on each pass through the loop. For example,

```
&DO I := 5 &REPEAT %I% * 5 &UNTIL %I% > 500
```

This example sets I to 5 on the first trip through the loop, then multiplies I by 5 on succeeding trips. This loop executes four times, with I set to 5, 25, 125, and 625. At the bottom of the fourth trip, the test 625 > 500 is true; so the loop terminates at the end of that iteration.

Note

A &REPEAT loop without a &WHILE or &UNTIL clause is an infinite loop; that is, it has no test for termination. If you write a &REPEAT loop without a &WHILE or &UNTIL clause, make sure you include some &RETURN or &GOTO directive inside the loop so that it can terminate.

&DO &LIST Loops

The format of the &DO &LIST loop is

&DO index-var &LIST list-of-items {&WHILE test} {&UNTIL test}

A &DO &LIST loop is a counted loop that executes as many times as the number of items in the &LIST *list-of-items*. Each time the loop executes, it sets *index-var* to the next item on the *list-of-items*. The loop executes until the list of items is exhausted.

index-var can be any valid variable name.

list-of-items is a list of items separated by blanks. Each item in the list can be a character string, a variable reference, or a function call. A variable reference or function call may itself evaluate to a list of items. The maximum length of *list-of-items* is 1024 characters. This maximum applies to both the list as written, and the evaluated value of the list.

A &DO &LIST loop is a zero-trip loop; that is, if the *list-of-items* contains no items, the loop is not executed. A null character ('') or a variable reference with a null value does count as an item in the *list-of-items* and causes the loop to execute. When the loop is executed, each iteration of the loop sets *index-var* to the next item on the list. When the list is exhausted, the loop terminates. For example,

```
&DO I &LIST alpha beta gamma
```

This statement executes a loop three times, with I equal to *alpha* on the first iteration, *beta* on the second iteration, and *gamma* on the third iteration.

The action of the &DO LIST loop is diagrammed in Figure 9-5.

Examples of the &DO &LIST Loop

The first example is a basic &DO &LIST loop:

```
&DO I &LIST 50 0 -50
```

This loop executes three times, with I set to 50 on the first iteration, 0 on the second, and -50 on the third.

The second example uses a single variable to specify multiple listed items:

```
&DO WORD &LIST [UNQUOTE %line_of_type%]
```

This statement evaluates the variable *line_of_type*, and assigns each blank-separated word or number found in that line to the index variable, WORD. For instance, if the value of *line_of_type* is 'THE QUICK BROWN FOX', then the loop executes four times, with WORD set to THE, QUICK, BROWN, and FOX. Because a quoted string is a single item, the value of *line_of_type* must be unquoted to be counted as four listed items. If *line_of_type* remains quoted, 'THE QUICK BROWN FOX', causes the loop to execute once, with *WORD* set to 'THE QUICK BROWN FOX'.

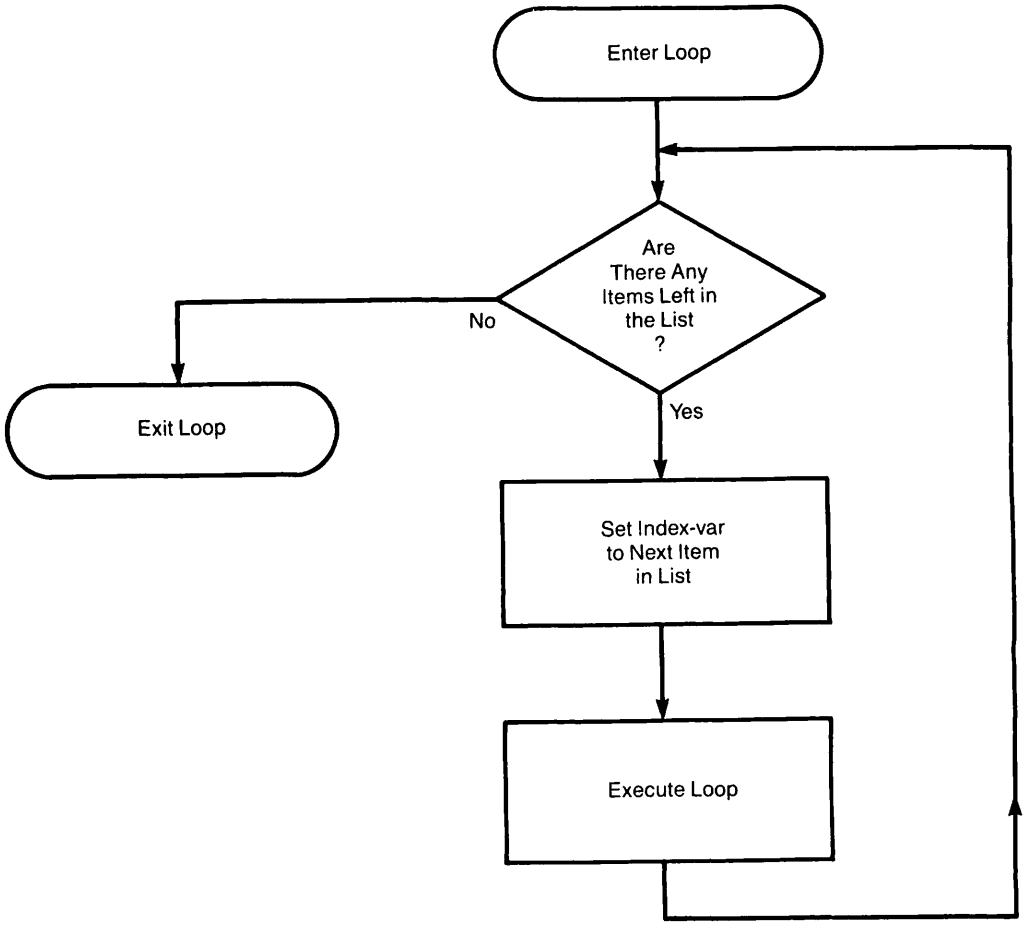


Figure 9-5
Action of &DO &LIST Loop (&WHILE and/or &UNTIL tests may be added)

The third example uses the &DO &LIST loop with the WILD function:

```
&DO I &LIST [WILD @.CBL]
   CBL %I%
 &END
```

This loop compiles all COBOL 74 files in the current directory. However, if the WILD function returns a list longer than 1024 characters, an error occurs that halts the CPL program. If you think this may happen in your program, use the &ITEMS loop, described below, instead of the &LIST loop.

The final example uses nested &DO &LIST loops:

```
&DO DEPT &LIST [WILD @@ -DIRS]                  /* Begin dept-loop
    A SALES>%DEPT%
    &DO REPORT &LIST [WILD @_REPORT -FILES]   /* Begin report-loop
        SPOOL %REPORT%
        &END                                   /* End report-loop
    &END                                       /* End dept-loop
A SALES                                  /* Re-attach to SALES directory
```

This example spools every report in every subdirectory belonging to the directory SALES. The outer loop attaches to each subdirectory in turn. The inner loop finds the files in that subdirectory that end in _REPORT, and spools them.

&DO &ITEMS Loops

The &DO &ITEMS loop is similar to the &DO &LIST loop in that it processes a sequence of items, and terminates when it has exhausted the items. It differs from &DO &LIST in that it does not have a list of items to read. Instead, the word &ITEMS is followed by an *expression* that is evaluated at each iteration. Usually, *expression* is the WILD function with the –SINGLE option, returning one filename per iteration.

The format of the &DO &ITEMS loop is

&DO index-var &ITEMS expression {&WHILE test} {UNTIL test}

It is equivalent to "&DO I := expression &REPEAT expression &WHILE ^ [NULL %I%]". The action of the &DO &ITEMS loop is shown in Figure 9-6.

The following example demonstrates the &DO &ITEMS loop:

```
&S UNIT := 0            /*This step is essential
&DO I &ITEMS [WILD @.CBL @.F77 -SINGLE UNIT]
    &S COMPILE := [AFTER %I% .]
    %COMPILE%  %I%
&END
```

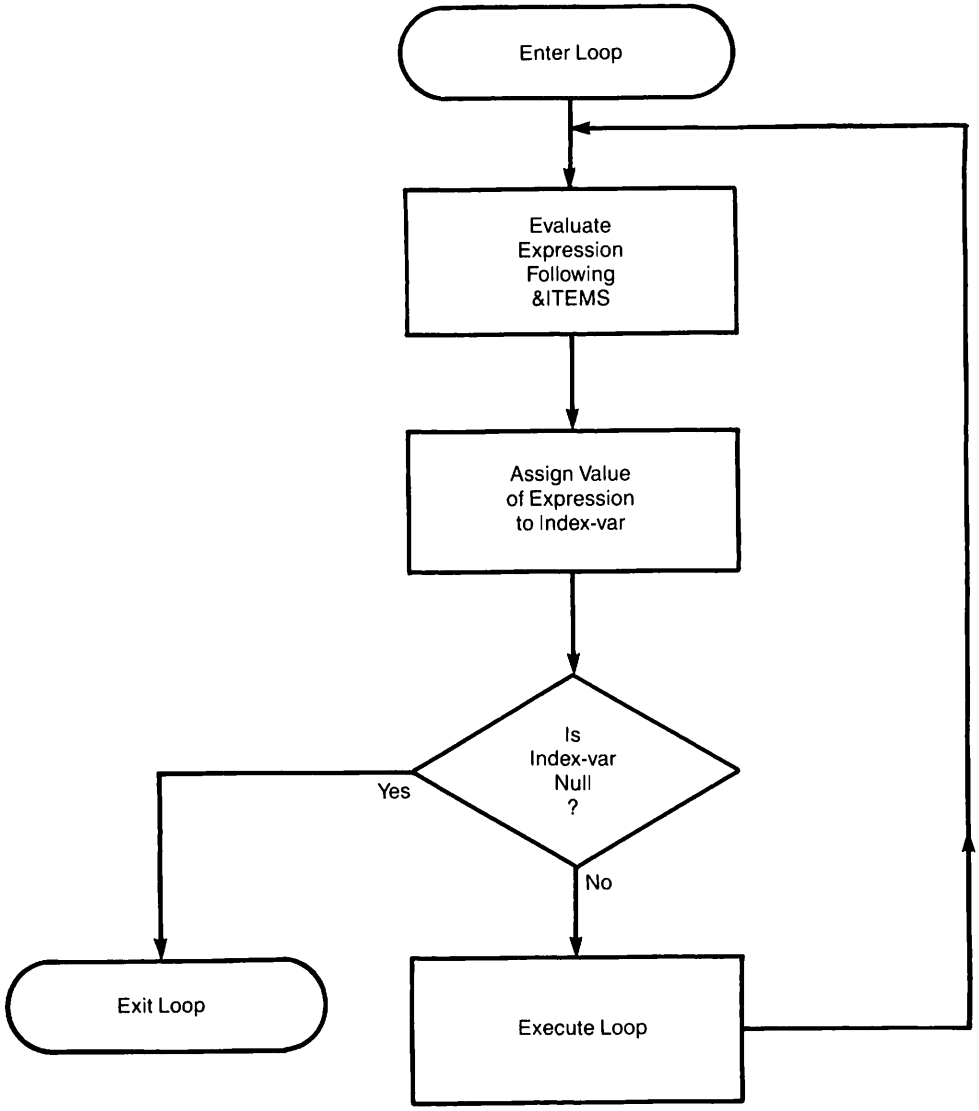


Figure 9-6
Action of &DO &ITEMS Loop (&WHILE and/or &UNTIL tests may be added)

This example compiles all COBOL 74 and FORTRAN 77 files in the user's current directory, no matter how many of them there are. It works as follows:

1. The directive &S UNIT := 0 initializes the variable *unit* with a value of zero. (Any variable name may be used; *unit* is only a handy mnemonic.)
2. The WILD function sees that *unit* is set to zero. It therefore opens the user's current directory on some available unit, then sets *unit* to the number of the unit that it is using. (It uses the decimal number of the file unit.)
3. Because the WILD function includes the -SINGLE option, the WILD function finds the first matching file, and returns that filename as its value.
4. The &DO processor assigns the value returned by the WILD function to *I*.
5. The loop executes.
6. When the loop returns to the &DO statement, the WILD function is re-invoked.
7. The WILD function reads the open unit number from *unit*, goes to that unit, and selects the next matching file.
8. The loop executes again.
9. When the WILD function finds no matching file, it returns a string of length zero and closes the file unit it was using. CPL resets *I* to the null string, and the loop terminates immediately.

This loop is equivalent to the following &REPEAT loop:

```
&S UNIT := 0
&DO I := [WILD @.CBL @.F77 -SINGLE UNIT] ~
    &REPEAT [WILD @.CBL @.F77 -SINGLE UNIT] ~
    &WHILE ^ [NULL %I%]
&S COMPILE := [AFTER %I% .]
%COMPILE% %I%
&END
```

A &DO &ITEMS Loop to Read and Write Files

The &DO &ITEMS loop can also be used with CPL's file I/O functions, as shown in the following example. (For information on these functions, see Chapter 12.)

```
   /* Open file ALPHA for reading and writing
&S UNIT := [OPEN_FILE ALPHA STATUS -MODE R]
   /* Read each line in turn
&DO I &ITEMS [READ_FILE %UNIT% STATUS]
     .
     .
     .
 &END
CLOSE ALPHA
```

This example performs the following steps:

1. Opens the file ALPHA for reading on some available unit, returning the number of the unit (in decimal) as the value of the variable, %UNIT%.
2. Reads one line from the file each time the &DO &ITEMS statement is encountered.
3. Terminates when it reaches the end of the file.

(Note that in this case, the file is not closed automatically. The user must close it after the loop is completed.)

10
Debugging and Error Handling

This chapter describes CPL directives that you can include in your program to deal with errors. CPL programs may encounter two types of errors:

- A CPL directive may be written incorrectly. For example, the word &THEN may have been omitted from an &IF statement.
- A PRIMOS command executed by the CPL program may produce a runtime error. For example, a command may try to open a file that does not exist.

You can locate errors in your CPL program by including the &DEBUG directive at the beginning of the program. Three &DEBUG options are useful for debugging: no-execute mode, echoing, and variable watching. These &DEBUG options are explained in the first half of this chapter.

You can detect and respond to runtime errors by including the &SEVERITY directive in your CPL program. &SEVERITY has several options that detect errors of differing severity, issue messages, halt execution, and invoke user-written error handling routines. The &SEVERITY directive is described in the second half of this chapter. More advanced methods of runtime error handling are described in Chapter 15.

Debugging CPL Programs: The &DEBUG Directive

All debugging operations are enabled and disabled using the &DEBUG directive. Its format is

&DEBUG {options}

Available options are shown in Table 10-1, and are explained below.

If no &DEBUG directive is given, debugging is disabled. (This is equivalent to &DEBUG &OFF.)

If &DEBUG is given without options, the result is equivalent to

```
&DEBUG &NO_EXECUTE &ECHO ALL
```

&DEBUG directives may appear anywhere in a CPL program. A &DEBUG directive takes effect when it is read, superseding any previous &DEBUG directives.

If a CPL program executes another program or calls a subroutine, the first program's debugging options are suspended while the called program or routine executes. The debugging options are re-enabled when execution of the first program resumes.

Table 10-1
&DEBUG Options

Option	*Action*
&OFF	Turns off all debugging options. Initially, all options are off.
&NO_EXECUTE	Suppresses execution of PRIMOS commands, but interprets CPL directives. Abbreviated &NEX.
&EXECUTE	Enables execution of PRIMOS commands. Abbreviated &EX.
&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 var2 ... 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 listed, &WATCH watches all variables.
&NO_WATCH {var1 var2 ... var16}	Removes the specified variables from the watchlist. If no variables are specified, watching is turned off completely.

The &NO_EXECUTE and &EXECUTE Options

This pair of options determines whether or not commands are executed when the CPL program is run. Specifying &DEBUG &NO_EXECUTE (or simply &DEBUG), allows you to run through, or "rehearse", a CPL program. When you run a CPL program that begins with &DEBUG &NO_EXECUTE, the CPL interpreter reads the CPL file and interprets its directives as usual. However, it does not pass any commands to PRIMOS. If a CPL error is found, the usual message is sent and execution is terminated.

The &NO_EXECUTE option thus lets you run through a program as many times as necessary to get rid of syntax errors before performing any of the commands the file contains. It is especially useful for CPL programs that

- Take a long time to execute
- Edit or update sensitive files
- Use peripheral equipment, such as magnetic tapes
- Contain any sequence of commands that should not be interrupted

&DEBUG &EXECUTE allows the execution of PRIMOS commands. You can place pairs of &DEBUG &EXECUTE and &DEBUG &NO_EXECUTE directives in your CPL program to prevent the execution of blocks of PRIMOS commands.

A &DEBUG directive with no options prevents the execution of PRIMOS commands. A &DEBUG directive with any options (for example, &DEBUG &ECHO) permits the execution of PRIMOS commands unless you specify &NO_EXECUTE.

The &ECHO and &NO_ECHO Options

The &ECHO and &NO_ECHO options control the echoing of commands and directives. Echoing displays each line of the CPL program on the terminal screen as it is encountered in program execution. You can echo all CPL program lines, CPL directives only, or PRIMOS commands only. If you specify &DEBUG with no options, it echos all CPL lines; if the &DEBUG directive is followed by options, you must specify &ECHO to echo the lines of the CPL program.

If &ECHO DIR is given, CPL directives are echoed on the terminal as they are read. (A loop directive echoes each time the loop is executed.) For example, this CPL program (named EX.CPL)

```
&DEBUG &ECHO DIR
&DO I := 1 &TO 3
   TYPE %I%
&END
```

produces this terminal session when run:

```
OK, R EX.CPL
&DO I := 1 &TO 3
1
&END
&DO I := 1 &TO 3
2
&END
&DO I := 1 &TO 3
3
&END
OK,
```

If &ECHO COM is given, PRIMOS commands are echoed. If this sample program contains the statement &DEBUG &ECHO COM, the terminal session looks like this:

```
OK, R EX.CPL
   TYPE 1
1
   TYPE 2
2
   TYPE 3
3
OK,
```

The &ECHO DIR and &ECHO COM options are additive. That is, if you specify &ECHO DIR, and then specify &ECHO COM later in the program, both directives and commands are echoed for the rest of the program.

If &ECHO ALL (or simply &ECHO) is given, both commands and directives are echoed. If this sample program contains the statement DEBUG &ECHO, the terminal session looks like this:

```
OK, R EX.CPL
&DO I := 1 &TO 3
   TYPE 1
1
&END
&DO I := 1 &TO 3
   TYPE 2
2
&END
&DO I := 1 &TO 3
   TYPE 3
3
&END
OK,
```

&NO_ECHO turns off echoing. You can specify &NO_ECHO ALL, &NO_ECHO COM or &NO_ECHO DIR. If a program begins with the directive &DEBUG &ECHO ALL, and later contains the directive &DEBUG &NO_ECHO COM, then echoing of commands is halted, but echoing of directives continues. &NO_ECHO is equivalent to &NO_ECHO ALL.

The &WATCH and &NO_WATCH Options

The &WATCH option lets you trace the values of local and/or global variables. &DEBUG &WATCH displays the value of a variable each time the variable is set by a &SET_VAR directive. (This includes changes made by the CPL interpreter itself, such as those that occur by setting the index variable of a loop or recording a new severity value. They do not include values set by the SET_VAR command or the GV$SET routine.)

&WATCH can be used in two ways. If you specify &DEBUG &WATCH, the values of all variables are displayed. If you specify &DEBUG &WATCH followed by a list of variables, only the values of the listed variables are displayed. You can list as many as 16 variables. Enclose the list of variables in parentheses; you must separate variables from the parentheses and from each other by blank spaces.

For example, the program

```
&DEBUG &WATCH ( I J )
&DO I := 1 &TO 5
   &S J := %I% * %I%
&END
```

produces the following result:

```
OK, R EX2.CPL
Variable "I" set to "1" at line 2.
Variable "J" set to "1" at line 3.
Variable "I" set to "2" at line 4.
Variable "J" set to "4" at line 3.
Variable "I" set to "3" at line 4.
Variable "J" set to "9" at line 3.
Variable "I" set to "4" at line 4.
Variable "J" set to "16" at line 3.
Variable "I" set to "5" at line 4.
Variable "J" set to "25" at line 3.
Variable "I" set to "6" at line 4.
OK,
```

Note that this display shows the loop's index as set to its first value at the top of the loop, then incremented at the &END statement each time thereafter.

&WATCH variables are additive. That is, if you specify &WATCH (A B), and then specify &WATCH (C D) later in the program, the variables A, B, C, and D are displayed for the rest of the program.

To cease displaying a variable, use the &NO_WATCH option. &DEBUG &NO_WATCH by itself halts the display of all variables. &DEBUG &NO_WATCH with a variable list halts the display of the listed variables only.

Error Handling: The &SEVERITY Directive

Whenever a PRIMOS command is executed, it produces an error code (known as a **severity code**). Possible severity codes are

Code	*Meaning*
0	No error
Positive integer	Error
Negative integer	Warning

CPL's default response to these severity codes is to ignore codes of 0 or less, but to halt execution of the CPL program if a severity code of 1 or greater is received.

The &SEVERITY directive allows CPL to perform error checking automatically after the execution of each command. Therefore, if you wish to alter CPL's default error handling during part or all of any CPL program, you may use a &SEVERITY directive to specify the action you want taken. Possible &SEVERITY directives are

Directive	*Meaning*
&SEVERITY	Ignore warnings, halt execution for errors (default).
&SEVERITY &WARNING &IGNORE	Ignore warnings, halt execution for errors (default).
&SEVERITY &ERROR &IGNORE	Ignore errors and warnings, continue execution.
&SEVERITY &WARNING &FAIL	Halt execution if any warning or error is received.
&SEVERITY &ERROR &FAIL	Ignore warnings, halt execution for errors (default).
&SEVERITY &ERROR &ROUTINE routine label	Invoke the specified routine if an error occurs. Ignore warnings.
&SEVERITY &WARNING &ROUTINE routine label	Invoke the specified routine if any warning or error is received.

&SEVERITY directives may be placed anywhere in a CPL program. They become effective when execution of the program reaches the line in which they occur, and they remain effective until the program either

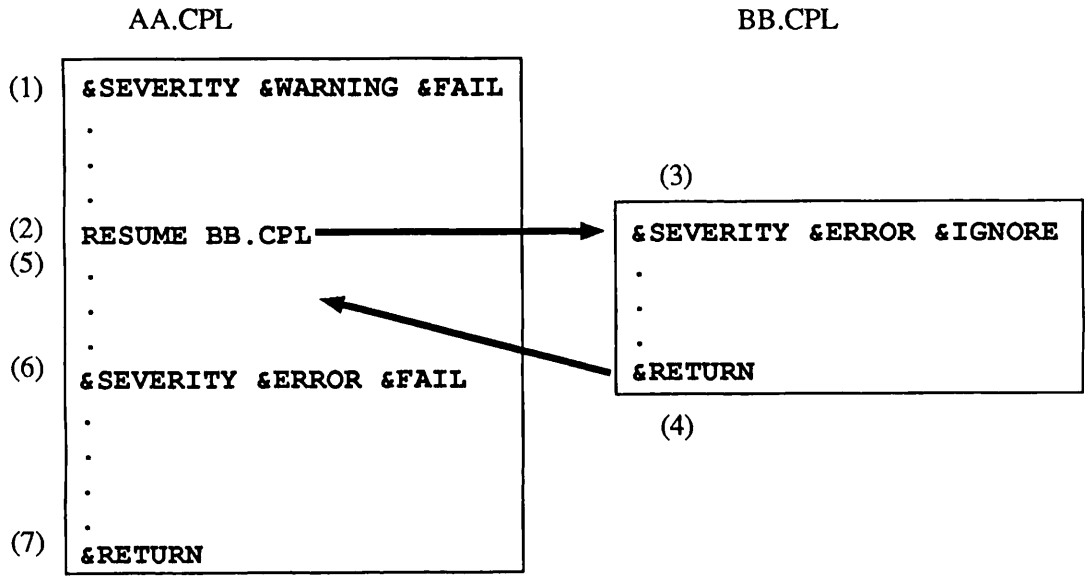
- Terminates
- Encounters a new &SEVERITY directive

If one CPL program invokes another (or if it invokes one of its routines), then the effectiveness of the &SEVERITY directive is suspended while the second program (or routine) executes. If the

invoked program or routine defines its own error handling, that takes effect. If the program defines no error handling, CPL's default error handling takes effect from the time the new program or routine is invoked until it returns to its caller (that is, to the first CPL program).

A possible sequence of error handling is shown in Figure 10-1. Chapter 15 contains a further explanation of CPL's error handling, including

- How to define your own error conditions
- How to write error-handling routines
- How to define your own condition handling
- How to make a &RETURN directive pass a severity code to its caller
- How to use the &STOP directive to halt a routine and its calling program simultaneously



Action	*Error Handling*
1. AA.CPL sets error handling to &SEVERITY &WARNING &FAIL.	1. Program will halt if it gets a warning message.
2. AA.CPL invokes BB.CPL.	2. AA.CPL's error handling is suspended.
3. BB.CPL sets error handling to &SEVERITY &ERROR &IGNORE.	3. No error codes can halt BB.CPL's execution. (PRIMOS condition codes, such as pointer faults or access violations, can still halt BB.CPL's execution.)
4. BB.CPL returns to AA.CPL.	4. BB.CPL's error handling is terminated.
5. Execution of AA.CPL continues.	5. Error handling is &SEVERITY &WARNING &FAIL again, as AA.CPL originally set it.
6. Execution of AA.CPL encounters &SEVERITY &ERROR &FAIL directive.	6. AA.CPL's error handling changes to system default error handling (that is, halt for errors, ignore warnings).
7. AA.CPL returns.	7. AA.CPL's error handling is terminated. (Error handling is determined by AA.CPL's caller.)

Figure 10-1
Scope of &SEVERITY Directive

Part III
Full CPL

11
Expression Evaluation

This chapter provides a detailed explanation of how CPL evaluates expressions. It describes

- How variables are defined and evaluated
- How function calls are used and evaluated
- How quoted strings are handled in variables, in function calls, and in CPL generally
- How to suppress and reactivate expression evaluation using the QUOTE, UNQUOTE, and RESCAN functions
- How arithmetic expressions are evaluated, and how to use the CALC function to force evaluation of arithmetic expressions
- How PRIMOS features such as ABBREV files and PRIMOS special characters are evaluated, and how these features interact with evaluation of CPL expressions

Variables

A CPL variable name can be as many as 32 characters in length. It can contain letters, digits, underscore characters (_), and dots (.). Names of local variables must start with a letter (to avoid confusion with numbers). Names of global variables must start with a dot (.). The dollar sign ($) is reserved for the names of predefined PRIMOS variables. Variables always take character strings as values; the maximum length of a value is 1024 characters.

Variables are not declared in CPL. They are defined by assigning them a value for the first time. There are two kinds of variables, **local** and **global**. Both may be assigned values using the &ARGS directive, the &SET_VAR directive, or the SET_VAR command. For example,

```
&SET_VAR PL1_PROG := RICHS>EVAL.PL1
```

sets the local variable *PL1_PROG* to the value RICHS>EVAL.PL1.

You can use the SET_VAR command to define global variables at command level. Use the &SET_VAR directive, which is both faster and more flexible, for defining variables within a CPL program.

Local Variables

Local variables cannot be set outside a CPL invocation. You can only define local variables in the CPL program in which they are used; you cannot define a local variable in a recursive invocation of the same CPL program, nor in an invocation of another CPL program. You can, however, pass values to local variables at runtime using the &ARGS directive. You can also use the &ARGS directive to pass local variable values as arguments to programs called from your CPL program.

CPL suspends local variables during a call to another program (using, for example, the RESUME command). Local variables are reactivated when the CPL program returns from the called program. Local variables remain active during a call to a CPL routine (using the &CALL directive). CPL deletes all local variables when the program in which they are defined finishes.

Global Variables

Global variables are distinguished from local variables by having names that start with a period. The CPL directive

```
&SET_VAR .HOME := RICHS
```

sets the global variable *.HOME* to the value RICHS. Global variables are associated with a particular user, and not with any program; they can be referenced in any CPL procedure invoked by that user. The names and values of global variables survive the invocation of a program in which they are used. Thus, if you run a CPL program that sets the global variable,

```
.A_GLOB_VAR,
```

you can later run another CPL program that references that variable. Furthermore, the names and values of global variables survive logout. When you log in, any global variables that you defined in previous sessions are still available.

Global variables survive program invocations and logouts because they are saved in a user-defined file. This file is defined by the DEFINE_GVAR internal command (see Chapter 4). If you intend to use global variables during a terminal session or in a CPL program, you must issue a DEFINE_GVAR command before the first global variable reference.

You can set global variable values using the SET_VAR command, the &SET_VAR directive, and the &ARGS directive. You can delete a global variable or a list of global variables using the PRIMOS command DELETE_VAR. The PRIMOS command LIST_VAR lists global variables and their values at your terminal. You can list all global variables, or list specified global variables. These commands and directives are further described in Chapter 4.

Evaluation of Variables

A variable is referenced by enclosing its name in percent signs, as in %variable_name%. An example of a statement referencing variables is

```
F77 %PATHNAME%.F77 -LIST %PATHNAME%.LIST -BIN %PATHNAME%.BIN
```

The string %PATHNAME% is replaced with the value of the variable *pathname*. For example, if *pathname* has the value HOBBIT, then the above statement is transformed into

```
F77 HOBBIT.F77 -LIST HOBBIT.LIST -BIN HOBBIT.BIN
```

When a statement contains variable references, all references are replaced by their values before the statement is executed. Variable evaluation is performed only once per statement. For example, if variable *VAR* has the value %XXX%, then when *VAR* is evaluated, CPL replaces the reference %VAR% with the value %XXX%. The value %XXX% may itself be a variable reference, but because a CPL statement is only evaluated once, this string remains as a literal in the program and is not evaluated as a variable reference.

Variable references inside single quotation marks are not evaluated.

Functions

Functions are procedures that return string values. These string values are substituted for the function call in the original statement. The maximum length of a function result is 1024 characters. A function call is indicated by square brackets:

[function-name arg1 ... argn]

where *function-name* is the name of the function, and *arg1* through *argn* are its arguments. An example of a function call is

```
PL1 %PL1_PROG% -L [BEFORE %PL1_PROG% .PL1].LIST
```

The function BEFORE returns that part of the value of %PL1_PROG% that occurs before the first occurrence of .PL1. In this case, the value returned by the BEFORE function is concatenated with the adjacent string .LIST. If %PL1_PROG% has the value RICHS>EVAL.PL1, then the statement is transformed into

```
PL1 RICHS>EVAL.PL1 -L RICHS>EVAL.LIST
```

before it is executed.

CPL evaluates variable references prior to function calls. This is illustrated by the example just given; CPL replaces the variable reference %PL1_PROG% with RICHS>EVAL.PL1 before it evaluates the BEFORE function call.

Function evaluation is done recursively; any or all of *function-name* or *arg1...argn* may themselves contain function calls. Innermost calls are done first. There is no implementation restriction on the depth of nesting.

Function calls inside single quotation marks are not evaluated.

Quoted Strings

To quote a string, enclose it in single quotation marks ('). You must quote a string if

- The string contains at least one blank, semicolon or comma, but is supposed to represent one token. For example,

  'a multiple token'

- The string contains a literal quotation mark, as in 'QUOTE-IT"S-INSIDE'. Note that to include a quotation mark in a string you must double it.
- The string contains at least one of the special characters described in Chapter 3, Rule 9, and the literal meaning of the character is desired. Since these characters have meanings in CPL syntax, these meanings must be suppressed by quoting the string. For example, 'HIDE_THIS_[FUNCTION CALL]'.

When you set a variable with a quoted string, for example

```
SET_VAR A := 'quotes_go_in'
```

the quotation marks become part of the value of that variable.

Enclosing a string in quotation marks establishes the string as an indivisible unit and prevents the evaluation of the contents of the string. Otherwise, the quotation marks that enclose the string are frequently not apparent. For example, the PRIMOS command TYPE does not display enclosing quotation marks. The LENGTH function does not count enclosing quotation marks when determining the length of a string.

Expression evaluation considers a quoted string to be a literal character string, and not an integer or a logical value. Therefore, do not use quoted strings in arithmetic expressions, logical expressions, or relational expressions other than string comparisons. (Chapter 4 provides examples of the use of quoted strings in expressions.)

A quoted string remains quoted until specifically unquoted using the UNQUOTE or RESCAN function.

Concatenating Strings

If two or more variable references or function calls are placed side by side, their values are concatenated. Thus, suppose *x* has the value 'ab' and *y* has the value 'cd'. Then

Typing	*Produces*
`%x%%y%`	'abcd'
`%x% %y%`	'ab' 'cd'

In the first example, the values of *x* and *y* are concatenated into a single string by removing their respective right and left quotation marks. In the second example, the intervening blank causes the references to be replaced without concatenation. Similar rules hold for function calls.

Variable Arrays

CPL's ability to concatenate variables permits you to create elements of an array. To create elements of an array, you concatenate the array name with the subscript index. For example, if the array is named TEST_SCORES, the sixth element of the array is named TEST_SCORES6.

The following example shows how to create an array named TEST_SCORES with 100 elements, and initialize the element values to zero:

```
&DO INDEX := 1 &TO 100 &BY 1
&SET_VAR TEST_SCORES%INDEX% := 0
&END
```

To retrieve the value of an array element, use the GET_VAR function:

```
&ARGS QUERY
TYPE Test score number %QUERY% is [GET_VAR TEST_SCORES%QUERY%]
```

Quoting and Unquoting Strings

CPL provides a function that unquotes strings. The UNQUOTE function has the format

[UNQUOTE string]

For example,

```
[UNQUOTE 'ab']
[UNQUOTE %x%]
```

The UNQUOTE function removes the outermost pair of quotation marks (if any) from *string*, then changes every remaining pair of adjacent quotation marks to a single quotation mark. UNQUOTE returns this unquoted version of *string*. Note that UNQUOTE only removes one layer of quotation marks; therefore, a string returned by UNQUOTE may, in some cases, still be a quoted string. The *string* and the value returned by UNQUOTE are shown in the following examples:

%x%	*[UNQUOTE %x%]*
ab	ab
'ab'	ab
'''ab'''	'ab'
'a"b'''	a'b'
'''a""b'''	'a"b'

CPL also provides a function that quotes strings. The QUOTE function has the format

[QUOTE string-1 {string-2} ...]

For example,

```
[QUOTE abc]
[QUOTE %a% %b% def]
```

The QUOTE function adds an outer pair of quotation marks to *string*. If there are multiple *strings*, QUOTE treats *string-1*, *string-2*, and so on, as a single quoted string. QUOTE also automatically doubles single quotation marks within *string*. CPL reads a doubled quotation mark as one literal quotation mark character, rather than as a quotation mark used to enclose a quoted string.

For example, suppose *x* has the value AB'C'D, then

```
[QUOTE %x%]
```

returns

'AB''C''D'

You can use the QUOTE function to quote an already quoted string. This works for any number of quote levels. For example, if *x* has the value AB'C'D,

```
[QUOTE [QUOTE %x%] ]
```

the outermost QUOTE function returns the string

'''AB''''C''''D'''

The RESCAN Function

The RESCAN function performs an unquote operation in which a string is evaluated, unquoted, and then evaluated again. The format of the RESCAN function is

[RESCAN string]

You can use the RESCAN function to force evaluation of *string* when *string* contains quoted variable references and function calls. Usually, RESCAN is used when *string* contains both variable references (such as %x%) and quoted variable references (such as '%x%').

RESCAN evaluates (scans) *string* twice. First, it evaluates any function calls or variable references that are not in quotation marks, and substitutes the resulting values into *string*. It then unquotes *string*. This step is identical to the operation performed by the UNQUOTE function. Finally, RESCAN evaluates any function calls or variable references in the string that are not still quoted.

To illustrate the use of this function, suppose a CPL program MYPROG.CPL has an argument *funcs_and_vars*. The value of *funcs_and_vars* is a string containing variable references and function calls. For example, the value of *funcs_and_vars* might be [LENGTH %holycow%]. If we try invoking MYPROG.CPL by

```
R MYPROG.CPL  [LENGTH %holycow%]
```

the CPL interpreter evaluates the [LENGTH %holycow%] argument value before passing it to the &ARGS directive in the program. This is not what was intended. Instead, we must type

```
R MYPROG.CPL  '[LENGTH %holycow%]'
```

The quotation marks suppress evaluation so that the string '[LENGTH %holycow%]' is assigned to *funcs_and_vars*.

Later, however, when MYPROG.CPL wants to evaluate the LENGTH function call in *funcs_and_vars*, using just %funcs_and_vars% would give the value with its quotation marks, again suppressing evaluation of the function. The RESCAN function must be used to strip the quotation marks from '[LENGTH %holycow%]', and then evaluate the %holycow% variable reference and the LENGTH function call. Thus, MYPROG.CPL might contain the statement:

```
&IF [RESCAN %funcs_and_vars%] > 100 &THEN &RETURN
```

In this example, CPL evaluates %funcs_and_vars% and returns '[LENGTH %holycow%]'. RESCAN unquotes '[LENGTH %holycow%]' and then evaluates [LENGTH %holycow%] to return an integer value for use by the &IF directive expression.

Using Abbreviations

PRIMOS provides an abbreviations facility that you can use to create your own short abbreviations for long sequences of PRIMOS commands. These abbreviations and their expansions are stored in an ABBREV file. The *Prime User's Guide* describes how to create abbreviations. This section describes how to use your abbreviations within a CPL program.

The &EXPAND Directive

The &EXPAND directive enables or disables the expansion of abbreviations from your ABBREV file within a CPL program. Its format is

&EXPAND $\left\{\begin{matrix}\text{ON}\\\text{OFF}\end{matrix}\right\}$

&EXPAND ON causes the CPL interpreter to pass each command in the CPL program to the abbreviation preprocessor for abbreviation expansion. After specifying &EXPAND ON, your CPL program can invoke the abbreviations in your ABBREV file as if they were PRIMOS commands.

The CPL interpreter passes each command to the abbreviation preprocessor for expansion (if necessary) *before* it performs variable evaluation, function evaluation, and execution. Therefore,

&EXPAND does not expand user-defined abbreviations that include variables or functions. For example, the following CPL program executes properly:

```
&ARGS SUBDIR
&S LISTEM := LD
&EXPAND ON
DOWN %SUBDIR%
%LISTEM%
```

The above example executes the user abbreviation DOWN and the PRIMOS command LD. The following example, however, *does not* execute properly:

```
&ARGS SUBDIR
&S LISTEM := LD
&S DESCEND := DOWN
&EXPAND ON
%DESCEND% %MYSUB%
%LISTEM%
```

In this example, the user abbreviation DOWN cannot be located because the CPL interpreter checks the abbreviation preprocessor before evaluating the %DESCEND% variable reference.

With the exception of &THEN and &ELSE clauses, the CPL interpreter cannot execute user abbreviations found within CPL directives (such as &DATA groups). This is because CPL directives are not passed to the abbreviation preprocessor. Use the command function [ABBREV –EXPAND text] to overcome these limitations.

In order for expansion to work, the PRIMOS command

ABBREV pathname -ON

must be given either at command level or within the CPL program before any abbreviations are used. This command activates the abbreviation file. *pathname* is the pathname of the user's abbreviation file. In many cases, this command is issued as part of a user's login procedure.

An &EXPAND directive takes effect when it is encountered. It is effective only for the procedure that invokes it; it does not carry over into programs or routines invoked by that procedure.

&EXPAND OFF disables abbreviation expansion. This is the default setting.

Evaluation of Expressions

A PRIMOS command that you issue at the terminal is evaluated before it is executed. Similarly, the CPL interpreter evaluates expressions in a CPL program before executing CPL directives and commands.

Evaluation at PRIMOS Command Level

A PRIMOS command that you issue at command level can contain function calls and references to global variables. When variables and functions are used at command level, the command processor evaluates these variable references and function calls.

Variable references are evaluated first. For example, suppose the variable *.SRC* has the value MYDIR and the variable *.FILE* has the value MYPROG.F77. If you type the following line at your terminal

```
F77 %.SRC%>%.FILE% -L [BEFORE %.FILE% F77].LIST -B NO
```

the command processor first evaluates the variable references, creating the following line:

```
F77 MYDIR>MYPROG.F77 -L [BEFORE MYPROG.F77 F77].LIST -B NO
```

Functions calls are evaluated second. Thus, the command processor converts the above line to

```
F77 MYDIR>MYPROG.F77 -L MYPROG.LIST -B NO
```

and then executes the command. This completes variable and function evaluation at command level.

If abbreviation processing has been enabled by the &EXPAND ON directive, the command line is passed to the abbreviation preprocessor for evaluation before variables and functions are evaluated.

Evaluation Within CPL: Arithmetic Expressions

When variables and functions are used inside a CPL program, the CPL interpreter evaluates the variable references and function calls. Evaluation is done by the CPL interpreter (rather than by the command processor) because these expressions must be evaluated in both CPL directives and PRIMOS commands.

In CPL Directives: As in interactive evaluation, CPL first evaluates variable references, then evaluates function calls. However, evaluating CPL directives requires a third step, an implicit call on the CALC function. (Chapter 12 describes CALC; briefly, this function calculates the values of arithmetic and Boolean expressions.) CPL calls CALC for any expression in a CPL directive. If the CALC function can evaluate the expression, it returns the resulting integer or Boolean value. If the CALC function cannot evaluate the expression, it returns the original string. Expressions evaluated by CALC must not be quoted and all operators must be delimited by blanks. Thus, instead of typing

```
&IF [CALC %I% > 5] &THEN &RETURN
```

you can type

```
&IF %I% > 5 &THEN &RETURN
```

The implicit call to CALC is done *last*, after variables and functions have been evaluated. Therefore, to evaluate an expression within a function call, you must issue an explicit call to CALC. For example, suppose *A* has the value 5 and *B* has the value 2 in the following example:

```
&IF %I% = [MOD [CALC %A% * %B%] %MODULUS%] &THEN &RETURN
```

The innermost nested function is evaluated first. The explicit call to CALC returns the value 10, which is then used as the first argument of the MOD function. Omitting the call to CALC causes the string '5 * 2' to be taken as the first argument of the MOD function. Since the string '5 * 2' is not an integer, the MOD function returns an error. (The MOD function does not understand that * means multiplication).

In PRIMOS Commands: All uses of arithmetic operators in PRIMOS commands must be inside an explicit CALC invocation. This is true both for PRIMOS commands issued from the command line and PRIMOS commands issued within a CPL program. For example, the PRIMOS command

```
SPOOL -CANCEL 5 + 1
```

treats 5, +, and 1 as three separate options. The PRIMOS command

```
SPOOL -CANCEL [CALC 5 + 1]
```

evaluates the arithmetic expression and returns 6 as an option of the SPOOL command.

Using PRIMOS Special Characters

CPL supports PRIMOS special characters. For example, the PRIMOS command separator character is the semicolon (;). You can use semicolons to separate multiple PRIMOS commands placed on a single CPL program line.

Syntax Suppression

The tilde is both the CPL line continuation character and the PRIMOS syntax suppression character. The tilde appears at the end of a line when it is used as the line continuation character:

```
This sentence is to be continued ~
on the next line.
```

The tilde appears at the beginning of a line when it is used as the PRIMOS syntax suppression character:

```
~ TYPE Evaluation of this: [LENGTH %STUFF% + 5]; is suppressed.
```

The PRIMOS syntax suppression character is evaluated before CPL special characters. This means that a tilde at the beginning of a line suppresses both PRIMOS special characters (such as semicolons), the evaluation of arithmetic expressions, function calls and variable references, and the use of the CPL line continuation character. A tilde at the beginning of a line does not suppress the comment indicator (/*), unless the comment is within a &DATA group.

Iteration

Parentheses are used as the PRIMOS iteration character. For example,

```
TYPE This is (John Mary Fred Kathy)'s slice of the pie.
```

Executing this command displays the sentence four times, as follows:

```
This is John's slice of the pie.
This is Mary's slice of the pie.
This is Fred's slice of the pie.
This is Kathy's slice of the pie.
```

Iteration is performed after evaluation of variable references and functions. Therefore, the expression

```
TYPE This is (%name% [LENGTH %name%]).
```

prints out as

```
This is Kathy.
This is 5.
```

To suppress iteration but permit CPL evaluation, use the UNQUOTE function and the PRIMOS syntax suppression character as follows:

```
[UNQUOTE '~'] TYPE There are two choices (%go% and CANCEL).
```

The above example evaluates the variable reference, then unquotes the PRIMOS syntax suppression character, which suppresses the iteration indicated by the parentheses. This displays the following single line:

```
There are two choices (PROCEED and CANCEL).
```

PRIMOS special characters are further described in the *PRIMOS Commands Reference Guide.*

12
Functions

This chapter describes CPL functions that you can call as part of any CPL statement. These functions perform specific operations, then return a value that replaces the function call in the CPL statement. For example, the LENGTH function returns the length of a string; therefore, 3 + [LENGTH fred] evaluates to 3 + 4.

This chapter divides the CPL function calls into four groups:

- Arithmetic functions that perform calculations or conversions of integers. The CALC function also performs comparisons of logical (Boolean) expressions.
- String functions that perform operations on a string specified within the function.
- File system functions that access files and directories on your system, including your global variable file.
- Operating system functions that access other PRIMOS facilities, such as the system clock, the ABBREVs facility, and the user terminal.

Note

Functions are listed alphabetically within each of these four groups.

Some functions quote their results and others do not. If the result of a function is most likely to be used as a single token, but contains a semicolon, comma, blank, or quotation mark, or if the result is an arithmetic or logical operator, the function quotes its result. If the result is most likely to be used as a list of multiple items, the result is not quoted. Automatic quoting is done only if the result contains one of the delimiters mentioned, or if it consists of an operator. Thus, the AFTER function quotes its result (when necessary), because the user most likely wants to treat it as one syntactic token. The WILD function, on the other hand, does not quote its result, because the user most likely wants to use the result as a blank-separated list of names rather than as a single string with embedded blanks. Functions that always return one token, such as LENGTH, do not quote their results.

Arithmetic Functions

▶ [CALC expression]

CALC evaluates arithmetic expressions and returns an integer value. CALC evaluates logical (Boolean) expressions and returns the value TRUE or FALSE.

An expression can contain the logical operators & (and), | (or), and ^ (not); the arithmetic operators +, -, *, /, unary +, and unary -; and the relational operators =, <, >, <=, >=, and ^=. The order of precedence is

```
Highest: ^ unary + unary -
   .     / *
   .     + -
   .     = ^= < > <= >=
   .     &
Lowest:  |
```

Parentheses may be used to alter the assigned precedence in the usual way. Five levels of nesting are allowed. Unparenthesized expressions containing operators of equal precedence are evaluated from left to right.

Notes

All operators that are to be evaluated by CALC *must* be delimited by blanks. This restriction resolves the ambiguity that can arise from the fact that *, <, and > are also valid pathname characters.

If you give CALC an expression containing more operators than it can handle, it displays the error message, `Operator stack overflow.` If you receive this message, rewrite the calculation to break it down into simpler expressions.

Logical and relational operators return Boolean values. The strings TRUE, true, T, and t all represent Boolean TRUE, while FALSE, false, F, and f represent Boolean FALSE.

Arithmetic operators return a character string representation of the numeric result. Arithmetic operators apply only to integer values; CPL has no floating point arithmetic.

All the arithmetic operators have the usual definition, except for /, which returns only the truncated integer part of any non-integer result. For example, [CALC 15 / 7] returns 2. Attempting to divide a number by a larger number or by zero returns zero; it does not indicate an error.

Arithmetic, logical, and relational operators have some restrictions on the kind of operands they accept:

- Arithmetic operators *must* have operands that convert to integers. Strings that convert to integers must contain only digits, the plus and minus signs, and leading and trailing blanks. An integer value must be in the range $-2^{31} + 1$ to $2^{31} - 1$. An arithmetic expression returns an integer value.

- Logical operators *must* have operands that are Boolean. Acceptable operands are T, TRUE, F, and FALSE (uppercase or lowercase). The value returned by CALC is TRUE if the logical operation's result is true, and FALSE otherwise.
- Relational operators accept either numeric or non-numeric operands. If a relational operator is given a non-numeric operand, CALC does a string comparison. If both operands are either numeric or Boolean, CALC does an arithmetic comparison. Boolean TRUE is interpreted as 1 and FALSE as 0. A relational expression returns TRUE or FALSE.

For example, suppose *tvar* and *fvar* are variables whose values are TRUE and FALSE, respectively, and *four*, *five*, and *six* are variables with the values 4, 5, and 6. Then

```
%four% + %five%
%six% * ( %four% - %five% )
%tvar% & %fvar%
^ ( %four% < %five% )
%tvar% | ( %four% < %five% )
%fvar% < %tvar%
```

are all valid expressions. However,

```
%tvar% | ( %four% + %five% )
```

is not valid, because %four% + %five% is not a Boolean expression.

▶ [HEX hex-string]

Converts a hexadecimal number to decimal. *hex-string* is an expression that must evaluate to a valid hexadecimal number. This function returns a string representation of the decimal equivalent of *hex-string*. For example, [HEX A] returns 10. Legal values for *hex-string* are 0-9, A-F, and a-f.

▶ [MOD decimal-string decimal-string]

Returns the modulus (remainder) from a division operation. Both arguments must be expressions that evaluate to decimal integers. [MOD dec1 dec2] returns the string representation of the value of *dec1* modulo *dec2*. That is, it returns the remainder resulting from division of *dec1* by *dec2*. For example, [MOD 27 4] returns 3. Attempting to divide a number by zero or by a larger number returns *dec1*; no error is indicated.

▶ [OCTAL octal-string]

Converts an octal number to decimal. *octal-string* is an expression that must evaluate to a valid octal number. This function returns a string representation of the decimal equivalent of *octal-string*. For example, [OCTAL 10] returns 8.

▶ **[TO_HEX decimal-string]**

Converts a decimal number to hexadecimal. *decimal-string* is an expression that must evaluate to a valid decimal number. This function returns a string representation of the hexadecimal equivalent of *decimal-string*. For example, [TO_HEX 15] returns F.

▶ **[TO_OCTAL decimal-string]**

Converts a decimal number to octal. *decimal-string* is an expression that must evaluate to a valid decimal number. This function returns a string representation of the octal equivalent of *decimal-string*. For example, [TO_OCTAL 8] returns 10.

String Functions

String functions take as input one or more strings of characters. An input string can contain quoted or unquoted literals, function calls, variable references or any combination of these. Some string functions return a quoted string, others return an unquoted string.

▶ **[AFTER string find-string]**

Returns the substring of *string* that occurs to the right of the leftmost occurrence of *find-string* in *string*. It returns the null string if *find-string* does not occur in *string* or if *find-string* is at the right end of *string*. For example, [AFTER abcxdefxg x] returns defxg. The AFTER function returns a quoted string.

▶ **[BEFORE string find-string]**

Returns the substring of *string* that occurs to the left of the leftmost occurrence of substring *find-string* in *string*. It returns *string* if *find-string* does not occur in *string*, and returns the null string if *find-string* is at the left end of *string*. For example, [BEFORE abcxdefxg x] returns abc. The BEFORE function returns a quoted string.

▶ **[INDEX string find-string]**

Returns the position of the leftmost occurrence of *find-string* within *string*. Positions are counted from 1. If *find-string* does not occur within *string*, INDEX returns 0. For example, [INDEX abcdef de] returns 4.

▶ [LENGTH string]

Returns the number of characters in *string*. The string does not need to be enclosed in single quotation marks unless it contains a percentage sign (%), a single quotation mark ('), or a square bracket character that is to be treated as a literal. A single quotation mark used as a literal must be doubled. Single quotation marks used to quote a string are not counted in the length of the string. For example, [LENGTH 'can"t'] returns a length of 5.

▶ [NULL string]

Returns TRUE if *string* is either the true null string or '', and FALSE otherwise.

▶ [QUOTE string1 {string2} {string3} {...}]

Returns the input *string(s)* as one quoted string. QUOTE encloses this string with a pair of single quotation marks, then doubles any single quotation marks within the quoted string. The CPL interpreter does not evaluate the contents of a quoted string. Therefore, this function is useful when it is necessary to suppress the meaning of special symbols in a text string. See Chapter 11 for a discussion of quoted strings. For example,

```
[QUOTE xy'|'z] returns 'xy"|"z'
[QUOTE abc 'd e' fg] returns 'abc "d e" fg'
```

▶ [RESCAN string_expression]

Performs a two-step unquote operation. First, RESCAN unquotes *string_expression*. It then rescans this unquoted string, evaluating any function calls or variable references that no longer appear within quotation marks. For example, [RESCAN '[BEFORE "[do not eval this]xxx" x]'] returns [do not eval this].

▶ [SEARCH string1 string2]

Returns the position (counting from 1) of the first character in *string1* that appears in the string *string2*. For example, [SEARCH abc.def <>.+] gives 4. If no character of *string1* appears in *string2*, the SEARCH function returns 0.

▶ [SUBST string1 string2 string3]

Replaces all occurrences of *string2* in *string1* with *string3*. For example, [SUBST aabbaabbaa bb qq] returns aaqqaaqqaa. The SUBST function returns a quoted string.

▶ [SUBSTR string start-pos {num-chars}]

Counts *start-pos* characters from the beginning of *string*, then returns a string of characters of the length specified in *num-chars*. *start-pos* is an integer that indicates the first string position to return. String positions are counted from left to right, starting with 1. *num-chars* is the number of characters to return; it must be either a positive integer or omitted.

If you omit *num-chars*, SUBSTR returns all characters in *string* starting with the position specified by *start-pos* and continuing to the end of the string. If *num-chars* is present, SUBSTR returns the characters in *string* starting with the position specified by *start-pos* and continuing for the number of characters specified in *num-chars*. If *start-pos* exceeds the number of characters in *string*, SUBSTR returns a null string. If *num-chars* exceeds the number of characters from *start-pos* to the end of the string, SUBSTR returns all of the characters from *start-pos* to the end of the string. Neither *start-pos* nor *num-chars* can take a value of zero or a negative number. The SUBSTR function returns a quoted string. For example,

```
[SUBSTR abcde 3 2] returns cd
[SUBSTR 'ab de' 2] returns 'b de'
```

▶ [TRANSLATE string {out-chars in-chars}]

Returns a string computed by the rule: for each character in *string*, if that character appears in the *n* position in *in-chars*, then replace it with the *n* character in *out-chars*. More explicitly,

for each character in *string*:
  if current_char_in_string is in the *n* position in *in-chars*
    then next_char_in_result = *n* character in *out-chars*
      else next_char_in_result = current_char_in_string

The TRANSLATE function returns a quoted string.

If both *out-chars* and *in-chars* are omitted, all lowercase letters in *string* are converted to uppercase, and that result is returned. If only *in-chars* is omitted, then *in-chars* is assumed to be the entire ASCII collating sequence. For example,

```
[TRANSLATE abc] returns ABC
[TRANSLATE 'abc' 123 cab] returns '231'
[TRANSLATE mixxpelled s x] returns misspelled
```

▶ [TRIM string {which-side} {trim-char}]

Trims a leading or trailing sequence of characters from *string*. *which-side* specifies which end of *string* to trim; it may be –RIGHT, –LEFT, or –BOTH. *trim-char* specifies the character to be trimmed. *trim-char* is always a single character. If *trim-char* is omitted, a blank is assumed. If *which-side* and *trim-char* are both omitted, leading and trailing blanks are trimmed. For example, [TRIM bbbabcbbb –both b] returns abc. The TRIM function returns a quoted string.

▶ **[UNQUOTE string]**

Removes the outermost pair of quotation marks from *string* and changes every remaining pair of adjacent quotation marks to a single quotation mark. See the discussion of quoted strings in Chapter 11. For example,

`[UNQUOTE '''xx''''yy''']` returns 'xx''yy'.

▶ **[VERIFY string1 string2]**

Returns the position (counting from 1) of the first character in *string1* that does not appear in *string2*. For example, [VERIFY 1298s8 0123456789] returns 5, because the 5th character in *string1* does not appear in *string2*. The VERIFY function returns 0 if all characters in *string1* appear in *string2*.

File System Functions

The following functions perform operations on existing files. Most of these functions request a pathname as input. This pathname can be a full pathname, for example, GLENN>TOOLS>HAMMER, or a filename, such as HAMMER. It is not necessary to specify the disk partition name, unless different partitions have the same top-level directory names. However, specifying the disk partition name can improve performance. If you specify a filename, these functions assume that the file is located in the current directory. With the exception of EXPAND_SEARCH_RULES and the serialization functions (KLMD, KLMF, and KLMT), none of these functions can use the search rules facility.

▶ **[ATTRIB path** { **–TYPE** / **–DTM** / **–LENGTH** } **{–BRIEF}]**

Returns information about the file specified by *path*. *path* must be a full pathname; this function cannot use the search rules facility. Specify one of the options, –TYPE, –LENGTH, or –DTM, when you call ATTRIB. The –TYPE option causes the function to return the type of the file *path*: SAM, DAM, SEGSAM, SEGDAM, UFD, ACAT, or UNKNOWN. The –DTM option returns the date/time modified information on the file in the format produced by [DATE –FULL]. The –LENGTH (or –LEN) option returns the length of the file in words.

The –BRIEF option, if used, suppresses the display of ATTRIB error messages.

▶ **[DIR path {–BRIEF}]**

Returns the directory portion of the pathname *path*. For example, [DIR smith>x>y] returns smith>x. An asterisk *, representing the home directory, is returned if the pathname is a simple filename. The DIR function returns a quoted string. The –BRIEF option, if used, suppresses the display of DIR error messages.

▶ **[ENTRYNAME path]**

Returns the entryname (filename) portion of the pathname *path*. For example,

```
[ENTRYNAME smith>x>y]
```

returns

```
y
```

▶ **[EXISTS path {type} {-BRIEF}]**

Returns TRUE if a file system object with pathname *path* of type *type* exists, and FALSE if not. If *type* is -ANY, any type of object is acceptable. *type* may also be -FILE, -DIRECTORY, -DIR, -SEGMENT_DIRECTORY, -SEGDIR, -ACCESS_CATEGORY, or -ACAT, to check for the existence of an object of that type. The default type is -ANY.

The -BRIEF option, if specified, suppresses the display of EXISTS error messages.

▶ **[EXPAND_SEARCH_RULES filename {-LIST_NAME listname} {type} {-REFERENCING_DIR pathname}] {-SUFFIX sfx1 {,...} }]**

Returns the fully qualified pathname of *filename*. *filename* can be the name of any file system object: file, directory, ACAT, or segment directory. This function uses the PRIMOS search rules facility to determine the fully qualified pathname of the file system object. It searches all of the locations listed in *listname* to locate the desired file system object. If *filename* cannot be found, EXPAND_SEARCH_RULES returns the value $ERROR$.

The -LIST_NAME (-LNAM) option indicates which search list to use to locate the file. The *listname* you specify must be a search list set for your user process. If you do not specify a -LIST_NAME option, the function uses the COMMAND$ search list for *filenames* that end in a .RUN, .SAVE, or .CPL suffix, and the ATTACH$ search list for all other *filenames*.

You can specify a *type* option of -FILE, -DIRECTORY (-DIR), -SEGMENT_DIRECTORY (-SEGDIR), or -ACCESS_CATEGORY (-ACAT). These options allow you to limit the search to that particular type of file system object.

The -REFERENCING_DIR (-REFDIR) option permits you to specify a search rule that PRIMOS substitutes for the [referencing_dir] entries in the search list. EXPAND_SEARCH_RULES uses this search rule to search for the file system object.

The -SUFFIX (-SFX) option specifies suffixes that PRIMOS appends to the objectname to conduct the search. The suffixes must begin with a period (for example, .RUN). You can specify up to eight suffixes following a -SUFFIX option. Suffixes are searched for in the order listed. If no match is found with all listed suffixes, PRIMOS searches for the object with no suffix.

You can call the EXPAND_SEARCH_RULES function using the abbreviated name ESR. The use of EXPAND_SEARCH_RULES is shown in the following example.

```
&ARGS FILENAME
&S PATHNAME := [ESR %FILENAME% -LNAM MYLIST]
```

This example uses the MYLIST search list to locate the directory containing *FILENAME*. It returns the absolute pathname of *FILENAME*.

For further details concerning the search rules facility, refer to the *Advanced Programmer's Guide, Volume II*. EXPAND_SEARCH_RULES can also be invoked as a PRIMOS command. For further details on the EXPAND_SEARCH_RULES command, refer to the *PRIMOS Commands Reference Guide*.

▶ [GET_VAR expr]

Returns the current value of a local variable or global variable. *expr* must evaluate to a valid variable name. GET_VAR returns the value of that variable if the variable has been defined, or the string $UNDEFINED$ if the variable is undefined.

If *expr* is a global variable, GET_VAR accesses your active global variable file and returns the value of the global variable named in *expr*. GET_VAR also returns $UNDEFINED$ if no global variable file is defined or active.

GET_VAR can also be used to get the value of a variable whose name is computed at runtime. This is useful for simulating indexing and indirection. For example, [GET_VAR a%i%] returns the value of variable *a1* if *i* has the value 1.

▶ [GVPATH]

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

▶ [KLMD pathname {option}]

Returns a string of information about Prime software named in *pathname*. This serialization information includes the name of the software, its revision number, and other attributes.

pathname can be a filename or a complete pathname of a system code object file (for example EMACS.RUN). If *pathname* is a filename, KLMD uses the search rules facility to search the directories listed in the COMMAND$ search list for the file. If KLMD cannot find the file, or if the file is not of the appropriate type, it issues an error.

option specifies which information is to be returned. The available options are

–STD	Returns standard data
–DST	Returns distribution data
–ALL	Returns all data

The standard data contains the following fields:

Product name	20 characters
Revision number	20 characters
Serial number	20 characters
Licensee	40 characters
Expiry date	18 characters
[undefined]	30 characters

The distribution data contains the following fields:

Organization	20 characters
Individual	6 characters
Issue date	18 characters
Order number	8 characters
Customer service number	10 characters

The –ALL option returns both the standard data and the distribution data. It returns the following fields:

Standard data	148 characters
[undefined]	88 characters
Distribution data	62 characters
[undefined]	130 characters

If you do not specify an *option*, KLMD returns the –STD (standard) data.

▶ [KLMF pathname {option}]

Returns the item of information specified in *option* from the Prime software named in *pathname*. The item of serialization information requested in *option* can be the name of the software, its revision number, or some other attribute.

pathname can be a filename or a complete pathname of a system code object file (for example EMACS.RUN). If *pathname* is a filename, KLMF uses the search rules facility to search the directories listed in the COMMAND$ search list for the file. If KLMF cannot find the file, or if the file is not of the appropriate type, it issues an error.

option indicates a specific item of information. For example, –REV returns the software revision number. You must specify one and only one option when you call KLMF. You can specify either the full name or an abbreviated name for each option. The available options are as follows:

Option	*Description*
–PRODUCT	Product name
–REVISION	Revision number
–SERIAL_NUMBER	Serial number of your copy of the software
–LICENSEE	Name of the software licensee
–EXPIRY_DATE	Date the software license expires
–ORGANIZATION	Prime software distribution organization
–INDIVIDUAL	Prime distribution contact
–ISSUE_DATE	Date this copy of the product issued
–ORDER_NUMBER	Order number used by Prime
–CSM_NUMBER	Customer service maintenance number

▶ [KLMT pathname {option {value}} {option {value}}...{-PART}]

Returns TRUE if *value* matches the corresponding attribute of the Prime software named in *pathname*. KLMT returns FALSE if the *value* you specify does not match the corresponding software attribute. KLMT matches the attribute specified in *option*. The available *options* are listed in the description of the KLMF function.

pathname can be a filename or a complete pathname of a Prime runfile (for example EMACS.RUN). If *pathname* is a filename, KLMT uses the search rules facility to search the directories listed in the COMMAND$ search list for the file. If KLMT cannot find the file, or if the file is not of the appropriate type, it issues an error.

You use *option* to indicate a specific item of information and *value* to specify its value. KLMT returns TRUE if the *value* you specify is the actual value in the software. For example, if you specify –REV 21.0.00, KLMT returns TRUE if the software revision number is 21.0.00. KLMT returns FALSE if the software revision number is any other value. *values* are converted to uppercase before they are compared.

You can specify multiple pairs of *option* and *value*, one value per option. KLMT returns TRUE if all of the *values* are correct, and FALSE if any of the *values* is incorrect. You can specify *options* in any sequence. If you specify an *option* but no *value*, KLMT considers *value* to be the null string.

–PART specifies that the matching of *values* with the software data tests only the part of the data specified in *value*. For example, if you specify –REV 21 –PART, KLMT only compares the first two digits of the revision number. If the –PART option is present, KLMT only compares the specified parts of all listed *values*.

▶ [OPEN_FILE pathname –MODE m status-var]

Opens a file for reading and writing. Unlike the OPEN command, the OPEN_FILE function does not require you to specify a unit number. The file specified by *pathname* is opened on some available unit; the unit number (in decimal) is returned as the value of the function. *pathname* must be either a full pathname, or the name of a file in the current directory; OPEN_FILE cannot use the PRIMOS search rules facility. The –MODE option indicates whether the file is to be opened for reading only (*m* = r or R), for writing only (*m* = w or W), or for reading and writing (*m* = wr or WR, position independent). The variable whose name is *status-var* is set to 0 if the operation is successful and is nonzero otherwise; *status-var* may be a local or global variable.

For example,

```
&S READ_UNIT := [OPEN_FILE ALPHA -MODE R OK]
```

In this example, the file named ALPHA is opened for reading. ALPHA must be located in the currently attached directory. The file unit number is returned as the value of the variable *READ_UNIT*. The variable *OK* is set to 0 if the file opening is successful. The variable *OK* is set to a nonzero value if the file opening is not successful. (Because the value of *OK* is being set, not referenced, by the function call, no percent signs surround the variable name.)

Note

CPL uses decimal numbers to refer to file units, *not* octal numbers. If you open a file using the statement

```
&SET_VAR A := [OPEN_FILE THISFILE -MODE R STATUS]
```

close it in one of the following three ways:

```
CLOSE THISFILE
CLOSE -UNIT %A%
CLOSE [TO_OCTAL %A%]
```

Do not say simply, CLOSE %A%; this syntax assumes that A is an octal value and does not work with a decimal unit number.

▶ [PATHNAME rel-path {-BRIEF}]

Returns the full pathname given the relative pathname *rel-path*. *rel-path* is a filename or partial pathname in the current directory. The PATHNAME function cannot use the search rules facility. Note that [DIR [PATHNAME *]] returns the pathname of the current directory.

The PATHNAME function works correctly whether or not the rightmost component of *rel-path* exists. But it produces an error if any other directory in *rel-path* does not exist. For example,

```
[PATHNAME *>FOO>BAR]
```

Returns a full pathname whether BAR exists or not, but produces an error if FOO does not exist.

The -BRIEF option, if specified, suppresses the display of PATHNAME error messages.

▶ [READ_FILE unit status-var {-BRIEF}]

Reads a record from the file open on *unit* and returns the quoted value of that record. By calling READ_FILE repeatedly, you can sequentially read the records in a file. *unit* is the file unit number used to open the file; for example, the value returned by the OPEN_FILE function. The value of *unit* must be a decimal integer. READ_FILE sets the variable *status-var* to 0 if the operation is successful and nonzero otherwise. *status-var* is set to 1 when End of File is reached.

The -BRIEF option, if specified, suppresses the display of READ_FILE error messages.

The following example opens a file using the OPEN_FILE function, then reads the first ten records:

```
&S R_STATUS := 0
&S R_UNIT := [OPEN_FILE GLENN>TOOLS>HAMMER -MODE R O_STATUS]
&DO I := 1 &TO 10 &WHILE %R_STATUS% = 0
&S LINE := [READ_FILE %R_UNIT% R_STATUS]
TYPE Record number %I% is: %LINE%
&END
```

Each time the READ_FILE function call is evaluated, it reads a one-line record from the file open on file unit %R-UNIT%. READ_FILE returns the line of text as the value of the variable LINE. The variable R_STATUS is set to 0 if the read is successful, to 1 if End of File has been reached, or to some other nonzero value if an error has occurred. Because the value of R_STATUS is being set each time the function call is evaluated, the variable name is not placed inside percent signs. READ_FILE does not close the file; you must issue an explicit CLOSE statement (as described in OPEN_FILE) when you wish to close the file.

▶ [WILD wild-path {wild-2 ... wild-n} {control} {–SINGLE unit} {–BRIEF}]

Produces a blank-separated list of entrynames representing the file system objects that match the specifications of *wild-path*, *wild*, and *control*. *wild-path* specifies the directory to consider, and the first wildcard name. The other *wild* arguments specify additional wildcard names (these cannot be pathnames). *control* specifies DTM (Date and Time last Modified) or type restrictions. The *control* options, and their abbreviations, are as follows:

–BEFORE date
–AFTER date
–FILE
–DIRECTORY
–SEGMENT_DIRECTORY
–ACCESS_CATEGORY

For example, [WILD @.PL1 @.F77 –FL] returns a list of files that end with the .PL1 and .F77 suffixes; for example, A.PL1 B.PL1 FOO.F77 BAR.F77 Z.PL1. Listed items are separated by spaces.

If a call on WILD is likely to produce a result longer than the 1024 character maximum, use the –SINGLE option. The –SINGLE option causes WILD to return matching names one at a time, rather than in one long string. Each time WILD is invoked it returns the next matching name. The –SINGLE option takes a variable name as an argument; for example,

```
[WILD london>@.pl1 -SINGLE unit]
```

You must initialize *unit* to zero before calling WILD. When WILD is called with the –SINGLE option and the value of *unit* is zero, WILD opens the specified directory on an available file unit, sets *unit* to the (decimal) number of that unit, and returns the first matching name as its value. Subsequent calls read the directory open on the unit, and return the remaining matching names one at a time. When no more matching names are found, the true null string is returned and the directory closed. Do not modify the value of *unit* between calls to WILD for the same directory.

The –SINGLE option is especially useful with the &ITEMS directive of the &DO statement. (See Chapter 9 for a discussion of the &DO &ITEMS loop and an example of the –SINGLE option.)

The –BRIEF option, if used, suppresses any messages from the WILD function.

▶ **[WRITE_FILE unit text]**

Writes a line of text into an open file. *text* can be any character string. WRITE_FILE strips one layer of quotation marks from *text* and then writes *text* into the file open on *unit. unit* is the file unit number of the open file, represented as a decimal integer. You can open a file using the OPEN_FILE function, which returns a decimal unit number. The WRITE_FILE function returns 0 if the operation is successful and nonzero otherwise. WRITE_FILE does not close the file; to close the file you must invoke a CLOSE operation, as described in the description of the OPEN_FILE function.

Operating System Functions

▶ **[ABBREV -EXPAND text]**

Expands the user abbreviation *text* and returns the expanded string as its result. *text* must be one of the current user's abbreviations. The user's abbreviation file must be active. ABBREV does not quote its result.

If *text* is not an abbreviation, *text* itself is returned. If no abbreviation file is active, an error is reported.

▶ **[CND_INFO control-flag]**

Returns information about the most recent condition on the stack. The function returns different information depending on the setting of *control-flag*. Three possible settings for *control-flag* are

- -NAME returns the name of the condition.
- -CONTINUE_SWITCH (or -CONT_SW) returns the Boolean value of the continue-to-signal switch.
- -RETURN_PERMIT (or -RET_PMT) returns the Boolean value of the return-permitted switch.

If no condition frame is on the stack, -NAME returns $NONE$, and -CONTINUE_SWITCH and -RETURN_PERMIT both return FALSE. The severity code is set to warning in this case. (For information on conditions and on the Prime Condition Mechanism, see the *Subroutines Reference Guide, Volume III.*)

▶ **[DATE {format}]**

Returns the current date/time in a variety of formats. If *format* is omitted, the date only is returned: 86-10-21. The other possibilities for format are

Format	*Example*
-FULL	86-10-21.13:24:48.Tue
-USA	10/21/86

Format	*Example*
-UFULL	10/21/86.13:24:48.Tue
-VFULL	21 Oct 86 10:54:32 Tuesday
-DAY	21
-MONTH	October
-YEAR	1986
-VIS	21 Oct 86
-TIME	13:24:48
-AMPM	1:24 PM
-DOW	Tuesday
-CAL	October 21, 1986
-TAG	861021
-FTAG	861021.132448

The DATE function returns a quoted string.

▶ [QUERY {text} {default} {-TTY}]

Displays a question on the user terminal and waits for a YES or NO answer. QUERY displays *text* on the user's terminal, following it with a question mark. QUERY then halts processing, awaiting a response.

You can respond to a QUERY function from a terminal or from a COMI file. The response must be YES, Y, OK, NO, N (in uppercase or lowercase), or a null response. Press the carriage return after typing your response. YES, Y, and OK return TRUE. NO and N return FALSE.

A carriage return by itself is a null response. A null response returns the *default* value; if no default is specified, a null response returns FALSE. The *default* option can be any text string; it is not limited to the values TRUE and FALSE. Lowercase letters in the *default* value are returned as uppercase letters.

Both *text* and *default* are optional. If you specify them, these options can be any text string, including the null string (represented as ''). If *text* or *default* contain blanks or special characters, you must enclose the string in single quotation marks.

The -TTY option forces the QUERY function to go to the terminal for input, no matter where the command stream that invoked it originated. Without this option, the function takes input from whatever command stream invoked the CPL program, whether that is a user at a terminal, a &DATA group within a CPL program, or a COMINPUT file.

▶ [RESPONSE {text} {default} {-TTY}]

Displays a question on the terminal and waits for a user response. RESPONSE returns the user response or a default value. This function can receive input from the user terminal or from a COMI file.

RESPONSE displays *text* on the user's terminal, following it with a colon. *text* can be any text string, including the null string. The program then waits for a user response.

The user responds by typing any text string, followed by a carriage return. A carriage return by itself is a null response. A null response returns the *default* value; if no *default* value is specified, a null response returns the null string. The value returned by RESPONSE is a quoted string.

The *text* and *default* options can be any text string, including the null string. If a *text* or *default* option contains blanks or special characters, it must be enclosed in single quotation marks.

The –TTY option forces the RESPONSE function to go to the terminal for input, no matter where the command stream that invoked it originated. Without this option, the function takes input from whatever command stream invoked the CPL program, whether that is a user at a terminal, a &DATA group within a CPL program, or a COMINPUT file.

13
Object Arguments and Option Arguments

This chapter provides a full reference for the use of arguments in CPL. CPL arguments are declared using the &ARGS directive; they receive their values from the command line when the program is executed.

The chapter discusses the format and use of

- Object arguments (position-dependent arguments)
- Option arguments (position-independent arguments)
- The REST and UNCL data types, which assign multiple command line values to a single argument

The &ARGS Directive

The &ARGS directive is a powerful tool for argument specification and validation. You use the &ARGS directive to declare local variables whose values are supplied in the command line used to invoke the CPL program.

The arguments described in Chapter 2 and Chapter 6 are object arguments. Object arguments are positional; that is, they must appear on the command line in the same order as they appear in the &ARGS directive. To allow position independence on the command line, you can define option arguments, which use flags to indicate the presence of specific arguments.

You can define a *type* for each argument, such as TREE or PTR, and have the &ARGS directive verify that the argument value supplied on the command line is of the declared type. You can also define a *default* value for each argument. The CPL interpreter assigns the default value to the argument if the command line does not supply a value for that argument.

The number of argument values that you supply in the command line is normally equal to or less than the number of arguments declared in the &ARGS directive. If the number of argument values in the command line exceeds the number of arguments declared in the &ARGS directive, a fatal error occurs, unless you have provided an argument to handle the excess. You can capture these excess argument values and prevent the error by declaring a REST or UNCL argument in the &ARGS directive.

An &ARGS directive may appear anywhere in a CPL program, although it must appear before the variables that it declares are used. When CPL encounters the &ARGS directive, it processes the arguments on the command line that invoked the CPL program. A CPL program may have more than one &ARGS directive. If more than one &ARGS directive is executed, CPL applies the same command line to each &ARGS directive.

Object Arguments

The format for object arguments is

&ARGS name{:{type}{=default}} {;name..}

Object arguments are positional; that is, they must appear on the command line in the same order as they appear in the &ARGS statement. For example, suppose CPL program MYPROG.CPL is to have three arguments. You include a statement like this:

```
&ARGS  SOURCE; DEST; NOLINES
```

If this program is invoked by the command line

```
R MYPROG.CPL Infile Outfile 100
```

then the variable *SOURCE* has the value INFILE, *DEST* the value OUTFILE, and *NOLINES* has the value 100. Note that in this simple default situation, lowercase letters on the command line are converted to uppercase letters. You can control conversion of letters by specifying a *type* for each argument.

An error occurs if you supply too many arguments on the command line. In the above example, typing

```
R MYPROG.CPL INFILE OUTFILE 100 LISTFILE
```

causes printing of the message `Too many object arguments specified. LISTFILE (cpl)`. If too few arguments are given, the omitted ones are assigned a system default value according to their *type*. Table 13-1 lists the supported data types and their default values. The default type is CHAR and its default value is the null string ('').

Specifying Types

You can specify a *type* for each argument by adding a colon (:) and the name of a *type* after the argument name. If you omit *type*, the argument's data type defaults to CHAR.

Specifying a *type* restricts the acceptable values for that argument. For example, if you specify &ARGS NOLINES:DEC, the NOLINES argument is restricted to decimal integer values. CPL

Table 13-1
Data Types for CPL Arguments

Type	*Default*	*Description*
CHAR	"	Any character string with a maximum of 1024 characters. Lowercase letters are shifted to uppercase. This is the default type.
CHARL	"	Any character string with a maximum of 1024 characters. Lowercase letters are not shifted to uppercase.
TREE	"	A PRIMOS pathname with a maximum of 128 characters.
DEC	0	A decimal integer.
OCT	0	An octal integer.
HEX	0	A hexadecimal integer.
ENTRY	"	A file entryname with a maximum of 32 characters.
PTR	7777/0	A virtual address in format *octal/octal.*
DATE	"	A calendar date in the form *mm/dd/yy hh:mm:ss day.*
REST	"	The remainder of the command line.
UNCL	"	All command line items not accounted for by the other &ARGS arguments.

checks the type of the argument value on the command line against the *type* declared for that argument in the &ARGS directive. The CPL interpreter generates a diagnostic and an error severity code if you supply an illegal argument value.

Within a CPL program, arguments are treated as ordinary variables. An argument value can be altered just like other variables; the type is not checked when an argument is assigned a value using the SET_VAR command or the &SET_VAR directive.

To add data types to the previous example, specify

```
&ARGS  SOURCE:TREE; DEST:TREE; NOLINES:DEC
```

A valid command line is

```
R MYPROG.CPL MYDIR>INFILE MYDIR>OUTFILE 50
```

How Null Strings Are Handled

If you do not specify an argument value on the command line, CPL assigns the corresponding argument a default value, as shown in Table 13-1. This default is frequently the null string ('').

The command processor removes all occurrences of the null string from a command before executing it. For example, if you create a CPL program named MYPROG.CPL, which declares an argument named F77ARGS,

```
&ARGS SOMEFILE; F77ARGS
```

and the CPL program uses the variable %F77ARGS% in this line:

```
F77 MYFILE.F77  %F77ARGS%
```

You could run this program using the command line

```
R MYPROG.CPL MYDIR>SOMEPROG
```

Note that this command line omits a value for the F77ARGS argument. If echoing is enabled when you run this CPL program, you see this echoed at your terminal:

```
F77 MYFILE.F77 ''
```

The '' indicates that the value of %F77ARGS% is the null string, which CPL assigned as the system default value for the omitted F77ARGS argument. The command processor removes the '' before executing the command.

Argument Defaults

You may specify a default value for each argument to override the system default values. If you omit a value for an argument from the command line, CPL assigns the *default* value to the argument. You can establish your own default value by typing an equal sign and the *default* after the declared *type*. For example,

```
&ARGS DIRECTORY:TREE=MYDIR
```

If you do not declare the *type*, the *default* appears as follows:

```
&ARGS DIRECTORY:=MYDIR
```

In this case, the *type* is taken to be CHAR.

Continuing the example, you can declare defaults like this:

```
&ARGS  SOURCE:TREE; DEST:TREE=MYDIR>OUTFILE; NOLINES:DEC=100
```

Typing

```
R MYPROG.CPL MYDIR>TEST
```

assigns MYDIR>TEST to SOURCE, the default value MYDIR>OUTFILE to DEST, and the default value 100 to NOLINES.

The default value must match the *type* of the argument. For example, NOLINES:BIN=100 is a valid value, but NOLINES:BIN=ABC is not. An argument of type CHAR can take a default value containing uppercase or lowercase letters; it converts the lowercase letters to uppercase

when it assigns the default value to variables. If a default value is of the wrong type, the CPL interpreter cancels program execution with an error. CPL issues the error even if the argument receives a value of the proper type from the command line.

You can use a function call as a default value. For example,

```
&ARGS REPORTDATE:=[DATE]
```

However, you cannot use a function call containing a blank, such as [DATE –USA]. An argument value containing a blank must be enclosed in quotation marks. Quoting a string disables its evaluation.

You can use a global variable reference as a default value. For example,

```
DEFINE_GVAR GLENN>GLOBALS
&ARGS DIRECTORY:CHAR=%.GVDIR%
```

You can use a local variable reference as a default value. For example, suppose the local variable *STANDARD_DIR* has the value LAUREL>HARDY, then the &ARGS directive

```
&ARGS COMPILE_DIR:CHAR=%STANDARD_DIR%
```

is transformed to

```
&ARGS COMPILE_DIR:CHAR=LAUREL>HARDY
```

and LAUREL>HARDY is assigned to COMPILE_DIR, if no value is supplied on the command line.

Variable references used in *default* may be references to other arguments in the same &ARGS directive. For example,

```
&ARGS COMPILE_DIR:CHAR=%STANDARD_DIR% ;OBJ_DIR:CHAR=%COMPILE_DIR%
```

This example uses the value assigned to argument COMPILE_DIR as the default value of argument OBJ_DIR; that is, if no value is typed in for OBJ_DIR on the command line, it defaults to be the same directory as COMPILE_DIR. This type of assignment works regardless of the order of the arguments in the &ARGS directive, because CPL first interprets all of the &ARGS directive, and then assigns default values to arguments.

Suppose a CPL program contains the following statements:

```
&SET_VAR ARG_VAR := ZYMURGY
&ARGS  ARG_VAR:CHAR; OTHER_ARG:CHAR=%ARG_VAR%
```

The first of these two statements has no effect whatsoever. Since the &ARGS directive is interpreted before the default values are assigned, the value used as the default of *OTHER_ARG* is whatever value was given to *ARG_VAR* in the command line, *not* the string ZYMURGY.

A local variable reference used as a default value cannot refer to another variable reference in the &ARGS directive. Indirect references, such as

```
&ARGS FIRST:=%SECOND%; SECOND:=%THIRD%; THIRD:=MYPROG
```

cancel program execution with an error message. Circular references like

```
&ARGS FIRST:=%SECOND%; SECOND:=%FIRST%
```

also cancel program execution with an error message.

Option Arguments

The format for option arguments is

&ARGS optname:-flaglist{ name{:{type}{=default}}}{ name..}{;optname...}

You can use option arguments to make CPL arguments position independent. An option argument identifies a particular command line argument or group of arguments by a flag name, rather than by its position on the command line.

CPL option arguments are similar to the options used in standard PRIMOS commands. In a PRIMOS command, you can specify options in any sequence. Each option is identified by a flag name that begins with a hyphen. For example, the F77 command takes several options:

```
F77 MYPROG.F77 -LISTING A_PROG.LIST -DEBUG
```

In the F77 command, –LISTING is an option that names the listing file; you specify the name of the listing file immediately after the –LISTING option. –DEBUG is an option that selects a compiler option. Reversing the sequence of these two options does not affect the execution of the F77 command.

In CPL, you can create your own option arguments and flag names. You specify the option argument name and a flag name in the &ARGS directive, and then type the flag name on the command line that runs the CPL program.

A flag name must begin with a hyphen; it can be as many as 32 characters in length. A flag name can contain any characters except the &ARGS directive delimiters (blank, comma, semicolon, colon, and equal sign) and certain PRIMOS command line characters (percentage sign, brackets, parentheses, single quotation mark, and asterisk). For example, –LISTING, –MYOPT.1, and –NO_BINARY are valid flag names. Avoid numeric flag names (for example, -123), because they may be mistaken for negative integers.

When you specify arguments in the &ARGS directive, you can specify all arguments as option arguments, or specify some arguments as object arguments and some as option arguments. In the &ARGS directive, all object arguments (that is, positional arguments) must precede any option arguments. The sequence of the option arguments themselves is not significant. For example,

```
&ARGS ARG_A; ARG_B; ARG_C; ARG2:-SECOND; ARG1:-FIRST; ARG3:-THIRD
```

This example specifies three object arguments followed by three option arguments. When you specify values for these arguments on the command line, you must specify the object arguments in the correct sequence. However, you can specify the flag names of the option arguments in any sequence. You can even specify these flag names before or between the positional object arguments. For example,

```
R MYPROG.CPL  A -THIRD B -FIRST -SECOND
```

Omitting an object argument value from the command line does not affect the option arguments.

Switches

The simplest option argument is a **switch.** A switch option argument is either active or inactive. If the flag name of the switch is present anywhere on the command line, the option argument is active. For example, if you specify –DEBUG on the command line, CPL activates the corresponding option argument in the &ARGS directive. Activating a switch assigns a flag name value to the option argument.

A switch has the following format:

&ARGS optname:-flaglist

-flaglist is a list of one or more flag names. This list of flag names allows you to specify multiple synonyms for the switch argument. When you specify any one of these flag names on the command line, the switch is active. Each flag name must begin with a hyphen. Flag names in *-flaglist* are separated by commas.

optname is the name of the option argument. When the switch is active, *optname* is set to the first name in the *-flaglist.* For example,

```
&ARGS LIST_SW:-LISTING, -L
```

If either –LISTING or –L appears on the command line, CPL assigns the value –LISTING to LIST_SW.

If none of the names in *-flaglist* appear on the command line, CPL sets *optname* to the null string (''). You cannot specify a data type or default value for *optname.* Omitting a flag name from the command line has no effect on the other arguments in the &ARGS directive.

Flags

You can use a flag name to identify one or more arguments that appear immediately after the flag name on the command line. Arguments associated with a flag name are known as an argument group. The flag name signals the presence of the argument group on the command line. Because the flag name identifies the group, the flag name and its group can appear anywhere on the command line.

A flag option argument has the following format:

&ARGS optname:-flaglist arg1 {arg2} ...

optname and *-flaglist* are defined in the previous section. They perform the switch operation described in that section. If you specify any of the *flaglist* flag name synonyms on the command line, CPL performs the switch operation. It sets *optname* to the first name in *-flaglist.* CPL then reads the argument group values that follow the flag name from the command line. It processes the arguments in the argument group as ordinary object arguments. If you do not specify a flag name on the command line, CPL sets *optname* to the null value and sets the members of its argument group to their default values.

arg1 is an object argument that is a member of the argument group. You can specify multiple object arguments (*arg1*, *arg2*, and so forth) in an argument group; you separate these object arguments with blank spaces.

For example, the program COMPILE_AND_GO.CPL contains the statement

```
&ARGS LIST_SW: -LISTING, -L LIST_FILE:TREE;~
      EXEC_SW: -EXECUTE, -E  OBJ_FILE:TREE LIBRARY:CHAR
```

This statement declares two argument groups: LIST_SW and EXEC_SW. The LIST_SW argument group contains the LIST_FILE argument. The EXEC_SW argument group contains two arguments, OBJ_FILE and LIBRARY. LIST_FILE is flagged by either –LISTING or –L: OBJ_FILE and LIBRARY are both flagged by either –EXECUTE or –E.

Some examples of valid invocations of COMPILE_AND_GO.CPL are

```
R COMPILE_AND_GO.CPL  -E GLENN>NEW_OBJ PL1LIB -L GLENN>NEW_LIST

R COMPILE_AND_GO.CPL  -L GLENN>NEW_LIST -E GLENN>NEW_OBJ PL1LIB
```

In these examples, each flag name (–E and –L) identifies an argument group. Each flag name is followed by its appropriate argument values. As shown in these examples, you can specify the argument groups in any sequence. However, you must specify the argument values within each group in the proper order. Following the –E flag name, you must specify the values for the OBJ_FILE and LIBRARY arguments in that order.

CPL assigns the command line argument values that follow the flag name to the arguments in the group until every argument in the group has received a value, or until CPL encounters another flag name or the end of the command line.

Continuing the example with program MYPROG.CPL,

```
&ARGS  ORIG_FILE:-SOURCE,-S SOURCE:TREE;~
       DEST_FILE:-DEST,-D   DEST:TREE=MYDIR>OUTFILE;~
       LINES:-LINES,-L      NOLINES:DEC=100
```

Notice the power of this brief statement. It defines three arguments: SOURCE, DEST, and NOLINES. SOURCE must be a pathname; the argument on the command line corresponding to

SOURCE is flagged by being preceded by either –SOURCE or –S. DEST must also be a pathname; its command line argument is flagged by either –DEST or –D, and defaults to MYDIR>OUTFILE. NOLINES must be a decimal integer; it is flagged by –LINES or –L, and defaults to 100. In addition, this &ARGS directive also defines three switches. It assigns ORIG_FILE the value –SOURCE if that flag (or its synonym) is present on the command line; similarly, DEST_FILE is assigned –DEST and LINES assigned –LINES, if those flags are present.

The following command line runs MYPROG.CPL:

```
R MYPROG.CPL  -D HISDIR>HIS_OUTFILE -S MYDIR>INTEST
```

and results in the following values:

```
%ORIG_FILE% = -SOURCE
%SOURCE%    = MYDIR>INTEST
%DEST_FILE% = -DEST
%DEST%      = HISDIR>HIS_OUTFILE
%LINES%     = ''
%NOLINES%   = 100
```

Another example,

```
&ARGS SOURCE:TREE; LIST_FLAG:-LIST,-L LIST_FILE:TREE;~
      FROM:-FROM FROM_S:DEC=1 FROM_E:DEC=9999
```

The command

```
R MYPROG.CPL MYFILE -FROM 6 -L MYFILE.LIST
```

results in the following values

```
%SOURCE%    = MYFILE
%LIST_FLAG% = -LIST
%LIST_FILE% = MYFILE.LIST
%FROM%      = -FROM
%FROM_S%    = 6
%FROM_E%    = 9999
```

REST and UNCL Data Types

The REST and UNCL data types assign multiple command line values to a single argument. REST and UNCL are invoked after the other &ARGS arguments have received their values from the command line. An argument with one of these data types is useful when you want CPL to assign a few command line values to arguments, and then assign whatever else is left on the

command line to a single argument. Only one of these arguments, either a REST or an UNCL argument, can appear in each &ARGS directive.

- An argument with the data type REST is used primarily with object arguments. When encountered, the REST argument is assigned the rest of the command line; that is, everything on the command line to the right of the last assigned argument.
- An argument with the data type UNCL is used with option arguments. It is assigned the unclaimed items that remain on the command line after all other arguments are assigned.

Both the REST and UNCL arguments prevent the error that otherwise occurs if you specify more arguments on the command line than occur are in the corresponding &ARGS directive.

The REST Argument

The REST argument is particularly useful when you want to assign multiple command line arguments to a single variable. For example, a CPL program that runs a compiler can receive many options for that compiler. Rather than establish a separate argument for each potential compiler option, you can create one REST argument and assign all of the compiler options to that one argument. This situation is shown in the following CPL program, MYPROG.CPL:

```
&ARGS FILENAME:TREE; F77ARGS:REST
    F77 %FILENAME%  %F77ARGS%
```

Typing

```
R MYPROG.CPL GLENN>TEST.F77 -LIST GLENN>LISTINGS>TEST -BIN GLENN>TEST.BIN
```

assigns to %F77ARGS% the rest of the command line:

```
-LIST GLENN>LISTINGS>TEST -BIN GLENN>TEST.BIN
```

The value assigned to the REST argument is not a quoted string, and therefore it can be evaluated without unquoting. The REST data type does not perform case mapping; a command line argument in lowercase letters is not converted to uppercase letters. You can specify a default value for a REST argument.

If an &ARGS statement contains an object argument of type REST, that argument must be the rightmost argument in the &ARGS directive, which must have no option arguments.

You can use multiple REST arguments in option argument groups. Each argument group can have only one REST argument, the REST argument must be the rightmost argument in the group, and only one argument group containing a REST argument can be invoked from the command line.

For example, the following CPL program takes as input the material, weight, and dimensions of various inventory items. You can express these measurements in either metric units or American units:

```
&ARGS MATERIAL; UNITS:-METRIC,-M KILOS:DEC MDIMENSIONS:REST; ~
UNITS:-AMERICAN,-A POUNDS:DEC ADIMENSIONS:REST
```

The following command line runs this program:

```
R INVENTORY.CPL WOOD -A 9 24in 11in 5.4in
```

It performs the following assignments:

```
%MATERIAL%     = WOOD
%UNITS%        = -AMERICAN
%POUNDS%       = 9
%ADIMENSIONS% = 24in 11in 5.4in
```

CPL stops parsing the command line when it encounters a REST type argument; therefore, whatever remains on the command line is assigned to the REST argument, even if it is a flag for another option argument.

The UNCL Argument

An argument of data type UNCL receives whatever is left unclaimed on the command line after all other arguments have been satisfied. Unlike the REST argument, the UNCL argument does not stop the parsing of the command line. Use the UNCL argument only in an &ARGS directive that contains at least one option argument.

Only one instance of the UNCL type may appear in an &ARGS directive. The UNCL argument may appear anywhere in the &ARGS directive, except that an option argument that receives a flag value may not have type UNCL.

For example, the following program, named STAFFING.CPL, contains arguments for a department name and the number of full-time and part-time employees. It uses the UNCL argument to capture command line values that cannot be assigned to those values.

```
&ARGS DEPARTMENT; OTHERS:UNCL; FULL:-FULLTIME,-F FCOUNT:DEC; ~
PART:-PARTTIME,-P PCOUNT:DEC
```

When given the command line

```
R STAFFING.CPL MARKETING -F 24 -CONSULTANTS 2 -P 5 -INTERN 1
```

it assigns the following variables:

```
%DEPARTMENT% = MARKETING
%FULL%       = -FULLTIME
%FCOUNT%     = 24
%PART%       = -PARTTIME
%PCOUNT%     = 5
%OTHER% -CONSULTANTS 2 -INTERN 1
```

As you can see, the UNCL argument can save unexpected command line data. The resulting value of the UNCL argument is an unquoted string. Lowercase letters are not converted to uppercase. You can assign a default value to an argument of type UNCL.

Use caution when declaring arguments of type UNCL. The UNCL type does not ensure order independence under all conditions. If a command line argument begins with a hyphen, and it is not an option flag recognized in the &ARGS directive, all command line arguments between it and the next identifiable option argument flag are assumed to belong to the unidentified argument, and are assigned to the UNCL argument.

For example, consider the following program, MYPROG.CPL:

```
&ARGS FILENAME:TREE; F77ARGS:UNCL
    F77 %FILENAME%  %F77ARGS%
```

Typing

```
R MYPROG.CPL GLENN>TEST.F77 -LIST GLENN>LISTINGS>TEST
```

assigns

```
%FILENAME% = GLENN>TEST.F77
%F77ARGS%  =  -LIST GLENN>LISTINGS>TEST
```

which is the desired result.

However, typing

```
R MYPROG.CPL  -LIST GLENN>LISTINGS>TEST GLENN>TEST.F77
```

assigns

```
%FILENAME% = ''
%F77ARGS%  =   -LIST GLENN>LISTINGS>TEST GLENN>TEST.F77
```

because everything between an undeclared option argument and the next option argument (or the end of the line) is assigned to the UNCL argument.

14
Writing Routines and Functions

This chapter describes how you can create your own routines and functions in a CPL program. It describes the use of &ROUTINE, &CALL, &RETURN, and &STOP directives for the definition of routines. It describes the use of the RESUME function call and the &RESULT directive for the definition of user-written functions.

A routine is a user-written group of CPL statements located within your CPL program. These CPL statements can be executed only by calling the routine (using the &CALL directive). A routine can be called from anywhere in your CPL program. The first part of this chapter explains the construction, invocation and execution of CPL routines. Chapter 15 explains how to use routines for error handling and condition handling.

This chapter also describes how to create user-defined CPL functions. A user-defined function is a group of CPL statements exterior to your CPL program that are executed by a function call in your CPL program. When this function completes, it returns a value to your CPL program. The last part of this chapter explains how to write and invoke functions.

A Note on Terminology

CPL routines are equivalent to subroutines or internal procedures in high-level languages. In PL/I, for example, any program or subroutine is called a **procedure**. A program is an **external procedure**. The subroutines it contains are **internal procedures**. And the main program, minus its subroutines, is the **main procedure**. Figure 14-1 diagrams this terminology.

In this guide, the term **routine** refers to a CPL routine. (For example, we say that every routine begins with a &ROUTINE directive.) The term **procedure** refers to a procedure of any sort (main, internal, or external). For example, we say that a &RETURN directive causes a procedure to return to its caller. This statement is equally true for internal and external procedures.

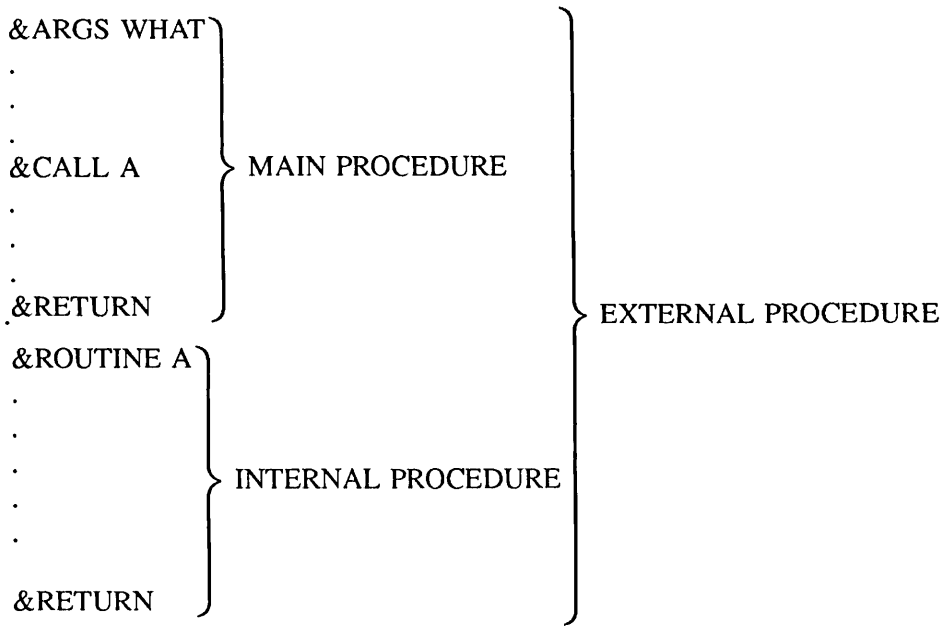


Figure 14-1
High-level Language Terminology

Writing Routines

Routines in CPL are intended primarily for error handling and condition handling. However, they can be used for any purpose for which subroutines are used in high-level languages. For example,

- A routine can replace a lengthy &DO group following a &THEN, &ELSE, or &WHEN. The routine call itself becomes the argument of the &THEN, &ELSE, or &WHEN directive (for example, &THEN &CALL ROUTINE_A).
- A routine can be used when one operation must be performed several times during the course of a program. You write the routine once, and place calls to the routine at all the points where the routine is needed.

How Routines Operate

Routines in CPL operate under the following rules:

- They begin with the directive

```
&ROUTINE routine_label
```

- They are invoked with the directive

```
&CALL routine_label
```

For example,

```
&CALL STARTUP      /* calls the routine named STARTUP
     .
     .
     .
&ROUTINE STARTUP   /* the beginning of STARTUP routine
```

- They can be invoked only by the CPL file within which they exist.
- Their execution is ended by one of the following:
  - A &RETURN directive
  - A &STOP directive
  - A nonlocal &GOTO (that is, a &GOTO to a label that is defined outside the routine containing the &GOTO)
- They are physically terminated by one of the following:
  - The presence of another &ROUTINE directive (signalling the start of another routine)
  - The end of the CPL file
- They use whatever variables the main CPL procedure has defined. They do not create their own copies of these variables. Rather, they act directly on the main procedure's copy. Thus, if a CPL program contains the following code,

```
&S NUMBER := 10
&CALL DOUBLE
TYPE %NUMBER%
&RETURN
&ROUTINE DOUBLE
&SET_VAR NUMBER := %NUMBER% * 2
&RETURN
```

  then the program, when invoked, types the number 20.
- They have their own &DEBUG, &SEVERITY, &CHECK, and &EXPAND settings. If a routine does not set these directives explicitly, then the directives are set to their default values when the routine is entered.

  When control returns to the main procedure, the directive values are re-set to whatever values were set by the main procedure.
- They may be invoked from within a &DATA group, but cannot supply commands to the subsystem being run in that &DATA group. For example, in a &DATA group that runs the BIND loader, you can call an exit routine, but not a routine that supplies a BIND load library.

Placement of Routines

Routines cease executing when they meet a &RETURN or a &STOP directive. However, they do not physically end until they encounter another &ROUTINE directive (signalling the start of the next routine), or the physical end of the CPL file.

A CPL program must not encounter a &ROUTINE directive during normal execution. Routines may be entered *only*

- By the &CALL directive
- By execution of the error-handling directives, &SEVERITY, &CHECK, or &ON

If a CPL program does encounter a &ROUTINE directive during normal execution, execution terminates with an error message.

The best place to put CPL routines, therefore, is at the end of the CPL file, following the main procedure. For example,

```
/*                              main procedure begins here
.
.
.
&CALL ROUTINE_1
&CALL ROUTINE_2
.
.
.
&RETURN                         /* end of main procedure
&ROUTINE ROUTINE_1              /* begin first routine
      .
      .
      .
      &RETURN                   /* first routine ends
&ROUTINE ROUTINE_2              /*  begin second routine
      .
      .
      .
      &RETURN                   /*  second routine ends
```

It is possible to place a routine in the middle of a CPL file. If you do so, however, your program must &GOTO around the routine. For example,

```
.
.
.
&GOTO SKIP_ROUTINE
   &ROUTINE OUT_OF_PLACE
      .
      .
      .
   &RETURN
&LABEL SKIP_ROUTINE
.
.
.
```

This is neither readable nor efficient code; its use is not recommended.

Note

You may leave a routine and enter your main program via a &GOTO, but you may not enter a routine via a &GOTO from the main procedure. Attempting to enter a routine via a &GOTO causes an error, and terminates execution of the CPL program.

Nesting Routines

A routine may call other routines. For example,

```
   .
   .
   .
&CALL A
   .
   .
   .
&RETURN
   &ROUTINE A
      .
      .
      .
   &CALL B
   &RETURN
      &ROUTINE B
         .
         .
         .
      &RETURN
```

Ending Routines: The &RETURN and &STOP Directives

There are two ways in which you may want to terminate a routine:

- If the routine performs correctly, you usually want it to return control to the main CPL program, so that the program can continue its execution. This is performed by the &RETURN directive.
- If the routine fails, or if the routine was called because an error occurred in the main program, you may want the routine to abort execution of the main program and return control to the main program's caller. This is done with the &STOP directive.

The &RETURN Directive: The &RETURN directive ends the routine and returns program execution to the point from which the routine was invoked. A routine can contain more than one &RETURN directive (for example, returns from &IF directive &THEN clauses). If a routine has no return statement, the physical termination of the routine invokes a return operation. A &RETURN directive can return a severity code and display a message text, as shown in Table 14-1. Further details on these uses of &RETURN are found in Chapter 15.

The &STOP Directive: The &STOP directive has the same format as the &RETURN directive. This is shown in Table 14-1.

If the &STOP directive is used in a main procedure, it acts just like the &RETURN directive. However, if the &STOP directive is used in an internal routine, it halts execution of the entire

Table 14-1
Forms of the &RETURN and &STOP Directives

Directive	*Action*
&RETURN	Halts execution of the procedure in which it occurs. Returns control to the procedure's caller.
&STOP	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 of any other routines that program may have active. Control returns to the main program's caller.
&RETURN &MESSAGE text &STOP &MESSAGE text	Halts execution, as above. Displays *text* on user's terminal (and writes it into command output files) when control returns.
&RETURN severity {&MESSAGE text} &STOP severity {&MESSAGE text}	Halts execution, as above. Returns a severity code to the caller. If the &MESSAGE directive is included, displays *text* at the user's terminal and writes it into command output files.

CPL program, and returns control to the program's caller. The following example shows the &STOP and &RETURN directives used in a routine:

```
&ARGS A
&CALL CHECKUP
TYPE A = %A%
&RETURN
&ROUTINE CHECKUP
  &IF %A% < 20 &THEN &RETURN &MESSAGE Arg A acceptable
    &ELSE &STOP &MESSAGE Argument A too large.
```

If the value of *A* in this example is less than 20, the &RETURN directive displays the message `Arg A acceptable`. The TYPE command then prints the value of *A*.

If the value of *A* is greater than 20, the &STOP directive displays the message `Argument A too large`. The &STOP directive also halts execution of the main CPL program. Therefore, the TYPE command is not executed. Instead, control returns to the main program's caller. The caller is either the user (if the user invoked the stopped program) or whatever CPL program invoked the program and passed argument *A* to it.

Writing Functions in CPL

You can define your own CPL functions by writing a CPL program and invoking it using a function call. The format of such a function call is

[RESUME program-name arg-list]

When a CPL program encounters such a function call, it executes program *program-name* as a function, passing it the arguments in *arg-list*. *program-name* is either the full pathname or the filename of the CPL program. If *program-name* is a filename, CPL searches for the program in the currently attached directory; it cannot search for the program using the search rules facility. *arg-list* is an optional list of values to pass to the invoked program. The items in *arg-list* are separated by blank spaces. Like any function call, the *program-name* and *arg-list* for this function call can include variable references, other function calls, and quoted strings.

The program invoked by a RESUME function is an ordinary CPL program. If the RESUME function passes *arg-list* arguments, the invoked program must include an &ARGS directive to receive these arguments.

If you want the invoked program to return a value, it must contain a &RESULT directive. The format for this directive is

&RESULT expression

expression is evaluated and returned as the value of the function, replacing the function call in the text of the calling program. The *expression* must evaluate to a single value; it can include variable references, function calls, or a quoted string. If you do not provide a &RESULT directive, the function call is replaced by a null value.

You can place the &RESULT directive anywhere in the invoked program, although it is usually placed at the end of the program, immediately before the &RETURN directive. A program can contain more than one &RESULT directive, although only one can be used during each invocation; the function returns the value of the last &RESULT directive it encounters before executing a &RETURN or &STOP directive. Therefore, the &RESULT directive can be used in &IF statement clauses such as

```
&IF test &THEN &DO
   &RESULT %value-1%
   &RETURN
   &END
&ELSE &RESULT %value-2%
&RETURN
```

For example, this user-written function, named SQUARE.CPL, either squares or cubes a given number, depending upon the arguments provided:

```
&ARGS X:DEC; EXPONENT:DEC
&IF %EXPONENT% = 2 &THEN &DO
     &RESULT %X% * %X%
     &RETURN
     &END
&IF %EXPONENT% = 3 &THEN &DO
     &RESULT %X% * ( %X% * %X% )
     &RETURN
     &END
&ELSE &STOP &MESSAGE Improper exponent
```

SQUARE.CPL is invoked by the following statement:

```
&S A := [RESUME SQUARE.CPL 5 3]
```

SQUARE.CPL takes 5 as the value for argument X and 3 as the value for argument EXPONENT. It therefore performs the cube of 5 and returns the integer 125. Variable A (in the calling program) is set to the returned value of 125.

If a CPL procedure that contains a &RESULT directive is not invoked as a CPL function (that is, if the invocation is not enclosed within function call brackets) executing the &RESULT directive causes an error.

15
Error and Condition Handling

This chapter discusses the following topics:

- Error handling, using the &SEVERITY and &CHECK directives
- Returning severity codes from routines and programs using the &RETURN and &STOP directives
- Condition handling, using the &ON, &REVERT, and &SIGNAL directives
- Handling errors and conditions using CPL routines

Error Handling

When a PRIMOS command executes, it produces an error code known as a severity code. Severity codes may take one of three values, as shown in the table below. After each PRIMOS command executes, the severity code it produces is placed in the system-defined local variable, SEVERITY$. The SEVERITY$ variable contains the severity code value for the most recently executed PRIMOS command.

Code	*Meaning*
0	No error
Positive integer	Error
Negative integer	Warning

Note

The system-defined variable SEVERITY$ is generated by the first PRIMOS command issued in a CPL program or routine; therefore, this first PRIMOS command cannot test the SEVERITY$ variable. Do not define a variable named SEVERITY$ in your CPL program. Doing so interferes with CPL's ability to handle errors.

When a CPL program is executing, the CPL interpreter checks the value of SEVERITY$ following the execution of each PRIMOS command (and following the execution of the &ARGS directive, as well). If SEVERITY$ has a value greater than zero, and the CPL program has not defined its own error handling parameters, the CPL interpreter terminates execution of the CPL program.

User-defined Error Handling

CPL programs can define their own error handling in four ways:

- They can use the &SEVERITY directive to modify the CPL interpreter's response to severity codes.
- They can use the &CHECK directive to define their own error conditions.
- They can use the &ROUTINE directive (in connection with either the &CHECK or the &SEVERITY directive) to define error handling subroutines.
- They can test the value of SEVERITY$ at some specific point in the program by using an &IF statement (for example, &IF %SEVERITY$% > 0 &THEN...).

The program's &SEVERITY directive or &CHECK directive is tested each time that a PRIMOS command is executed. Therefore, common programming practice is to place one &SEVERITY or &CHECK directive at the beginning of the CPL program.

The &SEVERITY directive takes precedence over the &CHECK directive. If a PRIMOS command is executed when both a &SEVERITY directive and a &CHECK directive are active, the CPL interpreter invokes the &SEVERITY directive first. If the &SEVERITY directive returns (that is, if it does not execute a &STOP directive or a &GOTO), the &CHECK directive is then executed.

The operation of the &SEVERITY and the &CHECK directives is explained in the following sections.

The &SEVERITY Directive

The &SEVERITY directive has the following format:

&SEVERITY {level action}

where *level* can be

&ERROR
&WARNING

and *action* can be

&FAIL
&IGNORE
&ROUTINE handler_label

For example, **&SEVERITY &ERROR &ROUTINE ERROR_HAPPENED**

The &SEVERITY directive checks the SEVERITY$ system-defined variable immediately after the execution of each PRIMOS command. Therefore, you can specify the &SEVERITY directive once in your CPL program, rather than writing an &IF directive to test the value of SEVERITY$ after each PRIMOS command.

The *action* clause specifies what is to be done if a severity code as bad as or worse than *level* is encountered. If *action* is &FAIL, the CPL interpreter terminates execution of the procedure, and returns a positive severity code to the caller of the procedure. If *action* is &IGNORE, execution continues. If *action* is &ROUTINE, CPL invokes the specified error handling routine. (Handlers are discussed under Condition Handling, later in this chapter.) *handler_label* must evaluate to a routine label.

If specified, *level* must be &ERROR or &WARNING. If *level* is omitted, *action* also must be omitted. Automatic severity handling is then disabled. Hence, typing just &SEVERITY is equivalent to &SEVERITY &WARNING &IGNORE; in other words, ignore all errors.

If the handler ends normally or executes a &RETURN statement, control passes to the statement following the one that caused the &SEVERITY handler to be invoked. The only exception to this is when the program contains both a &SEVERITY and a &CHECK directive. When the &SEVERITY handler returns, it tests the &CHECK directive. If &CHECK evaluates to TRUE, the CPL interpreter invokes the &CHECK handler before returning control to the next statement in the sequence.

If a CPL program contains multiple &SEVERITY directives, execution of a PRIMOS command is checked by the most recently encountered &SEVERITY directive; &SEVERITY directives issued earlier in the program are ignored.

The &CHECK Directive

The &CHECK directive invokes a handler if a given expression is true. The &CHECK directive has the following format:

&CHECK expression &ROUTINE handler

For example,

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

The &CHECK directive causes the CPL interpreter to check the current value of *expression* after executing each PRIMOS command. If *expression* evaluates to TRUE, the &CHECK handler is invoked; otherwise, no action is taken.

If the &CHECK handler ends normally or executes a &RETURN directive, control passes to the statement following the one that caused the invocation.

Usually, *expression* contains the severity level variable reference %SEVERITY$%. However, the &CHECK directive can check any expression that evaluates to a Boolean expression, even if the expression has nothing to do with severity levels. For example, a &CHECK directive can be used to maintain a log of PRIMOS commands executed during a CPL program.

If your program contains both a &SEVERITY directive and a &CHECK directive, the &SEVERITY directive is always invoked first. If the &SEVERITY handler returns, the &CHECK directive is then invoked. Suppose a CPL program contains the following statements:

```
&CHECK %THIS% > %THAT% &ROUTINE IT_WAS_GREATER
&SEVERITY &ERROR &ROUTINE ERROR_HAPPENED
```

In this example, when a PRIMOS command causes a positive severity code to be returned, it also causes variable *this* to become greater than variable *that.* In this case, the &SEVERITY handler ERROR_HAPPENED is invoked first. If ERROR_HAPPENED returns, the &CHECK handler IT_WAS_GREATER is invoked. If that handler returns, control is passed to the next statement after the PRIMOS command that caused the invocations.

If a CPL program contains multiple &CHECK directives, execution of a PRIMOS command is checked by the most recently encountered &CHECK directive; &CHECK directives issued earlier in the program are ignored.

Passing Severity Codes

Assume a CPL program that runs several other CPL programs. Its construction might look like this:

```
RESUME TASK1.CPL
RESUME TASK2.CPL
RESUME TASK3.CPL
```

You want this main program to know whether each of the programs it runs executes correctly. To accomplish this, have each of the three programs return a severity code as part of its &RETURN or &STOP directive. This severity code is a value that you establish within the program; it is not automatically generated.

The &RETURN Directive

The &RETURN directive has the following format:

&RETURN {severity} {&MESSAGE text}

For example: `&RETURN 1 &MESSAGE 'It failed'`

The &RETURN directive returns control to the statement in the main program immediately following the one that invoked the subprogram. In the example above, the &RETURN directive returns a severity code of 1 to the invoking program.

severity must evaluate to an integer. This integer is returned to the invoking program as a severity code. That is, this number is used to set the SEVERITY$ variable in the main program. If you omit *severity* from the &RETURN statement, it returns zero.

If the &MESSAGE clause is present, *text* is displayed at the user's terminal. (See a further discussion of &RETURN in Chapter 14.)

Note

When you define your own value for SEVERITY$ (as you do with this directive), you may assign it whatever integer value you please, and test for that value.

When you test for a system-supplied value for SEVERITY$, however, do not test for a specific integer. Rather, the test should be

- 0, for no error
- > 0, for an error
- < 0, for a warning

The &STOP Directive

The &STOP directive has the following format:

&STOP {severity} {&MESSAGE text}

For example, `&STOP 1 &MESSAGE wrong, Wrong WRONG!`

The &STOP directive is processed like the &RETURN directive if it occurs in a main CPL program. However, if it occurs in a subprogram, it halts both the routine in which it occurs and the procedure that invoked the routine.

The &STOP directive can return a user-defined severity code and/or display a message on the terminal. These features are identical to the features of the &RETURN directive.

The &STOP directive is explained more fully in the discussion of routines in Chapter 14.

Condition Handling

CPL provides an interface to the PRIMOS condition mechanism. This mechanism handles special conditions, such as the interruption of a running CPL program by pressing the CONTROL-P or BREAK keys. Refer to the *Prime User's Guide* for an introduction to the PRIMOS condition mechanism.

The PRIMOS condition mechanism uses a procedure known as an **on-unit** to handle a condition. It calls an on-unit each time a special condition is raised. Different on-units are called for different conditions. This is the default method of handling conditions in CPL.

You can also define your own routines for dealing with special conditions. These user-defined routines are called **handlers**. A condition handler is a CPL routine. First you define a routine in your CPL program using the &ROUTINE directive. Then you declare that routine as a condition handler by an &ON directive. You use a separate &ON directive for each condition. These handler declarations are generally placed at the beginning of your main CPL program. When the CPL interpreter encounters a handler declaration, it saves the name of the handler routine and the name of the condition it handles.

When a condition is raised, the CPL interpreter examines its list of handlers. If it finds a handler for the condition, it executes the handler. The handler may stop program execution, or may handle the problem and then return to the point of interruption. If you have not declared a handler for a condition, the PRIMOS condition mechanism searches the stack for an appropriate on-unit. This on-unit may be a system-defined on-unit, or may belong to another CPL invocation. Information in the condition stack frame is available through the CND_INFO command function. (See Chapter 12.)

Because of the overhead involved in searching the stack for a handler, signalling (deliberately raising) a condition is expensive. Therefore, only use condition signalling for unusual or unlikely events. (Issuing an &ON directive to declare a handler is not expensive.)

The &ON Directive

You can use the &ON directive to define a handler for a condition. The &ON directive has the following format:

&ON condition &ROUTINE handler_label

For example,

```
&ON bad_input &ROUTINE bad_inp_handler
```

This statement defines a handler *handler_label* for *condition*. *handler_label* and *condition* must evaluate to a routine label and an identifier, respectively. *condition* may be one of the predefined PRIMOS conditions (described in the *PRIMOS Subroutines Reference Guide, Volume III*) or one invented by the user. If the condition is raised, and the handler has not been **reverted** (see the next section, The &REVERT Directive), the handler is executed. (User-defined conditions are raised by using the &SIGNAL directive, explained later in this chapter.)

handler_label must be defined by a &ROUTINE directive elsewhere in the CPL program; it may not be defined by a &LABEL directive. If the end of the handler is reached, or if &RETURN is executed, control returns to the PRIMOS condition mechanism. If the handler executes a nonlocal &GOTO to a label outside itself, execution returns to the invocation of CPL in which the handler was defined (the stack is unwound if necessary), and then the &GOTO is executed. This aborts the command that raised the condition. By definition, a label is outside a handler if it occurs earlier in the file than the &ROUTINE directive in question.

The &REVERT Directive

The &REVERT directive has the following format:

&REVERT condition

For example,

```
&REVERT bad_input
```

condition is the name of an error condition. &REVERT cancels (reverts) the CPL program's handler for *condition*. If no handler for *condition* exists, &REVERT performs no operation.

The &ROUTINE Directive

The &ROUTINE directive designates the start of a routine. It has the following format:

&ROUTINE routine_label

For example,

```
&ROUTINE my_routine
```

This directive identifies the code that follows as an internal routine. The &ROUTINE code is terminated by another &ROUTINE directive (indicating the beginning of another internal routine) or by the end of the CPL file. A &ROUTINE directive may not occur inside a CPL statement group, such as &DO, &SELECT, or &DATA. &ROUTINE cannot be executed conditionally; that is, it may not be used inside an &IF or &ELSE statement.

Any routine may be invoked directly by using the &CALL directive (explained in Chapter 14). If the routine is declared as a condition handler by a &CHECK, &SEVERITY, or &ON directive, it may also be invoked by raising the condition it is intended to handle.

Internal routines may not be "fallen into", or entered by a &GOTO. If the &ROUTINE directive is accidently encountered during the normal execution of a CPL program, a fatal error occurs and execution of the program is terminated.

Execution of a routine terminates when it executes a &RETURN or &STOP directive, or when it executes a **nonlocal** &GOTO. A &GOTO is nonlocal if it jumps to a label that appears in the CPL file *before* the routine containing the &GOTO.

The &SIGNAL Directive

You can use the &SIGNAL directive to raise a condition. The &SIGNAL directive has the following format:

&SIGNAL condition {&NO_RETURN}

For example,

```
&SIGNAL bad_input
```

This directive raises the condition specified in *condition* and causes the CPL condition mechanism to search for a handler for that condition. The expression *condition* must evaluate to an identifier.

If there is no handler for *condition* in the CPL program, the PRIMOS condition mechanism continues to search the user's stack for on-units. If the user has written no on-units, PRIMOS condition handling is invoked.

&NO_RETURN may be omitted. If specified (as in &SIGNAL bad_input &NO_RETURN), then it is an error for the handler to return; execution must be aborted using the &STOP directive or a nonlocal &GOTO.

A Condition Handling Example

The following example shows the use of the &ON, &REVERT, &ROUTINE, and &SIGNAL directives in a CPL program:

```
&ON BADNUM &ROUTINE IMPROPER
&ON QUIT$ &ROUTINE IMPROPER
&S NUM := [RESPONSE 'how many']
&S OFNUM := [RESPONSE 'in a sample of']
&IF %OFNUM% = 0 &THEN &SIGNAL BADNUM
&IF %OFNUM% < %NUM% &THEN &SIGNAL BADNUM
&S PERCENT := ( %NUM% * 100 ) / %OFNUM%
&REVERT BADNUM
&ON BADNUM &ROUTINE TINY
&IF %PERCENT% = 0 &THEN &SIGNAL BADNUM
&ELSE TYPE %NUM% in a sample of %OFNUM% is %PERCENT% percent
&RETURN &MESSAGE Successful completion

&ROUTINE TINY
TYPE Percentage less than one percent.
&RETURN

&ROUTINE IMPROPER
TYPE This sample cannot be used.
&STOP &MESSAGE Failed
```

This program requests two integer arguments and returns what percent the first number is of the second number. If the sample is improper, or if the percentage is greater than 100 percent or less than 1 percent, it raises a user-defined condition.

The first &ON directive sets the user-defined condition BADNUM to call the routine IMPROPER. If the argument values are improper, the &SIGNAL directive raises the condition BADNUM, which is handled by the routine named IMPROPER. The second &ON condition sets the system-defined condition QUIT$ to the routine named IMPROPER. A QUIT$ condition occurs if the user presses the CONTROL-P key while this program is running.

Later in the program, the &REVERT directive disassociates the BADNUM condition from the IMPROPER routine, and the next &ON directive associates BADNUM with the TINY routine. Therefore, when the next &SIGNAL directive raises the condition BADNUM, the condition is handled by the TINY routine.

Appendices

A
Syntax Summary

▶ &ARGS

Syntax: &ARGS {name-1{:{type}{=default} }...{;name-n}
{optname:-flaglist{ name-1{:{type}{=default}}}...{ name-n}{;optname-2...}

Types: CHAR, CHARL, TREE, ENTRY, DEC, OCT, HEX, PTR,
DATE, REST, UNCL

Examples:
```
&args truth; beauty; charm
&args truth:dec; beauty:tree=mydir>file; charm:char
&args charm:char; tr_flag:-tr truth:dec;~
   be_flag:-be beauty:tree=mydir>file
```

▶ &CALL

Syntax: &CALL routine_name

Example:
```
&call this_routine
  .
  .
  .
&routine this_routine
```

▶ &CHECK

Syntax: &CHECK expr &ROUTINE handler

Example:
```
&check %this_var%>%that_var% &routine disaster
```

▶ &DATA

Syntax:
```
&DATA stmt
  data 1
  ...
  data n
&END
```

Example:
```
&data seg
   vl #prog
   &if %debugger_used%~
   &then lo *>bin>new_prog.bin.dbg
   &else lo *>bin>new_prog.bin
&end
```

▶ &DEBUG

Syntax: &DEBUG {option_list}

Options: &ON &OFF &ECHO &NO_ECHO &EXECUTE &NO_EXECUTE &WATCH &NO_WATCH

Example:
```
&debug &echo all &watch beserk_var
```

▶ &DO

Syntax:
```
&DO {iteration}
        stmt
        stmt
         .
         .
        stmt
     &END
```

where **iteration** is any one of the following:

1. null (statement grouping)
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_dir>@.pl1 -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

Syntax: &EXPAND { ON | OFF }

Example:

```
&expand on
```

▶ &GOTO

Syntax: &GOTO label

Example:

```
&goto a_label
```

▶ &IF-&THEN-&ELSE

Syntax &IF test &THEN true_stmt
{&ELSE false_stmt}

Example:
```
&if %i% > 5 &then type i = %i%
```

▶ &LABEL

Syntax: &LABEL label_name
stmt

Example:
```
&label a_label
 attach richs
```

▶ &ON

Syntax: &ON condition &ROUTINE handler_label

Example:
```
&on bad_input &routine bad_input_handler
```

▶ &RESULT

Syntax: &RESULT expr

Example:
```
&result 4 * 6
```

▶ &RETURN

Syntax: &RETURN {severity} {&MESSAGE text}

Examples:
```
&return
&return 1
&return %severity$%
&return &message Hello!
&return 1 &message Oops
```

▶ &REVERT

Syntax: &REVERT condition

Example:
```
&revert bad_input
```

▶ &ROUTINE

Syntax: &ROUTINE handler_name

Example:
```
&routine bad_inp_handler
```

▶ &SELECT

Syntax:
```
&SELECT expr
  &WHEN expr1 {,expr2,expr3, ... ,exprN}
  stmt
  &WHEN expr1 {,expr2,expr3, ... ,exprN}
  stmt
      .
      .
      .
  {&OTHERWISE
  stmt}
&END
```

Example:
```
&select %what_to_do%
 &when abc
 attach richs
 &when 6,%one_var% + %two_var%
 &return
 &otherwise
 resume not_one_of_those.cpl
&end
```

▶ &SET_VAR

Syntax: &S{ET_VAR} var1 {, var2, ..., varN} := value}

Examples:
```
&set_var this_var := this_string

&s this_var := this_string

&s a,b,c := 0
```

▶ &SEVERITY

Syntax: &SEVERITY {level action}

where *level* can be

&ERROR
&WARNING

and *action* can be

&FAIL
&IGNORE
&ROUTINE handler_label

Examples:
```
&severity &warning &ignore
&severity &error &routine fix_it
&severity &error &fail
&severity
```

▶ &SIGNAL

Syntax: &SIGNAL condition {&NO_RETURN}

Example:
```
&signal bad_bug &no_return
```

▶ &STOP

Syntax: &STOP {severity} {&MESSAGE text}

Example:
```
&stop 1 &message wrong, Wrong, WRONG!
```

B
Error Messages

Introduction

When an error occurs in a CPL program, the CPL interpreter displays four items of information:

1. A line of text giving
   - The error number
   - The line number in the CPL program in which the error occurred
   - If the errant text itself cannot be displayed, the last token (that is, the last word or operator) read before the error occurred
2. A full error message. If the error-causing text can be displayed, it is included as part of the message.
3. The text of the line of source code in which the error occurred.
4. A line describing the action taken by the CPL interpreter and giving the name of the program in which the error occurred. For example,

```
OK, r blunder

CPL ERROR 40 ON LINE 2. A reference to the undefined
variable
"FILLNAME" has been found in this statement.

SOURCE: como %fillname%.como

Execution of procedure terminated. BLUNDER (cpl)
ER!
```

In this example, program BLUNDER.CPL contained a misprint, FILLNAME, for the variable, FILENAME.

The rest of this appendix contains a list of CPL error messages. The term `text` marks the spot in a message where erroneous text from the running program is printed. Messages are given in order by number.

Error Messages

1 An error was encountered while attempting to read the source text of the procedure.

2 The token <text> was found where the keyword &THEN was expected. All &IF directives must contain a &THEN clause.

3 The keyword "&THEN" may only be used in the "&IF" directive.

4 The "&ELSE" directive may only be used as the directive immediately following an "&IF" directive.

5 The value <text> is not a number, but is used where a number is expected.

6 This "&END" directive could not be matched with a corresponding "&DO", "&DATA", or "&SELECT" directive.

7 Internal CPL error: the value of the loop control variable <text> for this iterative "&DO" loop could not be retrieved. Please contact your system administrator.

8 The value <text> is not Boolean (true/false), but is used where a Boolean value was expected.

9 The value <text> is not a legal variable name, but is used where one is expected.

10 The value <text> is not a valid statement label, or else a &GOTO directive has been used to transfer control to this routine.

11 A syntax error was found in this &ARGS directive.

13 Internal CPL error: the semantic stack has been overpopped. Please contact your system administrator.

14 The value of the &WHILE expression <text> in this &DO loop is not Boolean (true/false) as expected.

15 An unexpected problem was encountered while attempting to access the value of the variable <text> in this statement. Possible internal CPL error; please contact your system administrator.

16 A syntax error was found in a command function reference in this statement.

17 Internal CPL error: an unexpected error occurred while attempting to set the value of variable <text> in this statement. Please contact your system administrator.

18 The numeric value <text> used in this directive exceeds the value range limits of that directive.

19 The token <text> was found where the keyword "&ROUTINE" was expected.

20 The procedure has referenced the global variable <text>, but global variables have not been enabled in this process.

21 An unexpected error occurred while attempting to set or get the value of the global variable <text>. Check the global variables file for possible damage, accidental deletion, or lack of Write access.

22 The token <text> is unrecognized or appears in this iterative "&DO" directive in an unexpected place. This directive contains one or more illegal, duplicate, or out-of-order clauses.

23 The value <text> is not a valid routine name, or is a statement label used where a routine name was expected. A label may not be used as a condition, severity, or check routine.

24 Flow of control has dropped into the routine <text>. Control may be transferred to a routine only by means of a condition, severity, or check routine invocation.

25 The CPL expression <text> contains a non-numeric value where a numeric value was required, or an illegal combination of operators and/or values.

26 This directive ends before the appearance of one or more required clauses.

27 The text <text> follows the logical end of this statement.

28 The token <text> was found where one of the keywords &ERROR, &WARNING, &ROUTINE, &FAIL, or &IGNORE was expected.

29 The value of the check expression of the currently enabled check routine is <text>, which is not Boolean (true/false) as expected.

30 The token <text> was found where the keyword ":=" was expected.

31 The &DATA directive may not be nested.

32 An unexpected error was encountered while operating on the temporary file containing the data from this &DATA block. Check for insufficient access rights, disk full or offline, or the use of "CLOSE ALL" in the procedure.

33 Unable to create or open a temporary file with which to process this &DATA block. Check for insufficient access on the current directory.

34 A Primos command statement is required as an argument to the &DATA directive.

35 The Primos command invoked by this &DATA block has read all supplied input data and is requesting more. To suppress this message and continue execution using terminal input, use the &TTY directive.

37 The token <text> was found where the keyword "&MESSAGE" was expected.

38 An illegal option keyword has been found in this &DEBUG directive.

39 Insufficient storage was available to complete processing of this statement. Reduce the depth of nesting of the CPL program, or the length and/or number of local variables.

40 A reference to the undefined variable <text> has been found in this statement.

41 The text following <text> in this statement contains a syntax error in a variable reference.

42 The end of the CPL procedure file was reached before the logical end of the procedure. One or more &DO, &SELECT, or &DATA directives does not have a matching &END statement.

43 The initial-value, &TO or &BY expression in this numeric "&DO" directive has a non-numeric value.

44 Local command variables are not available at command level.

45 This line contains a command function reference, but the command function was not successfully invoked.

46 The token <text> was found where either &WHEN or &OTHERWISE was expected.

47 The keyword "&WHEN" may only be used in the "&SELECT" directive.

48 The keyword "&OTHERWISE" may only be used as the directive immediately following the last "&WHEN" of a "&SELECT" directive.

49 This command may only be invoked as a command function.

50 The token <text> was found in the options field of this "&SIGNAL" directive. The only option supported is "&NO_RETURN".

51 The token <text> has been found in the options field of this "&EXPAND" directive. The only options supported are "ON" and "OFF".

52 <text> is not a directive recognized by CPL.

53 Abbreviation expansion is enabled for this statement, but the expansion could not be successfully performed.

54 Too many variables have been placed on the watchlist.

55 The &RESULT directive may only be executed in a CPL program invoked as a command function.

56 The label or routine name <text> could not be found in this CPL procedure. It was used as the target of a &GOTO, &CALL, or &ROUTINE directive elsewhere in the procedure.

1001 A null argument (two successive semicolons) was found in this &ARGS directive.

1002 This &ARGS directive contains a syntax error which most likely is an invalid or missing delimiter character.

1003 An illegal option argument name (keyword) has been found in this &ARGS directive.

1004 Repeat counts (indicated by *) are not presently implemented in the &ARGS directive.

1005 An unrecognized data type name has been found in this &ARGS directive.

1006 Internal CPL error: a bad state was encountered during parse of this &ARGS directive. Please contact your system administrator.

1007 A word or token in this &ARGS directive exceeds the implementation maximum limit of 1024 characters.

1008 In this &ARGS directive, an object argument specifier appears to the right of one or more option argument (keyword) specifiers. All object arguments must appear to the left of the first option argument.

1014 The default value specified for an argument in this &ARGS directive is not the correct data type.

1015 In this &ARGS directive, a default value has been specified for a data type for which default values are not supported.

1017 In this &ARGS directive, a default value expression contains an undefined variable reference, or a syntax error in a variable reference.

1018 In this &ARGS directive, the data type UNCL has been specified more than once or for an option (keyword) argument. The UNCL data type may be used only for a single object argument.

1019 This &ARGS directive contains a global variable name (a name starting with "."). Only local variable names may appear in an &ARGS directive.

1020 This &ARGS directive contains an illegal variable name.

1021 The &ARGS directive does not accept numeric option arguments. Option arguments must contain at least one alphabetic character.

C
Running CPL Programs as Batch Jobs and Phantoms

Running CPL Programs as Batch Jobs

To run a CPL program as a Batch job, use the command

JOB pathname {–CPL} {batch_options} {–ARGS CPL_arguments}

pathname is the pathname of the CPL job, with or without the .CPL suffix.

Batch looks for *pathname*.CPL. If it finds it, it runs the file as a CPL job. If Batch does not find *pathname*.CPL, it looks for *pathname*. If it finds *pathname*, it runs it as a command input (COMINPUT) file.

The –CPL option may be used to force Batch to run a file as a CPL file, whether it ends in .CPL or not.

This option may be placed in the command line, or in the $$ JOB line within the CPL file itself. (If a $$ JOB line is used, it must be the first non-comment line of the CPL file.)

batch-options are the usual options that govern control of Batch jobs:

–ACCT information

–CPTIME { seconds / NONE }

–ETIME { minutes / NONE }

–HOME pathname

–PRIORITY value

–QUEUE queuename

–RESTART { YES / NO }

For information on these options, see the *Prime User's Guide* or the *PRIMOS Commands Reference Guide.*

Note

Batch's –FUNIT option cannot be used with CPL programs. File units for CPL jobs are allocated dynamically.

The –ARGS option is used to pass arguments to the CPL program. Everything (except comments when abbrev processing is on) following the word –ARGS is passed as arguments to the CPL program when it is run. For this reason, the –ARGS option must be the last option on the command line or in the $$ JOB line. If any Batch options follow the –ARGS option, Batch ignores them and passes them to the CPL file instead.

Job Displays for CPL Jobs

The JOB –DISPLAY command tells whether a job is a regular job (that is, a COMINPUT file), or a CPL job. Displays for CPL jobs begin with the words **Cpl job**. If the –ARGS option is used, the arguments are shown as the final line of the display (or before **Accts:** if –ACCT is specified).

An Example

Assume a CPL program, named TEST.CPL, that contains the following &ARGS statement:

```
&ARGS WHAT:TREE; HOWMANY:DEC = 0
```

Running this program as a job displays the following lines:

```
OK, JOB TEST -ARGS SMITH>TESTBED 50
[JOB Rev. 20.0  Copyright (c) Prime Computer, Inc. 1985]
Your job, #00009, was submitted to queue normal-1.
Home=<ADVERT>JONES>BATCH_JOBS
OK, JOB -DISPLAY
[JOB Rev. 20.0 Copyright (c) Prime Computer, Inc. 1985]

Cpl job TEST(#00009), user JONES executing (queue normal-1).
Submitted today at 9:05:49 a.m., initiated today at 9:05:58 a.m.
Funit=6, priority=5, cpu limit=None, elapsed limit=None.
Project=DEFAULT, Notify=No.
Args: SMITH>TESTBED 50
Home ufd=<ADVERT>JONES>BATCH_JOBS
OK,
```

Running CPL Programs as Phantoms

Any CPL program that does not request terminal input can be run as a phantom job, using the command

PHANTOM pathname [cpl-arguments]

Notes

You cannot use the PHANTOM command's FUNIT argument when running a CPL program as a phantom job. If you try to do so, the *funit* specification is passed as an argument to the CPL program. (PRIMOS allocates file units dynamically for CPL programs, thus guarding against conflicts.)

A CPL program running as a phantom does not need to use the LOGOUT command to log out the phantom. The &RETURN directive (implicit or explicit), which concludes a CPL program, causes the phantom to log out in an orderly fashion.

D
COMINPUT and CPL Compared

This appendix explains the similarities and differences between CPL programs and command input files (COMINPUT files). It also illustrates, by means of several sample programs, how command input files may be converted into CPL programs.

Comparisons

The questions that arise when comparing CPL files (or programs) and command input files are the following:

- How are the files executed?
- How do they execute other files and programs?
- What commands can they execute?
- What special commands must they contain?
- How can they control the execution of the commands they contain?
- What error-handling capabilities do they have?
- What use can they make of variables?
- What use can they make of user-defined abbreviations?
- How do they handle interactive utilities (such as ED and BIND), and user programs?

The answers to these questions are given below.

Execution of CPL and COMINPUT Files

CPL programs are executed by the RESUME or CPL commands. For example,

```
R  MYPROG.CPL
```

Command input files are executed by the COMINPUT command. For example,

```
CO FILE.COMI
```

Execution of Programs by CPL and COMINPUT Files

CPL programs use the CPL command to execute other CPL programs, the RESUME command to execute most user programs, and BASIC or BASICV to execute BASIC programs. (CPL programs cannot use the COMINPUT command. Therefore, they cannot execute COMINPUT files.)

CPL programs do not need to specify the file units on which other programs are to be opened. The CPL interpreter assigns the units automatically.

Similarly, CPL programs do not need to close the file units after the programs they call have finished running. The CPL interpreter closes them automatically.

Command input files use the COMINPUT command to execute other command files. They execute CPL programs and most user programs with the RESUME command, and BASIC or BASICV to execute BASIC programs.

The command input file *must* specify the file unit on which the called command file is to be opened, and must use the CLOSE command to close the file unit when the called command file has finished running.

What Commands Can Be Used?

CPL programs can contain (and execute) any PRIMOS commands except

- COMINPUT
- CLOSE ALL
- DELSEG ALL
- LOGIN, LOGOUT
- ICE

Command input files can contain any PRIMOS command except

- CLOSE ALL
- DELSEG ALL
- LOGIN, LOGOUT
- ICE

Special Commands Needed

A CPL file needs no special commands. (A CPL program always ends with a &RETURN statement, but the CPL interpreter adds that statement for you if you do not put it in yourself.)

Command input files must end with CO -END, CO -TTY, or CO -CONTINUE.

Control of Execution

CPL programs can control the execution of the commands they contain by evaluating flow of control directives, such as &IF, &DO, and &GOTO, contained in the CPL programs. (These directives are explained in Chapters 2, 8, and 9.)

Command input files allow no control of execution. They must execute every command they contain, in the order in which the commands appear in the file.

Error Handling

CPL programs may use the PRIMOS default mechanisms for error handling, severity code handling, and condition handling. Or, they may use CPL directives and/or routines to define their own error handling, severity code handling, and condition handling. (See Chapter 15 for details.)

Command input files must use PRIMOS default mechanisms for error and condition handling.

Use of Variables

CPL programs can use both local and global variables, as explained in Chapter 4. Command input files can use only global variables. They must use the SET_VAR command to set or change the value of these variables.

Use of Abbreviations

The &EXPAND directive allows a CPL program to pass commands to the abbreviation preprocessor for expansion. Thus, users can use their own abbreviations for PRIMOS commands and their arguments inside CPL files, as well as at command level.

Command input files cannot use the abbreviation preprocessor. The commands they contain can use system-defined abbreviations only.

Use of Interactive Utilities and User Programs

CPL files handle interactive utilities and user programs in three ways:

- If the command that invokes the program or utility appears by itself (for example, BIND), the CPL interpreter invokes the program or utility, and transfers control to the user at the terminal. The user provides the data needed by the utility. When the user leaves the utility (for example, by typing QUIT or FILE), control returns to the CPL program.

- If the command that invokes the utility or user program is preceded by a &DATA directive (for example, &DATA BIND), the CPL interpreter constructs a temporary file to contain the data (or subcommands) needed by the program or utility. Construction of the temporary file terminates when the CPL interpreter reads an &END directive. When the temporary file is complete, the CPL interpreter invokes the utility or user program and gives it the data or commands contained in the temporary file.

  Note

  If the CPL program is attached to one directory when it begins execution of the &DATA group, and to another directory at the end of the &DATA group, it cannot delete its temporary file. The file therefore remains in the directory in which it was created.

  If the &DATA group contains a &TTY directive immediately preceding the &END directive, the temporary file is built, the utility or program invoked, and the data or commands from the temporary file passed to it. When the end of the temporary file is reached, control passes to the user at the terminal. When the user finishes with the program or utility, the CPL file resumes control.

CPL programs may also request specific items of information from the user during their execution by the use of the QUERY and RESPONSE functions (explained in Chapter 5).

Command input files do not distinguish between commands that invoke utilities and other commands.

- A utility is invoked when the command that invokes it is read.
- Once the utility has been invoked, succeeding commands in the COMINPUT file are passed to the utility until some command relinquishes control of the utility.
- If a CO –TTY command appears during this time, control passes to the user at the terminal. If the user types CO –CONTINUE while still inside the utility, the command file resumes passing commands to the utility. If the user leaves the utility and then types CO –CONTINUE, the COMINPUT file resumes passing commands to PRIMOS.

Sample Files

Here are some sample command input files. To demonstrate the comparison between command input files and CPL files, each file has been rewritten twice: once as a CPL file without variables, once as a CPL file with variables.

A Simple File

Here is a simple command file, C_TEST, that compiles and links a FORTRAN program:

```
/*BEGIN TEST OF COMMAND FILE
COMOUTPUT O_TEST
DATE
/*COMPILE THE PROGRAM
F77 TEST.F77
/*LINK THE PROGRAM
BIND
LO  TEST.BIN
LI
DYNT -ALL
FILE
/*COMMAND FILE TEST COMPLETED
DATE
COMO -END
CO -END
```

If C_TEST is rewritten as a CPL program, it looks like this:

```
/*BEGIN TEST OF COMMAND FILE
COMOUTPUT O_TEST
DATE
/*COMPILE THE PROGRAM
F77 TEST.F77
/*LINK THE PROGRAM
&DATA  BIND            /* First change
LO  TEST.BIN
LI
DYNT -ALL
FILE
&END                    /* Second change
/*COMMAND FILE TEST COMPLETED
DATE
COMO -END
```

With the addition of variables you get the following:

```
&ARGS WHAT : TREE = TEST
/*BEGIN TEST OF COMMAND FILE
COMOUTPUT O_TEST
DATE
/*COMPILE THE PROGRAM
F77 %WHAT%.F77
/*LINK THE PROGRAM
&DATA  BIND
LO  %WHAT%.BIN
LI
DYNT -ALL
FILE
&END
/*COMMAND FILE TEST COMPLETED
DATE
COMO -END
```

Command Files That Run Other Command Files

The –CONTINUE option of COMINPUT allows command files to be chained. The following example illustrates the chaining of three command files, and shows how file unit conflicts can be avoided. The command file C_GO contains the following commands:

```
/* Compile the program
F77 TEST.F77 -64V
/* Invoke the BIND command file
COMINPUT C_BINDTEST 7
CLOSE 7
/* Return command to user terminal
COMINPUT -TTY
```

The command file C_BINDTEST contains the following commands:

```
/* BINDTEST command file
BIND
LO TEST.BIN
LI
DYNT -ALL
FILE TEST.RUN
/* Invoke the RUN command file
COMINPUT C_RUNTEST 10
CLOSE 10
COMINPUT -CONTINUE
```

The command file C_RUNTEST contains the following commands:

```
/* RUNTEST command file
R TEST.RUN
/* Return to 'calling' command file
COMINPUT -CONTINUE 7
```

The calls and returns involved in this sequence are much simpler with CPL files. The CPL versions of these three files look like this:

```
/*  GO.CPL, a translation of C_GO
/* Compile the program
F77 TEST.F77 -64V
/* Invoke the BIND operation
R BINDTEST.CPL
```

```
/* BINDTEST command file, CPL version
&DATA BIND              /* Add &DATA directive
LO TEST.BIN
LI
DYNT -ALL
FILE TEST.RUN
&END                    /* Add &END directive
R RUNTEST.CPL           /* R replaces CO
                        /* The CLOSE command has been removed
&RETURN                 /* &RETURN replaces CO -CONTINUE
```

```
/* RUNTEST command file, CPL version
R TEST.RUN
&RETURN
```

If the three files want to pass the name of a local variable among themselves, they can do that as well:

```
/*  New version of GO.CPL
&ARGS WHAT : TREE = TEST
/*  Compile program
F77 %WHAT%.F77 -64V
/*  Pass program name to BINDTEST.CPL
R BINDTEST.CPL %WHAT%
```

```
/*  New version of BINDTEST.CPL
&ARGS WHAT
&DATA BIND
LO %WHAT%.BIN
LI
DYNT -ALL
FILE %WHAT%.RUN
&END
/*  Pass argument to third CPL program
R RUNTEST.CPL %WHAT%
```

```
/* New version of RUNTEST.CPL
&ARGS WHAT
R %WHAT%
/*  Control returns to BINDTEST.CPL automatically
```

A Final Note

If a pathname begins with a quotation mark, COMINPUT programs assume the closing quotation mark. If the programmer forgets to type the closing quotation mark, the COMINPUT program supplies it. CPL programs, on the other hand, neither assume nor supply the final quotation mark. If you have pathname problems when you convert a COMINPUT program to a CPL program, check the pathnames to be sure that each opening quotation mark is balanced by a final quotation mark.

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Surveys

READER RESPONSE FORM

CPL User's Guide
DOC4302-3LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

3. What faults or errors in this manual gave you problems?

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

Name:
Position:
Company:
Address:

Postal Code:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 21
Prime Park, Natick, Ma. 01760