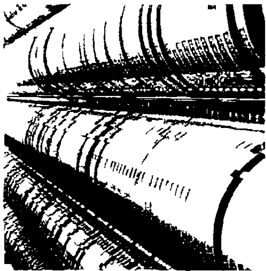
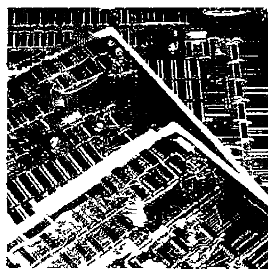
A6

Prime Computer, Inc.

DOC4302-190P
CPL User's Guide
Revision 19.0





CPL User's Guide

DOC4302-190

Revision 19

by

Alice Landy

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19 (Rev. 19).

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

COPYRIGHT INFORMATION

The information in this document is subject to change without notice and should not be construed as a commitment by Prime Computer Corporation. Prime Computer Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Copyright © 1982 by
Prime Computer, Incorporated
500 Old Connecticut Path
Framingham, Massachusetts 01701

PRIME and PRIMOS are registered trademarks of Prime Computer, Inc.

PRIMENET, RINGNET, PRIME INFORMATION, and THE PROGRAMMER'S COMPANION are trademarks of Prime Computer, Inc.

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053, 2054

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000, X4837

PRIME INFORMATION

Contact your PRIME INFORMATION dealer.

PRINTING HISTORY — CPL User's Guide

Edition	Date	Number	Documents Rev.
First Edition	January, 1981	IDR4302	18.1
Second Edition	July, 1982	DOC4302-190	19.0

Changes made to the text since the last printing have been indicated with change bars in the margin. Change bars with numbers indicate technical changes. Those without numbers indicate rewrites for clarification or additional information.

SUGGESTION BOX

All correspondence on suggested changes to this document should be directed to:

Alice Landy
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

Contents

ABOUT THIS BOOK ix

PART I - THE BASIC SUBSET

1 INTRODUCTION

What Is CPL? 1-1
How Might You Use CPL? 1-1
Naming CPL Programs 1-2
Running CPL Programs 1-2
Variables, Functions, and Directives in CPL Programs 1-3
How Does CPL Work? 1-3
CPL Features 1-8
Who Wants Which Features? 1-8

2 THE BASICS OF CPL

PRIMOS Commands in CPL Programs 2-1
Using Variables in CPL Programs 2-5
Decision-Making in CPL Programs 2-7
Other Conditional Actions 2-15
Using CPL with Subsystems: &DATA Groups 2-18
When Errors Occur 2-23
How CPL Programs End: The &RETURN Directive 2-24
When One CPL Program Runs Another 2-24

3 CPL FORMAT

CPL Format Rules 3-1

PART II - THE INTERMEDIATE SUBSET

4 VARIABLES IN CPL

Introduction 4-1
The &SET_VAR Directive 4-1
Integer Values for Variables 4-3
Logical Values for Variables 4-4

Local and Global Variables 4-5
PRIMOS Commands 4-7

5 TERMINAL INPUT AND OUTPUT IN CPL

Overview 5-1
Terminal Input 5-2
Terminal Output 5-8

6 ARGUMENTS WITH TYPE-CHECKING AND DEFAULT VALUES

Introduction 6-1
Type Checking and Default Specification 6-2
Using REST Arguments 6-6

7 PROCESSING GROUPS OF FILES

Grouping Files and Directories 7-1
Filename Conventions 7-1
Using Suffixes: The BEFORE and AFTER Functions 7-2
Wildcards 7-4
The WILD Function 7-6
Using the WILD Function in Loops 7-9

8 DECISION-MAKING IN CPL PROGRAMS

Control Directives 8-1
Single &IF Statements 8-1
Nested &IF Statements 8-3
The &SELECT Directive 8-6

9 LOOPS IN CPL

Using Loops 9-1
Overview 9-1
Counted Loops 9-7
&DO &WHILE Loops 9-9
&DO &UNTIL Loops 9-10
Loops that Combine Counting, &WHILE, and &UNTIL Tests 9-11
&REPEAT Loops 9-11
&DO &LIST Loops 9-11
&DO &ITEMS Loops 9-14

10 DEBUGGING AND ERROR HANDLING IN CPL

Encountering Errors 10-1
Debugging CPL Programs 10-1
&NO EXECUTE/&EXECUTE 10-2
&ECHO/&NO ECHO 10-4
&WATCH/&NO WATCH 10-5
Error Handling 10-6

PART III - FULL CPL

11 EXPRESSION EVALUATION IN CPL

Introduction	11-1
Variables	11-2
Functions	11-3
Quoted Strings	11-4
Using Abbreviations	11-7
Evaluation of Expressions	11-7

12 COMMAND FUNCTIONS

The CALC Function	12-1
Other Arithmetic Functions	12-3
String Functions	12-4
File System Functions	12-7
Miscellaneous Functions	12-11

13 ARGUMENTS

Introduction	13-1
The &ARGS Directive	13-1
Object Arguments	13-2
Specifying Types	13-3
How Null Strings are Handled	13-3
Argument Defaults	13-5
Option Arguments	13-6
REST and UNCL Data Types	13-8

14 WRITING SUBROUTINES AND FUNCTIONS IN CPL

Introduction	14-1
Writing Routines	14-2
Writing Functions in CPL	14-8

15 ERROR AND CONDITION HANDLING IN CPL

Introduction	15-1
Error Handling	15-1
Passing Severity Codes	15-4
Condition Handling	15-6

APPENDIXES

A SYNTAX SUMMARY A-1

B CPL ERROR MESSAGES

Introduction B-1
Error Messages B-2

C RUNNING CPL PROGRAMS AS BATCH JOBS AND PHANTOMS

Running CPL Programs as Batch Jobs C-1
Job Displays for CPL Jobs C-2
Running CPL Programs as Phantoms C-3

D COMINPUT AND CPL COMPARED

Comparisons D-1
Sample Files D-5
A Final Note D-9

E GLOBAL VARIABLE ROUTINES

Introduction E-1
GV$SET E-1
GV$GET E-2
Data-Type Conversions for
FORTRAN and COBOL E-2

INDEX X-1

About This Book

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

This book is divided into three parts.

- Part I introduces CPL and teaches the basics of CPL programming. We advise all readers who are new to CPL to read through these chapters in order.

  Some readers will find their needs satisfied by the features provided by this basic subset of CPL. They will not need to 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, particularly applications programmers, will want to work with this subset.

- Part III presents the additional features which 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 might want to refer to some part of this material, Part III as a whole will probably be of most use to systems programmers.

We assume that the readers of this book are programmers who are already somewhat familiar with Prime's operating system, PRIMOS, and its editor, ED. If you're not familiar with PRIMOS, you should read:

- The Prime User's Guide, Chapters 1-7

- The New User's Guide to EDITOR and RUNOFF, Chapter 3

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 ones are:

- Conway and Gries, An Introduction to Programming: A Structured Approach, Winthrop, Cambridge, MA, 1973
- Xenakis, Structured PL/I Programming, 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. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id
abbreviations	If a command or statement has an abbreviation, it is indicated by underlining. In cases where the command or directive itself contains an underscore, the abbreviation is shown below the full name, and the name and abbreviation are placed within braces.	LOGOUT { &SET_VAR &S }
underlining in examples	In examples, user input is underlined but system prompts and output are not.	OK, RESUME MY_PROG This is the output of MY_PROG.CPL OK,

Large brackets	Large brackets enclose a list of two or more optional items. Choose none, one, or more of these items.	SPOOL [-LIST / -CANCEL]
Large braces	Large braces enclose a list of items. Choose one and only one of these items.	CLOSE {filename / ALL}
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x{,item-y}...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	DIM array (row,col)
Hyphen -	Wherever a hyphen appears as the first letter of an option, it is a required part of that option.	SPOOL -LIST

ADDITIONAL CONVENTIONS FOR THIS BOOK

Braces { }	Braces indicate that the item enclosed is optional.	DATE {option}
Brackets [ ]	Brackets indicate a CPL function call. They must be entered literally.	[EXISTS object]

PART I
The Basic Subset

1 Introduction

WHAT IS CPL?

CPL is Prime's Command Procedure Language. It makes use of such "high-level language" features as branching and argument transfer to simplify and automate long command sequences and to provide decision-making and computational power at the command level.

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
```

That's an annoying amount to type many times a day. But you can type it once, with the Editor, to create a CPL program (named, say, COMP.CPL). Then you can run the CPL program with the simple command:

```
R COMP
```

to compile all three programs.

NAMING CPL PROGRAMS

CPL programs must have names ending in .CPL (e.g., TEST.CPL, COMP.CPL). The .CPL suffix identifies the file as a CPL program to the RESUME, JOB, and PHANTOM commands. However, you do not have to specify the .CPL suffix when you invoke the program. (You may specify it if you wish.)

RUNNING CPL PROGRAMS

CPL programs may be run interactively by the RESUME or CPL commands. They may be run as phantoms by the PHANTOM command, and as Batch jobs by the JOB command. Thus, our sample program, COMP.CPL, could be run by the commands:

- RESUME COMP (or R COMP)
- CPL COMP
- PHANTOM COMP (or PH COMP)
- JOB COMP

When given the filename COMP, the CPL, PHANTOM, and JOB commands look for the file COMP.CPL. Finding it, they run it as a CPL program. (The RESUME command looks first for a runfile (that is, a compiled and loaded program) named COMP.SAVE. If it doesn't find that, it looks for a CPL program named COMP.CPL.)

If COMP.CPL didn't exist, the four commands would then look for plain COMP. If COMP existed, the CPL command would run it as a CPL program. RESUME would run it as a runfile. PHANTOM and JOB would run it as a command input file.

Note

If a CPL program is used by many people, the System Administrator may put it into the system commands directory, CMDNC0. Then it is invoked by typing its name alone. For example:

```
COMP
```

VARIABLES, FUNCTIONS, AND DIRECTIVES IN CPL PROGRAMS

The convenience gained by creating programs composed exclusively of PRIMOS commands is just the beginning of what CPL offers. CPL is modelled 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

Variable references in CPL are identified by being set within percent signs (e.g., %VAR%). Function calls are enclosed in brackets (e.g., [NULL A]). Control directives are preceded by ampersands (e.g., &IF, &GOTO). Through these simple means, you can write CPL programs of great power and flexibility.

HOW DOES CPL WORK?

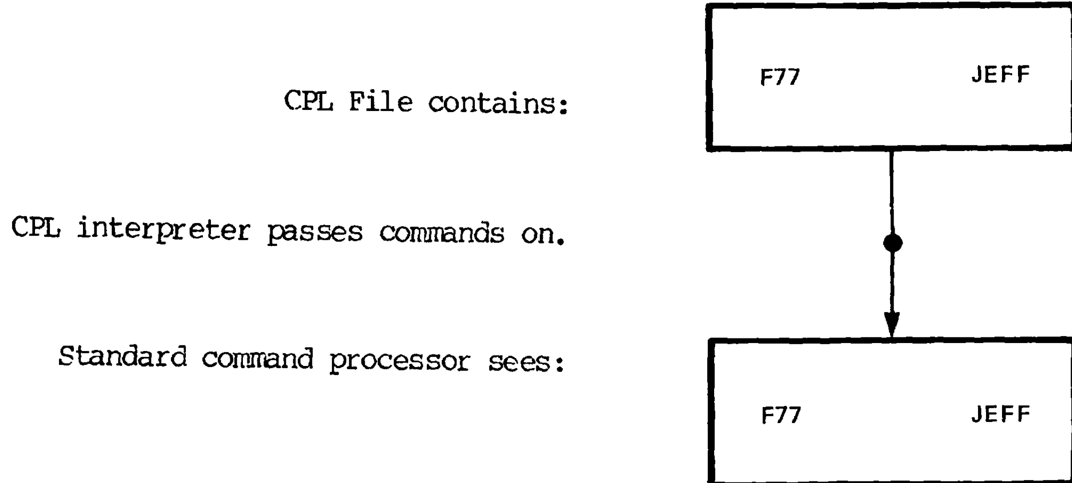
CPL has two parts: the language and the interpreter. The CPL language allows users to write CPL programs which contain either a sequence of PRIMOS commands or a combination of PRIMOS commands and CPL directives. The commands give instructions to PRIMOS, or to one of its subsystems. The directives give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

When the programs are executed, the CPL intepreter 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 commands to PRIMOS for execution. Thus, a lengthy series of commands can be set in motion by a single command, relieving the user of much repetitive typing; yet run-time decisions can be made at any time during the file's execution.

Let's take a closer look at how the interpreter accomplishes this.

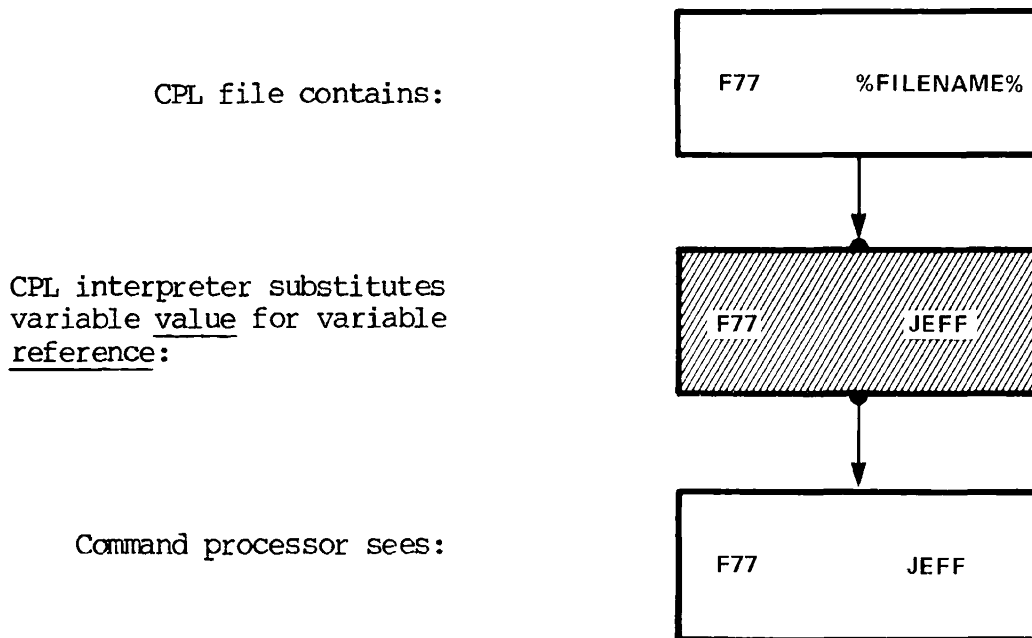
The CPL Interpreter

When a CPL file is run, each line in turn is handed 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.



Command Execution via CPL
Figure 1-1

Variables: If the command contains either variables or function calls, the interpreter evaluates the references, and substitutes the correct values before passing the command to PRIMOS. For example, if JEFF is the current value of a variable called FILENAME, the actions shown in Figure 1-2 are taken. The interpreter, seeing the percent signs surrounding FILENAME, recognizes that they signal a variable reference. It therefore removes the characters %FILENAME% from the command line and replaces them with the characters JEFF. Then it hands the modified command to the command interpreter for execution.



Execution of Command Containing a Variable
Figure 1-2

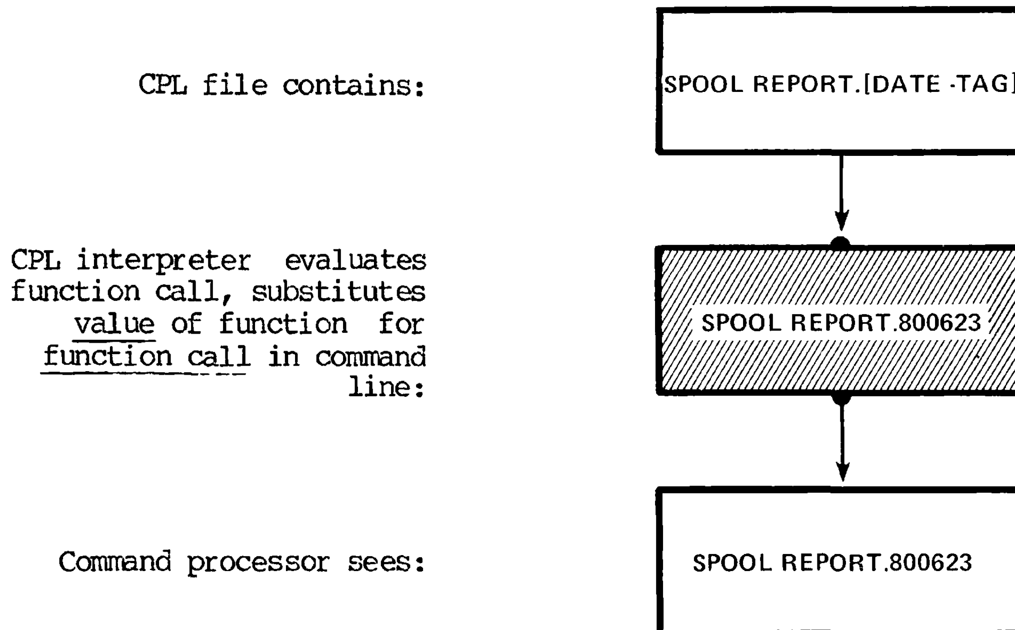
Function Calls: Function calls are treated similarly to variables. That is, if the CPL interpreter finds a function call in a command line, it evaluates that function call and substitutes the character string returned by the function call for the call itself in the command line.

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

Figure 1-3 shows an example of a function call in a command. A CPL program wishes to spool a report it has created. The report is labelled, via a call on the CPL function DATE, with the date of its creation.

The DATE function has several formats. The one shown in this example, DATE -TAG, provides year, month, and day: an ordering which alphabetizes accurately and is thus excellent for "tagging" reports, data files, or listing files.

When the CPL interpreter sees the square brackets that mark the function call, it evaluates the function. In this example, it locates the current date in the correct format. It then substitutes the character string representing this date, 800623 (that is June 23, 1980), for the character string [DATE -TAG]. This completes the interpreter's work on this sample command, so it now passes the command to PRIMOS for execution.



Execution of Command Containing a Function Call
Figure 1-3

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

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

In this example, the interpreter replaces the variable references %A%, %B%, and %FILENAME%, with their current values (say, 3, 1, and JEFF). It tests to see if 3 is greater than 1. Since 3 is greater, it 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.

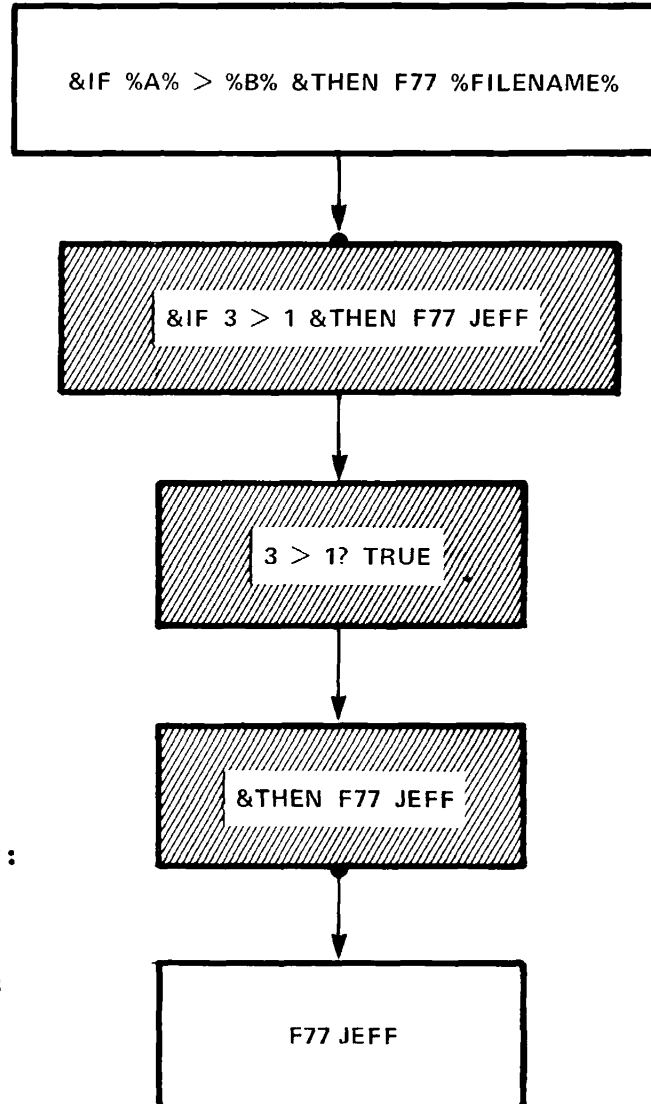
1. CPL file contains the statement:

2. The CPL interpreter reads the statement, substituting current values for variable references:

3. The CPL interpreter tests:

4. Since the test condition is true, CPL executes the &THEN statement, passing the command "F77 JEFF" to the Standard Command Processor:

5. Command processor executes the command:



Execution of a Sample CPL Directive
Figure 1-4

CPL FEATURES

The above examples demonstrate only two of the time-saving features offered by CPL. But they also show how simple CPL programs can be. CPL has features designed to appeal to everyone, from the applications programmer who wants a language that's:

- Simple
- Easy to remember
- Unambiguous

to the system programmer who wants:

- Maximum control
- Flexibility
- The power to write his own commands and command functions

WHO WANTS WHICH FEATURES?

To help users find the features of CPL that will be most useful to them, we have divided this book into three parts, each presenting one "subset" of CPL, as follows:

Part I: The Basic Subset

This is the subset everyone needs to know. It contains the following features:

- Arguments — Allow user to supply values for CPL variables when the CPL file is invoked.
- Flow-of-control directives — Allow CPL interpreter to make tests (e.g., &IF) and take conditional action, (e.g., &THEN &GOTO LABEL) at run time.
- &DATA groups — Allow CPL programs to pass data to user programs and subsystems (e.g., ED, SORT, SEG), and to accept input from the user's terminal.

Part I also explains and demonstrates CPL's format and CPL's default error handling. Sample programs demonstrate CPL programs that:

- Compile programs
- Compile, load and execute programs
- Set your own editor symbols at the start of each edit session

Part II: The Intermediate Subset

This subset contains extensions allowing considerably more flexibility and control while still being easy to use. It offers:

• Arguments with default values and type checking	Useful in CPL programs meant for use by several people.
• "Rest" type arguments	Useful for specifying command options as arguments.
• Further flow-of-control directives	Include loops and case (&SELECT) statements.
• Local variables	Allow variables to be defined within CPL programs, values to be computed at runtime.
• Global variables	Allow variables to be defined at command level, within CPL programs, or within user programs; variables last until the user deletes them.
• Terminal output functions and commands	Allow CPL programs to print messages at the terminal or in output files.
• Terminal input functions	Allow CPL programs to request and use information from user at terminal.
• Simple error handling	Allows users to override CPL's error-handling defaults.
• Debugging	Allows easy debugging of CPL programs.

- Wildcards and their use — Allow easy specification of groups of files and directories.

Part III: Full CPL

Part III is addressed to the programmer who wants the full power and flexibility offered by CPL. Features at this level include:

- Option arguments — Allow the creation of PRIMOS-like commands with position-independent arguments.
- The "unclaimed" argument type — Allows the use of a variable number of arguments, with some position independence.
- Full error handling — Allows users to write their own error-handling routines.
- Full condition handling — Provides an interface to PRIMOS's condition mechanism.
- Abbreviation expansion — Allows CPL programs to use PRIME's ABBREV preprocessor.
- CPL command functions — Provide built-in CPL functions for arithmetic, Boolean, string-handling, and file-handling.
- User-written command functions — Allow users to define their own functions to supplement those provided by CPL.

Further Information

Further information on CPL is provided in appendixes.

- Appendix A summarizes the syntax of CPL.
- Appendix B lists CPL's error messages.
- Appendix C shows how to run CPL programs as Batch jobs or phantoms at Rev 18.1.
- Appendix D tells how to convert existing command input files to CPL programs.
- Appendix E contains two routines by which user programs can define and reference global variables.

2
The Basics of CPL

PRIMOS COMMANDS IN CPL PROGRAMS

The simplest CPL programs are those composed entirely of PRIMOS commands: for example, a CPL file that opens a comoutput file and then compiles three FORTRAN 77 programs. Such a file might be named COMPILE.CPL . It might look like this:

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

Note

> The format of CPL programs is quite simple, being based on the principle of "one statement per line." Each example in this chapter demonstrates correct format. Formatting rules are discussed in Chapter 3.

COMPILE.CPL is run by the command "R COMPILE", "R COMPILE.CPL", or "CPL COMPILE". (Since PRIMOS automatically searches for files with a .CPL suffix whenever the RESUME command is given, you do not have to type the suffix explicitly.

Note

If the System Administrator had placed COMPILE.CPL in the system command UFD, CMDNC0, it would be invoked by its name alone (e.g., "COMPILE") and would behave in all respects like a PRIMOS external command. Similarly, if users define abbreviations for their "Resume CPLfile" commands, they can run those commands by simply typing the abbreviations. (See the Prime User's Guide or the PRIMOS Commands Reference Guide for an explanation of ABBREV.)

Since CPL programs can serve many useful purposes, some of them may well be installed in CMDNC0. Many more will be invoked by user defined abbreviations. However, since these invocations vary from user to user, all examples in this guide will use the RESUME command.

Terminal Displays With CPL

The commands contained in CPL programs are not normally printed out. Thus, if you ran COMPILE.CPL, this is what you would see at your terminal:

```
OK, r compile
24 Dec 81 11:41:52 Thursday

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
```

The COMOUTPUT file would contain the following statements:

```
24 Dec 81 11:42:38 Thursday

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
```

If you want the commands to be printed, you can preface the CPL file with the "&DEBUG &ECHO COM" directive. This directive tells the CPL interpreter to print all commands at the terminal and into output files.

Note

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

If the &DEBUG &ECHO directive were included, the COMPILE.CPL file would look 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
DATE
24 Dec 81 11:45:44 Thursday
F77 THISFILE -XREF

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
F77 THATFILE -XREF -32I

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
F77 TOTHERFILE -DEBUG

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
COMO -E
```

The COMOUTPUT file contains this:

```
DATE
24 Dec 81 11:48:21 Thursday
F77 THISFILE -XREF

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
F77 THATFILE -XREF -32I

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
F77 TOTHERFILE -DEBUG

[FORTRAN 77 19.0]
0000 ERRORS [<.MAIN.> F77-REV 19.0]
COMO -E
```

Which PRIMOS Commands Can You Use?

CPL programs that consist entirely of PRIMOS commands can use the following commands:

- All compiler commands: COBOL, F77, FTN, PL1G, PMA, RPG, etc.
- All commands which execute programs. For example:

  ```
  SEG THISFILE.SEG
  R THATFILE.SAVE
  R FILE.CPL
  BASICV ANYFILE
  ```

- Any user commands which do not invoke a subsystem or initiate a dialog. For example, you may use:

  ```
  ATTACH
  LISTF
  CREATE
  DELETE
  CNAME
  SET_ACCESS
  SET_DELETE
  SIZE
  ```

- Commands that invoke interactive subsystems or user programs, if the user is going to supply the data or subcommands from the terminal at runtime. For example:

  ```
  ED
  SEG
  MAGNET
  SORT
  ```

  If you want the CPL program itself to supply the data or subcommands, you must use CPL's &DATA directive, explained later in this chapter.

What Commands Can't You Use?

Do <u>not</u> use the commands:

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

in a CPL file. Any of these commands will abort execution of the file.

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, they are established with the &ARGS directive. For example, a CPL file (named F7.CPL) that compiles any F77 source file might be:

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

In this example, the &ARGS directive defines one variable, FILENAME. When the file is invoked, the name of the file to be compiled is supplied as an argument, following the name of the CPL file. For example:

```
R F7 JEFF
```

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

From now on, each time a variable reference, %FILENAME%, is found, 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.

Note

> 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

CPL programs can contain multiple arguments. When multiple arguments are given, the variable names in the &ARGS directive must be separated by 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
```

Invoking this file by typing,

```
R COMPILE_ALL JEFF FTN
```

creates the command,

```
FTN JEFF -64V -DEBUG
```

In general, arguments are defined by their position in the command line. In the above example, the first argument, "JEFF", became the value of the first variable in the &ARGS line, "FILENAME". The second argument, "FTN", was assigned to the second variable, "COMPILER". Giving the arguments in reverse order:

```
R COMPILE_ALL FTN JEFF
```

would assign "FTN" to "FILENAME" and "JEFF" to "COMPILER".

Omitted Arguments

If an argument is omitted from the command line, the CPL interpreter sets its value to the explicit null string, ''. The PRIMOS command processor then removes the null string before executing the command.

In the above example, the command:

```
R COMPILE_ALL TESTFILE
```

assigns the value TESTFILE to the variable FILENAME, and assigns the null string to the 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.

CPL offers several ways to deal with null arguments. Some simple ones are explained later in this chapter, in Chapter 5, and in Chapter 13. CPL's &ARGS directive can also 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 file 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 form of the &IF directive is:

```
&IF test &THEN statement
```

Test is a logical test which can be answered TRUE or FALSE (for example, &IF A = B, &IF %NUMBER% < 10). *Statement* is either a command or a CPL directive.

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

- &IF %A% = 10 (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 command or directive that forms the &THEN statement.

An Example: Suppose you compile a program frequently, but only occasionally want to spool the listing file. You could use an argument and the &IF directive to tell the CPL program whether or not to spool the listing file. Here's a program to do it (called CNS.CPL):

Note

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

```
&DEBUG &ECHO COM
     /*This program compiles and optionally spools
     /*an F77 program.
     /*Give the argument "SP" to spool the listing file.
&ARGS FILENAME; SP
     /*Open the COMOUTPUT file and compile the program
COMO %FILENAME%.COMO
DATE
F77 %FILENAME% -L %FILENAME%.LIST -XREF
     /*If desired, spool it.
&IF %SP% = SP  &THEN SPOOL %FILENAME%.LIST -AT MS3
COMO -E
```

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	
&	and
|	or
^	not
Relational Operators	
=	equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
^=	not equal

If you give the command

```
R CNS JEFF SP
```

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

```
R CNS JEFF
```

the test is false (the null string does not equal "SP"). 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.

The &ELSE Directive

The &IF directive may be used by itself, as in the example above; or it may be followed by the &ELSE directive. When used by itself, &IF tells the interpreter either to execute or to ignore some statement. (In the example, spool the file, or don't spool it.) When the &IF and &ELSE directives are used together, they tell the interpreter to choose between two courses of action.

The form of the paired directives is:

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

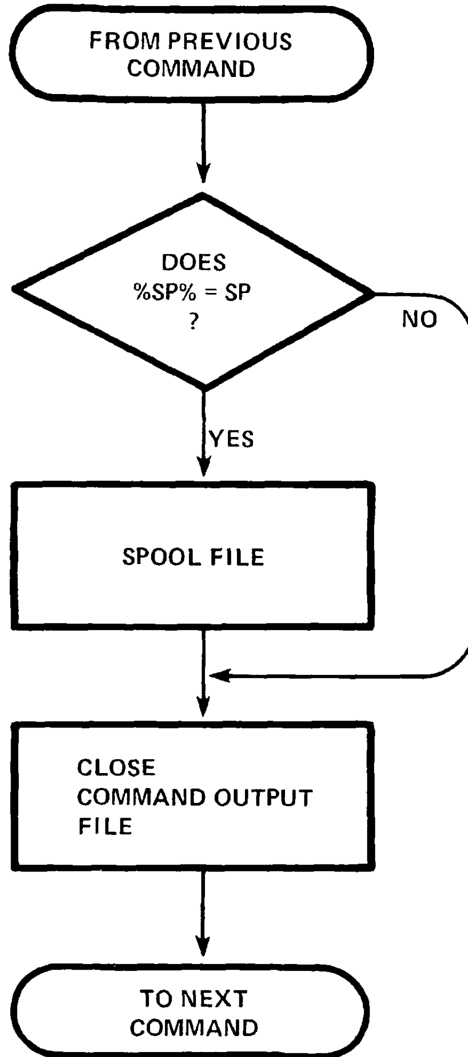
If _test_ is TRUE, _statement-1_ is executed. If _test_ is false, _statement-2_ is executed. For example, suppose you compile many FTN programs and a few F77 programs. You might want a program (called COMPILE2.CPL) that looked like this:

```
&ARGS FILENAME; COMPILER
&IF %COMPILER% = F77 &THEN F77 %FILENAME% -DEBUG -32I
&ELSE FTN %FILENAME% -64V
```

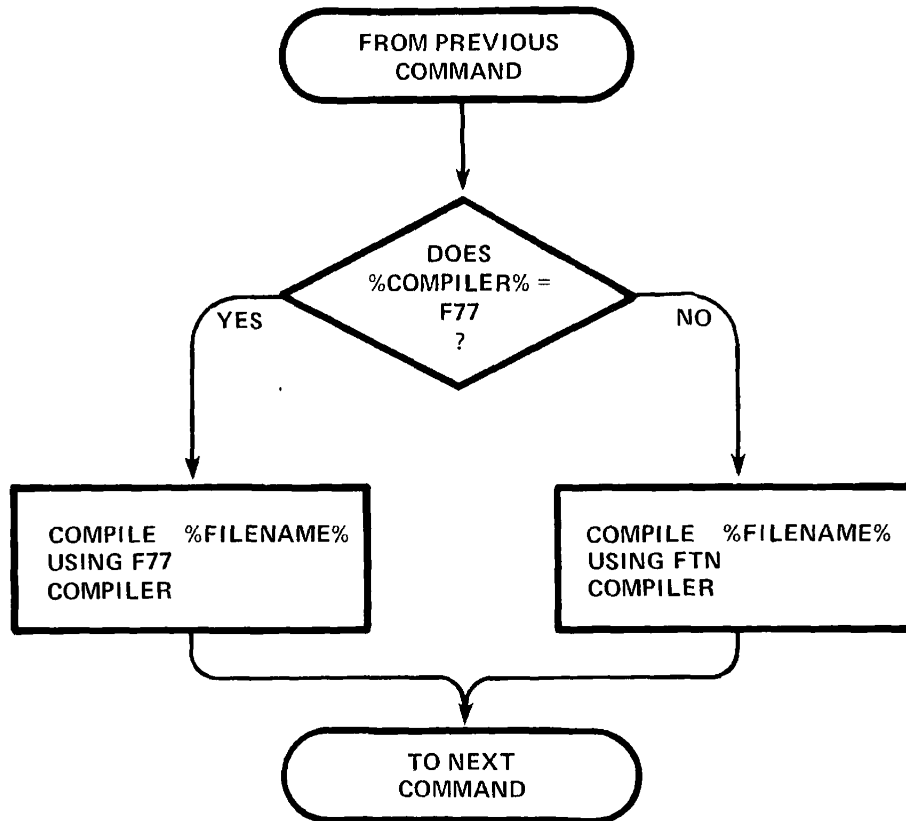
If you give the command "R COMPILE2 THISFILE F77", the test (F77 = F77) becomes true, and THISFILE is compiled by the F77 compiler. If you give any other value for the "compiler" argument--or if you omit that argument altogether--THISFILE is compiled by the FTN compiler. Figure 2-2 shows the flow chart for these statements.

Nested &IFs

&IF directives may be nested: that is, either the &THEN or the &ELSE action of one &IF directive may be another &IF directive. Nested &IF statements are discussed in Chapter 8.



Sample &IF Statement
Figure 2-1



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

Using Functions in &IF Statements

Like other high-level languages, CPL provides built-in functions to simplify frequently made tests and computations. Functions appear in CPL programs in the form of function calls; that is, functions and their arguments enclosed in square brackets (i.e., [FUNCTION arg]). When a function call appears in a command or directive, the CPL interpreter performs the required test or computation, and substitutes the character string thus produced for the character string represented by the function call.

The NULL Function: One of the most useful CPL functions is the NULL function. Its form is

```
[NULL var]
```

where var is any CPL variable.

The NULL function tests for a null character string, returning the character string TRUE if it finds one and the character string FALSE if it does not. 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: A test for a null argument might be used to set the home UFD for some procedure. For example, a CPL file might begin

```
&ARGS WHERE
IF [NULL %WHERE%] &THEN ATTACH MY_UFD
                  &ELSE ATTACH %WHERE%
```

Specifying WHERE allows you to make any desired ATTACH; omitting WHERE attaches you to your default choice (MY_UFD).

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:

- Whether or not a file system object exists
- Whether it matches a specified type (file, directory, or segment directory)

The form of the function call is:

```
[EXISTS pathname {type}]
```

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

type is one of the following:

19.0

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

If type is present, then the EXISTS function returns the value TRUE if pathname does exist 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 UFD that contains three files: PAYROLL.COBOL, COMPILE_ALL.CPL, and PHONE_LIST. Assume that it also contains two sub-UFD's, WORKFILES and MEMOS. If you were attached to this UFD, the function call

```
[EXISTS PHONE_LIST -FILE]
```

would return

```
TRUE
```

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

```
[EXISTS MEMOS -SEGDIR]
```

would return 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.

Continuing with examples from our imaginary directory,

```
[EXISTS MEMOS]
```

returns TRUE, while

```
[EXISTS PAYROLL.FTN]
```

returns FALSE.

Examples: The first example checks to see if a "new" file has been written. If it has, it calls ED to allow its user to edit the 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 wants to attach to a specific directory. If the directory doesn't exist, it will create it before doing the ATTACH:

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

OTHER CONDITIONAL ACTIONS

In the examples above, 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 the command groups. (&THEN and &ELSE directives may also execute GOTO's, as discussed later in this chapter.)

DO Groups

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 the interpreter is asked to perform. In a &DO group, however, all the statements between the &DO and the &END represent a single action to the interpreter. Thus, instead of saying

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

we can say

```
&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:

```
&DEBUG &ECHO COM
&ARGS FILENAME; COMPILER
&IF [NULL %COMPILER%] &THEN &DO
   COBOL %FILENAME%1
   COBOL %FILENAME%2
   COBOL %FILENAME%3
   &END
 &ELSE &DO
   %COMPILER% %FILENAME%1 -64V
   %COMPILER% %FILENAME%2 -64V
   %COMPILER% %FILENAME%3 -64V
   &END
```

Terminal sessions using this program might look like this:

```
OK, R COB_ALL MODULE
   COBOL MODULE1
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI


No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.2 <MODULE>
```

```
  COBOL MODULE2
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI


No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.2 <MODULE>

  COBOL MODULE3
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI


No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.2 <MODULE>

OK,
```

Two further things should be noted in this example:

- The argument MODULE is transformed into the filenames MODULE1, MODULE2, and MODULE3. This is made possible by CPL's method of string substitution. Since %FILENAME%1 appears as a single word (that is, it contains no blanks), the string produced by substituting "MODULE" for "%FILENAME%" is also a single word.
- The statements inside the &DO group are indented. This is done for ease of reading. CPL allows indentation wherever you wish it. It never demands it.

&GOTOs

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. 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 to the statement following the &LABEL directive. Example: &GOTO HERE.

The form is:

```
&GOTO label-name
.
.
.
&LABEL label-name
.
.
```

Once control has passed to the labelled 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 null compiler
&IF [NULL %COMPILER%] &THEN &GOTO DFLT
      &ELSE &GOTO ANY
     /*
&LABEL DFLT /*First alternative
      FTN %FILENAME% -L %FILENAME%.LIST -64V
      &GOTO WRAPUP
     /*
&LABEL ANY /*Second alternative
      %COMPILER% %FILENAME% -L %FILENAME%.LIST -64V
     /*
     /*Both alternatives finish off the same way
&LABEL WRAPUP
      SPOOL %FILENAME%.LIST
      COMO -E
```

USING CPL WITH SUBSYSTEMS: &DATA GROUPS

Many of Prime's utilities, such as ED (the text editor) and SEG (the V-mode and I-mode loader), 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.

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

The form of the &DATA group is:

```
&DATA command
Statement-1
Statement-2
.
.
.
Statement-n
&END
```

Command is the command that invokes the subsystem or utility; for example: "&DATA ED 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.

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

```
/*CPL program to compile, load, and execute a PL1G program
/*usage:  R CLR FILENAME
/*
&ARGS FILENAME
PL1G %FILENAME% -DEBUG -B %FILENAME%.BIN   /*Compile program
/*
&DATA SEG                  /*Invoke SEG
  VLOAD %FILENAME%.SEG     /*Provide SEG commands
  LOAD %FILENAME%.BIN      /*via &data directives
  LI PL1GLB
  LI
  SA
  QU
&END                       /*end of &data group
SEG %FILENAME%.SEG         /*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 CPL's &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, we suggest that you place the &TTY directive at the end of the &DATA group, just before the &END statement.

19.0

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. When the user leaves the subsystem, control returns to the CPL file. Leaving a subsystem happens in a variety of ways; for example:

- The user types QUIT in SEG or CONCAT.
- The user types QUIT, FILE, or FILE filename in the Editor.
- RUNOFF or SORT finish their work and return control to command level automatically.

Conditional Use of the &TTY Directive: You can use the &TTY directive as part of an &IF...&THEN or &IF...&THEN...&ELSE directive. For example, you could say:

19.0

```
&IF something &THEN &TTY
   &ELSE another_statement
```

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

A Sample Program Using the &TTY Directive

One use of the &TTY directive might be to "customize" the Editor for your own use by writing a CPL file that

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" when the edit session is finished, thus returning you to PRIMOS command level.

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

Note

This program uses CPL's &SET_VAR directive followed by a carriage return to define a variable, EMPTYLINE, and 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.

```
/* 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
```

Some terminal sessions using this program might look like this:

```
OK, r edd
INPUT

EDIT
    SYMBOL SEMICO }
    MODE COLUMN

INPUT
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890123456789
This is a sample file
This is the second line of the file

EDIT
file sample
OK,
```

```
OK, r edd 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

Another example shows how the &TTY directive might work 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) might use a CPL program (named P.CPL) to invoke the PURCHASE program and pass it its first two items of information. The statements would look like this:

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

The example as shown could be a complete CPL program. Or, it might be part of a larger program.

A terminal session might look like this:

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

Notes

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

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

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 would produce 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: error or warning. If a warning occurs, the CPL file continues operation. If an error occurs, the file execution is halted and the user is returned to PRIMOS command level, usually with an error message and ER! prompt. (The error message generally includes the name of the command or subsystem that generated it.)

Users can override the handling of PRIMOS errors. Chapters 10 and 15 show how to do this. They cannot change the handling of CPL errors.

HOW CPL PROGRAMS END: THE &RETURN DIRECTIVE

Every CPL program ends with the directive &RETURN. You may either supply this directive as the last line of the CPL file or may allow the CPL interpreter to add the directive at the file's end.

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 ONE CPL PROGRAM RUNS ANOTHER

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

```
COMO ACCTS_UPDATE.COMO
DATE
RESUME NEW_ACCTS
RESUME ACCTS_CLOSED
RESUME ADDRESS_CHANGES
COMO -E
SPOOL ACCTS_UPDATE.COMO
```

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.

For example, when a user runs ACCTS_UPDATE, the following actions occur:

1. The user gives the command,

   RESUME 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 the user.

5. The user gives the next command.

Similarly, when ACCTS_UPDATE invokes NEW_ACCTS:

1. ACCTS_UPDATE passes the command RESUME 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, ACCT_UPDATES must pass them to it, as in:

```
RESUME NEW_ACCTS WEST_BRANCH
```

3
CPL Format

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 existing high-level programming languages. Moreover, CPL's format supports the PRIMOS command line syntax unchanged, for ease of writing and use. This means that:

- PRIMOS commands may be written into a CPL program just as they would be typed interactively.
- CPL programs support PRIMOS's use of the semicolon as a command delimiter. This allows you to write two or more PRIMOS commands (separated by semicolons) on a single line.

▶ 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.) Examples:

```
A MY_UFD
```

This statement shows a single command on a line by itself.

```
CR SUBUFD1;  A *>SUBUFD1
```

This statement represents two commands separated by a semicolon.

```
&IF %VAR% = 1 &THEN SEG #FIRSTFILE
```

The &THEN directive is the argument for the &IF directive. The command SEG #FIRSTFILE is the argument for the &THEN directive. Thus, this line represents one directive plus arguments.

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

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

```
&DO
  SEG #FIRSTFILE
  SEG #SECONDFILE
&END
```

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

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

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

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

This allows you to create whatever indentations you like. For example:

```
&IF %VAR% = 1~
  &THEN SEG #FIRSTFILE
  &ELSE~
    &IF %VAR% = 2~
      &THEN SEG #SECONDFILE
      &ELSE SEG #LASTFILE
```

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
```

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

Examples:

```
SEG #FIRSTFILE                      /*FIRSTFILE does such-and-so

&IF %VAR% = 1 &THEN SEG #FIRSTFILE  /*Test for case 1
```

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.

Thus, the statement:

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

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

▶ RULE 5: Every CPL file ends with a &RETURN directive. If the user omits the &RETURN directive, it is supplied automatically by CPL.

As its name implies, the &RETURN directive halts execution of the CPL procedure and returns control to its "caller". For detailed information on the &RETURN directive, see Chapter 15.

▶ RULE 6: Filenames for CPL programs follow Prime's standard rules for filenames and end with .CPL.

Filenames must not exceed 32 characters. Allowable characters are A-Z, 0-9, _ # $ - . * &. The first character may not be numeric. The CPL interpreter translates lower case characters to upper case. 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 may not exceed 32 characters in length. They may contain only the characters A-Z (upper and lower case), 0-9, underscore (_), and dot (.). (The CPL interpreter translates lower case letters to upper case.) 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: Any 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. Insisting on the use of spaces in the logical expression keeps the distinction clear for users and for the CPL interpreter.

▶ RULE 9. Any string containing blanks or special characters (defined below) must be placed inside single quotes when the string is used as the value of a variable.

Special characters are:

- Single quotes (these must be doubled inside the string). For example:

  '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, when the string is not a CPL option argument. (Option arguments are explained in Chapter 13.)

  '-64V is a FORTRAN option'

- CPL expressions, if you don't want them evaluated.

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

CPL does not evaluate variable references, function calls, or expressions inside quoted strings. Thus, 2 + 3 is an expression but '2 + 3' (quoted) is merely a string. Hence,

```
2 + 3 = 5
```

is TRUE, since 2 plus 3 equals 5; but

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

is FALSE, since the strings "'2 + 3'" and the string "5" are not identical.

Note

> If operator characters are not set off by blanks, quotes are not needed. Thus, the expression A > B (which contains the > operator) must be quoted if it is to remain the character string A > B, rather than being evaluated and replaced with the character string TRUE or FALSE; but the pathname A>B does not need to be quoted.

Using Quoted Strings

Whenever you use a quoted string in CPL, the quotes are considered part of the string. They do not disappear unless you remove them with the CPL unquote function (discussed below). Thus, you can pass quoted strings to PRIMOS.

For example, assume a program SP.CPL which begins:

```
&ARGS pathname
SPOOL %pathname% -FORM WHITE
```

Suppose you needed to pass this program a pathname containing a password. PRIMOS demands that you put such a pathname inside quotation marks. Therefore, you would type:

```
R SP 'TOP SECRET>NEEDLESS>REPORT'
```

SP.CPL would pass PRIMOS this command, with the pathname correctly quoted inside it:

```
SPOOL 'TOP SECRET>NEEDLESS>REPORT' -FORM WHITE
```

Concatenating Quoted Strings: 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'
```

Quoting and Unquoting Strings: CPL provides built-in QUOTE and UNQUOTE functions to place quotes around strings and to remove quotes from strings. The UNQUOTE function is particularly useful, as it allows you to use quoted strings as arguments for a CPL program, then remove the strings inside the program. For example, you might want to pass some PRIMOS command options, which begin with hyphens, as arguments. You could write a CPL file (F.CPL) like this:

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

With this program, the command:

```
R F FOO '-XREF -EXPLIST'
```

produces the command

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

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

F.CPL can also be invoked by the command:

```
R F FOO
```

This invocation produces the PRIMOS command:

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

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.

Note

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 in CPL

INTRODUCTION

This chapter discusses:

- Defining variables with the &SET_VAR (&S) directive.
- The three types of values--string, integer, and logical--that variables can possess.
- 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

The &SET_VAR directive has the form:

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

name-1 through name-n are either:

- Valid variable names (for local or global variables)
- Expressions that evaluate to valid variable names. This allows you to simulate array variables, as in:

```
&SET_VAR A%I% := 30
```

(See Chapter 11 for details.)

value may be

- A character string (up to 1024 characters, quoted if necessary)
- An integer (-2**31 + 1 to 2**31 - 1)
- A logical value (some form of TRUE or FALSE)
- An expression which evaluates to any of the above

Note

Real numbers may not be used as variable values.

The assignment symbol (:=) must be given explicitly.

Examples

```
&S A, B, C := 0
```

This example defines three local variables, A, B, and C, and sets the value of each to zero.

```
&ARGS UFD
&IF %UFD% = N ~
   &THEN &S UFD := ACCTS>RECV>NORTH
   &ELSE &IF %UFD% = S ~
      &THEN &S UFD := ACCTS>RECV>SOUTH
      &ELSE &S UFD := ACCTS>RECV>CENTRAL
```

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

INTEGER VALUES FOR VARIABLES

All CPL variable values are character strings. However, some character strings (such as 3, 259, -6847) can be interpreted as integer values. CPL allows standard arithmetic operations on these integers. (For a summary, see Table 2-1.) 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)
```

Note

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

Examples Using String and Integer Values

```
&SET_VAR A := ELLEN
&SET_VAR B := [LENGTH %A%]
```

This example gives the value ELLEN to the variable A. Then it uses CPL's LENGTH function to set the variable B to the length of A. B therefore has the value 5.

Note

Do not try to perform arithmetic operations on character-string variables. For example, do NOT say:

```
&SET_VAR A := ELLEN
&SET_VAR B := %A% + 1
```

Executing these commands produces an error message and aborts execution of the CPL program.

```
▶ &SET_VAR A := MY_UFD
  &SET_VAR B := >THISFILE
  &SET_VAR C := %A%%B%
```

This example uses CPL's automatic string concatenation to set the value of C to the pathname MY_UFD>THISFILE. Since CPL merely substitutes the value of the variable for the reference--that is, substitutes "MY_UFD" for "%A%" and ">THISFILE" for "%B%",—a value composed of two juxtaposed variable references evaluates to a single character string.

```
▶ &SET_VAR A := 5
  &SET_VAR B := 6
  &SET_VAR A := %A%%B%
  &SET_VAR B := %A% + %B%
```

Since integers, in CPL, are actually character strings which evaluate to integer values, integers too can be concatenated. When the four commands in this example have been executed, the value of A is 56 and the value of B is 62.

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 which 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.

Note

The logical operators -- >, >=, =, <=, and < — perform string comparisons if either operand is a character string. 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

BARREL > APPLE

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

All variables shown so far have been local variables. Local variables:

- Are defined inside a running CPL program
- Are defined by either:
  - -- The &ARGS directive (shown in earlier examples)
  - — The &SET_VAR (or &S) directive (explained in this chapter)
  - — The SET_VAR command (explained in this chapter)
- Are known only to the program that creates them
- 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. Global variables are stored in one or more files inside your UFD (or inside a subdirectory). When you activate a global variable file, all the variables it contains can be used by you, interactively, for PRIMOS commands, by all your CPL programs, and by programs written in high-level languages. Global variables survive program termination and logouts. Once defined, they last until you delete them.

The PRIMOS commands governing 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, 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 providing easy communication of variable values among programs, 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. Attempts to use them for that purpose are not guaranteed to work.

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

    .SIZE, .UFD

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.) They are defined from high-level programs by the GV$SET routine, and referenced within high-level language programs by the GV$GET routine. These routines are described in Appendix E.

PRIMOS COMMANDS

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 form:

    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 define any global variables; and you must activate the global variable file before using the variables it contains.

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 the directory containing the global variable is protected by a password, then the user must provide the full pathname of the file within the DEFINE_GVAR command. For example:

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

Whenever the file is active, you may add to, delete, list, and make use of any variables it contains. If you reference a global variable in a CPL program without having defined a global variable file, the program aborts with an error message.

A user may create more than one global variable file, but may only have one global variable file active at any time. Therefore, the DEFINE_GVAR command activates the named file and turns off any global variable file already active. Logging out also deactivates an active global variable file. Global variable files may also be deactivated by the command:

```
DEFINE_GVAR -OFF
```

Pathnames cannot be used with this form of the command.

19.0

Global variable files may be deleted with the standard PRIMOS DELETE command. Make sure the file is inactive (using the command DEFGV -OFF, if necessary) before you delete it. (If you fail to do this, you will create a confusing situation in which you will be able to list variables from your deleted file, but will not be able to add or modify any variables.)

The SET_VAR Command

The SET_VAR command has the format:

```
SET_VAR name {:=} value
```

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

<u>value</u> can be:

- Any character string, up to 1024 characters long. Lowercase characters are not converted to uppercase. If the string contains special characters (as explained in Chapter 3), it must be enclosed in single quotes. The single quotes are included in the character count.

- A numeric character string representing an integer between the values of -2**31 + 1 to 2**31 - 1.

- A character string consisting of the logical value TRUE or FALSE (the forms TRUE, T, true, t, FALSE, F, false, and f are acceptable).

The assignment symbol (:=) is optional.
For example:

```
SET_VAR .A ALPHA
```

defines the global variable .A and assigns it the value ALPHA.

You can use the SET_VAR command interactively, at command level, to define global variables. Or, you may 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, we recommend that you use the SET_VAR command at command level only, and use the &SET_VAR directive inside CPL programs.

For example, a CPL program (FOO.CPL) might contain the following statements:

```
.
.
.
/*Set variable .ERR_REPORT to the null string
&SET_VAR .ERR_REPORT :=
RESUME BAR.CPL
   /* BAR may change value of .ERR_REPORT
   /* When BAR returns, FOO checks to see
   /* if value has been changed
&IF [NULL %.ERR_REPORT%] &THEN RESUME BAR2
   /*If .ERR_REPORT is still null,
   /*everything's OK, keep going
&ELSE &DO
/*Something went wrong;  send message and
/* halt execution
   TYPE Error reported by program BAR
   &RETURN &MESSAGE %.ERR_REPORT%
&END
.
.
.
```

The DELETE_VAR Command

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

```
DELETE_VAR id1 {...idn}
```

id1 through idn may be names of global variables, they may be wildcards, or they may be variable references or function calls which evaluate to the names of global variables. All variables in the list are deleted from the file. For example:

```
DEFINE_GVAR MY_VARS
DELETE_VAR .UFD
```

deletes the variable .UFD from the file MY_VARS. The command:

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

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

```
DELETE_VAR .AB@@
```

deletes all variables in the file that begin with the letters .AB.

The LIST_VAR Command

The command LIST_VAR lists some or all global variables contained in an active global variable file, with their values. Its form is:

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

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.

For example:

```
OK, list_var
.ERR_MESSAGE                        Sorry, try again!
.UFD                                alice
.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. For example:

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

5

Terminal Input and Output in CPL

OVERVIEW

Input

CPL provides three facilities for input from the terminal:

- The &TTY directive (or its extension, the &TTY_CONTINUE directive) is used within a &DATA group to allow the user to enter information interactively within a utility or a user program. 19.0

- The QUERY function, a logical (Boolean) function, prints a question at the user's terminal and accepts a YES or NO answer. The QUERY function interprets "YES" answers as TRUE and "NO" answers as "FALSE". If any other answer is given, it prompts "Please answer YES or NO".

- The RESPONSE function prints a request for information at the user's terminal. RESPONSE accepts any character string the user types in. The string is put in quotes, if it contains special characters, and is then returned as the value of the function: that is, the string replaces the function call.

The &TTY directive is discussed in Chapter 2. The &TTY_CONTINUE directive is discussed in the second section of this chapter, The Command Input Stream. The QUERY and RESPONSE directives are discussed in both the first and second sections of this chapter, Terminal Input and The Command Input Stream. The QUERY and RESPONSE functions are discussed in the next section of this chapter. 19.0

Output

Two facilities allow output to be printed at the terminal or into COMOUTPUT files:

- The PRIMOS TYPE command prints any message. This command can be placed anywhere within a CPL file.
- The &MESSAGE clause of the &RETURN and &STOP directives send a message when the CPL program "returns", or ends. It is particularly useful for announcing the success or failure of a program, or for warning a user that a command line has been entered incorrectly.

Examples of the TYPE command and the &MESSAGE clause in use are given in the last section of this chapter. (The &MESSAGE clause may also be used with the error-handling &STOP directive. See Chapter 15, ERROR AND CONDITION HANDLING, for details.)

TERMINAL INPUT

The QUERY Function

The form of the QUERY function is:

19.0 | [QUERY {text} {default} {-TTY}]

Example:

    [QUERY 'Et tu, Brute' TRUE]

When the QUERY function is encountered, CPL prints text on the user's terminal, follows it with a question mark, and then waits for the user to type an answer. The QUERY function returns either TRUE or FALSE, depending on the user's response, as described below.

Text: The text for this function may be any character string up to 1024 characters long. If text contains blanks, it must be placed inside single quotes.

If text is omitted, or is the null string, no prompt is printed. This option is provided for use when prompts or instructions are printed by some other means, such as the TYPE command or output from a user program.

Quoted Strings in Text: Variables and function calls are not evaluated inside quoted strings. If you write [QUERY 'SPOOL %FILE%'], the user will see:

```
SPOOL %FILE%?
```

at the terminal. Writing:

```
&SET_VAR T := 'SPOOL '%FILE%
[QUERY %T%]
```

lets you use the actual filename in the query.

Default: The default (if given) should be either TRUE, T, FALSE, or F (upper or lower case). If default is specified, then a null response to the query (that is, a carriage-return, or empty line), is taken as the function's default response. If no default is specified, a carriage-return is interpreted as FALSE. QUERY accepts YES, yes, Y, y, OK, and ok as TRUE answers. It accepts NO, no, N, and n 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. (For example, the function call shown above would abort any Batch or phantom program that contained it.)

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. Or, it can be a &DATA block inside another CPL program. Or, it can be a command input file.

19.0

These mechanisms are slightly more complex than those involved in simply going to the terminal for a response. Therefore, they are discussed in the second section of this chapter, after the discussion of the QUERY and RESPONSE functions themselves.

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" or "NO"?
```

For example, if the above program were named TEST.CPL, this might happen:

```
OK, R TEST BOOK
SPOOL FILE? SURE
Please answer "YES" or "NO"? Y
[SPOOL rev 18.0]
PRT011 spooled,  records:    1, name:BOOK
OK,
```

▶
```
&ARGS FILENAME
FTN %FILENAME% -64V -L %FILENAME%.LIST
&IF [QUERY 'SPOOL LISTING FILE' TRUE] ~
   &THEN SPOOL %FILENAME%.LIST
&RETURN
```

Again, the user chooses whether or not a file will be spooled. This time, however, default has been given as TRUE. Therefore, a carriage-return as answer will spool the file.

```
OK, R FTN_TEST THISFILE
0000 ERRORS [<.MAIN.>FTN-REV18.0]
SPOOL LISTING FILE?
[SPOOL rev 18.0]
PRT008 spooled,  records:    1, name:THISFILE.LIST
OK,
```

The RESPONSE Function

The form of the RESPONSE function parallels the form of the QUERY function:

19.0|
```
[RESPONSE {text} {default} {-TTY}]
```

This function returns the text string typed by the user (up to 1024 characters).

Text, again, is a character string of up to 1024 characters, quoted if it contains blanks. The text, followed by a colon, appears at the user's terminal. Default, if given, is another character string. If it contains blanks, it too must be quoted. If no default is specified, the default answer (produced by a carriage-return) is the null string.

For example:

```
&ARGS BOOK
&IF [NULL %BOOK%]~
     &THEN &SET_VAR BOOK := [RESPONSE 'Which Book']
```

This example tests for a null argument. If it finds one, it asks the user explicitly for the argument, then uses the &SET_VAR directive to give the variable its correct value.

If text and default are omitted, or if text is the null string (''), no prompt is printed.

The -TTY option for the RESPONSE function is identical to that for the QUERY function.

The Command Input Stream

As stated earlier, the &TTY directive, and the QUERY and RESPONSE functions used with the -TTY option, all insist on input from the terminal. CPL programs employing these statements canot be invoked as phantoms or Batch jobs; the request 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 &DATA group in a CPL program
- a COMINPUT file

19.0

If the CPL program demanding the input was invoked from the terminal, it takes its input from the terminal. If it was invoked from a command input file, it seeks its input there. If it was invoked by a &DATA directive, it gets its input from the &DATA group.

Here is a sample CPL program containing 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 might look like this:

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

Invoking the Program From a COMINPUT File

LENGTHEN_FILE can also be invoked from a command input file such as this one:

```
R LENGTHEN_FILE.CPL
Add this line
And this one
And this one.
;
FILE
CO -TTY
```

19.0

The first line of this file invokes the CPL program shown above. The second, third, and fourth lines contain input to be added to TESTFILE. The fifth line returns to EDIT mode, and the sixth line files TESTFILE and returns from the EDITOR.

At this point, control returns to LENGTHEN_FILE, which in turn returns to its caller, the command input file, which then returns to the terminal. A terminal session which ran the COMINPUT file might look 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
```

Invoking the Program From a &DATA Group

If a CPL program were to invoke LENGTHEN_FILE, it would do it like this:

```
&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; the next three lengthen it; and the fourth and fifth close the file, put it away, and leave the EDITOR.

A terminal session might look like this:

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

19.0

How Errors Are Handled

What would happen if the programmer forgot the semicolon or FILE statment in the CPL or COMINPUT file? The COMINPUT program would add every line in its file (including the CO -TTY which should terminate the file) to TESTFILE. Then it would return to the terminal with an error message and a request for input. The user would then have to leave the EDITOR interactively in order to return to PRIMOS command level. The sequence of events would look 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, would recognize that an error had occurred when it came to the &END statement in the &DATA group. It would simply terminate 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!
```

19.0

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.

TERMINAL OUTPUT

The TYPE Command

The PRIMOS TYPE command has the form:

```
TYPE text
```

text is a character string of up to 251 characters. When the TYPE command is executed, text is typed at the user's terminal.

Everything following "TYPE" is taken as text, so there is no need to quote strings. (TYPE does remove one set of quotes from around text before it prints it.) Since TYPE is an internal command, it can be used whenever a PRIMOS-level command can be used within a CPL file. Since text does not have to be quoted, it can contain variables and function calls.

For example, we might write a program, called ED_TEST, 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 might look like this:

```
OK, r ed_test 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 18.0]
PRT022 spooled,  records:    1, name:SAMPLE
Thank you.
Good-bye.
OK,
```

The &MESSAGE Clause

The &MESSAGE clause is used in the &RETURN directive to cause a CPL program to print a message and return to its caller. Thus, it is useful for announcing the success or failure of a program. Its form is:

```
&RETURN &MESSAGE text
```

text may be any character string up to 1024 characters. It does not need to be quoted if it contains blanks. For example:

```
&IF %LEFTOVERS% = 0 &THEN~
    &RETURN &MESSAGE It worked!
&ELSE~
    &RETURN &MESSAGE %LEFTOVERS% left undone.
```

The &MESSAGE clause can also be used to send PRIMOS-like messages warning users of the correct command line format for a CPL file. For example,

```
&ARGS UFD
&IF [NULL %UFD%] &THEN &RETURN &MESSAGE ~
     Usage:  R EXAMPLE UFD
```

6

Arguments With Type-checking and Default Values

INTRODUCTION

Previous chapters of this guide have included examples of programs that checked for the existence of needed arguments and took action if they did not find them.

The methods shown have included:

1. Setting up a default action (shown in Chapter 2)

```
&ARGS UFD
&IF [NULL %UFD%]~
    &THEN ATTACH MY_UFD
    &ELSE ATTACH %UFD%
```

2. Using CPL's RESPONSE function to demand the argument from the user (shown in Chapter 5)

```
&ARGS UFD
&IF [NULL %UFD%]~
    &THEN &SET_VAR UFD :=~
    [RESPONSE 'which UFD do you~
    want to attach to']
```

3. Using CPL's &RETURN &MESSAGE directive to terminate the CPL program and tell the user the appropriate command format (shown in Chapter 5)

```
&ARGS
&IF [NULL %UFD%] &THEN~
    &RETURN &MESSAGE~
    USAGE: R EXAMPLE ufd-name
```

This chapter introduces:

- A method of establishing default values for arguments within the &ARGS directive itself. With this method, each argument omitted from the command line is automatically assigned its designated default value, rather than being set to the system default.
- A method for setting a type specification (character string, integer, etc.) for each argument in the &ARGS directive. When this is done, each argument given in the command line is checked against the specified type. If the types do not match, the CPL program terminates with an explanatory error message.
- A special type of argument, REST.

TYPE CHECKING AND DEFAULT SPECIFICATION

The form of the &ARGS directive that provides type checking and default specification is:

```
&ARGS name-1 : type-1 = default-1{;...name-N : type-N = default-N}
```

Either type or default (or both) may be omitted for any name. If default is omitted, the equals sign that precedes it is also omitted. The colon that follows name is omitted only when both type and default are omitted. Spaces may precede or follow the equals sign, colon, and semicolon. They are not required.

Default may 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. Table 6-1 shows all types and their defaults.

Examples

▶ &ARGS FILENAME

Filename is established as a variable of type CHAR (i.e., character string). Its default is the null string ('').

▶ &ARGS UFD:TREE=MY_UFD

The variable UFD must be a valid treename (that is, a pathname or directory name). Its default value is MY_UFD.

Table 6-1
CPL Argument Types

Argument Type	Explanation	CPL Default Value
CHAR	Any character string up to 1024 characters long, mapped to upper case (default)	''
CHARL	Any character string up to 1024 characters long, no case shifting	''
TREE	A filename, directory name, or pathname, up to 128 characters long. The last element of the pathname (that is, the final file or directory name) may contain wildcard characters. (A)	''
ENTRY	A filename up to 32 characters long; may contain wildcard characters. (A)	''
DEC	A decimal integer (B)	0
OCT	An octal integer (B)	0
HEX	A hexadecimal integer (B)	0
PTR	Pointer; a virtual address in the format "octal/octal" (segno/wordno) (C)	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 tokens not accounted for in the &ARGS picture. (Unclaimed arguments are discussed in Chapter 13).	''

(A) See Chapter 7 for explanation of wildcard characters.
(B) Numeric arguments must be within the range -2**31+1...2**31-1.
(C) User specified default values are not supported for this datatype.

▶ &ARGS NAME:=XXXXX; NUMBER:DEC

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

▶ &ARGS UFD:TREE=%.UFD%

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

```
OK, r xl

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

SOURCE: &args ufd :tree =

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

▶ &ARGS HEX:HEX = 4AB

This directive declares one variable named HEX, giving it a default value of 4AB (1195 decimal). The argument HEX will 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, 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 would interpret the decimal number 20 as hexadecimal 20 (decimal 32).

▶ &ARGS EIGHT_BALL:OCT

This directive defines an octal variable named EIGHT_BALL. Octal numbers can contain only the digits 0-7; therefore, a value for EIGHTBALL containing any other digits or characters will be rejected with the message:

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

The arguments discussed in this chapter are position dependent arguments. The first value found on the command line is assigned to name-1, the second to name-2 and so on. (For position independent arguments in CPL, see the discussion of Option Arguments in Chapter 13.)

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 arguments given to the variable-names 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 no default has been specified, 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.

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 prints an explanatory message and returns the user to command level with an ER! prompt. For example:

   ```
   OK, R EXAMPLE 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).

Example

Assume that X.CPL contains the directive:

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

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

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

USING REST ARGUMENTS

REST is a special argument type that allows the remainder of a command line (after any other arguments have been read) to be passed as is to a single variable without quoting.

REST arguments are designed for passing PRIMOS option arguments as positional arguments to CPL programs, without having to quote 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
```

The first argument on the command line must be a filename (or pathname). Everything that follows the filename becomes the value of OTHER_ARGS.

A Sample Program

A sample program using the directive shown above might spool a file on a particular printer, giving the user the choice of specifying additional options at run time:

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

Here are some sample terminal sessions:

```
OK, R SPL X1.CPL
[SPOOL rev 18.0]
PRT003 spooled,  records:    1, name:X1.CPL
OK, R SPL X1.CPL -FORM NOW -LIST
[SPOOL rev 18.0]
PRT004 spooled,  records:    1, name:X1.CPL

 user  prt  time  name        size  opts/#   form  defer  at: CAROUSEL
------ --- -----  ----------  ----  ------  ------ -----  ------------
SMITH  001  0:20  MEMO.41       14
JONES  002  2:33  CL-DEPT.O      6                          QA.TST
BROWN  003 13:22  X1.CPL         1
BROWN  004 13:23  X1.CPL         1           NOW
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 terminal session with this program might look like this:

```
OK, R SPL2 X1.CPL
[SPOOL rev 18.0]
PRT003 spooled,  records:    1, name:X1.CPL

 user  prt  time  name          size  opts/#   form  defer  at: CAROUSEL
------ --- -----  ------------  -----  ------  ------ -----  ------------
SMITH  001  0:20  MEMO.41         14
JONES  002  2:33  CL-DEPT.O        6                           QA.TST
BROWN  003 13:27  X1.CPL           1
OK,
```

7
Processing Groups of Files

GROUPING FILES AND DIRECTORIES

CPL and PRIMOS have several methods for providing easy access to groups of files and directories:

- Prime's file-naming conventions help you set up your directory so that you can see easily what types of files it contains.

- Prime's wildcard facility lets you access groups of similarly named files (or directories) within a directory.

- CPL functions and loops take advantage of wildcards and naming conventions to let you perform operations on specified groups of files or directories.

This chapter explains each of these topics, in turn.

FILENAME CONVENTIONS

Prime's filename conventions use suffixes to identify various sorts of files. Using these conventions, a filename is divided into two components: the base name and the suffix. A dot separates the components.

(There may be any number of components in a filename, separated by dots. However, only the final component is considered to be the suffix. Names with more than three components are not recommended.)

USING SUFFIXES: THE BEFORE AND AFTER FUNCTIONS

CPL's BEFORE and AFTER functions make it easy to break a filename into its separate components. Thus, the name of a source file can be separated into "filename" and "compiler name", dropping the dot in the process.

The BEFORE Function

The form of the BEFORE function is:

```
[BEFORE string-1 string-2]
```

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

```
[BEFORE ABCD C]
```

returns

```
AB
```

Hence

```
&S FILE := [BEFORE SOURCE.PL1G   .]
```

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   .]
```

returns

```
SOURCE
```

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

The AFTER Function

The form of the AFTER function is

```
[AFTER string-1 string-2]
```

The AFTER function returns as its value that portion of string-1 which occurs after string-2.

For example,

```
[AFTER ABCD C]
```

returns

```
D
```

Hence,

```
&S COMPILER := [AFTER SOURCE.PL1G   .]
```

sets the value of COMPILER to PL1G.

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, loads, and runs any 64-V mode program, using the filename as its argument.

(This program, named CLR_ALL, is a revision of the "compile, load, and run" program shown in Chapter 2.)

```
/* CPL program to compile, load and
/* execute any -64V mode program
/* 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%
/*
   &DATA SEG -LOAD              /* SEG names output file source.SEG
    LOAD %SOURCE%               /* SEG finds file source.BIN
    LI VCOBLB
    LI PL1GLB
    LI VFORMS
```

```
        LI VAPPLB
        LI VSSRTLI
        LI
        SA
        QU
       &END
    /*
       SEG %source%                    /* execute runfile
    &RETURN
```

WILDCARDS

Wildcards allow you to specify groups of files using a single wildcard name. A wildcard name is a file or directory 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.

Some Examples

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

```
FOO.COBOL      BARR1.COBOL      BARR1.SEG
BARR2.COBOL    BARR2.SEG        FOO.SEG
CLR.CPL        EDD.CPL          SCROLL
EDD.COMO       EDD.COMO.OLD
```

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

```
FOO.COBOL      FOO.SEG
```

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

```
BARR1.SEG      BARR2.SEG        FOO.SEG
```

The wildcard name BARR+.COBOL matches

```
BARR1.COBOL    BARR2.COBOL
```

The wildcard name BARR+.@ matches

```
BARR1.COBOL    BARR2.COBOL
BARR1.SEG      BARR2.SEG
```

The wildcard name EDD.@ matches:

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

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 <u>don't</u> match the rest of the wildcard name.

It 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.COBOL	BARR1.COBOL	BARR2.COBOL
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.COBOL	BARR1.COBOL	BARR1.SEG
FOO.SEG	BARR2.COBOL	BARR2.SEG
SCROLL	EDD.COMO	EDD.COMO.OLD

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

FOO.COBOL	BARR1.COBOL	BARR1.SEG
BARR2.COBOL	BARR2.SEG	FOO.SEG
CLR.CPL	EDD.CPL	SCROLL
EDD.COMO	EDD.COMO.OLD	

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 option, 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 which aborts the CPL program. Since the -SINGLE option only returns one name at a time, it can handle cases which would produce over-long lists.

Basic Format of the WILD Function

The basic format of the WILD function is:

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

wild-name-1 through wild-name-n are wildcard names which the WILD function will match. If wild-name-1 is a pathname, all the wild-names are looked for in the directory that wild-name-1 specifies. Otherwise, all names are searched for in the current directory. (Wild-name-2 through wild-name-n may *not* be pathnames.) For example:

```
ATTACH MYUFD
&SET_VAR SOURCES := [WILD @.COBOL @.PMA]
```

creates a list of all COBOL and PMA source files in MYUFD, and stores the list in the variable, SOURCES.

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

creates and stores a list of all COBOL and PMA source files in the directory SMITH.

options represent one or more optional option arguments. These place limits on the objects matched by the specified wildcard names. options are as follows:

Option	Meaning
-ACL	Select only ACLs. (19.0)
-AFTER date	Select only objects created or modified after the date specified by date. (This information is stored as the file's DTM, "date and time modified." Its format is mo/da/yr.)
-BEFORE date	Select only objects created or last modified before the specified date.
-DIRECTORY	Select only directories.
-FILE	Select only files. (19.0)
-SEGMENT_DIRECTORY	Select only segment directories.

Some examples using options are as follows:

▶ SET_VAR .OBJ := [WILD @@ -SEGDIR]

creates a list containing the names of all segment directories in the current directory. For example:

```
SET_VAR .OBJ := [WILD @@ -SEGDIR]
TYPE %.OBJ%
FOO.SEG  BAR.SEG
```

19.0

▶ SET_VAR .OBJ := [WILD @.PL1G -BF 05/30/80]

lists all PL/I, Subset G, source files created or last modified before May 30, 1980. For example:

```
SET_VAR .OBJ := [WILD @.PL1G -BF 05/30/80]
TYPE %.OBJ%
FOO.PL1G
```

▶ SET_VAR .OBJ := [WILD MYUFD>@@ -DIR]

lists all subdirectories in the UFD, MYUFD. For example:

```
SET_VAR .OBJ := [WILD MYUFD>@@ -DIR]
TYPE %.OBJ%
REPORTS MEMOS OTHER_STUFF
```

The -SINGLE Option

The -SINGLE option 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 might overrun its limit of 1024 characters or when it's more convenient to deal with the file names one at a time.

The format of this version of the WILD function is:

```
[WILD wild-path {wild-2 ...  wild-n} {options} -SINGLE unit-var]
```

options are the same as those for the plain WILD function.

unit-var is a variable-name in which WILD puts the number of the file unit it has used to open the directory to search for objects. unit-var must be set to zero before WILD is invoked, so that WILD can distinguish the first call (in which it opens the unit and returns the first matching name) from subsequent calls (in which it takes the next

name from the open 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 directive defines the variable UN and sets it to zero. The second causes CPL to:

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 would be that you want to do something with each of the files or directories on the list, for example, spool all your runoff files, obtain a listing of the contents of each of your subdirectories, update a group of reports or data files.

Tasks like these can be achieved easily by using the WILD function to control a loop, thus performing the desired process once for each item on the list.

CPL Loops

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. The &DO &LIST loop is used when the WILD function is used to get the entire list of file or directory names at one time. The &DO &ITEMS loop is used with WILD's -SINGLE option. An example of each of these types of loop is shown here, to demonstrate its use with the WILD function. Full explanations are given in Chapter 9.

Example of &DO &LIST Loop

The following program spools all runoff files (ending in .RUNO) in any directory specified by the user:

```
&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 had many runoff files in your directory, you could write the same program with a &DO &ITEMS loop, as follows:

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

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

8

Decision-making in CPL Programs

CONTROL DIRECTIVES

Among the most powerful features of CPL are its "flow of control" directives. These are the statements by which users specify what tests they want performed at run-time and what actions they want taken depending on the result of those tests. Table 8-1 shows the flow of control directives offered by CPL.

&IF...&THEN...&ELSE, &GOTO, and &DO groups are explained in Chapter 2. Nested &IF statements and &SELECT statements are discussed in this chapter. &DO loops are discussed in Chapter 9.

SINGLE &IF STATEMENTS

A single &IF...&THEN...&ELSE statement can choose between any two alternatives. For example:

```
&IF %A% > 10 &THEN SEG MYFILE
  &ELSE SEG HISFILE
```

Multiple expressions can be combined into a single test by the use of the logical AND (&) and inclusive OR (|). When logical AND is used, both expressions must be true for the test to be true. With logical OR, on the other hand, if either expression (or both) is true, the test is true.

Table 8-1
Flow-of-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 was 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 run-time. • _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.

For example:

```
&IF %A% > 10 & %B% > 10 ~
  &THEN SEG MYFILE
  &ELSE SEG HISFILE
```

In this example, MYFILE will not be executed unless both A and B have values that are greater than 10.

```
&IF %A% > 10 | %B% > 10 ~
  &THEN SEG MYFILE
  &ELSE SEG HISFILE
```

In this example, MYFILE will be executed if the value of either (or both) A or B exceeds 10.

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 runs the yearly W-2 program. 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 RUN THE 941 REPORT
&IF ( ( ( %DAY% = 31 ) ~
     & ( ( %MONTH% = March ) | ( %MONTH% = December ) ) ) ~
   | ( ( %DAY% = 30 ) ~
     & ( ( %MONTH% = June ) | ( %MONTH% = September ) ) ) ) ~
   &THEN SEG PGM941
/* IF THIS IS THE END OF THE YEAR, THEN RUN THE W-2 PROGRAM
&IF ( ( %DAY% = 31 ) & ( %MONTH% = December ) ) ~
   &THEN SEG W2FORM
/* ALWAYS RUN THE PAYROLL PROGRAM
SEG PAYROLL
```

NESTED &IF STATEMENTS

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

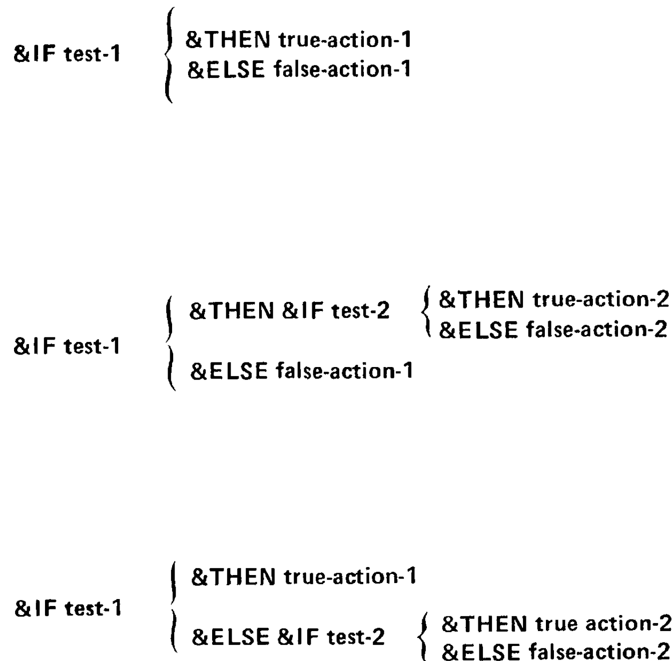
```
&IF %A% > 10 &THEN SEG MYFILE
  &ELSE &IF %A% = 10 &THEN SEG HISFILE
    &ELSE SEG HERFILE
```

In this example, MYFILE will be executed if the value of A is greater than 10; HISFILE will be executed if the value of A is equal to 10; and HERFILE will be executed if the value of A is less than 10. (Note that each &ELSE statement matches, or depends on, the &THEN statement immediately preceding it.) There is no limit to the number of &IF statements which 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
```

More Nested &IF Statements

A more complex form of nested &IF statement is one in which both &IF and &ELSE statements are nested. With this construction, the rule for matching &THEN and &ELSE statements is: An &ELSE statement matches the last &THEN statement preceding it that is not already matched by an &ELSE statement. Examples of such matching are shown in Figure 8-1.



Matching of &THEN and &ELSE Statements
Figure 8-1

Here is an example of this sort of construction:

```
&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 8-2. 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.

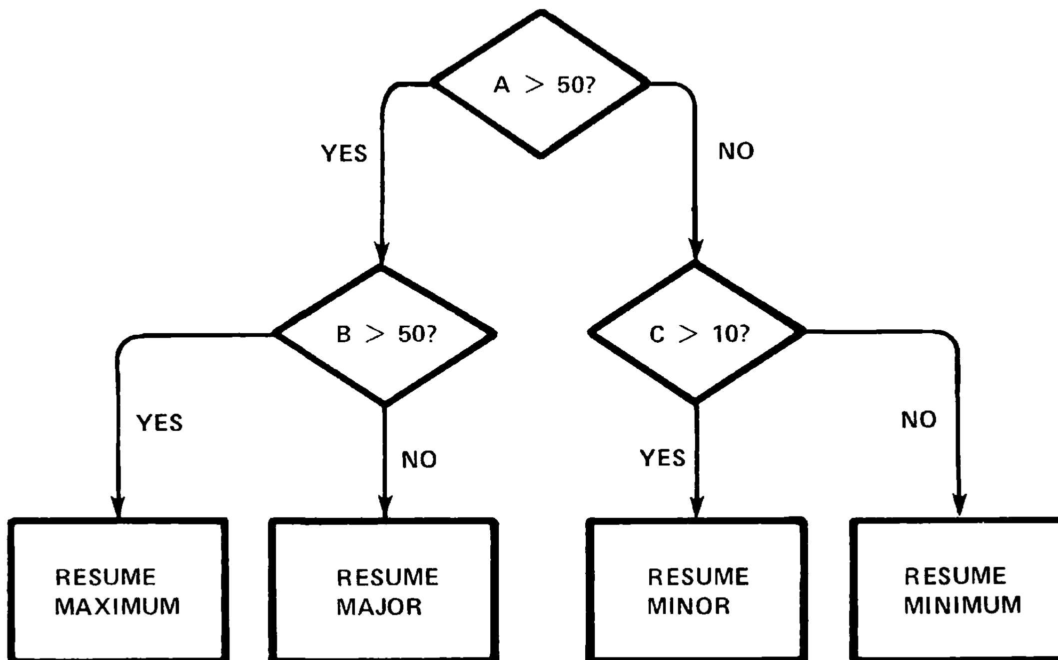


Diagram of Nested &IF Statement
Figure 8-2

THE &SELECT DIRECTIVE

Since CPL &IF directives can be nested, they can handle any situation in which you need to test at run time and then take action based on the result of that test. However, deeply nested &IF statements, as well as &IF statements containing logical OR's, are often difficult to read. When you want to choose between several alternatives, therefore, you may prefer to use the &SELECT directive to provide a neatly set out grouping of a test condition, its possible results, and the action to be taken in every case. Figures 8-3 and 8-4 compare the nested &IF statement and the &SELECT statement.

&SELECT Directive Format

The form of the &SELECT directive is as follows:

```
&SELECT key-expression
   &WHEN expression-A1  {,expression-A2...,expression-An}
         action-A
   &WHEN expression-B1  {,expression-B2...,expression-Bn}
         action-B
   .
   .
   .
   {&OTHERWISE
         action-n}
&END
```

key-expression may be a variable reference, a function call, or a string, arithmetic, or Boolean expression. For example:

```
&SELECT %COMPILER%
&SELECT %A% + %B%
&SELECT [DATE -DOW]
```

expressions-1 through n represent possible values of key-expression. In the example, "&SELECT %COMPILER%", all further expressions would represent possible values of the variable, %COMPILER%. In the example "&SELECT [DATE -DOW]", all further expressions would represent the possible results returned by the DATE function. In the example, "&SELECT %A% + %B%", all further expressions would be integers or arithmetic expressions representing possible values of the sum of the current values of A + B.

Action-A 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.

Actions Taken by the &SELECT Directive

When the CPL interpreter reads an &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 key-expression.

3. When it finds a match, it executes the statement immediately following that &WHEN directive.

4. As soon as it has found that first match and executed the accompanying statement, it drops to the &END statement that concludes the &SELECT group, and continues execution with the following statement.

5. If it finds no match, but does find an &OTHERWISE directive, it executes the statement immediately following the &OTHERWISE directive.

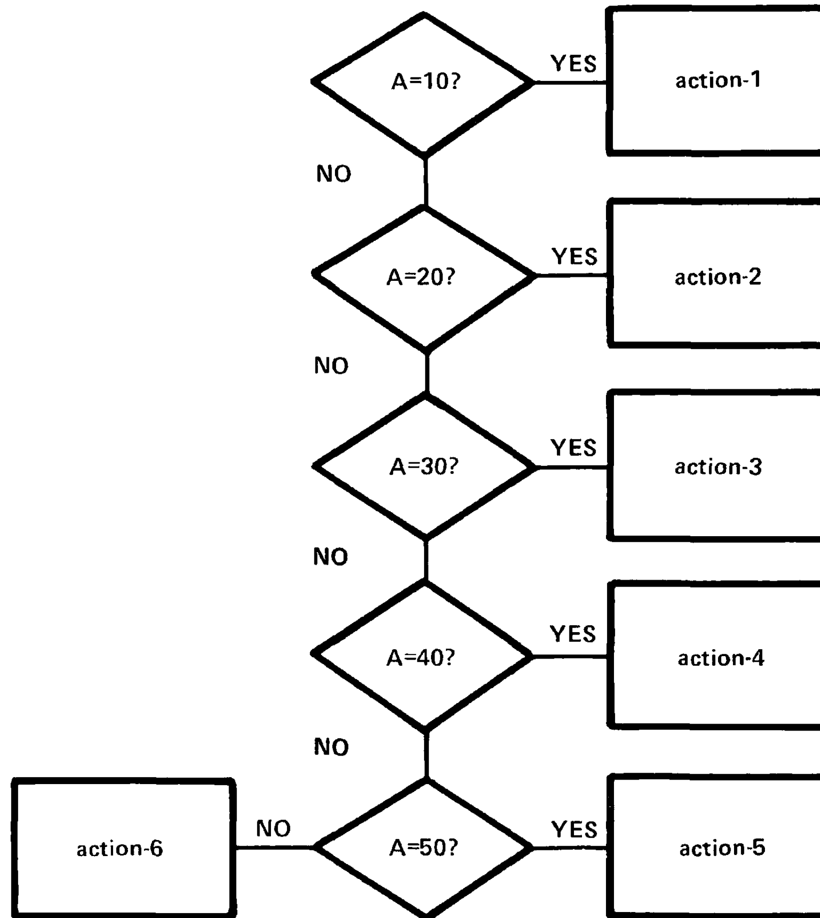
6. If it finds neither a match nor an &OTHERWISE directive, it executes none of the &SELECT group's statements, but continues reading the CPL file at the statement following the &SELECT group's &END statement.

Using Variable References

A variable reference used in a &SELECT statement evaluates to a single expression. For example, assume that variable A has the value "5, 10, 15", and that it is used in a &SELECT statement beginning:

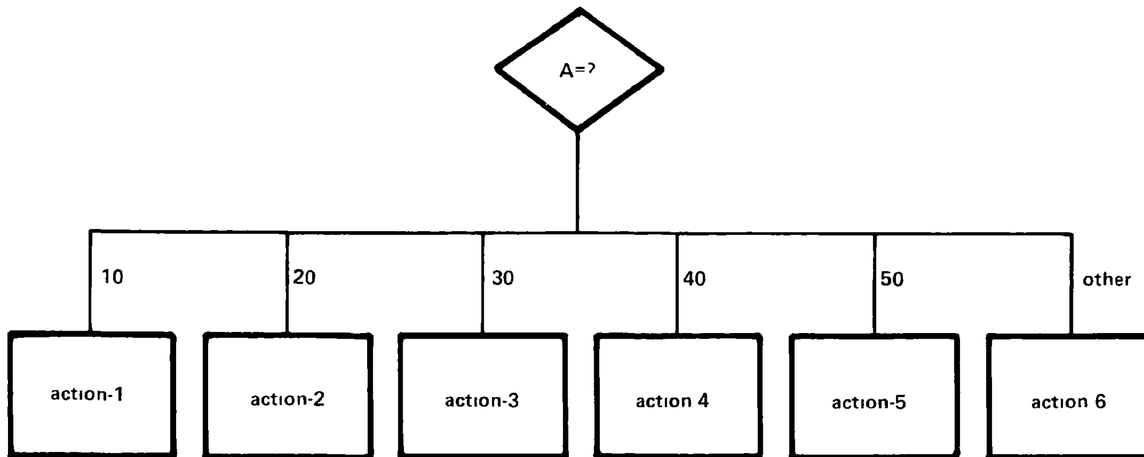
```
&SELECT %B%
&WHEN %A%, 20, 25
```

This statement tests the value of B three times: once against the 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 B is 20, the &WHEN test is TRUE; if the value of B is 10, the &WHEN test is FALSE.




```
&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
```

Nested &IF Statement
Figure 8-3




```
&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
```

The &SELECT Statement
Figure 8-4

&SELECT Examples

The first example demonstrates the use of multiple expressions in &WHEN statements. In this example, the &SELECT statement 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
```

A second example applies the &SELECT statement 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, 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 third example, EDDD.CPL, invokes the EDITOR and sets editor symbols in accordance with the language in which the source file is being written. It takes one argument, which may be

- The name of the file to be edited
- The name of a compiler or language: COBOL, CPL, FTN, F77, PL1G, or RPG

If the argument is supplied, EDDD.CPL decides whether the user wants to edit an existing or create a new file in the specified language. If the argument is omitted, EDDD.CPL assumes that an ASCII file is to be created (that is, a data file, a report or memo, etc.).

EDDD.CPL is a variant of EDD.CPL, which was shown in Chapter 2.

```
/* EDDD.CPL is a fancier variant of EDD.CPL
/*
&args name
&set_var empty_line :=
&select %name%        /* Is name a filename or a compiler name
    &when FTN, F77, PL1G, COBOL, CPL, RPG
    &do
         &s language := %name%
         &s name :=
         &s input_mode := true
         &end
    &otherwise
    &do
         &if [null %name%]  &then &do
               &s input_mode := true
               &s language :=  ASCII
               &end
           &else &s input_mode := false
           &end
     &end        /* end select
/*
/* If we've got a genuine filename, check for compiler suffix
/*
&if ^  %input_mode%  &then ~
   &if [index %name% .] ^= 0 &then ~
     &s language := [after %name% .]
     &else &s language := ASCII
/*
/* enter editor
/*
&data ed %name%
   &if %input_mode%  &then %empty_line%
   &select  %language%
         &when FTN, F77
               TABSET 7 45
         &when PL1G, CPL
         &DO
               TABSET 3 6 9 12 15 18 21 24
               SYMBOL SEMICO }
               &END
         &when COBOL
         &DO
               MODE COLUMN
               TABSET 7 45
               &END
        &when RPG
               MODE COLUMN
        &otherwise      /* set characteristics for report writing
            &DO
               SYMBOL SEMICO }
               TABSET 5 10 15 20 25 30
               &END
```

```
        &END   /*  end &select
    &IF %input_mode%  &THEN %empty_line%  /* return to input mode
    &TTY   /*  give user control
    &END   /* end &data group
    &RETURN
```

Some sample sessions using this program might be:

```
/* This example sets Editor characteristics for standard report generation.
OK, R EDDD
INPUT

EDIT
              SYMBOL SEMICO }
              TABSET 5 10 15 20 25 30

INPUT
type in whatever you want

EDIT
file it
OK,
```

```
/*  This example sets Editor characteristics for an RPG program.
OK, R EDDD RPG
INPUT

EDIT
              MODE COLUMN

INPUT
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890123456789

EDIT
Q
OK,
```

```
/*  This example sets Editor characteristics for a PL1G program.
OK, R EDDD SOMETHING.PL1G
EDIT
              TABSET 3 6 9 12 15 18 21 24
              SYMBOL SEMICO }
p3
.NULL.
/* this is a sample PL/I program
DCL A FIXED BIN
q
OK,
```

The final example uses a &SELECT group inside a MAGSAV routine, selecting on the DATE function's DAY-OF-WEEK option to pick the directories to be backed up on tape.

```
/*
/*Assign a tape drive and
/*have operator mount a tape
/*
ASSIGN MTX -ALIAS MT0 -TPID BACKUP.NEW -RINGON -1600BPI
&DATA MAGSAV
0             /*Response to "Tape Unit" prompt
1             /*Response to "Enter logical tape number"
BACKUP.[DATE -TAG]  /*Response to "Tape" prompt
[DATE -USA]         /*Response to "Date" prompt
0             /*Response to "rev no:" prompt
    /*&SELECT is now used to respond to
    /*"NAME OR COMMAND" prompts
&SELECT [DATE -DOW]
   &WHEN Monday, Wednesday    /*DATE returns day of week
     &DO                      /*in upper and lower case format
       UFD1                   /*Back up these UFD's
       UFD2>SUBUFD1           /*on Monday and
       UFD3                   /*Wednesday
     &END
   &WHEN Tuesday, Thursday
     &DO
       UFD2>SUBUFD2           /*Back up these UFD's
       UFD4                   /*on Tuesday and Thursday
       UFD5
     &OTHERWISE
       MFD                    /*Back up the whole MFD on Friday
&END          /*End &SELECT
/*Now tell MAGSAV to finish tape, rewind,
/*and return to PRIMOS
$R
&END          /*End &DATA group
/*Unassign the tape drive
  UNASSIGN -ALIAS MT0
  &RETURN
```

9

Loops in CPL

USING LOOPS

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" 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" loop combined with a "while" or "until" test: for example, &DO I := 1 &TO 100 &WHILE %J% > 20

- The "&REPEAT" loop, which is usually combined with a "while" or "until" test:
  for example, &DO I := 50 &REPEAT %I% * %I% &WHILE %I% <= 100000

- The "&LIST" loop
  for example, &DO I &LIST %var_list%
  or &DO I &LIST 5 36 489

- The "&ITEMS" loop, a variant of the "&LIST" loop:
  for example, &DO I &ITEMS [WILD @.FTN -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 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 prints 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 a loop is encountered in the execution of a CPL program, the following actions occur. (Figure 9-1 contains the corresponding flow chart.)

1. If index-var is present, it is set to its initial value. The value is tested for loop completion.

2. If a &WHILE clause is present, it is tested for loop completion.

3. If the loop has not yet been completed, statement-1 through statement-n are executed.

4. When the &END statement that closes the loop is reached, the &UNTIL clause (if there is one) is tested. 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.

6. index-var is set to its next value.

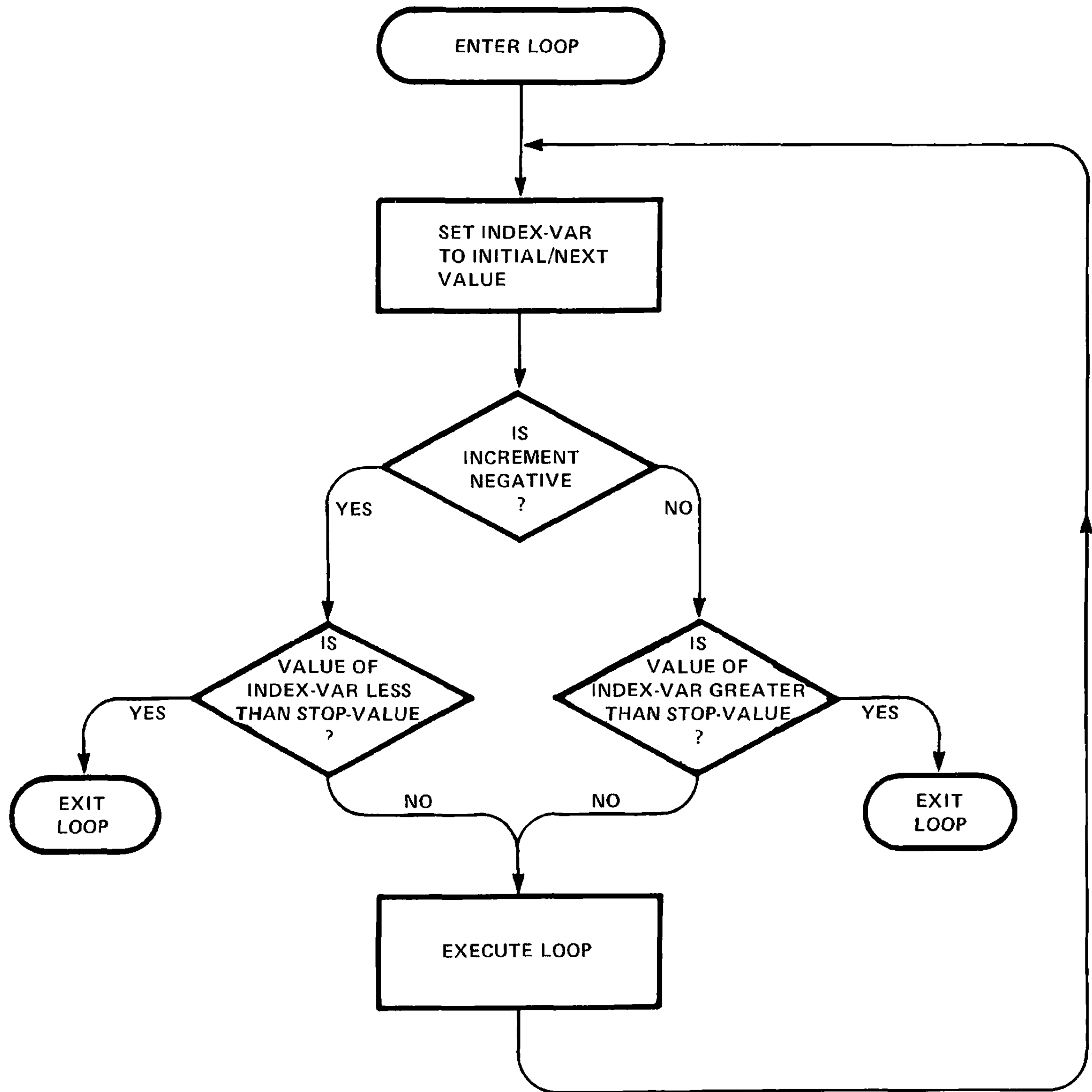
7. Tests for index-var and/or &WHILE clauses are made.

8. If tests are not TRUE, statement-1 through statement-n are executed again.

9. 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 hits 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 will be the first out-of-range value reached. For example, if a loop said "&DO I := 1 &TO 10", the value of index-var at termination would be 11. When &DO &LIST and &DO &ITEMS loops terminate, their index-vars are null.

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



Flow of Control in CPL Loops
Figure 9-1

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 will 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.

Nested Loops

Loops in CPL may be nested: that is, one loop may be written inside another. A trivial example, called NEST.CPL, is:

```
&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's 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
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.

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

Legal Use of &GOTO
Figure 9-2

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

Illegal Use of &GOTO
Figure 9-3

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 a &BY clause is included, increment must also evaluate to an integer. If a &BY clause is not included, increment defaults to 1. Negative increments or limits may be used: for example, &DO I := 10 &TO -10 &BY -1.

Execution of Counted Loops

When a counted loop executes, index-var is set to start-value. start-value is tested to see that it is less than or equal to stop-value. If it is, the loop executes. When control returns to the top of the loop, the value of index-var is incremented by increment, and re-tested. When the value of index-var exceeds stop-value, execution passes to the statement following 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: if the initial value of index-var is out of range, the loop is not executed.

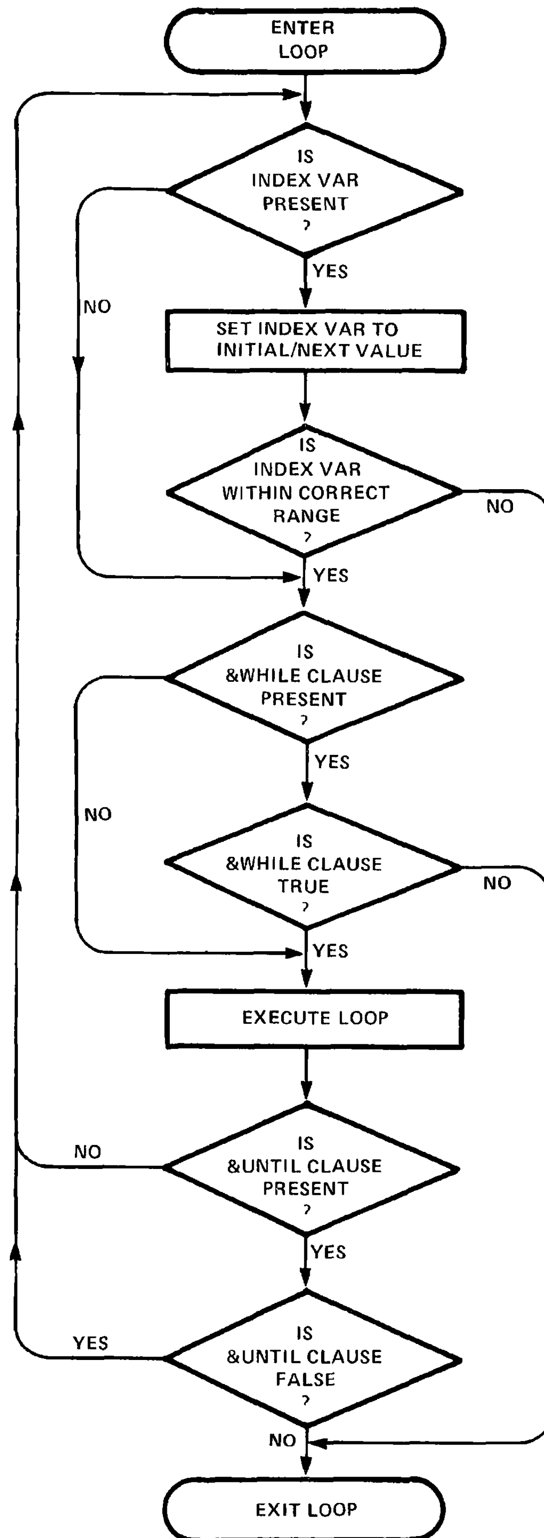
An example of a counted loop might be:

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

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

Omitted &TO and &BY Clauses

If you omit the &BY clause in a counted loop, it defaults to "&BY 1". If you omit the &TO clause, index-var has no stop-value, but may be incremented 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.



Action of Counted &DO Loop
Figure 9-4

The following can be used 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 to a point outside the loop

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

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

would initiate such a loop. More efficient code to do the same thing would be:

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

&DO &WHILE LOOPS

The form of the &DO &WHILE statement is:

```
&DO &WHILE test
```

test can be any expression which evaluates to TRUE or FALSE. A TRUE result of the text allows the loop to execute. A FALSE result precludes execution.

Some example 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 will not execute at all if the first test shows the loop to have completed.

Since &DO &WHILE loops are tested at the top of the loop, they require that any variable they test have some value assigned to it before or during the execution of the &DO statement. In the examples above, %A%, %B% and %STRING% must have been assigned some values before the &DO statement is executed.

Here is an example of a &DO &WHILE loop. This 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 form of the &DO &UNTIL loop is:

```
&DO &UNTIL test
```

For example:

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

test is any expression which 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 will always execute at least one time. A trivial example follows.

```
&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 printing 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

"Counted" loops, &DO &WHILE tests, and &DO &UNTIL tests may all be combined. Some possible combinations 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
```

These loops execute until any one of their tests signals completion. See Figure 9-1 for the test points they can contain.

&REPEAT LOOPS

The form of the &REPEAT loop is:

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

index-var is any valid variable name. start-value may be any string or arithmetic expression. Expression is another string or arithmetic expression which tells 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

> If no &WHILE or &UNTIL clause is used, &REPEAT loops are "infinite loops"; that is, they have 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 form of the &DO &LIST loop is:

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

index-var is any valid variable name. list-of-items can be a list of items separated by blanks, a variable reference, or a function call. (The variable reference or function call may itself evaluate to a list of items.) The maximum length of list-of-items is 1024 characters. At

each iteration of the loop, index-var is set 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.

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

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

```
&DO WORD &LIST %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 were "How now, brown cow?", then the loop would execute four times, with word set to "How", "now,", "brown" and "cow?". A quoted string is a single item. If line_of_type were 'How now, brown cow', the loop would execute once, with word set to 'How now, brown cow'.

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

The &LIST loop can also be used with the WILD function: for example,

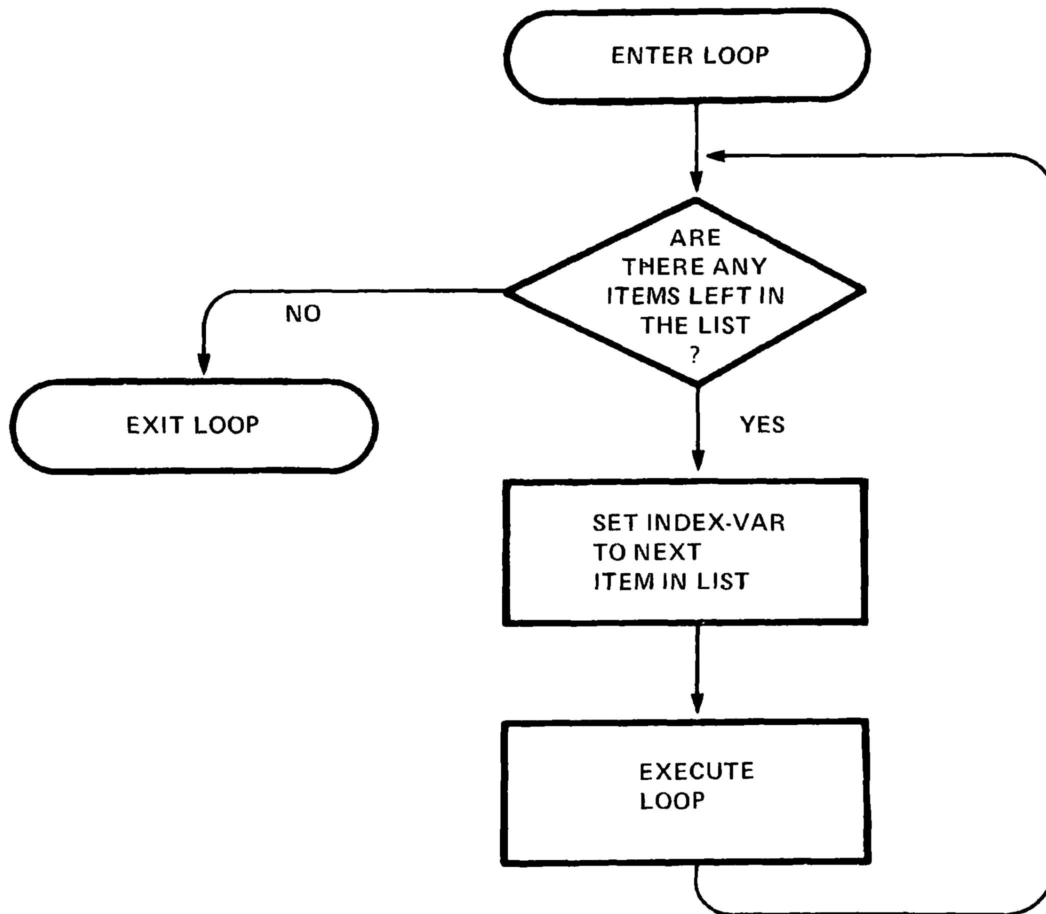
```
&DO I &LIST [WILD @.COBOL]
   COBOL %I%
 &END
```

This loop compiles all COBOL 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.

A Nested Example

This module uses nested loops to spool every report in every top-level sub-UFD belonging to UFD SALES. The outer loop attaches to each sub-UFD in turn. The inner loop finds the files in that sub-UFD that end in "_REPORT", and spools them.

```
&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                                /* Attach back to UFD SALES
```



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

Figure 9-5

&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 which is evaluated at each iteration. Usually, expression is the WILD function with the -SINGLE option, returning one filename per iteration.

The form 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.

An example of a &DO &ITEMS loop is:

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

This example compiles all COBOL and FORTRAN 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, and resets unit to identify the unit it's using. (It uses the decimal number of the file unit.)

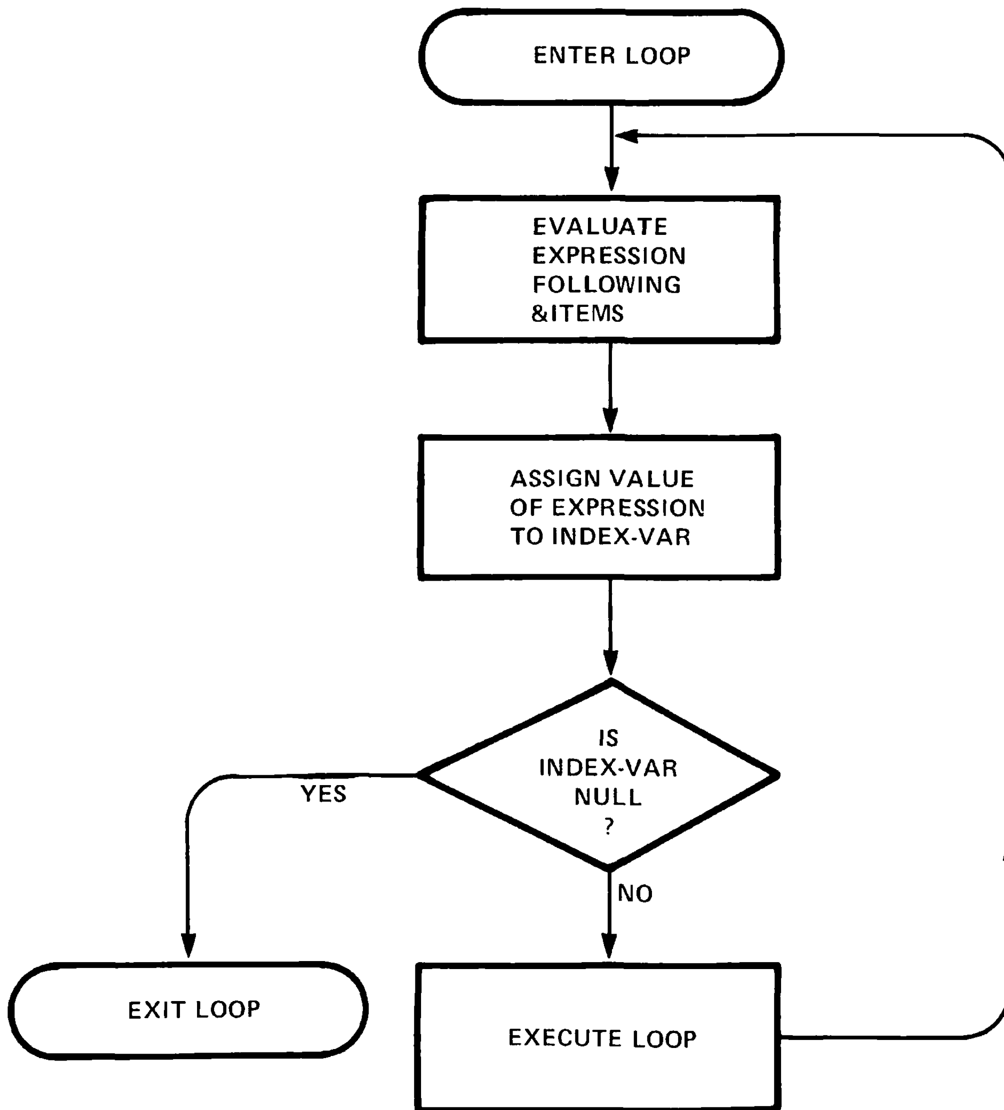
3. Since the option -SINGLE has been given, the WILD function finds the first matching file, and returns that filename as its value.

4. The &DO processor assigns the value of the WILD function to I.

5. The loop executes.

6. When the loop returns to the &DO statement, the WILD function is re-invoked. It reads the open unit number from unit, goes to that unit, and selects the next matching file.

7. The loop executes again.



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

Figure 9-6

8. When the WILD function finds no matching file, it returns a string of length zero and closes the file unit it was using. I is then set to the null string, and the loop terminates immediately.

This loop is equivalent to the following &REPEAT loop:

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

Loops That 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:

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 in CPL

ENCOUNTERING ERRORS

CPL programs may encounter two types of errors:

- The commands executed by the CPL program may produce run-time errors: for example, a command may try to open a file that does not exist.
- The CPL directives themselves may be written incorrectly: for example, the word &THEN may have been omitted from an &IF statement.

CPL offers several levels of control in dealing with run-time errors. The simplest method is shown in the second half of this chapter. More advanced methods are shown in Chapter 15.

There is only one way to deal with CPL errors: that is, to debug the program. CPL provides three useful tools for debugging: no-execute mode, echoing, and variable watching. These are explained in the first half of this chapter.

DEBUGGING CPL PROGRAMS

Debugging is enabled and disabled by 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 one CPL program RESUMES 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.

&NO_EXECUTE/&EXECUTE

This pair of options determines whether or not commands will be executed when the CPL program is run. Specifying &DEBUG &NO_EXECUTE (or saying simply "&DEBUG"), allows you to run through, or "rehearse", a CPL program. When you give the RESUME command for a program which 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 you need to get rid of syntax errors before performing any of the commands the file contains. It is especially useful for the CPL programs which:

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

&EXECUTE allows the execution of PRIMOS commands.

If neither &EXECUTE nor &NO_EXECUTE is specified, the default is &EXECUTE.

Table 10-1
&DEBUG OPTIONS

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

&ECHO/&NO_ECHO

&ECHO and &NO_ECHO control the echoing of commands and directives. If neither is specified, default is &NO_ECHO.

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, resuming this CPL program:

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

produces this terminal session:

```
OK, R EX
&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 our sample program read "&DEBUG &ECHO COM", its execution would look like this:

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

If &ECHO ALL (or simply &ECHO) is given, commands and directives are both echoed. If our sample program said "&DEBUG &ECHO", a terminal session would look like this:

```
OK, R EX
&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. 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.

&WATCH/&NO WATCH

The &WATCH directive lets you trace the values of up to 16 local and/or global variables, watching whatever changes are made by the &SET_VAR directive. (This includes changes made by the CPL interpreter itself, such as those which 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.) For example, this trivial program:

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

produces the following result:

```
OK, R EX2
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 the loop's index is shown as being set to its first value at the top of the loop, but as being incremented at the &END statement each time thereafter.

ERROR HANDLING

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 {&WARNING | &ERROR} &IGNORE	Ignore all error codes, continue execution.
&SEVERITY &WARNING &FAIL	Halt execution if any warning or error is received.
&SEVERITY &ERROR &FAIL	Ignore warnings (code < 0), halt execution for errors (code > 0). (Default)
&SEVERITY	(equal to &SEVERITY &WARNING &IGNORE)
&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.

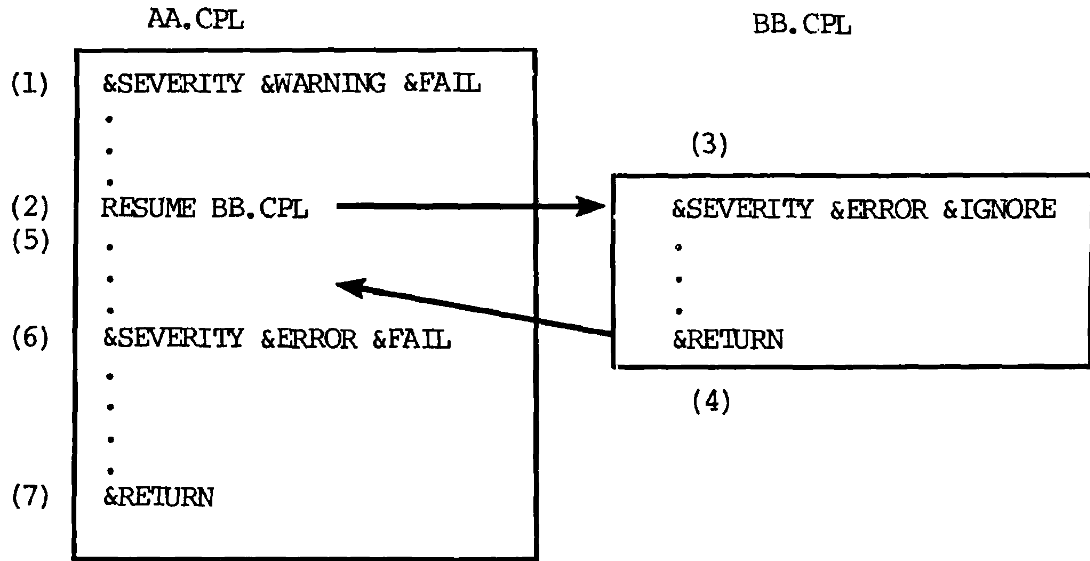
&SEVERITY directives may be placed anywhere in a CPL program. They become effective when execution of the program reaches the line in which they occur, and they remain effective until either

- The program terminates
- A new &SEVERITY directive is encountered

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 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.)

Scope of &SEVERITY Directive
Figure 10-1

PART III
Full CPL

11

Expression Evaluation in CPL

INTRODUCTION

This chapter provides a detailed explanation of how CPL handles character strings and arithmetic expressions. The first half of the chapter discusses string handling, with particular reference to:

- 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 the RESCAN function may be used to force evaluation of a quoted string containing variables or function calls.
- How the &EXPAND directive may be used to allow commands in a CPL program to contain abbreviations from an ABBREV file.

The second half of the chapter discusses the evaluation of arithmetic expressions in CPL. It also explains the CALC function, with which users may force evaluation of an arithmetic expression within a command or a function call.

VARIABLES

A CPL variable name may be up to 32 characters in length. It may contain the letters, digits, "_", or ".". Names of local variables must start with a letter (to avoid confusion with numbers). Names of global variables must start with ".". "$" is reserved for 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".

The SET_VAR command is used to define global variables at command level. The &SET_VAR directive, which is both faster and more flexible, should be used for defining variables within a CPL program.

Local Variables

Local variables are defined only in the activation of the CPL program in which they are set; they are not defined in a recursive invocation of the same CPL program, nor in an invocation of another CPL program. If their values are needed in an invocation of a CPL program, local variables must be passed as arguments. Local variables are deleted when the program in which they are defined finishes. Local variables cannot be set outside a CPL invocation.

Global Variables

Globabl variables are distinguished from local variables by having names that start with a period. So,

```
&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 a user ran a CPL program which set a global variable, .A_GLOB_VAR, he could run another CPL program which referenced that variable. Furthermore, the names and values of global variables survive logout. When a user logs in, any global variables he defined in a previous session 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 a user intends to use global variables during a terminal session, he must use a DEFINE_GVAR command before the first global variable reference.

Global variables are deleted using the PRIMOS command DELETE_VAR:

```
DELETE_VAR id1 id2 ... idn
```

Each id is an expression which must evaluate to a variable name; these variables are deleted.

The PRIMOS command LIST_VAR {wild_name1, ..., wild_nameN} lists global variables and their values at the user's terminal. If no wildcard names are given, all variables are listed; if present, only those matching the given wildcard names are listed.

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:

```
FTN %PATHNAME%.FTN -LIST %PATHNAME%.LIST -BIN %PATHNAME%.BIN -DYNM
```

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:

```
FTN HOBBIT -LIST HOBBIT.LIST -BIN HOBBIT.BIN -DYNM
```

When a statement contains variable references, all references are replaced by the their values before the statement is executed. Variable evaluation is performed only once per statement. If variable var has the value "%XXX%", then when var is evaluated the reference is replaced by "%XXX%" and that string remains in the text. It is not reevaluated as a variable reference.

Variable references are not evaluated inside quotes.

FUNCTIONS

Functions are procedures which 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 will return that part of the value of PL1_PROG that occurs before the first occurrence of ".PL1". 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.

Variable references are evaluated prior to function calls; this is illustrated by the example just given; the variable reference "%PL1_PROG%" is replaced by "RICHS>EVAL.PL1" before the call to the function is evaluated.

Function evaluation is done recursively; any or all of function-name or argi may themselves contain function calls. Innermost calls are done first. There is no implementation restriction on the depth of nesting.

Function calls are not evaluated inside quotes.

QUOTED STRINGS

CPL uses the single quote when it is necessary to quote a string. In particular, quotes must be used if:

- A string contains a literal quote character, as in

  'quote''inside'

  Note that a quote is included in a string by doubling it.

- A string contains at least one blank or comma, but is supposed to represent one token. For example,

  'a multiple token'

- A string contains at least one of the characters "[];,%", and the literal meaning of the characters is desired. Since these characters have meanings in CPL syntax, these meanings must be suppressed by quotes. For example,

  'hide_this_[function call]'

- A string contains an arithmetic operator surrounded by blanks, and the literal meaning is desired. For example,

  'not a + operator'

- A string begins with the character "-", but it is not intended that the value represent a command control argument, as in

  '-not_a_control_argument'

Note that

    SET_VAR A := 'quotes_go_in'

sets A to "'quotes_go_in'", i.e., the quotes are part of the value.

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 This	Produces This
%x%%y%	'abcd'
%x% %y%	'ab' 'cd'

In the first example, the values of x and y are concatenated by removing the their right and left quotes respectively. In the second example, the intervening blank causes the references to be replaced without concatenation. Similar rules hold for function calls.

CPL provides a function which will unquote strings. A call on the function UNQUOTE has the form

    [UNQUOTE string]

For example,

    [UNQUOTE 'ab']
    [UNQUOTE %x%]

The UNQUOTE function removes the outermost pair of quotes (if any), and changes every pair of adjacent quotes to a single quote:

%x%	[unquote %x%]
ab	ab
'ab'	ab
'''ab'''	'ab'
'a''b'''	a'b'
'''a''''b'''	'a''b'

No other action is taken.

The QUOTE function will quote strings. A call on this function has the form:

    [QUOTE string-1 {string-2} ...]

For example,

    [QUOTE abc]

```
[QUOTE %a% %b% def]
```

The QUOTE function adds an outer pair of quotes to its arguments. If an argument of the quote function already contains quotes, these quotes will be automatically doubled to preserve the original meaning of the string. This works for any number of quote levels. So, suppose x has the value "ab'c'd", then

```
[QUOTE %x%] returns 'ab''c''d'
```

If this result were the argument of another call on QUOTE, as in

```
[QUOTE [QUOTE %x%] ]
```

then the result would be

```
'''ab''''c''''d'''
```

The RESCAN Function

The RESCAN function may be used to force evaluation of quoted variable references and function calls. This function strips one layer of quotes from its argument and evaluates any function calls or variable references which are not still quoted. To illustrate the use of this function, suppose a CPL program test_a_fun has an argument funs_and_vars, which has as its value a string containing variable references and function calls. That is, funs_and_vars might be "[length %holycow%]". If we try invoking test_a_fun by:

```
r test_a_fun [length %holycow%]
```

the call on function length and the variable reference will be evaluated, not at all what was intended. Clearly, we must type:

```
r test_a_fun '[length %holycow%]'
```

The quotes will suppress evaluation and "'[length %holycow%]'" will be assigned to funs_and_vars. However, when test_a_fun wants to evaluate the function call in funs_and_vars, using just %funs_and_vars% would give the value with its quotes, again suppressing evaluation of the function. The rescan function must be used to strip the quotes and evaluate the string "[length %holycow%]". Thus, test_a_fun might contain the statement:

```
&if [rescan %funs_and_vars%] > 100 &then &return
```

which causes test_a_fun to return if the length of the value of holycow is greater than 100.

USING ABBREVIATIONS

The &EXPAND Directive

The &EXPAND directive enables and disables statement expansion within a CPL program. Its form is:

```
&EXPAND  {ON }
         {OFF}
```

&EXPAND ON causes the CPL interpreter to pass each command in the CPL file to the abbreviation pre-processor for abbreviation expansion. The command is passed before variable evaluation, function evaluation, and execution. Directives are not passed to the pre-processor. Therefore, user-defined abbreviations cannot be used in CPL directives.

In order for expansion to work, the command

```
ABBREV pathname -ON
```

must be given either at command level or within the CPL program before any abbreviations are used.

&EXPAND directives take effect when they are read. They are effective only for the procedure that defines them; they do not carry over into programs or routines invoked by that procedure.

&EXPAND OFF disables expansion. This is the default setting.

EVALUATION OF EXPRESSIONS

Evaluation at PRIMOS Command Level

When variables and functions are used interactively, the command processor evaluates references and calls. Variable references are done first. So, suppose the variable .SRC has the value "MY_UFD" and variable .FILE the value "MY_PROG.FTN". The line:

```
FTN %.SRC%>%.FILE% -L [BEFORE %.FILE% FTN].LIST -B NO
```

would first have its variable references replaced:

```
FTN MY_UFD>MY_PROG.FTN -L [BEFORE MY_PROG.FTN FTN].LIST -B NO
```

Functions calls are done second. Thus, the above line would be converted to:

```
FTN MY_UFD>MY_PROG.FTN -L MY_PROG.LIST -B NO
```

before it is executed. 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 a CPL Invocation

When variables and functions are used inside a CPL program, the CPL interpreter evaluates references and calls. This is done because expressions must be evaluated in CPL directives (which the command processor does not understand) as well as in commands.

As in interactive evaluation, variable references are processed before function calls. However, in CPL directives there is a third step: an implicit call on the CALC function. (Chapter 12 describes CALC; briefly, this function calculates the values of arithmetic expressions.) CPL calls CALC on any expression in a CPL directive. If the expression contains operators delimited by blanks which are recognized by CALC, the operations are done; otherwise, the original string is returned. If operators are not to be interpreted by CALC, they must be quoted or not delimited by blanks. Thus, instead of saying:

```
&IF [CALC %I% > 5] &THEN &RETURN
```

we can say:

```
&IF %I% > 5 &THEN &RETURN
```

The implicit call to CALC is done <u>last</u>, after variables and functions have been evaluated. This implies that if infix operators are used inside a function call, CALC must be called explicitly. For example, suppose <u>A</u> has the value "5" and <u>B</u> the value "2". We must say

```
&IF %I% = [MOD [CALC %A% * %B%] %MODULUS%] &THEN &RETURN
```

because omitting the call on CALC would cause the string "5 * 2" to be taken as the first argument of the MOD function. Since "5 * 2" does not convert to an integer, an error will result (the function MOD does not "understand" that "*" means multiplication).

All uses of arithmetic operators in PRIMOS commands must be inside an explicit CALC invocation. For example, the command line

```
primos_command 1+5
```

represents a use of the string "1+5", while

```
primos_command [calc 1 + 5]
```

represents a use of the addition operator.

If statement expansion is in effect and the current statement is not a CPL directive, the statement is then passed to the abbreviation preprocessor. The string returned by the preprocessor is then passed to the system command processor for execution.

12
Command Functions

THE CALC FUNCTION

Arithmetic expressions may be evaluated using the function CALC. Its form is:

```
[CALC infix_expression]
```

This function evaluates expressions containing the logical operators & (and), | (or), and ^ (not); the arithmetic operators +, -, *, /, unary +, and unary -; and the relational operators =, <, >, <=, >=, and ^=. The 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

1. All operators which are to be evaluated by CALC must be delimited by blanks. This restriction resolves the ambiguity which can arise from the fact that "*", "<", and ">" are also valid pathname characters.

2. If CALC is given an expression containing more operators than it can handle, it prints 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 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. The final result is converted to a string and that string is returned as the value of CALC.

Arithmetic, logical, and relational operators have some restrictions on the kind of operands they accept. Arithmetic operators must have operands which convert to integers. (Strings which convert to integers must contain only digits, except possibly for a preceding sign and leading and trailing blanks: the resulting value must be in the range -2**31 + 1 ... 2**31 - 1.)

Logical operators must have operands which are Boolean. 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

```
%tvar% & %fvar%
^ ( %four% < %five% )
%tvar% | ( %four% < %five% )
```

are all valid expressions. However,

```
%tvar% | ( %four% + %five% )
```

is not valid since "%four% + %five%" is not a Boolean expression. The value returned by CALC is "TRUE" if the logical operations 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, a string comparison will be done. If both operands are either numeric or Boolean, an arithmetic comparison is done. Boolean true is interpreted as "1" and false as "0".

If we used relational operators with our sample variables, the expressions:

```
%four% + %five%
%six% * ( %four% - %five% )
```

would be legal, while:

```
%four% + 'i''m_not_a_number'
%four% + %tvar%
```

would not.

As in the other functions, the expression in CALC may contain function calls and variable references. The expression left after these are evaluated should, of course, be a valid logical or arithmetic expression.

OTHER ARITHMETIC FUNCTIONS

▶ [HEX hex-string]

hex-string is an expression which 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".

▶ [MOD decimal-string decimal-string]

Both arguments must be expressions that evaluate to decimal numbers. [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".

▶ [OCTAL octal-string]

octal-string is an expression which 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]

decimal-string is an expression which 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]

decimal-string is an expression which must evaluate to a valid decimal number. This function returns a string representation of the octal equivalent of decimal-string. Example: [TO_OCTAL 8] returns "10".

STRING FUNCTIONS

Some of the following functions will quote their results and others will not. If the result of a function is most likely to be used as a single token, but contains a semicolon, comma, blank, or quote, or if the result is an arithmetic or logical operator, the function will quote its result. If the result is most likely to be used as a list of multiple items, the result is not quoted. The 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 since the user most likely wants to treat it as one syntactic token. The WILD function, on the other hand, does not quote its result since 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 which always return one token, such as LENGTH, do not quote their results.

In the list that follows, an asterisk precedes the description of any function that quotes its results.

▶ * [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 abc.def.x .] returns "def.x".

▶ * [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 abc.def.x .] returns "abc".

▶ [INDEX string find-string]

returns the position of the leftmost occurrence of find-string within string. If find-string does not occur withing string, INDEX returns "0". For example: [INDEX abcdef de] returns "4".

▶ [LENGTH string]

returns the number of characters in string.

▶ [NULL string]

returns "TRUE" if string is the true null string or '', and "FALSE" otherwise.

▶ [QUOTE string1 string2 string3 ...]

adds an outer pair of quotes and doubles the quotes already in strings string{i}. This function is useful when it is necessary to suppress the meaning of special symbols through calls to subsystems (see Chapter 11 for a discussion of quotes). Examples:

[QUOTE xy'|'z] returns "'xy''|''z'"
[QUOTE abc 'd e' fg] returns "'abc ''d e'' fg'"

▶ [SEARCH string1 string2]

returns the index (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. Example: [SUBST aabbaabbaa bb qq] returns "aaqqaaqqaa".

▶ * [SUBSTR string start-pos {num-chars}]

start-pos must be numeric, and num-chars must be either omitted or numeric. If we count positions from left to right starting at 1, then [SUBSTR string start-pos] returns all characters in string in positions start-pos, start-pos+1, start-pos+2, etc. to the end of string. If num-chars is present, [SUBSTR string start-pos num-chars] returns the first num-chars characters in string to the right of and including the character in position start-pos. If start-pos and/or num-chars specify a substring that runs off the end of string, then start-pos and/or num-chars are reduced until the substring is proper or the null string results. Examples:

```
[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 ith position in in-chars, then replace it with the ith character in out-chars. More explicitly,

```
for each character in string:
    if current_char_in_string is in the ith position in in-chars
        then next_char_in_result = ith character in out-chars
            else next_char_in_result = current_char_in_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. Examples:

```
[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 from string. If which-side and trim-char are both omitted, leading and trailing blanks are trimmed. which-side specifies where the trimming occurs, and may be any of "-right", "-left", or "-both". trim-char specifies the character to be trimmed. If only trim-char is omitted, a blank is assumed. Example: [TRIM bbbabcbbb -both b] returns "abc".

▶ [UNQUOTE string]

removes one outer pair of quotes and changes every pair of adjacent quotes remaining to a single quote. (See discussion of quotes in Chapter 11). For example:

    [UNQUOTE '''xx''''yy'''] returns 'xx''yy'.

▶ [VERIFY string1 string2]

returns the index (counting from 1) of the first character in string1 that DOES NOT appear in the string string2. For example: [VERIFY 1298s8 0123456789] gives "5". The VERIFY function returns 0 if all characters in string1 appear in string2.

FILE SYSTEM FUNCTIONS

▶ [ATTRIB path {-TYPE | -DTM | -LENGTH} {-BRIEF}] | 19.0

This function returns information about the file specified by path. Exactly one of the options -TYPE, -LENGTH, or -DTM, must be given on each call. 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 (-LEN) option returns the length of the file in words. | 19.0

The -BRIEF option, if used, suppresses the printing of messages by ATTRIB. | 19.0

▶ * [DIR path {-BRIEF}] | 19.0

returns the directory portion of the pathname path. For example: [DIR smith>x>y] returns "smith>x". "*" (representing the home directory) is returned if the pathname is a simple filename.

The -BRIEF option, if used, suppresses the printing of messages by the -DIR function. | 19.0

▶ [ENTRYNAME path]

returns the entryname portion of the pathname path. Example:

    [ENTRYNAME smith>x>y] returns "y".

19.0 ▶ [EXISTS path {type} {-BRIEF}]

returns "TRUE" if there exists a file system object with pathname path of type type, and "FALSE" if not. If type is -ANY, any type of object will suffice. 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 can be used to suppress the printing of messages by the EXISTS function.

19.0

▶ [GVPATH]

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

▶ [OPEN_FILE pathname -MODE m status-var]

This function is useful when a user wants to open a file for reading/writing without having to specify a unit number as in the PRIMOS open command. The file specified by pathname is opened on some available unit; the unit number is returned as the value of the function. The mode indicates whether the file is to be opened for reading only (m = "r" or "R"), writing only (m = "w" or "W"), or 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 local or global.

For example, a call on OPEN_FILE might be:

```
&S READ_UNIT := [OPEN_FILE ALPHA -MODE R OK]
```

In this example, the file named ALPHA will be opened, and the number of the unit returned as the value of the variable READ_UNIT. The variable OK will be set to 0 if the file opening is successful. It will be set to a non-zero 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.)

▶ [PATHNAME rel-path {-BRIEF}] | 19.0

returns the full pathname given the relative pathname rel-path. Note that [DIR [PATHNAME x]] returns the pathname of the home 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 can be used to suppress the printing of messages by the PATHNAME function. | 19.0

▶ * [READ_FILE unit status-var {-BRIEF}] | 19.0

This function reads a record from the file open on unit (a decimal integer) and returns the quoted value of that record as its value (that is, the text read replaces the function call in the CPL program text). The variable status-var is set to "0" if the operation is successful and nonzero otherwise. (It is set to "1" when End of File is reached.)

The -BRIEF option may be used to suppress any messages READ_FILE might print. For example, a call on READ_FILE, following the opening of the file shown in the example above, might be: | 19.0

```
&S LINE := [READ_FILE %READ_UNIT% OK]
```

When this function call is evaluated, a line is read from the file previously opened on UNIT %READ_unit%. The line of text is then returned as the value of the variable LINE. The variable OK is set to 0 if the read is successful, to 1 if End of File has been reached, or to some other non-zero value if an error has occurred. Again, since the value of OK is being set each time the function call is evaluated, the variable name is not placed inside percent signs.

▶ [WILD wild-path wild-2 ... wild-n {control} {-BRIEF}] | 19.0

produces a blank-separated list of entrynames representing the file system objects that match the specifications of wild-path, wild-i and control. wild-path specifies the directory to consider, and the first wildcard name. The wild-i specify additional wildcard names (these may not be pathnames). control specifies DTM or type restrictions: -BEFORE date, -BF date, -AFTER date, -AF date, -FILE, -FL, -DIRECTORY, -DIR, -SEGMENT_DIRECTORY, -SEGDIR, -ACCESS_CATEGORY, -ACAT.

Example: [WILD @.pl1 @.ftn -fl] might produce the list "a.pl1 b.pl1 foo.ftn bar.ftn z.pl1".

It is easy for a call on wild to produce a result longer than the 1024 character maximum. The -single (-sgl) option causes wild to return matching names one at a time, rather than in one long string. This option takes a variable name as an argument; for example

```
[WILD london>@.pl1 -single unit-var]
```

unit-var must be initialized to zero by the user before calling wild to get the first name. When wild is called with the -single option and the value of unit-var is zero, wild opens the specifed directory on an available unit, sets unit-var to the (decimal) number of that unit, and returns the first matching name as its value. Subsequent calls will 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. The user must not modify the value of unit-var between calls on 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 &ITEMS directive and an example of the -SINGLE option).

19.0 | The -BRIEF option, if used, suppresses any messages from the WILD function.

▶ [WRITE_FILE unit text]

This is the inverse of the read_file function. The text is stripped of one layer of quotes and written on the file open on unit (a decimal integer). The function returns "0" if the operation is successful and nonzero otherwise.

Note

CPL uses decimal numbers to refer to file units, not octal numbers. If you open a file by saying:

```
&SET_VAR A := [OPEN_FILE THISFILE -MODE R STATUS]
```

you should close it in one of the following three ways:

```
CLOSE THISFILE
CLOSE -UNIT %A%
CLOSE [TO_OCTAL %A%]
```

19.0

Do not say simply, "CLOSE %A%"; this syntax assumes that A is an octal value and therefore does not work.

MISCELLANEOUS FUNCTIONS

▶ [ABBREV -EXPAND text]

expands text (if text is in fact an abbreviation and if you have an abbreviation file active), and returns the expanded string as its result. It does NOT requote the result.

19.0

If text is not an abbreviation, text itself is returned. If no abbreviation file is active, an error is reported.

▶ [CND_INFO control-flag]

This function allows a condition handler to examine the condition information of the most recent condition on the stack. The function returns different information depending on the setting of control-flag. If control-flag is "-name", the name of the condition is returned. For "-continue_switch", "-cont_sw" the Boolean value of the continue-to-signal switch is returned. For "-return_permit", "-ret_pmt" the Boolean value of the return-permitted switch is returned. If no condition frame is on the stack, -name returns "$NONE$", and -continue_sw and -return_permit both return "FALSE". The severity code is set to warning in this case. (For information on conditions and on Prime's Condition Mechanism, see the Subroutines Reference Guide.)

▶ * [DATE {format}]

returns the current date/time in a variety of formats. If format is omitted, the date only is returned: 81-10-21. The other possibilities are:

19.0

-FULL	81-10-21.13:24:48.Tue	
-USA	10/21/81	
-UFULL	10/21/81.13:24:48.Tue	
-VFULL	27 Apr 82 10:54:32 Tuesday	19.0
-DAY	21	
-MONTH	October	
-YEAR	1981	
-VIS	27 Apr 82	19.0
-TIME	13.24.48	
-AMPM	1:24 PM	
-DOW	Tuesday	
-CAL	October 21, 1981	
-TAG	811021	
-FTAG	811021.132448	

▶ [GET_VAR expr]

19.0 | 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 it is undefined. GET_VAR also returns $UNDEFINED$ if no global variable file is defined or active. This function can be used to test if a variable has been set. Example: [GET_VAR undefined_var] returns "$UNDEFINED$", assuming undefined_var is indeed undefined.

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. Example: [GET_VAR a%i%] returns the value of variable a1 if i has the value "1".

19.0 | ▶ [QUERY text {default} {-TTY}]

prints text on the user's terminal output stream, following it with a question mark. Printing is suppressed if text is null. The command input stream will be read for the user's reply, which must be "yes", "y", "ok", "no", "n" or null (case-insensitive). Null input causes the default to be returned; if the default is not specified, it is taken to be "FALSE". Otherwise, the function returns "TRUE" if the answer was "yes", and "FALSE" if it was "no". If text and default contain embedded blanks, they must be enclosed in quotes.

19.0 | The -TTY option forces the QUERY function to go 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 command line or CPL program containing it, whether that is a user at a terminal, a &DATA group within a CPL program, or a COMINPUT file.

▶ [RESCAN string]

returns the result of stripping one level of quotes from string and evaluating any function calls or variable references no longer appearing in quotes. Example: [RESCAN '[BEFORE ''[do not eval this]xxx'' x]'] returns "[do not eval this]".

19.0 | ▶ * [RESPONSE text {default} {-TTY}]

prints text on the user's terminal output stream, following it with a colon. Printing is suppressed if text is null. The command input stream will be read for the user's reply, which is returned (possibly quoted) as the value of the function. If a null reply is entered, the default is returned. If default is omitted, it is taken to be the null string. If text and default have embedded blanks, they must be quoted.

The -TTY option forces the RESPONSE function to go 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 command line or CPL program containing it, whether that is a user at a terminal, a &DATA group within a CPL program, or a COMINPUT file.

19.0

13
Arguments

INTRODUCTION

This chapter provides a full reference for the use of arguments in CPL.

It discusses the format and use of:

- Object arguments (that is, positional arguments)
- Option arguments
- Two special argument types, REST and UNCL

THE &ARGS DIRECTIVE

```
Syntax: &ARGS {name{:{type}{=default}  }...}~
        {name: -option_list{ name{:{type}{=default} };...} }
```

```
Examples:  &ARGS TRUTH;  BEAUTY;  CHARM

           &ARGS TRUTH:DEC; BEAUTY:TREE=A_UFD>FILE; CHARM:CHAR

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

CPL provides a powerful argument specification and validation facility. The &ARGS directive defines a "picture" of the command line that will be used to invoke this CPL procedure and declares local variables whose values will be those of the actual arguments. If the command line typed does not match the picture, a diagnostic is printed and the severity code set to error (see Chapter 15).

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, the user may define option arguments which flag the presence of specific arguments. In addition, a user may define arguments to be of a particular type, such as "tree" or "ptr", and have the &ARGS directive verify that the arguments supplied on the command line match the declared types. Finally, default values may be defined for arguments; these values are assigned to the arguments if they are omitted from the command line. All these features are discussed in detail in the following chapter.

An &ARGS directive may appear anywhere in a CPL procedure; the arguments on the command line are processed when the &ARGS directive is encountered. A procedure may have more than one &ARGS directive. If more than one &ARGS directive is executed, the same command line is parsed each time.

OBJECT ARGUMENTS

All object arguments are positional; they must appear on the command line in the same order as they appear in the &ARGS statement. For example, suppose CPL program X.CPL is to have three arguments. We might include a statement like this:

```
&ARGS SOURCE; DEST; NOLINES
```

If this program is invoked by the command line:

```
R X A B C
```

then the variable source has the value "A", dest the value "B", and nolines the the value "C". Note that in this simple case, lower case is mapped. to upper case. An error occurs if the user gives too many object arguments. In the above example, typing:

```
R X A B C D
```

would cause the message "Too many object arguments specified. D (cpl)" to be printed. If too few arguments are given, the omitted ones are assigned the system default value according to their type, as shown in Table 13-1.

SPECIFYING TYPES

The user can specify types for his arguments, by adding ":type" after the argument name. (If ":type" is omitted, the type defaults to "char".) Specifying a type restricts the form of the string which the &ARGS directive will accept as a value of the arguments. CPL will check that the type of the actual argument is the same as the declared type of the formal argument. A diagnostic is produced if a faulty argument is found, and an error severity code produced.

Arguments are just variables and their values may be altered just like other variables; the type is not checked if an argument is assigned a value using the SET_VAR command or the &SET_VAR directive.

The types supported, and their system default values, are shown in Table 13-1.

So, in the example above we might say:

```
&ARGS SOURCE:TREE; DEST:TREE; NOLINES:DEC
```

A valid call would be:

```
R X RICHS>EVAL.PL1 MY_SOURCE 50
```

HOW NULL STRINGS ARE HANDLED

The standard command processor will remove all occurences of the explicit null string from a command line before executing it. This allows a user to use omitted arguments on PRIMOS command lines without an error. For example, assume that a user defines an argument ftn_args:

```
&ARGS OTHER_ARGS:CHAR; FTN_ARGS:REST
```

and that ftn_args is going to be used in this line:

```
FTN %CURRENT_FILE% %FTN_ARGS%
```

If the user has echoing enabled and omits FTN_ARGS from his invocation of the CPL program (in order to get the default compiler options), he will see this echoed at his terminal:

```
FTN value_of_current_file ''
```

The '' indicates that ftn_args has been omitted and was assigned the system default value; the command processor will remove the '' before executing the command.

Table 13-1
Argument Types Supported in CPL

Type	Default	Description
char	''	Any character string up to 1024 characters long, mapped to upper case
charl	''	Any character string up to 1024 characters long, no case shifting
tree	''	A PRIMOS pathname up to 128 characters long
dec	0	Decimal integer
oct	0	Octal integer
hex	0	Hexadecimal integer
entry	''	File entryname up to 32 characters long
ptr	7777/0	Virtual address in format "octal/octal"
date	''	Calendar date in the form "mm/dd/yy hh:mm:ss day"
rest	''	The remainder of the command line
uncl	''	All tokens not accounted for by the &ARGS picture

ARGUMENT DEFAULTS

Users may specify a default value for each argument to override the system default values. If the argument is omitted in the command line used to invoke the program, the default value is assigned to it. Defaults are specified by typing "=default" after the declared type, or after the colon if type is omitted; the type is taken to be char in that case. Continuing our example, we might declare defaults like this:

```
&ARGS SOURCE:TREE; DEST:TREE=MY_UFD>MY_SOURCE.PL1; NOLINES:DEC=100
```

Typing:

```
R X RICHS>EVAL.PL1
```

would assign "RICHS>EVAL.PL1" to source and the default values "MY_UFD>MY_SOURCE.PL1" and "100" to dest and nolines respectively.

Default values may contain local variable references. For example, suppose the local variable standard_ufd has the value "LAUREL>HARDY", then the &ARGS directive:

```
&ARGS COMPILE_UFD:CHAR=%STANDARD_UFD%
```

would be transformed to:

```
&ARGS COMPILE_UFD:CHAR=LAUREL>HARDY
```

and "laurel>hardy" would be assigned to compile_ufd 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. So, we could say:

```
&ARGS COMPILE_UFD:CHAR=%STANDARD_UFD% ; OBJ_UFD:CHAR=%COMPILE_UFD%
```

which uses the value assigned to argument compile_ufd as the default value of argument obj_ufd; that is, if no value is typed in for obj_ufd on the command line, it defaults to be the same ufd as compile_ufd. This construction is possible because the &ARGS directive is interpreted first, and default values assigned second. Thus, if references to other argument variables are used in default, the value used as the default is the value assigned to that variable after the &ARGS directive has been interpreted.

Suppose a CPL program contains the statements:

```
&SET_VAR ARG_VAR := ZYMURGY
&ARGS OTHER_ARG:CHAR=%ARG_VAR%; ARG_VAR:CHAR
```

The first of these two statements has no effect whatever. 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". Circular references like:

```
&ARGS OTHER_ARG:CHAR=%ARG_VAR%;  ARG_VAR:CHAR=%OTHER_ARG%
```

are not permitted and give undefined results.

OPTION ARGUMENTS

Option arguments may be used to make some or all of the arguments order independent. They are used to identify a particular argument or group of arguments, or to select a specific program option; they are similar to option arguments used in standard PRIMOS command lines. For example, in:

```
FTN A_PROG.FTN -LISTING A_PROG.LIST -BINARY A_PROG.BIN -DEBUG
```

-listing and -binary are option arguments which identify the names of the listing and object files, respectively; -debug is an option argument which selects a compiler option. Option argument names must begin with a hyphen; the rest of the name may contain any character which is not a delimiter in the &ARGS directive (blank, comma, semicolon, colon, equal sign). For example, -listing, -no_binary.

Note

> Names of option arguments must contain at least one alphabetic character. Numeric names for option arguments (e.g., -123) are illegal, as they may be mistaken for negative integers.

An important restriction to remember is that in the &ARGS directive any object arguments (that is, positional arguments) must precede any option arguments used. This restriction does not apply to the command line being parsed.

Switches

The simplest option argument is a switch that is either present on the command line or not. A switch is used to select a specific program option (the -debug option in the ftn invocation above is an example of a switch). A switch is declared by:

```
&ARGS flag-var:name-list
```

name-list is a list of one or more option argument names separated by commas. The additional names are commonly used to provide short synonyms, as in:

```
&ARGS LIST_SW:-LISTING, -L
```

flag-var is the name of a local variable which will be set to the first name in the name-list of the option argument if any of the names in name-list appear on the command line. It will be set to the null string '' if none of the names appear. In the above example, if "-listing" or "-l" appears on the command line, list_sw is set to "-LISTING"; it is set to '' if neither appears.

Flags

The definition of an option argument may specify one or more arguments that will follow the option argument on the command line. In this case, the option argument acts as a flag which signals the presence of the argument group on the command line. Since the flag identifies the group, the group's position on the command line is independent of the position of its declaration in the corresponding &ARGS directive. Within a group flagged by one option argument, however, the arguments are position dependent. So, a program compile_and_go might have the statement:

```
&ARGS LIST_SW: -LISTING, -L LIST_FILE:TREE;~
      EXEC_SW: -EXECUTE, -E  OBJ_FILE:TREE; LIBRARY:CHAR
```

This statement declares three arguments: a listing file "list_file", which is flagged by either "-listing" or "-l"; and a binary file "obj_file" and a library-name "library", both of which are flagged by either "-execute" or "-e". Valid calls on compile_and_go are:

```
R COMPILE_AND_GO -EXECUTE RICHS>NEW_OBJ PLPLIB -L RICHS>NEW_LIST

R COMPILE_AND_GO -L RICH>NEW_LIST -E RICHS>NEW_OBJ PLPLIB
```

While list_file and the pair of arguments obj_file and library may be first or last on the command line, obj_file and library must appear in that order after their flag. As in a simple switch, flag_var is set to the first name in name_list if the control argument appears on the command line.

Continuing our example with program X, we might now say:

```
&ARGS  ORIG_FILE:-SOURCE,-S SOURCE:TREE;~
       DEST_FILE:-DEST,-D   DEST:TREE=MY_UFD>MY_SOURCE.PL1;~
       LINES:-LINES,-L      NOLINES:DEC=100
```

Notice the power of this brief statement. We have defined three arguments: source, dest, and nolines. source must be a pathname; the actual argument on the command line corresponding to source is flagged

by being preceded by either "-source" or "-s". dest must also be a pathname; the actual argument is flagged by either "-dest" or "-d", and defaults to "my_ufd>my_source.pl1". nolines must be a decimal integer, is flagged by "-lines" or "-l", and will be assigned the value "100" if it is omitted. In addition, the variable orig_file is assigned "-SOURCE" if that argument is present; similarly, dest_file is assigned "-DEST" and lines assigned "-LINES", if those arguments are present. A call on program X might now look like:

```
R X -D HIS_UFD>HIS_SOURCE -S RICHS>EVAL.PL1
```

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 X MYFILE -FROM 6 -L MYFILE.LIST
```

results in the 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 types are useful when the user wants CPL to interpret some arguments and take whatever else is on the command line (after variable references and function calls have been evaluated) as is. Consider the following CPL procedure, run_ftn.cpl:

```
&ARGS FTN_ARGS:REST
   FTN MUMBLE %FTN_ARGS%
```

typing:

```
r run_ftn -list stevec>mumble.list
```

assigns "-list stevec>mumble.list" to ftn_args. Note that no case mapping occurs for a rest type argument.

Another example:

```
&ARGS DIR:TREE; DEEP:-DEPTH; LIM_F:-LIMIT,-LIM  LIM_V:DEC;~
      CL_F:-COM_LINE,-CL  CL_V:REST
```

when given the command line:

```
command -LIM 50 ABC>DEF -DEPTH -CL LS -SORT_DIM -LONG
```

causes dir to be set to "ABC>DEF", deep to "-DEPTH", lim_f to "-LIMIT", lim_v to "50", cl_f to "-COM_LINE", and cl_v to "LS -SORT_DIM -LONG".

The parsing of the command line stops when a rest type argument is encountered in the picture; whatever remains on the command line is assigned to the rest argument. This means that only one rest type argument may appear among the object arguments, and it should be the rightmost of these; if there are no rest type object arguments, one or more option arguments may have a rest type as the rightmost of their positional arguments. Furthermore, all characters that are to be assigned to a rest argument must appear last in the command line; some order dependence has been introduced.

An uncl argument in the &ARGS picture does not stop the parsing of the command line. The parsing continues, and any tokens not assigned to an argument when the parse ends are concatenated and assigned to the uncl argument rather than causing an error. If an option argument that is not in the &ARGS picture is encountered, all arguments between it and the next option argument or the end of line are assumed to belong to the first option argument. For example:

```
&ARGS UNCLAIMED:UNCL;  FNL:-FUNNY_LIST,-FYL
```

when given the command line:

```
command source -b source.bin -dynm -funny_list -limit 26
```

causes variable fnl to be set to "-funny_list", and variable unclaimed to be set to "source -b source.bin -dynm -limit 26". Note that no case mapping occurs for arguments of type uncl.

Caution should be used when arguments are declared to be type uncl. The uncl type does not ensure order independence. Suppose run_ftn were to take two arguments, ftn_args and another_arg, and we use this &ARGS directive:

```
&ARGS ANOTHER_ARG:OCTAL;  FTN_ARGS:UNCL
```

Typing:

```
r ftn_args 777 -list stevec>mumble.list
```

assigns "777" to another_arg and "-list stevec>mumble.list" to ftn_args. However,

```
r ftn_args -list stevec>mumble.list 777
```

assigns "-list stevec>mumble.list 777" to ftn_args (since everything between an undeclared option argument and the next option argument, or the end of the line, is assigned to the uncl argument), and "0" to another_arg (the default for numeric types which are omitted from the command line).

Only one instance of the uncl type may appear in an &ARGS directive. An argument flagged by a option argument may not have type uncl.

14

Writing Subroutines and Functions in CPL

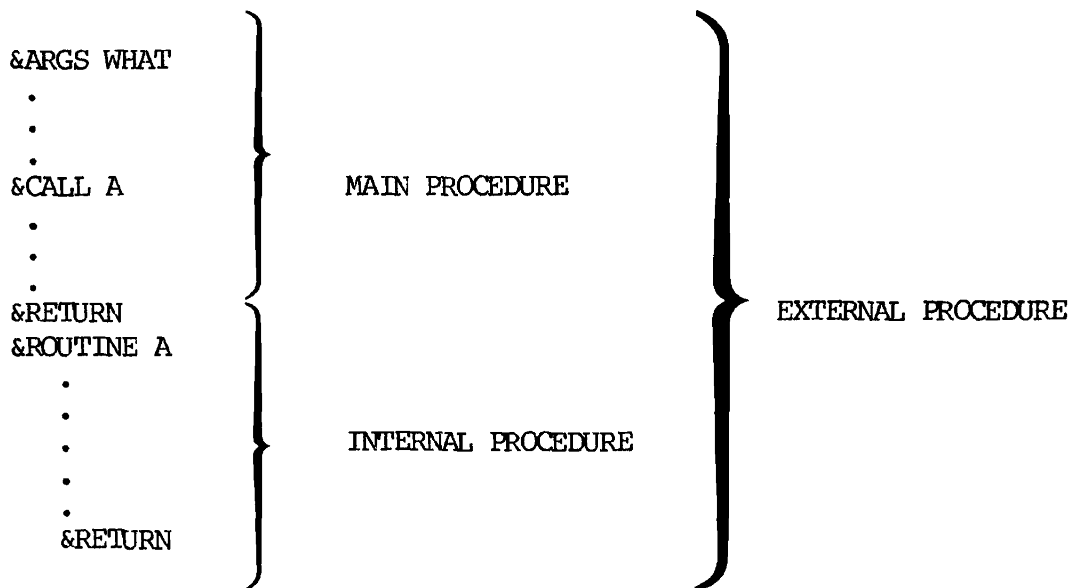
INTRODUCTION

CPL programs may contain internal "routines". These are equivalent to subroutines or internal procedures in high-level languages. 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.

CPL programs may also contain user-defined functions. The last part of this chapter explains how to write and invoke functions.

A Note on Terminology

In PL/I, 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. This terminology is diagrammed below.



In this guide, we use the term routine when we refer to a CPL routine. (For example, we say that every routine begins with a &ROUTINE directive.) We use the term procedure when we refer to a main-or-internal-or-external procedure. (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.)

WRITING ROUTINES

Routines in CPL are intended primarily for error handling and condition handling. However, they may be used for any purpose for which subroutines are used in high-level languages. For example:

- A routine might replace a lengthy &DO group following a &THEN, &ELSE, or &WHEN. The routine call itself would then be the argument of the &THEN, &ELSE, or &WHEN directive (for example, &THEN &CALL ROUTINE_A).

- A routine may be used when one operation must be performed several times during the course of a program. The routine can thus be written once, and calls to the routine placed 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
    .
    .
    .
&ROUTINE STARTUP
```

- They may be invoked only by the CPL file within which they exist

- They are ended by either:

  -- 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:

  -- 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 contained the following code:

```
&S NUMBER := 10
&CALL DOUBLE
TYPE %NUMBER%
&RETURN
&ROUTINE DOUBLE
&SET_VAR NUMBER := %NUMBER% * 2
&RETURN
```

  then the program, when invoked, would type 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.

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, &ON, &CHECK, or &HANDLER

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 routine begins here
.
.
.
&CALL ROUTINE_1
&CALL ROUTINE_2
.
.
.
&RETURN                         /* end of main program
&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. We do not recommend its use.

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. Entering a routine via a &GOTO causes an error, and terminates execution of the CPL program.

Nesting Routines

Internal procedures may call other internal procedures. An example of this would be:

```
  .
  .
  .
&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 subroutines:

- If the routine performs correctly, you usually want it to return control to the main CPL program, so that that 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 &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 CPL program, and returns control to the program's caller. Here is a trivial example that 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 prints 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 prints 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": that is, either the user (if the user had invoked the stopped program) or whatever CPL program had invoked the program and passed argument A to it.

Table 14-1
Forms of the &RETURN and &STOP Directives

Directive	Action
&RETURN	Halts execution of procedure in which it occurs. Returns control to procedure's caller.
&STOP	Halts execution of 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. Prints 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 severity code to caller. If &MESSAGE directive is included, prints text at terminal and writes it into command output files.

WRITING FUNCTIONS IN CPL

Users may define their own functions by writing a CPL program and invoking it via 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, passing it the arguments in arg-list.

A program invoked as a function must contain a &RESULT directive. Its format 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.

An Example

Here is a trivial program named DOUBLE.CPL:

```
&ARGS X: dec
&RESULT %X% * 2
&RETURN
```

DOUBLE could be invoked by the following statement:

```
&S A := [RESUME DOUBLE 5]
```

DOUBLE would take the integer "5" as its argument, double it, and return the integer "10". Variable A (in the calling program) would then be set to the value "10".

Using the &RESULT Directive

A CPL procedure may have more than one &RESULT directive; the last one encountered before the procedure executes a &RETURN or &STOP directive will be the function's value. If no &RESULT directive is executed, the value of the function is the null string, ''.

If a CPL procedure is not invoked as a function (that is, if the invocation is not enclosed within function call brackets) executing a &RESULT directive is an error.

15

Error and Condition Handling in CPL

INTRODUCTION

This chapter discusses:

- Error handling in CPL
- How CPL programs and routines can pass severity codes to each other
- Condition handling in CPL
- The use of routines for error-handling in CPL

ERROR HANDLING

Each executed PRIMOS command produces an error code known as a severity code. Severity codes may take one of three values, as shown in the table below. After a PRIMOS command is executed, the severity code it produces is available in the system-defined local variable, SEVERITY$.

Code	Meaning
0	No error
Positive integer	Error
Negative integer	Warning

Note

The user should never define SEVERITY$ as a variable himself. Doing so will interfere with CPL's ability to handle errors.

How CPL Handles 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.

How CPL Programs Can Handle Errors

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 ...")

&SEVERITY handling takes precedence over &CHECK handling. If the execution of a PRIMOS command activates both a &SEVERITY handler and a &CHECK handler, the &SEVERITY handler is invoked first. If the &SEVERITY handler returns (that is, if it does not execute a &STOP directive or a &GOTO), the &CHECK handler is executed.

The operation of the &SEVERITY directive and the &CHECK directive are explained below.

▶ &SEVERITY - SPECIFY SEVERITY HANDLING

Syntax: &SEVERITY {level action}

where level is any of:

&ERROR
&WARNING

and action is any of:

&FAIL
&IGNORE
&ROUTINE handler_label

Example: &SEVERITY &ERROR &ROUTINE ERROR_HAPPENED

This directive is provided as a convenience, since checking the value of SEVERITY$ and taking corrective action accordingly is expected to be a common operation. The statement is a shorthand for a &CHECK statement which checks the value of SEVERITY$.

The action clause specifies what is to be done if a severity code as bad as or worse than level is ever produced. If action is &FAIL, execution is terminated, and a positive severity code is returned to the caller of this CPL procedure. If action is &IGNORE, execution continues. If action is "&ROUTINE handler_label", CPL will invoke that 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 handler to be invoked. The only exception to this is the case when a &CHECK handler has been declared and the check expression evaluates to "TRUE". In this case, the check handler will be invoked after the &SEVERITY handler returns (if it does return), and before control returns to the next statement in the sequence.

▶ &CHECK - INVOKE A HANDLER IF A GIVEN EXPRESSION IS TRUE

Syntax: &CHECK expression &ROUTINE handler

Example: &CHECK %THIS_VAR% > %THAT_VAR% &ROUTINE DISASTER

After each PRIMOS command is executed the expression expression is evaluated. If the expression is true, the specified handler is invoked; otherwise, no action is taken.

If the handler ends normally or executes a &RETURN directive, control passes to the statement following the one that caused the invocation.

If a PRIMOS command generates a severity code and causes a check expression to become "TRUE", and both a check handler and a severity handler exist, then the severity handler is always invoked first; if that handler returns, the check handler will be invoked. Suppose a CPL program contains the statements:

```
&CHECK %THIS% > %THAT% &ROUTINE IT_WAS_GREATER
&SEVERITY &ERROR &ROUTINE ERROR_HAPPENED
```

If a PRIMOS command causes a positive severity code to be returned, and also causes variable this to become greater than variable that, then the handler error_happened will be invoked before the check handler. If error_happened returns, the handler it_was_greater will be invoked. If that handler returns, control will pass to the statement following the one that caused the invocations.

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
```

Assume also that you would like this program to know whether each of the programs it runs executes correctly, or whether their execution ran into problems. You would do this by having the three programs (or whatever error-handling routines they defined) return a severity code as part of the &RETURN or &STOP directive with which they end. The format of these two directives, when used to return severity codes, is as follows:

▶ &RETURN

Syntax: &RETURN severity {&MESSAGE text}

Example: &return 1

severity must evaluate to a string convertible to an integer. This integer is returned to the invoker as a severity (error) code. If severity is omitted, "0" is returned. The point of return is determined as in the simple &RETURN discussed in Chapter 2.

If the &MESSAGE clause is present, text is printed 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, you should not test for a specific integer. Rather, the test should be:

- 0, for no error
- > 0, for an error
- < 0, for a warning

▶ &STOP

Syntax: &STOP {severity} {&MESSAGE text}

Example: &STOP 1 &MESSAGE wrong, Wrong WRONG!

The &STOP directive

- Is processed like the &RETURN directive if it occurs in a main CPL program
- Halts both the routine in which it occurs and the procedure that invoked the routine, if it occurs within a routine.

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 is useful for handling exceptional conditions. Some familiarity with the PRIMOS condition mechanism is assumed. (See *The Prime User's Guide* for an introduction to the PRIMOS condition mechanism.)

An *on_unit* is a procedure which is called only when some special condition is raised. Since on-units can be complicated, CPL provides one for the user, and also makes it possible for the user to define simpler procedures called *handlers*. A handler is a &ROUTINE (as described later in this chapter) which has been declared as a condition handler by a &CHECK, &SEVERITY, or &ON directive (described below). When a handler is declared, its name and the name of the condition it will handle are saved by the CPL interpreter. When a condition is raised, CPL's on-unit is invoked; it examines CPL's list of handlers. If it finds a handler for the condition, the handler is executed. When the handler returns, CPL's on-unit returns to the point of interruption. If no handler is found, the PRIMOS condition mechanism is instructed to continue its search of the stack for other on-units (which may or may not 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 a condition is expensive. Therefore, condition handling should be reserved for unusual or unlikely events. (It is not expensive merely to declare a handler with the &ON directive.)

▶ &ON - DEFINE A HANDLER FOR A CONDITION

Syntax: &ON condition &ROUTINE handler_label

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) or one invented by the user. If the condition is raised, and the handler has not been *reverted* (see below), the handler is executed. (User-defined conditions are raised by using the &SIGNAL directive, explained below.)

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, the invocation of CPL in which the handler was defined is returned to (the stack is unwound if necessary),

and then the goto is executed. This aborts the command that raised the condition. A label is defined as being outside if it occurs earlier in the file than the &ROUTINE directive in question.

▶ &REVERT

Syntax: &REVERT condition

Example: &REVERT bad_input

The expression condition must evaluate to an identifier. The CPL program's handler for condition, if any, is reverted (cancelled).

▶ &ROUTINE - DESIGNATE START OF A ROUTINE

Syntax: &ROUTINE routine_label

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. &ROUTINE may not define the start of a routine to be inside any statement group (&DO, &SELECT, &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 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 is to a label which appears in the CPL file before the &ROUTINE containing the &GOTO.

&SIGNAL - RAISE A CONDITION

Syntax: &SIGNAL condition {&NO_RETURN}

Example: &SIGNAL bad_input

This directive raises the condition 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 will continue searching the user's stack for on-units. If the user has written no on-units, PRIMOS's own condition handling will be 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

Syntax Summary

▶ &ARGS

Syntax: &ARGS {name{:{type}{=default} }...}~
        {name: -control_list{ name{:{type}{=default} };...} }

Types: CHAR, CHARL, TREE, ENTRY, DEC, OCT, HEX, PTR,
       DATE, REST, UNCL

Examples:

```
&args truth; beauty; charm

&args truth:dec; beauty:tree=a_ufd>file; charm:char

&args charm:char; tr_flag:-tr truth:dec;~
      be_flag:-be beauty:tree=a_ufd>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:

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%

&do a &items [wild a_ufd>@@.pll -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 [&ERROR | &WARNING [&FAIL | &IGNORE | &ROUTINE label]]

```
Syntax: &SEVERITY   [ &ERROR     [ &FAIL          ] ]
                    [ &WARNING   [ &IGNORE        ] ]
                    [            [ &ROUTINE 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
CPL Error Messages

INTRODUCTION

When an error occurs in a CPL program, the CPL interpreter prints out 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 printed, 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 printed, it will be 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.

19.0

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

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 will look for pathname.CPL. If it finds it, it runs the file as a CPL job. If Batch doesn't 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, they will be ignored by Batch and passed 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 was used, the arguments are shown as the final line of the display (or before "Accts:" if -ACCT was specified).

An Example

Assume a CPL program, named TEST.CPL, that contains the following &ARGS statement:

```
&ARGS WHAT:TREE; HOWMANY:DEC = 0
```

This program might be run and displayed as follows:

```
OK, JOB TEST -ARGS SMITH>TESTBED 50
[JOB rev 18.1]
Your job, #00009, was submitted to queue Normal-1.
Home=<ADVERT>JONES>BATCH_JOBS
OK, JOB -DISPLAY
[JOB rev 18.1]


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.
Args: SMITH>TESTBED 50
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]
```

Two points should be noted:

- 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. 19.0

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:

1. How are the files executed?
2. How do they execute other files and programs?
3. What commands can they execute?
4. What special commands must they contain?
5. How can they control the execution of the commands they contain?
6. What error-handling capabilities do they have?

7. What use can they make of variables?

8. What use can they make of user-defined abbreviations?

9. How do they handle interactive utilities (such as ED and SEG) 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 PROG
```

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 RESUME or CPL commands to execute other CPL programs and R-mode user programs. They use SEG to execute V-mode and I-mode 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 R-mode programs and CPL programs with the RESUME command; they execute V-mode and I-mode programs with SEG; and they execute BASIC programs with BASIC or BASICV.

The command input file MUST specify the file unit on which the called command file is to be opened, and must use the CLOSE comand 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

Command input files can contain any PRIMOS command except:

- CLOSE ALL
- DELSEG ALL

Special Commands Needed

There are no special commands needed in a CPL file. (A CPL program always ends with a &RETURN statement, but the CPL interpreter will add that statement for you if you don't 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 PRIMOS's default mechanisms for error handling, severity code handling, and condition handling. Or, they may use CPL directives and/or subroutines to define their own error handling, severity code handling, and condition handling. (See Chapter 15 for details.)

Command input files must use PRIMOS's 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 PRIMOS's SET_VAR command to set or change the value of these variables.

Use of Abbreviations

CPL's &EXPAND directive allows commands to be passed from CPL files 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: SEG), 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 SEG), 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 loads a FORTRAN program:

```
/*BEGIN TEST OF COMMAND FILE
COMOUTPUT O_TEST
DATE
/*COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
/*LOAD THE PROGRAM
SEG
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
```

```
QU
/*COMMAND FILE TEST COMPLETED
DATE
COMO -END
CO -END
```

If C_TEST were rewritten as a CPL program, it would look like this:

```
/*BEGIN TEST OF COMMAND FILE
COMOUTPUT O_TEST
DATE
/*COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
/*LOAD THE PROGRAM
&DATA SEG     /*First change
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
&END          /*Second change
/*COMMAND FILE TEST COMPLETED
DATE
COMO -END
```

With the addition of variables, and the use of the new filename conventions, you would get:

```
&ARGS WHAT : TREE = TEST
COMOUTPUT TEST.COMO
DATE
FTN %WHAT% -64V
&DATA SEG -LOAD    /* Let SEG create default filename
  LO %WHAT%
  LI
  SA
  MAP %WHAT%.MAP 7
  MAP %WHAT%.UNSAT 3
  QU
  &END
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 IN 64V MODE
FTN FTN.TEST -64V
/* LOAD THE PROGRAM
COMINPUT C_LOADTEST 7
CLOSE 7
/* RETURN COMMAND TO USER TERMINAL
COMINPUT -TTY
```

The command file C_LOADTEST contains the following commands:

```
/* LOADTEST COMMAND FILE
SEG
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
QU
COMINPUT C_MAPS 10
CLOSE 10
COMINPUT -CONTINUE
```

The command file C_MAPS contains the following commands:

```
/* GET FULL MAP AND UNSATISFIED REFERENCES
SEG
VLOAD * #FTN.TEST
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
/* 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 would look like this:

```
/* GO.CPL, a translation of C_GO
/* Compile the program in 64V mode
FTN FTN.TEST -64V
/* Load the program
R LOADTEST
```

```
/*LOADTEST COMMAND FILE, CPL version
&DATA SEG   /*  Add &DATA directive
  VLOAD #FTN.TEST
  LO B_FTN.TEST
  LI
  SA
  QU
&END    /* Add &END directive
R MAPS.CPL      /* Resume replaces CO
   /* Remove CLOSE command
&RETURN  /* Change CO -CONTINUE to (optional) &RETURN
```

```
/*  MAPS.CPL, a CPL version of C_MAPS
/*  Get full map and unsatisfied references
&DATA SEG     /* Add &DATA directive
VLOAD * #FTN.TEST
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
&END   /* Add &END directive
/*  Return to 'calling&"' program
 &RETURN    /* This line is optional
```

If the three files wanted to pass the name of a local variable among themselves, they could do that as well:

```
/*  New version of GO.CPL
&ARGS WHAT : TREE = TEST
/*  Compile program
FTN %WHAT% -64V
/*  Pass program name to LOADTEST.CPL
R LOADTEST %WHAT%
```

```
/*  New version of LOADTEST.CPL
&ARGS WHAT
&DATA SEG -LOAD
  LO %WHAT%
  LI
  SA
  QU
  &END  /*  End &DATA group
/*  Pass argument to third CPL program
R MAPS.CPL %WHAT%
```

```
/* New version of MAPS.CPL
&ARGS WHAT
&DATA SEG
  VLOAD * %WHAT%  /* SEG looks for file ending in .SEG
  MAP %WHAT%.MAP
  MAP %WHAT%.UNSAT
  QU
  &END  /* End &DATA group
/* Control returns to LOADTEST.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.

E
Global Variable Routines

INTRODUCTION

Two routines are available for the accessing and setting of global variables from inside a user program. GV$SET sets the value of a global variable, and GV$GET retrieves the value.

GV$SET and GV$GET routines, as shown, use PL/I data types and declaration statements. Data type conversions for FORTRAN and COBOL are shown at the end of this appendix.

The Primos command DEFINE_GVAR must be used to define the global variable file before either of these two procedures is called.

GV$SET

GV$SET allows a user to set the value of a global variable. Its calling sequence is:

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

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

var-name is the name of the global variable to be set. This name must follow the rules for CPL global variable names. All letters must be upper case.

var-value is the new value of the variable var-name.

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

GV$GET

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

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

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

var-name is the name of the global variable whose value is to be retrieved. The name must follow the rules for CPL global variable names and must be in upper case.

var-value is returned value of variable var-name.

value-size is the length of the user's buffer var-value in characters.

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

DATA-TYPE CONVERSIONS FOR FORTRAN AND COBOL

The CHAR(*) VAR Data Type

The PL/I data type "char(*) var" is a varying-length character string. The first word of such a variable contains the length of the character string currently stored by the variable. The remaining words contain the string itself. (Note that the user does not supply the length; the PL/I, or PL1G, compiler, determines the length of the string itself, and updates the first word accordingly.)

FORTRAN Equivalent of CHAR(*) VAR: The FORTRAN equivalent of the "char(*) var" data type is an integer array. The first word of the array must store the length of the string to be passed. The other words in the array store the string itself, 2 characters per word. (Thus, a string 32 letters long would require an array of 17 words.)

COBOL Equivalent of CHAR(*) VAR: COBOL programs should create a record structure in which the first word contains the length of the character string to be passed, while the remaining words contain the character string itself.

The FIXED BIN Data Type

The FORTRAN equivalent of FIXED BIN is INTEGER*2. The COBOL equivalent is COMP.

Index

% 2-5

&ARGS directive:
  format of 2-5, 13-1
  multiple arguments 2-6
  omitted arguments 2-6

&BY clauses, in loops 9-7

&CALL directive 14-3

&CHECK directive 15-4

&DATA groups:
  &TTY directive 2-19
  CPL programs invoked from 5-7
  defined 2-18
  terminal input in 2-19

&DEBUG directive 10-1

&DEBUG:
  &ECHO 2-2, 10-2
  &EXECUTE 10-2
  &NO_ECHO 10-2
  &NO_EXECUTE 10-2
  &NO_WATCH 10-2
  &OFF 10-2
  in routines 10-1

&DO &ITEMS loops 7-10, 9-14

&DO &LIST loops 7-10, 9-11

&DO &UNTIL loops 9-10

&DO &WHILE loops 9-9

&DO groups 2-15

&DO loops 9-1

&ECHO 10-1, 10-3, 10-4

&ELSE directive:
  diagram of 2-12
  format of 2-10

&EXECUTE 10-2, 10-3

&EXPAND directive 11-7

&GOTO directive 2-17

&GOTOs, and routines 14-5

&IF directive:
  diagrammed 2-11
  format of 8-1
  nested 2-10, 8-3
  simplest form 2-7

use of EXISTS function with 2-15
use of NULL function with 2-13

&IF statements 8-1

&IF statements, nested 8-3

&ITEMS loops 7-10, 9-14

&LABEL directive 2-17

&LIST loops 7-10, 9-11

&MESSAGE directive 5-1, 5-9

&NO_ECHO 10-3, 10-4

&NO_EXECUTE 10-2, 10-3

&NO_WATCH 10-3, 10-5

&ON directive 15-6

&REPEAT loops 9-11

&RESULT directive 14-8

&RETURN directive 2-24, 3-4, 14-6, 14-7, 15-5

&REVERT directive 15-7

&ROUTINE directive 14-2, 15-7

&SELECT directive 8-6, 8-9

&SELECT, variable references in 8-9

&SET_VAR directive 4-1, 11-2

&SEVERITY directive 10-6, 15-3

&SIGNAL directive 15-8

&STOP directive 14-6, 14-7, 15-5

&TO clauses, in loops 9-7

&TTY directive 2-19, 5-1

&TTY directive:
conditional use of 2-20

&TTY_CONTINUE directive 2-23, 5-1

&WATCH 10-3, 10-5

+ (Wild character) 7-5

@ (Wild character) 7-5

@@ (Wild character) 7-5

ABBREV 2-1, 11-7

Abbreviations:
in CPL programs 11-7
used to invoke CPL programs 2-2

AFTER function 7-2, 12-4

Arguments:
default specification 6-2
default values for 13-5
defined by position 2-6
in CPL 13-2
multiple 2-6
null, handling of 13-3
object 13-2
omitted 2-6
option 13-6
REST type 13-8
supplied to CPL programs 2-5
type checking 6-2
types of 6-3, 13-4
UNCL type 13-8

Arithmetic expressions 12-2

Arithmetic operators 2-9, 11-8, 12-1, 12-2

ATTRIB function 12-7

Batch execution of CPL programs C-1

Batch jobs C-1

BEFORE function 7-2, 12-4

Boolean operators 2-9

Boolean values 12-1

Braces, in documentation xi

Brackets:
  for function calls 1-6
  in documentation xi

CALC function 12-1

CALC function, implicit calls on 11-8

CALC function, use of 12-2

Calling routines 14-2, 14-3

CHAR, type of argument 6-3

CHARL, type of argument 6-3

Check handling 15-3, 15-4

CMDNC0 (system commands directory) 1-2

CND_INFO function 12-11

COMINPUT files D-1

COMINPUT files, input from 5-6

Command input stream 5-5

Commands:
  ABBREV 2-2
  CPL 1-2
  DEFINE_GVAR 4-6, 4-7, 11-3
  DELETE_VAR 4-6, 4-10, 11-3
  in CPL programs 2-1, 2-4
  LIST_VAR 4-6, 4-10, 11-3
  RESUME 1-2
  SET_VAR 4-6, 4-8, 11-2
  TYPE 5-8

Comments in CPL programs 3-3

Concatenation:
  of command lines 3-3, 3-4
  of strings 3-7
  of variables 2-6, 4-4

Condition handling 15-6

Condition mechanism 15-6

Conventions:
  filename 7-1
  in examples x
  used in this book x

CPL:
  command 1-2
  directives 1-3
  errors 2-23
  features 1-8
  interpreter 1-3, 1-4
  invoking programs 1-2
  language 1-3
  program names 1-2
  subsets of 1-8
  suffix 1-2, 2-1

DATE function 1-5, 12-11

DATE, type of argument 6-3

Debugging CPL programs 10-1

DEC, type of argument 6-3

Decrementing counted loops 9-6

Default checking, for arguments 6-5

Default error handling 15-1

Default specification, for arguments 6-2

Default values for arguments 13-5

DEFINE_GVAR command 4-6, 4-7, 11-3

Defining functions 14-8

Defining global and local variables 2-5, 11-2

Defining variables, with &ARGS directive 2-5

DELETE_VAR command 4-6, 4-10, 11-3

DIR function 12-7

Directives:
- &ARGS 2-5, 13-1
- &CALL 14-2
- &CHECK 15-4
- &DATA 1-8
- &DEBUG 10-1
- &DO 2-15
- &ELSE 2-10
- &EXPAND 11-7
- &GOTO 2-17
- &IF 2-7
- &LABEL 2-17
- &MESSAGE 5-9
- &ON 15-6
- &RESULT 14-8
- &RETURN 2-24, 14-6, 14-7, 15-5
- &REVERT 15-7
- &ROUTINE 14-2, 15-7
- &SELECT 8-5, 8-9
- &SET_VAR 4-1, 11-2
- &SEVERITY 10-6, 15-3
- &SIGNAL 15-8
- &STOP 14-6, 14-7, 15-5
- &TTY 2-19
- &TTY_CONTINUE 2-23
- flow-of-control 8-2
- handled by CPL interpreter 1-6

DO groups 2-15

Echo/no_echo, for debugging 10-4

Echoing commands in CPL programs 2-2

Ellipsis, in documentation xi

Ending routines 14-6

ENTRY, type of argument 6-3

ENTRYNAME function 12-7

Equality, testing for 2-8, 4-5, 11-8

Equals sign 2-9, 3-4, 12-1

Error handling:
- default 2-23
- in subroutines 10-7
- user specified 10-6, 15-2

Error messages B-1

Evaluation:
- at PRIMOS command level 11-7
- implicit calls on CALC 11-8
- of arithmetic expressions 12-1
- of Boolean relations 12-1
- of expressions 11-7
- of functions 11-4
- of quoted strings 11-6
- of variables 11-3
- precedence in 12-1
- within a CPL program 11-8

Execute/no_execute, for debugging 10-2

Execution of loops 8-2

Execution, Batch C-1

EXISTS function 12-8

Expressions, evaluation of 11-1, 11-7, 12-1, 12-2

External procedures 14-1

File I/O 9-16

File system functions 12-7

Filename conventions 7-1

Filenames for CPL programs 3-4

Flags, in option arguments 13-6

Flow of control 2-7

Flow-of-control directives 8-2, 9-1

Format rules 3-1

Function Calls:
- defined 2-13
- evaluated by CPL interpreter 1-5
- format of 11-3
- in &IF statements 2-13

Functions:
- AFTER 7-2, 12-4
- ATTRIB 12-7
- BEFORE 7-2, 12-4
- CALC 12-1
- CND_INFO 12-11
- DATE 12-11
- DIR 12-7
- ENTRYNAME 12-7
- evaluation of 11-4
- EXISTS 2-14, 12-8
- GET_VAR 12-12
- GVPATH 12-8
- HEX 12-3
- INDEX 12-5
- LENGTH 12-5
- MOD 12-3
- NULL 2-13, 12-5
- OCTAL 12-3
- OPEN_FILE 12-8
- PATHNAME 12-9
- QUERY 5-2, 12-12
- QUOTE 11-5, 12-5
- READ_FILE 12-9
- RESCAN 11-6, 12-12
- RESPONSE 5-4, 12-13
- SEARCH 12-5
- SUBST 12-6
- SUBSTR 12-5
- supplied by CPL 11-3
- TO_HEX 12-4
- TO_OCTAL 12-4
- TRANSLATE 12-6
- TRIM 12-6
- UNQUOTE 11-5, 12-6
- user-defined 14-8
- VERIFY 12-7
- WILD 7-6, 12-9
- WRITE_FILE 12-10

GET_VAR function 12-12

Global variables 4-6, 11-2

GOTO 2-17

Grouping statements:
- &DATA groups 2-18
- &DO groups 2-15

GV$GET routine E-2

GV$SET routine E-1

GVPATH function 12-8

HEX function 12-3

HEX, type of argument 6-3

High-level Languages:
- programming concepts x, 1-3

IF statements:
- diagram of 2-11
- format of 2-7

IF_THEN_ELSE statements:
- diagrammed 2-12
- format of 2-10
- with function calls 2-13

Incrementing counted loops 9-2, 9-6

Indentation of lines in CPL programs 3-2

INDEX function 12-5

Inequality, testing for 2-8, 4-5, 11-8

Integer values for variables 4-3

Internal procedures 14-1

Invocation of routines 14-3, 14-4

LENGTH function 12-5

LIST_VAR command 4-6, 4-10, 11-3

Local variables 4-5, 11-2

Logical expressions 12-2

Logical operators 2-9, 12-1, 12-2

Logical values for variables 4-4

Loop formats 8-2

Loops 9-1

Loops, execution of 8-2

Loops:
&BY clauses 9-7
&DO &ITEMS 7-10, 9-14
&DO &LIST 7-10, 9-11
&DO &UNTIL 9-10
&DO &WHILE 9-9
&REPEAT 9-11
&TO clauses 9-7
counted 9-7
counted, execution of 9-7
decrementing 9-7
formats of 8-2
incrementing 9-2, 9-7
nested 9-5
with file I/O 9-16

Lower case, in documentation x

Main procedures 14-1

Miscellaneous functions 12-11

MOD function 12-3

Multiple arguments 2-5

Names:
of CPL programs 1-2
of global variables 4-7

Nested loops 9-5

Nesting routines 14-5

No_execute, debugging technique 10-2

NULL function 2-13, 12-5

Null strings:
explicit 2-6
handling of 13-3
removed by command processor 2-6
supplied for omitted arguments 2-6

Object arguments 13-1, 13-2

OCT, type of argument 6-3

OCTAL function 12-3

Omitted arguments 2-6

OPEN_FILE function 12-8

Operators 2-8, 12-1

Operators, preceded and followed by spaces 3-4

Option arguments 13-1, 13-6

Output, as seen on terminal 2-2

Passing severity codes 15-4

PATHNAME function 12-9

Percent signs 1-4, 2-5

Phantom execution of CPL programs C-3

Phantoms C-3

Placement of routines 14-4

Positional arguments 6-5, 13-2, 13-2

Precedence of operators 12-1

PRIMOS commands:
ABBREV 2-2
in CPL programs 1-3, 1-4, 2-1, 2-4

not available for CPL 2-4

PRIMOS:
information on 1-3
interaction with CPL 1-3

Procedures 14-1

PTR, type of argument 6-3

QUERY function 5-1, 5-2, 12-12

QUOTE function 11-5, 12-5

Quoted strings 3-5, 11-4

Quoted strings, evaluation of 11-6

Quoting strings 3-5

Reading files 9-16

READ_FILE function 12-9

Relational operators 2-9, 12-2

RESCAN function 11-6, 12-12

RESPONSE function 5-1, 5-4, 12-13

REST arguments 13-8

REST, type of argument 6-3, 6-6

RESUME (command) 1-2

Routines, in CPL 14-1

Routines:
calling 14-3
ending 14-3, 14-6
nesting 14-5
placement of 14-4

Running CPL programs:
as Batch jobs C-1
as phantoms C-3
interactively 1-2

Scope of &SEVERITY directive 10-8

Scope of variables 14-3

SEARCH function 12-5

Semicolons:
as command delimiters 3-1
to separate arguments 2-6

Sequential execution of CPL programs 2-7

SET_VAR command 4-6, 4-8, 11-2

Severity codes 10-6, 15-1

Severity codes, passing 15-4

SEVERITY$ 15-1, 15-5

SINGLE option, to WILD function 7-8

Square brackets 1-6

Statement, CPL 3-2

String functions 12-4

String values for variables 4-3

Strings:
concatenation 2-6, 3-7
quoting 3-5
unquoting 3-7

Structured programming concepts 1-3

Subsets of CPL 1-8

SUBST function 12-6

SUBSTR function 12-5

Suffixes:
CPL 1-2
filename 7-1

Switches, option arguments as 13-6

Syntax, summary of A-1

Terminal display of CPL programs 2-2

Terminal input:
- from QUERY function 5-2
- from RESPONSE function 5-4
- in &DATA groups 2-19

Terminal output:
- from &MESSAGE directive 5-9
- from &RETURN directive 14-6
- from &STOP directive 14-6
- from TYPE command 5-8

Termination of routines 14-3

Tilde (~) 3-3

TO_HEX function 12-4

TO_OCTAL function 12-4

Transfer of control between CPL programs 2-24

TRANSLATE function 12-6

TREE, type of argument 6-3

TRIM function 12-6

Type checking, for arguments 6-2, 6-5, 13-3

TYPE command 5-2, 5-8

Types, of arguments 13-4

UNCL arguments 13-8

UNCL, type of argument 6-3

Underlined words:
- in command formats x
- in examples x

UNQUOTE function 11-5, 12-7

Unquoting strings 3-7

Upper case, in documentation x

User-defined functions 14-8

Using ABBREV files 11-7

Variable names 11-2

Variable names, rules for 3-4

Variable references:
- concatenating 2-6
- defined 2-5
- evaluated by CPL interpreter 1-4
- inside function calls 1-5
- use of percent signs with 1-4

Variable watching 10-5

Variables:
- defined 11-2
- evaluation of 11-3
- global 4-6
- handled by CPL interpreter 1-4
- in &SELECT directives 8-9
- in CPL programs 2-5
- Integer values for 4-3
- local 4-5
- logical values for 4-4
- names vs. values 2-5
- string values for 4-3
- used by routines 14-3

VERIFY function 12-7

Watch/No_watch, debugging technique 10-5

Wild characters 7-5

WILD function 7-6, 12-9

WILD function, options for 7-7

Wildcards 7-4

WRITE_FILE function 12-10

Writing files 9-16

Writing functions 14-8

^ (Wild character) 7-5