decsystem10/20

CONVERSATIONAL PROGRAMMING LANGUAGE USER'S MANUAL

DEC-10-LCPLA-B-D

digital equipment corporation · maynard, massachusetts

First Printing, March 1976

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

SUMMARY OF TABLE OF CONTENTS

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

TO THE READER

The experienced PL/I programmer may find the last chapter, "CPL SUMMARY" sufficient to learn the CPL system. He can also refer to the chapter, "CPL PROGRAMMING EXAMPLES," which is near the end of the manual.

This manual is intended to be both a tutorial and a reference manual. Each chapter, and sometimes each section, will discuss features in a general way which the beginner should be able to understand. Later in the chapter or section there will follow more detailed and comprehensive technical coverage of the subject. Therefore the reader should approach this manual as follows: At the first reading, read quickly, skimming over or skipping sections which you do not understand. Later, when you need more technical details, you can go back and read the more difficult sections.

To help you even further, each chapter and heading line is following by a letter (A), (B), (C), (D) and (R), indication the type and level of material. These codes have the following meanings:

(A) Beginning material (Chapter 1 only) which is suitable for the person who has never programmed before or who has never used CPL before.

(B) Material which covers the simple data types (FIXED and FLOAT), arrays, built-in functions, and the simple programming statements (IF, DO, DECLARE, labels, GOTO, direct statements, END, GET LIST, PUT LIST).

(C) More advanced programming statements, which permit more powerful programs to be written, including subroutine and function PROCEDUREs, error handling with ON, PUT and GET to arbitrary disk files, CHARACTER and BIT data types, and data conversions.

(D) The most advanced CPL programming functions, including some esoteric things. Sections at this level discuss storage classes, BASED storage and POINTERs, RECORD input/output to arbitrary files, full scope and invocation rules for blocks including recursive blocks, and the DEFAULT statement.

(R) These chapters are not meant to be read. They are to be used as reference chapters by users at all levels. This level includes the list of error messages, the list of built-in functions and pseudo-variables and the comparison of CPL with the ANSI PL/I standard.

CHAPTER 1

INTRODUCTION TO CPL (A)

This chapter is for beginners -- people who have never used a computer before, as well as experienced programmers who have never used CPL before. After you read this chapter, you will be better prepared to use the rest of the manual. When you read the rest of the manual, keep in mind that each chapter contains both easy and difficult material. Furthermore, many chapters are sufficiently self-contained so that you can understand it without understanding preceding chapters.

For this reason, you should always feel free to use refer to only those parts of the manual that you need. You should never be afraid to try to read a section, just because you have not understood the preceding material.

If you do not know how to log on and start CPL running, please refer to the chapter "Running CPL under TOPS-10" or to the chapter "Running CPL under TOPS-20."

1.1 DESK CALCULATOR MODE (A)

CPL can be used as a "desk calculator." This means that you can use CPL to get the values of computations.

When you enter CPL, CPL types out an asterisk. This signifies that CPL is ready for commands, which are called "statements."

The first statement you will use is the "?" statement. (? is an abbreviation for PUT LIST. The PUT statement is described in a later chapter.)

For example, if you type the statement

```
?2+3
```

then CPL will type back the sum, 5.

You may specify any expression using the operators +, -, /, * (for multiplication) and ** (for exponentiation). You may also use parentheses, just as you would in a mathematical formula to specify the order in which operations are to take place.

For example, if you type

```
?(23.45-6)*(.0183+16.3487)
```

then CPL will type back the answer, 285.60414.

The "**" operator stands for exponentiation. M**N stands for M raised to the power N. For example, if you type

```
?3.45**6.2
```

then CPL will type back the answer, 2160.122.

1.2 SCIENTIFIC NOTATION AND E-TYPE CONSTANTS (A)

You can express very large or very small constants using E-type notation. For example, the constant 23.4E7 stands for 23.4 times 10 to the power 7, or 234000000.

The number following the letter E is the power of 10 with which the first part of the number must be multiplied. This number may also be negative. For example, 23.4E-5 is the same as .000234.

For example if you type the statement

```
?25E10 * 16
```

then CPL will type back 4E12.

1.3 BUILT-IN FUNCTIONS (A)

Suppose you want to compute the hypotenuse of a right triangle. The formula for that computation, given the sides A and B, is to take the square root of A**2+B**2. CPL provides a built-in function, called SQRT, which allows you to compute square roots.

For example, if you type

```
?SQRT(3**2 + 4**2)
```

then CPL will type back 5, the length of the hypotenuse of a 3-4-5 triangle.

If you type `?SQRT(7.5**2 + 8.3**2)`

then CPL types back 11.186599, the length of the hypotenuse of a right triangle with legs 7.5 and 8.3.

CPL contains many built-in functions. They are all described in the chapter entitled "BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES."

Some of the most commonly used ones are:

1. SQRT(x) computes the square root of x.

2. SIN(x) and COS(x) compute the sine and cosine, respectively, of x, where x is an angle measured in radians.

3. SIND(x) and COSD(x) compute the sine and cosine, respectively, of x, where x is an angle measured in degrees.

4. ABS(x) computes the absolute value of x.

5. LOG(x) computes the natural logarithm of x.

6. LOG10(x) computes the common logarithm of x (the logarithm to the base 10).

1.4 THE ASSIGNMENT STATEMENT (A)

You may wish to save the values of some intermediate computations, and use those values later. You can do this by assigning the value of an expression to a CPL variable.

For example, if you type the statement,

```
VALUE=23.4+15
```

then CPL computes the value of 23.4+15 and assigns that value to the CPL variable VALUE. If you want to know the value of VALUE, you can type

```
?VALUE
```

to which CPL will type out 38.4.

You can also use VALUE in another expression. For example, you can type

```
?SQRT(VALUE+1)
```

and CPL will type out 6.2769419, the value of the square root of 39.4.

1.5 USE OF "COLLECT" STATEMENTS (A)

Up to this point, all statements which we have discussed have been in "direct" or "desk calculator" mode.

There is another mode of typing in statements. A "collect" statement is one which you type in preceded by a line number. Such statements are not executed immediately; instead, they are saved as part of your program, to be executed later with your whole program.

Suppose, for example, you wish to write a program which computes both the area and hypotenuse of a right triangle with sides A and B and types out these values. You can type in the 2 statements:

```
10 ?A*B/2
20 ?SQRT(A**2 + B**2)
```

These two statements are typed in with line numbers (10 and 20). As a result, they are not executed immediately; instead they are stored with the program.

You can list your collected program by typing "LIST". If you type "LIST", then CPL will type out the two statements which you have just typed in with the line numbers 10 and 20.

Now, you can execute the program by doing something like the following:

```
          A=8.4
          B=35.3
          XEQ
```

The first two lines assign the values 8.4 and 35.3 to A and B, respectively. The third line, the "XEQ" statement, tells CPL to execute the collected program. The two statements which are in the collected are executed, and CPL types out the values

```
          148.26 36.285672
```

The first of these figures, 148.26, is the area, computed by statement 10. The second, the hypotenuse, is computed by statement 20.

You can now type the following:

```
          A=10.4
          B=13.5
          XEQ
```

and CPL will type out the values

```
          70.2 17.04142
```

Note that there is no longer any need to retype the the two formulas in statements 10 and 20. All you have to do is change the values of A and B and they type XEQ, and your program will be executed. This can save you a lot of time if you wish to compute the same formulas over and over with different values of the variables.

1.6 MODIFYING YOUR PROGRAM (A)

If you have followed the example given above, your collect program contains two statements. Let us see how you can modify your program:

```
          5 ?'THE VALUES OF A AND B ARE',A,B
          6 PUT SKIP
          10 ?'THE AREA AND HYPOTENUSE ARE', A*B/2
          LIST
```

The first two statements have line number 5 and 6, so they are inserted into your program before line 10. The third statement has the line number 10, and so it will replace the statement in your program with line number 10.

The last line is the LIST statement. It will type out your program as it stands currently:

```
 5.        ?'THE VALUE OF A AND B ARE',A,B;
 6.        PUT SKIP;
10.        ?'THE AREA AND HYPOTENUSE ARE',A*B/2;
20.        ?SQRT(A**2+B**2);
```

Now, if you type

```
          A=5
          B=12
          XEQ
```

then CPL will type out the following:

```
          THE VALUE OF A AND B ARE 5 12
          THE AREA AND HYPOTENUSE ARE 30 13
```

You can entirely erase a statement by means of the ERASE statement. For example, if you type "ERASE 20", then CPL will delete statement 20.

You can erase your entire program by typing "ERASE THRU ...".

1.7 STATEMENTS LABELS AND THE GOTO STATEMENT (A)

Suppose you want to type out a table of numbers and their square roots. You could type in the following program:

```
10.        I=0;
20.       LOOP: I=I+1;
30.        PUT SKIP;
40.        ?I,SQRT(I);
50.        GO TO LOOP;
```

Here is what each of the statements in this program does:

Statement 10 is an assignment statement. It assigns the value 0 to the variable I.

Statement 20 is another assignment statement, which increases the value of I by 1. It does this by computing the value of I+1 and assigning that value as the new value of I.

Statement 30 types out a carriage return.

Statement 40 types out the values of two quantities: I and SQRT(I).

Statement 50 is a GOTO statement. It indicates that control is to transfer to the statement which has the statement label LOOP. Statement 20 is such a statement -- the assignment statement I=I+1 is preceded by "LOOP:". The colon (:) tells CPL that the preceding identifier is to be considered a statement label for that statement. Therefore, after statement 50 is executed, CPL go back and re-execute statement 20, and continue executing from there.

If you type "XEQ", then you will be able to execute the above program. However, you should be aware that it contains what is known as an "infinite loop." An infinite loop is one which contains no conditions for stopping. The above program will go on and on forever, incrementing I by 1 and typing out the square root of the new value of I.

If you do execute the above program it will never stop by itself. If you wish to stop it, then you must type Control-C.

1.8 THE IF STATEMENT (A)

The IF statement provides a means for conditional execution. You can specify that if a certain thing is true, then CPL should do a certain thing.

For example, let us replace statement 50 in the last example with the statement

```
50.       IF I<=10 THEN GOTO LOOP ;
```

This statement tells CPL to do the following: Check to see whether the value of I is less than or equal to 10. If it is, then GOTO the statement with label LOOP. If it is not, then continue with the next statement. (In this case there is no next statement, so execution would stop.)

With this program modification, if you type XEQ, then the program will compute the square roots of all the numbers between 1 and 10 and print them out. Then the program will stop.

The program still contains a "loop," but it is not an "infinite loop."

1.9 THE DO STATEMENT (A)

Another way to write a program containing a loop is to use the DO statement. For example, suppose you erase the above program and type in the following program:

```
10.       DO I=1 TO 20;
20.       PUT SKIP
30.       ?I,SQRT(I)
40.       END
```

The DO statement in statement 10 and the END statement in statement 40 enclose a group of statements known as a DO-group.

The DO statement in statement 10 says to do the following:

Execute all the statements in the DO-group. The first time you execute them, let I = 1. The next time, let I=2. Continue executing them until I=20, then transfer control to the statement following the END statement. (In this case there is no such statement, so execution will stop.)

If you type XEQ, then the above program will print out a table of square roots of all numbers between 1 and 20, and then stop.

The DO statement is a complex statement with many optional specifications. A complete description, as well as further examples, will be found in the chapter entitled "THE DO AND END STATEMENTS."

CHAPTER 2

BASIC CPL LANGUAGE ELEMENTS (B)

2.1 THE CPL CHARACTER SET (B)

CPL recognizes all 128 ASCII characters. Not all of these, however, may appear in CPL programs. The rules governing the use of characters are explained below.

2.2 ALPHABETIC, NUMERIC AND ALPHAMERIC CHARACTERS (B)

CPL uses all the letters of the alphabet in upper and lower cases (A-Z and a-z), and the digits (0-9). Thus there are 62 alphameric (alphanumeric) characters.

2.3 SPECIAL CHARACTERS AND OPERATORS (B)

2.3.1 Special Characters (B)

The following are the special characters used by CPL, with descriptions:

CHAR	MEANING
<blank>	No meaning; separates elements of a statement
<tab>	Treated like a blank except in character strings
=	"Equal to" or assignment operator
+	Plus operator
-	Minus operator
*	Asterisk or Multiplication operator
/	Slash or Division operator
(	Left (or open) parenthesis
)	Right (or close) parenthesis
,	Comma
.	Decimal point or period
:	Colon
;	Semicolon
^	"Not" operator
!	"Or" operator
&	"And" operator
>	"Greater than" operator
<	"Less than" operator
'	Apostrophe or quotation mark
?	Question mark
_	Break character

2.3.2 Two-character Operators (B)

The following operators require two characters:

OPERATOR	MEANING
**	Exponentiation operator
>=	"Greater than or equal to" operator
<=	"Less than or equal to" operator
^=	"Not equal" operator
^>	"Not greater than" operator
^<	"Not less than" operator
->	Pointer qualifier
!!	Concatenation operator
/*	Start of comment
*/	End of comment

2.4 CHARACTER-STRING-ONLY CHARACTERS (B)

All the characters not discussed in preceding paragraphs are illegal in CPL programs, except inside character strings.

All ASCII characters may appear in character string constants in your program with the following exceptions: The carriage return (octal 15) and ilne feed (octal 12) may not appear in a character string constant.

It was mentioned above that CPL will treat a "tab" character appearing in your program as a blank. However, when you put a tab character into a character string constant, CPL does not change it to a blank or to a sequence of blanks; it remains as a single tab character.

2.5 IDENTIFIERS (B)

All variable names and keywords are identifiers.

An identifier contains between 1 and 31 characters, subject to the following rules:

1. The first character must be a letter.
2. All other characters must be alphabetic, numeric, or the break character (_).

2.6 VARIABLE NAMES (B)

You may use any identifier as a variable name.

If you have an upper/lower case terminal, and if you use two variable names which differ only in the case of the letters, then CPL will treat them as different variable names. For example, CPL will treat ABC, abc, AbC and aBc as four distinct variables.

CPL has no reserved words. For example, you may use the identifier IF in the IF statement, and, in the program (or even the same statement) you may use it as a variable name. CPL will determine from the context of the identifier whether you are referring to the keyword or to the variable name.

There is one partial exception to this rule concerning reserved words. The identifiers SYSIN and SYSPRINT are the default filenames used in the GET and PUT statements. Although you could, if you wanted to, use the DECLARE statement to give these identifiers attributes other than FILE, the PUT and GET statements without any FILE option and the ? statement would no longer work. For this reason, you are advised to treat these identifiers as reserved.

2.7 KEYWORDS (B)

Keywords are also identifiers. In CPL, all keywords are less than 16 characters long, and contain only letters.

A keyword will be recognized regardless of whether the letters are upper case, lower case, or mixed. Thus, DECLARE, DeClArE and dEcLaRe are all the same keyword.

2.8 COMMENTS (B)

You may put comments into any CPL statement. The comment is legal anywhere that a blank is legal. You begin the comment with the characters "/*" and you end the comment with the characters "*/".

EXAMPLE: A=/*COMMENT*/A+1; is the same as A=A+1;.

2.9 ABBREVIATIONS AND ALTERNATE KEYWORDS (B)

Some long keywords have abbreviations and alternate forms.

For example, EXECUTE may be typed XEQ, and DECLARE may be typed DCL.

A later chapter, "List of All CPL Abbreviations" provides a reference on CPL abbreviations.

CHAPTER 3

PROGRAMMING ELEMENTS (B)

3.1 DIRECT AND COLLECT STATEMENTS (B)

A "direct" statement is entered without a line number. CPL executes the statement immediately.

For example, the statement:

```
PUT LIST(A+B)
```

would be executed immediately.

You enter a "collect" statement with a line number. CPL will not execute such a statement immediately; instead, CPL will store such a statement with your stored program, to be executed later at your command. For example, the statement

```
10.  PUT LIST(A+B)
```

would be stored as your line 10, and executed later with your stored program.

3.2 TERMINATING SEMICOLON (B)

All CPL statements end with a semicolon. If you type in a line without ending it with a semicolon, then CPL will insert one for you.

3.3 LINE NUMBERS (B)

"Collect" statements, described above, are indicated by means of a line number.

A line number lies in the range 1 to 9999.99. It may have up to 2 fractional digits.

3.4 MULTIPLE STATEMENTS PER LINE (B)

You may type in several statements per line,separating them with semicolons. For example, the statements

```
 10.         I=5;  J=10;  PUT LIST(I+J)
```

will all be stored as your line 10.

3.5 CONTINUING STATEMENTS ON ADDITIONAL LINES (B)

If a statement is too long to fit on one line, then you may continue it on an additional line by the following method: If the last character on the line is an ampersand ("&"), then CPL will allow you to continue the statment on the following line.

The ampersand character is not considered to be part of the statement text.

The ampersand may appear in the following positions:

1. Anywhere where a blank character can be used as a separator

2. Anywhere in a CHARACTER or BIT string constant

The ampersand may not split an identifier or a numeric constant.

Statements may be continued on several lines in this manner.

NOTE: This method of continuing statements on additional lines is not standard PL/I. Also, it is a very ugly feature which I put in under duress. Most users should avoid using it.

3.6 STATEMENT NUMBERS (B)

Sometimes it is necessary to refer specifically to one of several statements appearing on the same line. A statement number provides this capability.

The first statement on line 10.33 will have the statement number 10.33 or 10.33+0. The next statement will have the statement number 10.33+1, and so forth.

Note, in addition, that the THEN clause of an IF statement and the on-unit clause of an ON statement have separate statement numbers.

For example, consider the line

```
10.33 I=5;  IF J>I THEN I=6;  ELSE GO TO Y;  K=I+J;
```

In this line, the following are the individual statements:

```
10.33+0     I=5
10.33+1     IF J>I
10.33+2     THEN I=6
10.33+3     ELSE GO TO Y
10.33+4     K=I+J
```

3.7 <u>PLACEMENT OF COLLECT STATEMENTS (B)</u>

The user may enter collect statements in any order he wishes. When he does, they will be inserted into his program in the order specified by his line numbers.

3.8 <u>REPLACING COLLECT STATEMENTS (B)</u>

If a user types in a statement with the same line number as a line which already exists in his program, then the new line replaces the old.

CHAPTER 4

PROGRAM MANIPULATION STATEMENTS (B)

You will use the statements in this chapter to manipulate your program, as well as to perform other general functions associated with the CPL system.

You use the LIST statement to list your program, the ERASE statement to erase parts of your program, the NUMBER statement to cause CPL to generate line numbers automatically, the SAVE statement to save your program on disk, the LOAD and WEAVE statements to recall stored programs from disk, the EXECUTE (XEQ) and CONTINUE statements to execute your program, the BREAK and NOBREAK statements to control debugging breakpoints, and the MONITOR statement to leave CPL and return to the monitor.

4.1 THE LIST STATEMENT (DIRECT ONLY) (B)

You use the LIST statement to list your stored program.

Format: LIST [specification [,specification ...]]

where the optional "specification" has the format:

1. line-number
2. line-number THRU line-number
3. THRU line-number
4. line-number THRU ...
5. THRU ...

You use the unmodified LIST command to list your entire program. By means of specifications containing line numbers, you can specify which statements that you want listed. You may use the phrase "THRU ..." to indicate that you wish to list to the end of the program.

For example, the statement

```
LIST 2, 3 THRU 5, 2000.55 THRU ...
```

will list statement 2, all statements in the range 3 to 5, and all statements from 2000.55 to the end of the program.

If you specify a line number which does not exist in the program, then CPL will list the statements immediately preceding and immediately following the specified line number. Thus, for example, to find out the last statement of your program, type "LIST 9999.99."

4.2 THE ERASE STATEMENT (DIRECT ONLY) (B)

The ERASE statement is used to erase parts of your stored program.

Format: ERASE specification [,specification]

where the specification has the same format as for the LIST statement. (But note that here the specification is required.)

This statement is used to erase statements in the stored program.

ERASE is similar to LIST except that it deletes statements rather than listing them. One other difference is that if an invalid line number is specified, then ERASE will take no action other than typing an error message.

You may not erase individual statements in a line. You must erase an entire line.

4.3 THE NUMBER STATEMENT (DIRECT ONLY) (B)

Format: NUMBER line-number [BY increment]

When you are typing in a long program, you may find it annoying to have to enter a line number at the beginning of each line. By means of the NUMBER statement, you can cause CPL to generate line numbers for you.

The "line-number" argument to the statement specifies the first line number which CPL will generate. CPL will generate subsequent line numbers by adding 10.00 to the preceding line number, unless you have specified a different increment in the "BY" clause of the NUMBER statement.

After you have entered the NUMBER statement, CPL will prompt you with line numbers. Every statement entered will be in collect mode.

In order to leave automatic line-numbering mode, do the following: After CPL has prompted you with a line number, type a single "#" character, followed by a carriage return. The terminal will then return to normal mode.

EXAMPLE: The statement NUMBER 3 BY 1.5; will cause CPL to generate line numbers starting with 3 and continuing with 4.5, 6.0, 7.5, etc.

4.4 THE SAVE STATEMENT (DIRECT STATEMENT) (B)

Format: SAVE 'file-id'

This statement is used to save a stored program on disk. It is necessary to specify only the filename in the file-id; the device will default to DSK, and the filename extension will default to CPL.

If you wish, however, you may specify a full file-id, including a device name, filename and extension, and project-programmer number.

You may edit this file on disk using any of the standard DEC editors, and then use the LOAD command to reload the edited file into CPL.

EXAMPLE: The statement "SAVE 'DSKC:TX.XXX[10,4433]';" causes the current collected program to be saved on disk in the specified file.

4.5 THE LOAD STATEMENT (DIRECT ONLY) (B)

FORMAT: LOAD 'file-id' [NUMBER line# [BY incr]]

This statement is used to load the specified program from disk into CPL. Default conventions for the file-id are the same as in the SAVE statement.

The arguments to the NUMBER option, if specified, are the same as the arguments to the NUMBER statement, described above. The NUMBER option specifies how the statements in the file are to be numbered.

If the statements in the file already have line numbers, then the NUMBER option need not be used; if it is used, then the generated line numbers supersede the line numbers in the file. If the statements in the file do not have line numbers, then the NUMBER option must be used.

Before executing the LOAD statement, CPL erases all storage and program variables in the old program, and types out the message "ALL STORAGE RESET" to remind you that your old variables are no longer defined.

4.6 THE WEAVE STATEMENT (DIRECT ONLY) (B)

Format: WEAVE 'file-id' [NUMBER line [BY incr]]

You use this statement when you wish to combine two separate files into one program. The WEAVE statement is like the LOAD statement, except that CPL does not erase your old program before loading the new one.

Usually you will use the NUMBER option with the WEAVE statement, since that is the only way you can control where, in the existing program, CPL will place the new program.

For example, if your existing program runs through statement 2000, then you may load the new program with "WEAVE 'name' NUMBER 2010" to load the new program starting at line 2010.

4.7 THE EXECUTE STATEMENT (DIRECT ONLY) (B)

Abbrev: XEQ for EXECUTE

Format: XEQ [FROM statement-number]

This statement causes your collected program to be executed.

If only XEQ is typed, then execution begins with the first statement of the program. If the FROM option is used, then execution begins with the specified statement number.

When the FROM option is used to specify a specific statement number, and that statement happens to have a breakpoint set, then the breakpoint will not be taken for the first execution of that statement. Of course, if control returns to that statement, then a breakpoint will occur.

Prior to beginning execution of your program, XEQ will perform some initialization functions which are usually meaningful only if you are restarting a program. These initialization functions are:

1. All BEGIN and PROCEDURE blocks are terminated.

2. All files are closed.

3. All storage allocated by ALLOCATE statement for CONTROLLED and BASED identifiers is released. All POINTER variables pointing to such storage are made invalid.

If you wish to keep CPL from performing these initialization functions, then use the CONTINUE statement.

4.8 THE CONTINUE STATEMENT (DIRECT ONLY) (B)

Abbrev: CONT for CONTINUE

Format: CONTINUE [FROM statement-number]

If your program has stopped executing due to an error or to a breakpoint, then you may make the changes you desire and continue executing the program by using the "CONTINUE" statement.

If you specify no FROM clause with the CONTINUE statement, the CPL will restart execution with the statement which contained the error or on which the breakpoint occurred. If you wish to continue from a different statement, you must specify it with the FROM clause.

If you erase or replace the statement which contained the error or which caused the breakpoint, then you must use a FROM clause with the CONTINUE statement.

4.9 THE BREAK STATEMENT (DIRECT ONLY) (B)

Format: BREAK statement-number [,statement-number ...]

This statement specifies a list of one or more statements to be flagged as having "breakpoints." If control passes to any such statement, then CPL will not execute the statement; instead, CPL will type a "breakpoint" message and pass control to you. You can then examine values of variables, set values, and even modify your program.

You may continue execution of your program from the by typing CONTINUE.

If you ERASE or replace a line in your program, then all breakpoints for statements on that line are removed.

4.10 THE NOBREAK STATEMENT (DIRECT ONLY) (B)

Format: NOBREAK [statement-number [,statement-number ...]]

This statement removes breakpoints from specified statements.

If no statement numbers are specified, then all breakpoints in the program are removed.

4.11 THE MONITOR STATEMENT (DIRECT ONLY) (B)

Abbrev: MON for MONITOR

Format: MONITOR

Execution of this statement causes CPL to relinquish control and return you to the monitor level.

To continue from where you left off, you may type the monitor command "CONTINUE".

CHAPTER 5

THE DECLARE AND DEFAULT STATEMENTS (B-D)

This chapter summarizes the formats of these two statements. The meanings of the various data types and attributes are given in later chapters. Therefore, this chapter can be skipped or skimmed and referred to later.

5.1 THE DECLARE STATEMENT (COLLECT ONLY) (B-D)

Abbrev: DCL for DECLARE

The following is the basic format of the DECLARE statement. For the more sophisticated formats, see the later section, "Multiple Declarations and Attribute Factoring."

```
Format:   DECLARE identifier-name [attribute-list] ;
or        DECLARE id-name (dimension-list) [attribute-list];
```

This statement is used to specify a list of attributes or properties that are to be associated with the variable. The attributes specify whether the variable is to be integer or floating point, a character or bit string, an array or scalar, and how and when storage for the variable is allocated.

5.1.1 Data Type Attributes (B)

The following attributes are used to specify data types:

1. FIXED -- to specify that the variable is an integer. (In PL/I terminology, this specifies that the variable is BINARY(35) FIXED.)

2. FLOAT -- to specify that the variable is a floating point number. (In PL/I terminology, this is fully expressed as BINARY(27) FLOAT.)

3. CHARACTER (length) [VARYING] -- to specify that the variable is a character string of the specified (maximum) length.

4. BIT (length) [VARYING] -- to specify that the variable is a bit string of the specified (maximum) length.

5. POINTER -- to specify that the variable is have as a value the address of a data item. This is a "non-computational" data type, in that arithmetic operations cannot be performed on it.

Examples:
```
10.       DECLARE A FIXED ;
20.       DECLARE B FLOAT ;
30.       DECLARE C CHARACTER(15);
40.       DECLARE D BIT(5) VARYING;
```

5.1.2 ARRAYS (B)

Arrays are specified by enclosing the list of subscript ranges in parentheses following the identifier name.

Examples:
```
10.       DECLARE A(10) ;
20.       DECLARE B(5,6);
30.       DECLARE C(0:10) ;
40.       DECLARE D(10) FIXED ;
50.       DECLARE E(5,5) CHARACTER (10) VARYING ;
```

The use of arrays is discussed in a later chapter.

5.1.3 Alternate Method For Specifying Arrays (C)

Instead of placing the list of dimension bounds in parentheses following the identifier name, it is possible to specify the dimension list by means of the attribute DIMENSION(dimension-list).

For example, the following two statements are equivalent:
```
10.       DECLARE A(5,0:6) BIT(10) ;
10.       DECLARE A BIT(10) DIMENSION(5,0:6) ;
```

5.1.4 Storage Class Attributes (D)

The following attributes are used to specify the storage class:

1. AUTOMATIC -- this attribute indicates that storage is to be allocated when the block in which the declaration appears is invoked.

2. STATIC -- this attribute indicates that storage is to be allocated immediately when the declaration is typed in on the terminal.

3. CONTROLLED -- this attribute indicates that allocation is under explicit control of the programmer.

4. BASED -- this attribute specifies that each reference to the identifier will be accompanied by a POINTER qualifier to specify the address of the storage area.

5. PARAMETER -- this attribute specifies that the declaration lies inside a PROCEDURE block, and that the identifier also appears in the parameter list of the PROCEDURE statement. (It is not necessary to specify PARAMETER, unless you have specified "*" for a string length or an array bound.)

5.1.5 Other Attributes (D)

1. BUILTIN -- this attribute indicates that CPL should consider the identifier to be one of its built-in functions or pseudo-variables. Such a declaration is legal only if the identifier is a real CPL built-in function name. Normally you would specifically declare an identifier to have the BUILTIN attribute only if you wish to make this usage clear to another person reading your program. See the chapter on built-in functions for further details on recognition.

2. NONVARYING -- this attribute may be specified with either CHARACTER or BIT. It is the opposite of VARYING, and its specification has no effect since it is the default.

3. The file attributes are described in the chapter on input/output to arbitrary files. These attributes are: INPUT, OUTPUT, STREAM, RECORD, PRINT and and the ENVIRONMENT attributes VFORM, APPEND and NOPAGE.

5.1.6 Default Attributes (B)

If a variable is referenced without appearing in a DECLARE statement, then it will be given the default attributes FLOAT and AUTOMATIC.

(The identifiers SYSIN and SYSPRINT, which are used in conjunction with the GET and PUT statements, are exceptions to the above rule.)

If a variable appears in a DECLARE statement, but no data type is specified, then it will be FLOAT. If it appears in a DECLARE statement but no storage class is specified, then it will be STATIC.

These defaults can be changed by means of the DEFAULT statement.

5.1.7 Multiple Declarations And Attribute Factoring (C)

CPL permits many variables to be declared in the same DECLARE statement. This is done by separating the declarations with commas, as in the following statement:

```
  10.     DECLARE A(10), B CHAR(5) ;
```

It is also possible to "factor" out attributes to simplify the DECLARE statement. For example,

```
  10.     DECLARE (A,B,C) (10) BIT(1) ;
```

is the same as

```
  10.     DECLARE A(10) BIT(1), B(10) BIT(1), C(10) BIT(1) ;
```

Similarly, these two statements have the same effect:

```
  10.     DECLARE (A(10), B VAR) CHAR(5) ;
  10.     DECLARE A(10) CHAR(5), B VAR CHAR(5) ;
```

Finally, attributes can be factored to any level. For example,

```
  10.     DECLARE((A FIXED, B)(10),(C CHAR(5),D BIT(3))VAR)STATIC ;
```

has the same effect as

```
  10.     DECLARE A(10) FIXED STATIC, B(10) STATIC,
                  C CHAR(5) VAR STATIC, D BIT(3) VAR STATIC;
```

5.2 THE DEFAULT STATEMENT (COLLECT ONLY) (D)

Abbrev: DFT for DEFAULT

Format: DEFAULT (range-spec) default-spec ;

where the "range-spec" specifies a group of letters of the alphabet, and default-spec specifies the default attributes. They are described in more detail in the following paragraphs.

This specifies that all identifiers beginning with one of the specified letters have the default attributes.

5.2.1 The Range-spec (D)

The range-spec specifies a letter or group of letters to which the defaults apply. The specification RANGE(letter) gives a single letter for which the defaults will apply. The specification RANGE(letter:letter) specifies a range of letters for which the defaults will apply. The specification RANGE(*) indicates that the default applies to all identifiers.

For example, RANGE(I) indicates that the defaults will apply to all identifiers beginning with the letter I, while the specification RANGE(A:D) indicates that the defaults apply to all identifiers beginning with the letters A through D.

It is possible to combine such range specifications by separating them with ! (the "or" symbol). For example, the specification

```
RANGE(A:D)!RANGE(M)!RANGE(Z)
```

indicates that the defaults will apply to all identifiers beginning with the letters A, B, C, D, M and Z.

5.2.2 The Default-spec (D)

This part of the DEFAULT statement specifies the default attributes that will apply to the identifiers beginning with the specified letters.

The default-spec may contain a data type attribute or a storage class attribute or both. The data type attribute must be either FIXED or FLOAT, and the storage class attribute must be either STATIC or AUTOMATIC.

5.2.3 Conflicting DEFAULT Statements (D)

If two default statements in a program indicate conflicting attributes for the same letter of the alphabet, then the one later in the program takes precedence.

5.2.4 Default Rules In Absence Of DEFAULT Statement (D)

As stated above in the discussion of the DECLARE statement, each CPL program begins with the following implicit DEFAULT statement:

```
          DEFAULT (RANGE(A:Z)) FLOAT,AUTOMATIC ;
```

Any DEFAULT statement which you enter will take precedence over this implicit statement.

5.2.5 Examples (D)

1. The I-N rule

Many programmers are familiar with the rule that all variables beginning with the letters I-N are FIXED, while all others are FLOAT. This rules holds in all FORTRAN implementations, as well as in some IBM PL/I implementations.

You may effect this rule in CPL by inserting the DEFAULT statement

```
          DEFAULT (RANGE(I:N)) FLOAT ;
```

at the beginning of your program.

2. All Variables STATIC

You may be able to save some execution time in your program by making the default for all variables STATIC rather than AUTOMATIC. This is done by means of the statement:

```
          DEFAULT (RANGE(*)) STATIC ;
```

This may save some execution time in your program if your program contains a number of declarations inside BEGIN or PROCEDURE blocks which are frequently invoked. This is true because it is necessary for CPL to allocate AUTOMATIC storage each time the block each time the block is invoked, but STATIC storage is allocated only once.

5.2.6 Overriding Defaults With DECLARE Statement (D)

Of course, the rules specified by a DEFAULT statement are only defaults. If a DECLARE statement specifies a data type or a storage class, then it overrides the data type or storage class specified by a DEFAULT statement.

CHAPTER 6

DATA TYPES (B-D)

Each CPL variable has a "data type." The data type simply specifies what kinds of values the variable can take on. For example, a FIXED variable can take on integer values. A FLOAT variable can take on real numbers as values. A CHARACTER variable can take on strings of characters as values. A BIT variable can take on strings of bits as values. A POINTER variable can take on addresses as values.

Unless you specify otherwise, all CPL variables have the FLOAT data type. You may use a DECLARE statement to specify a different data type for a specific identifier, or you may use the DEFAULT to change the default from FLOAT to FIXED in all or selected cases. These statements are described in a preceding chapter.

6.1 "FIXED" DATA TYPE (B)

A variable having the attribute FIXED can have only integers as values. Its range of values is from -34359738368 to 34359738367.

If a non-integer value is assigned to a FIXED variable, then the value is first "truncated" to an integer value. This means that any fractional part is removed.

For example, the value 3.7 would be truncated to the value 3 before assignment to a FIXED variable. Similarly, the value -20.5 would be truncated to -20. Note that no rounding takes place.

6.2 "FLOAT" DATA TYPE (B)

A variable having the FLOAT data type can assume fractional values.

You represent CPL floating point constants by a form of scientific notation which specifies a mantissa and an "exponent" (power of 10). For example, 23.87E5 stands for the value 2387000 (since the "5" is a power of 10 to be multiplied by 23.87), and 142E-8 stands for the value .00000142 (where here the power of 10 is -8).

FLOAT values may be, in absolute value, as high as 1.7014118E+38.

6.3 "CHARACTER" DATA TYPE (C)

Abbrev: CHAR for CHARACTER, VAR for VARYING

A variable having this data type does not take numeric values. Instead, the value of such a variable will be a character string -- a string of ASCII characters.

In CPL, character string constants are enclosed in single quotes. An example is 'CHARACTER-STRING-VALUE'. If a character string constant is to contain the character "single-quote", it is represented by two single quotes, as in the example 'DON''T GO'.

You may precede a character string constant by a parenthesized integer. This integer is called a "repetition factor" and specifies that the character string constant is to be repeated that number of times. For example, the character string constant (5)'AB' is the same as 'ABABABABAB'.

When a character string variable is declared in a DECLARE statement, you must make two choices:

1. You must choose the length of the character string values the string can assume.
2. You must decide whether the variable is to have VARYING or NONVARYING length.

For example,

```
10.       DECLARE A CHAR(20), V CHAR(30) VARYING ;
```

specifies that A and V are to be character string variables. The variable A is NONVARYING (the default), and its length is 20, meaning that the value of A will always be a string of exactly 20 characters.

The variable V in the above example is VARYING and has a length of 30. In this case, the length should be interpreted as a maximum length -- V can equal a string of characters of length 30 or less. This length can even be zero, in which case the value is called the "null character string."

6.4 "BIT" DATA TYPE (C)

In many ways, the BIT data type is similar to the CHARACTER data type. Variables with the BIT data type take as values strings of bits -- that is, each element of the string must be a 0 or 1.

In CPL, a BIT string constant is written by enclosing a string of 0's and 1's in single quotes, followed by the letter "B". For example, the BIT string constant '1101001'B represents a string of 7 bits.

Notice that that BIT strings are represented by numbers in the binary number system. CPL permits you to specify the same bit strings by means of numbers in the base 4, octal, and hexadecimal number systems. To specify one of these systems, follow the quoted digit string in the base 4, octal or hexadecimal number base with B2, B3 or B4, respectively.

For example, the string '12322'B2 is the same bit string as '0110111010'B, '37402'B3 is the same as '011111100000010'B, and '1ABF2C'B4 is the same as '000110101011111100101100'B.

You may precede a bit string constant by a parenthesized integer. This integer is called a "repetition factor" and specifies that the bit string constant is to be repeated that number of times. For example, the bit string constant (4)'1011'B is the same as '1011101110111011'B.

As in the case of CHARACTER strings, you must make two choices when declaring BIT string variables:

1. The number of bits (length) of the BIT string variable, and

2. Whether the length will be VARYING or NONVARYING.

If the BIT string is NONVARYING (the default), then it will always contain the number of bits specified by the length. If it is VARYING, then it will contain at most the number of bits specified in the length.

6.5 "POINTER" DATA TYPE (D)

This is a "non-computational" data type. It is described in the chapter entitled "BASED storage and POINTERs."

CHAPTER 7

THE ASSIGNMENT STATEMENT AND CPL EXPRESSIONS (B-C)

7.1 THE ASSIGNMENT STATEMENT (DIRECT OR COLLECT) (B)

```
Format:   variable = expression ;
or        variable,variable [,variable ...] = expression ;
```

This statement causes the expression on the right-hand side of the equal sign to be evaluated, and the result assigned to the variable or list of variables appearing on the left-hand side of the equal sign.

7.2 CPL EXPRESSIONS (B)

Expressions are combinations of constants, variables and operators which specify that CPL is to make certain computations. For example, the assignment statement:

```
A=B+C-5 ;
```

causes CPL to subtract the constant 5 from the sum of the variables B and C, and store the result into the variable A.

7.3 DATA CONVERSIONS (C)

When an expression is being evaluated, it may be that an operation cannot be performed since the quantities being operated upon are the wrong data type. For example, the expression A+B cannot be directly computed if the variables A and B happen to be CHARACTER strings.

Whenever such an event occurs then CPL automatically converts the operands to the correct data types. These conversions are additional operations that take place automatically before the desired operation can take place.

For example, if A+B is to be computed, where A and B are CHARACTER string variables, then each of the CHARACTER strings must be converted to FIXED before the addition can take place.

These conversions take place according to specific rules and with specific restrictions, as described in a later chapter, entitled "Conversions among Computational Data Types."

7.4 CPL EXPRESSION OPERATORS (B-C)

This sections lists all the CPL operators. It tells what each operator does, what data types are required by each operator, and how conversions are made, if necessary.

7.4.1 +, - And * (Addition, Subtraction And Multiplication) (B)

Each of these operators takes two operands -- two numbers to be added, subtracted or multiplied. Either both operands must be FIXED, in which case the result of the operation is FIXED, or both operands must be FLOAT, in which case the result of the operation is FLOAT.

If you specify operands which are not both FIXED or both FLOAT, then CPL must convert one or both operands. The rules are as follows: If either one of the operands is FLOAT, then CPL will convert the other to FLOAT; otherwise, CPL will convert both operands to FIXED, unless they are already FIXED.

7.4.2 / (Division) (B)

This operation follows the same rules as addition, subtraction and multiplication, with the following exception: At least one of the operands must be FLOAT (so that the other operand will also be converted to FLOAT).

It is illegal in CPL to divide one FIXED value by aonther. If you wish to divide two FIXED values, then you can do one of the following:

1. You may force conversion of the numerator (or denominator) to FLOAT, so that CPL will perform floating point division. (For example, instead of specifying I/J, you specify (0E0+I)/J.) You may then assign the floating point result to a FIXED variable to provide truncation.

2. You may use the DIVI or DIVF built-in functions. DIVI(I,J) will return a truncated FIXED quotient, while DIVF(I,J) will return the FLOAT quotient.

   WARNING: The functions DIVI and DIVF are not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

NOTE: Full PL/I, as specified by the ANSI PL/I standard, permits FIXED division, but specifies it in such a way that it cannot be implemented correctly in any system which does not support scaled FIXED data types. Since CPL could not support FIXED division correctly, we decided not to permit it at all. The DIVI and DIVF functions represent a kind of compromise. It is also worth noting that FIXED division in full PL/I is very error-prone in use, and can lead to unexpected errors. The most famous example is that the simple statement, "A=25 + 1/3;" will, according to the standard, cause your program to abort with a FIXEDOVERFLOW error interrupt.

7.4.3 ** (Exponentiation) (B)

This operator takes two operands. The first operand is raised to the power of the second operand. The result is always FLOAT.

For example, 2**3 returns the value 8, while X**.1 returns the 10'th root of X.

7.4.4 Comparison Operators (B)

There are eight comparison operators, as follows:

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
^<	No less than (same as >=)
^>	Not greater than (same as <=)
=	Equal to
^=	Not equal to

Each of these comparison operators takes two operands. The operands may be of any computational data type, but they must both be of the same data type. (Conversion rules are described below, in case data types are different.)

The operands are compared according to the specified comparison operator, and a BIT(1) value is returned. If the comparison yields "true," then the value '1'B is returned. If the comparison yields "false," then the value '0'B is returned.

7.4.4.1 Conversions For Comparisons (C) - The comparison operators require that both operands be of the same data type. If they are not of the same data type, then the one of the "lower" data type is converted to the "higher" data type. The computational data types, in order from "highest" to "lowest," are as follows:

1. FLOAT
2. FIXED
3. CHARACTER
4. BIT

For example, if the two operands are FLOAT and CHARACTER, then CPL converts the CHARACTER operand to FLOAT.

7.4.4.2 Character String Comparisons (C) - When CPL compares two character strings it compares them on a character-by-character basis as follows: First, if the strings are of unequal length, then CPL pads the shorter one with blanks on the end until it is as long as the longer one. Then CPL takes characters from the first string and compares them, one by one, with corresponding characters from the second string, until two corresponding characters are found to be unequal. CPL compares these two characters according to the standard ASCII collating sequence. The string which contains the greater character, according to this collating sequence, is the greater character string.

If CPL finds no unequal corresponding characters, then the two character strings are equal.

EXAMPLES:

1. 'XYZ' is equal to the character string 'XYZ   '.

2. 'ABCDZZZZZZZZ' is smaller than 'ABCEAAAA'.

7.4.4.3 BIT String Comparisons (C) - The method of comparison is similar to that for CHARACTER strings. CPL pads the shorter string by adding 0's to the end of it, until it is as long as the longer string. Then CPL compares the strings on a bit-by-bit basis, until two unequal corresponding bits are found. When these bits are found, then the BIT string containing the 0-bit is considered smaller than the BIT string containing the 1-bit.

7.4.4.4 POINTER Comparisons (D) - These are described in detail in the chapter on BASED storage and POINTERs. POINTERs can be tested only for equality or inequality (= or ^=).

7.4.5 + And - (Prefix Plus And Minus) (B)

These two operators take a single operand. the prefix minus operator negates a FIXED or FLOAT operand. The prefix plus operator has no effect on a FIXED or FLOAT operand.

If the operand of the prefix plus or minus operator is CHARACTER or BIT, then it is converted to FIXED.

7.4.6 & And ! (Logical And and Or) (B)

These two operators require BIT operands. The shorter of the two operands is padded with zeros to the length of the other operand. The resulting BIT string is the same length as the operands, and is formed by taking the logical And or Or of corresponding bits.

EXAMPLE: '1101'B&'0110111'B yields the result '1000000'B, while '1101'B!'0110111'B yields '1111111'B.

7.4.7 ^ (Logical Not Operator) (B)

This operator takes a single BIT string operand. The resulting BIT string is the same length as the operand, with each 0-bit changed to a 1-bit and each 1-bit changed to a 0-bit.

7.4.8 !! (String Concatenation Operator) (C)

This operator requires two operands, both of which must be either CHARACTER strings or BIT strings. The two strings are combined to form one long string consisting of all characters or bits of both strings.

If both operands are BIT strings, then BIT string concatenation takes place; otherwise, both operands are converted to CHARACTER, if necessary, and CHARACTER string concatenation takes place.

EXAMPLE: 'ABC'!!'12345XYZ' yields the result 'ABC12345XYZ'.

7.5 PRECEDENCE OF OPERATORS (B)

You may use parentheses to specify any operator order that you desire. For example, in the expression A*(B+C), CPL will perform the addition before the multiplication.

However, whenever you do not use parentheses to specify an operator priority, then CPL evaluates the operators according to a priority algorithm. The operators are arranged in the following priority groups, from highest to lowest:

1. ^, **, prefix +, prefix -
2. *, /
3. Infix +, infix -
4. !!
5. All comparison operators (<, >, >=, <=, ^>, ^<, =, ^=)
6. &
7. !

Unless you override the priorities with parentheses, operators of higher priority are performed before operators of a lower priority. Within each group, operators have equal priority. Operations from the same group are performed in the sequence in which they appear in a PL/I statement -- that is, from left to right -- except that operations in Group 1 are performed from right to left.

EXAMPLE: The numbers show the order in which the operations are performed in the following expression:

```
A< -B+(C/(D*E/F))**G ! H^=I & J>K
 7 5 6  3  1 2   4    11 8   10 9
```

In general, an operation can be performed only after its operands have been evaluated.

When in doubt, parenthesize!

CHAPTER 8

LABELS AND "GO TO" AND "IF" STATEMENTS (B)

As a general rule, when CPL finishes executing one statement in a program, it moves to the next statement. Some statements, however, modify this normal flow. The GO TO and IF statements are such statements, as is the DO statement discussed in the next chapter.

8.1 STATEMENT LABELS (COLLECT ONLY) (B)

```
Format:   ident:  statement;
```

The identifier "ident" is a statement label for the "statement". Any statement except DECLARE and DEFAULT may be labeled. The PROCEDURE statement must be labeled.

A statement may have more than one statement label. For example, in the line:

```
120.      LAB1:  LAB2:  STOP;  XYZ:  I=5;
```

the STOP statement has two statement labels, LAB1 and LAB2, and the assignment statement has one label, XYZ.

A PROCEDURE statement may have only one label.

8.2 THE GO TO STATEMENT (COLLECT ONLY) (B)

```
Format:   GO TO label;
or        GOTO label;
```

Here "label" is a statement label which appears elsewhere in the same program.

When the GO TO statement is executed, it causes control to pass to the statement with the specified statement label.

8.3 THE IF STATEMENT (COLLECT ONLY) (B)

```
Basic format:  IF expr THEN stmt1;  ELSE stmt2;
```

CPL evaluates the logical expression "expr." If the result is true, then CPL executes "stmtl"; if the result is false, then "stmt2" is executed.

In the IF statement, you will probably be using the comparison and logical operators. Typical logical expressions are "A>B" and "A>2&B=3". These express relationships among variables whose values have already been computed, and return what may be called a "truth value." (A truth value is really a BIT(1) value. BIT(1) values will be discussed below.)

For example, the statement

```
          IF A>B THEN GO TO XYZ;  ELSE GO TO UVW;
```

causes CPL to compare the values of A and B. If A is larger, then control transfers to the statement labeled XYZ; if A is less than or equal to B, then control transfers to the statement labeled UVW.

Any statement may be used in the THEN or ELSE clauses. For example, suppose you wish to add to I the maximum of J and K. You can do it with the statement

```
          I=I+MAX(J,K);
```

but you can also do it with the statement

```
          IF J>K THEN I=I+J;  ELSE I=I+K;
```

Note that no matter which of the alternatives is taken, execution will continue with the statement following the ELSE clause in the program. (This is usually the statement on the next line.)

8.3.1 Omitting The ELSE Clause (B)

If desired, you may omit the ELSE clause from your IF statement. For example, suppose your program executes

```
          IF A>B & C=2 THEN GO TO LAB1;
```

If A is greater than B and C equals 2 then control will pass to statement LAB1; otherwise, control will pass to the statement following the IF statement.

8.3.2 Separating The ELSE Clause (B)

The ELSE clause need not appear on the same line as the IF statement.

8.3.3 Nested IF Statements (B)

The THEN or ELSE clause of an IF statement may itself be another IF statement. An example is the following group of statements:

```
IF A>B THEN IF C=5 THEN GO TO XYZ;
                   ELSE GO TO ABC;
    ELSE IF A=B THEN IF C=5 THEN GO TO UVW;
              ELSE GO TO MNO;
```

When matching up ELSE clauses to IF statements, CPL will match up an ELSE clause with the last unmatched IF statement.

8.3.4 DO/END Groups As THEN/ELSE Clauses (B)

If you wish the THEN or ELSE clause of an IF statement to consist of several statements, you may use a DO/END group. DO statements will be discussed in detail in a later chapter; here we will illustrate a simple example of their usage.

The general format is:

```
IF expr THEN DO;  ...;  END;
 ELSE DO;  ...;  END;
```

The two occurrences of "..." in the above format stand for any group of statements. The first group will be executed if the "expr" is true, and the second group will be executed if the "expr" is false.

Of course, all of the above statements can be split up onto separate lines.

8.3.5 BIT Values In The IF Expression (D)

Recall that the format of the IF statement is "IF expr THEN stmt1; ELSE stmt2;". "Expr" is a logical expression which is, in all the examples, an expression with a BIT(1) value -- that is, an expression with a value of '0'B (for "false") or '1'B (for "true").

Any BIT expression can be used as the first argument to the IF statement. If the BIT string contains any 1-bit, then the THEN clause will be taken. But if the BIT string is the null BIT string, or if it contains only 0-bits, then the ELSE clause, if any, will be taken.

CHAPTER 9

THE DO AND END STATEMENTS (B-D)

The DO and END statements enclose and identify a group of statements which you wish to think of as a single unit. The DO and END statements also provide a means for you to specify that this group of statements is to be executed more than once (iteratively).

Terminology: The term "DO-group" will refer, throughout this manual, to a DO statement and its corresponding END statement and to the statements they enclose.

9.1 THE NON-ITERATIVE DO-GROUP (COLLECT ONLY) (B)

Format: DO; ... END;

This form of the DO statement is generally useful only to specify a group of statements to be used as a THEN clause or ELSE clause of an IF statement. This usage has been discussed in the description of the IF statement.

This simple form of the DO statement specifies that the group of statements is to be executed once.

9.2 THE WHILE-ONLY DO GROUP (COLLECT ONLY) (B)

Format: DO WHILE(exp); ...; END;

The expression is a "logical expression" with the same properties as the expression which serves as the first argument to the IF statement. The expression is evaluated. If its value is "false," then the group of statements is not executed at all; instead, control transfers immediately to the statement following the END statement.

If the value of the expression is "true," then the group of statements is executed. After the group has executed, the expression in the DO statement is re-computed. If it is now "false," then control transfers to the statement following the END statement. But if it is still "true," then the group of statements is executed again. The group of statements is executed over and over until evaluation of the expression in the WHILE clause yields a value of "false."

EXAMPLE: Suppose file DSK:F.DT contains input values, and we wish to find the first input value which is greater than 100. We can use the following statements:

```
10.     DECLARE F FILE;
20.     I=0;
30.     DO WHILE (I<=100);
40.     GET FILE(F) LIST(I);
50.     END;
```

Here there is only a single GET statement in the DO-group. This statement will be executed over and over until it yields a value of I which exceeds 100. (The GET statement is described in another chapter. To understand this example you need only know that statement 40 above will read a single value from file F and place its value into the variable I.)

9.3 THE DO-GROUP WITH DO VARIABLE (COLLECT ONLY) (B-C)

```
Format:   DO vble = spec [,spec ...] ;
          ...; END;
```

The "vble" is a scalar variable, or a subscripted array element, which is to take on specified values as the statements in the DO-group are repetitively executed. The variable is called the "DO-group variable" or "DO-loop variable."

The "spec" is a specification which defines what values the DO-group variable is to be assigned, and how many times the DO-group of statements is to be executed.

Before proceeding to the general definition of this specification, let us consider the following:

EXAMPLE: The following program will print out a table of the square roots of all numbers from 1 to 50.

```
10.     DECLARE A FLOAT;
20.     DO I = 1 TO 50;
30.     A = SQRT(I);
40.     PUT SKIP LIST(I,A);
50.     END;
```

Statement 20. of the above program requests the following: Execute all statements in the DO group 50 times. During the first iteration, I will have the value 1. For each subsequent iteration, the value of I will be incremented by one, so that the DO-group will be executed with I having the values 2, 3, 4, and so forth. On the last iteration, I will equal 50. It will then be incremented once more -- to 51. As soon as the program finds this value greater than the TO value (50), it will end execution of the DO group and will pass control to the statement after END.

The portion "1 TO 50" of statement 20 is a "spec," which provides the following information:

1. The values which the DO-group variable will assume: 1, 2, 3, ..., 50, and, by implication, the number of times that the DO-group will be executed.

2. A condition for determining whether the DO-group should be executed one more time. In this case, the test is whether the value of I has exceeded 50.

Sometimes, the "spec" will specify the first of the above, but not the second. In that case, the DO-group will never terminate except by "abnormal" means. (Abnormal DO-group terminations are discussed in a later section, "Normal and Abnormal Termination of DO Groups.")

9.3.1 The Format Of The "spec" (B)

The format of a "spec" is the following:

```
initial-expression [iteration-part] [WHILE(exp)]
```

where the optional "iteration-part" has one of the following formats:

1. BY expression

   The DO-group variable assumes the "initial-expression" value for the first iteration of the group. For each subsequent iteration, the value of the variable is incremented by the value of the "by-expression" (as the expression following the BY keyword is called). If the value of the by-expression is negative, then the DO-variable will be decremented rather than incremented.

   This form of the "spec" provides no means for normal termination (unless a WHILE clause is present -- this will be discussed later).

2. BY expression TO expression

   For the first iteration of the DO-group, the variable will assume the value of the "initial-expression." For subsequent iterations, the value of the DO-group variable will be incremented by the value of the by-expression (or decremented, if the by-expression is negative).

   This form of the "spec" provides a test for termination of the DO-group. The DO-group will be executed only as long as the DO-variable does not exceed the value of the to-expression.

   Exception: If the value of the by-expression is negative, then the statements in the DO-group will be executed only as long as the DO-variable is larger than or equal to the value of the to-expression. the to-expression.

3. TO expression BY expression

   This case has the same effect as the preceding case.

4. TO expression

   This case is equivalent to "BY 1 TO expression".

5. REPEAT(expression)

   The REPEAT option allows you to vary the value of the DO-variable according to a specific formula. This is very useful when you wish the sequence of DO-variable values to be other than an arithmetic series.

This form of the iteration specification is quite different from that of the others. For the first iteration of the DO-group, the DO-variable will assume the value of the "initial expression." After each iteration of the DO-group, CPL recomputes the value of the repeat-expression (as the "expression" following the REPEAT keyword is called), and assigns that value as the new value of the DO-variable. This form of the "spec" provides no test for termination (unless a WHILE clause is also specified, as described below).

EXAMPLES.

1. `DO I = 2 BY 3;  ...;  END;`

   The DO-variable I will assume the values 2, 5, 8, 11, ... The iterations will never terminate normally.

2. `DO I = 2 BY 3 TO 18;  ...;  END;`

   I will assume the values 2, 5, 8, 11, 14, and 17. Then I will be set to 20, which exceeds the value, 18, of the to-expression, so that control will pass to the statement following the END statement.

3. `DO I = 2 TO 5;  ...;  END;`

   The statement group will be executed 4 times, with I having the values 2, 3, 4 and 5.

4. `DO I = 5 TO 2 BY -1;  ...;  END;`

   The statement group will be executed 4 times, with I assuming the values 5, 4, 3 and 2.

5.
   ```
   DECLARE A FLOAT;
   DO A = 2.3 TO 6.3 BY 1.34;  ...;  END;
   ```

   The FLOAT DO-variable will assume the values 2.3, 3.64, and 4.98.

6. `DO I = 1 REPEAT(2*I);  ...;  END;`

   The variable I will assume the values 1, 2, 4, 8, 16, 32, ... The DO-group will not terminate normally.

9.3.2 Completely Unsatisfied Specifications (B)

Consider the following program segment:

```
J = 0;
DO I = 1 TO J;
...; END;
```

Since the value of J is 0, the second statement is equivalent to "DO I=1 to 0". In this event, the initial value, 1, is assigned to the DO-variable I, and then, since I exceeds the value, 0, of the to-expression, control passes immediately to the statement following the END statement. The statements in the group are not executed even once.

9.3.3 The Effect Of The WHILE Clause (B)

You use the WHILE clause with an iteration specification in much the same way you used it in the WHILE-only DO statement described before.

As stated above, the "iteration specification" may or may not specify a test for ending the iterations. The WHILE clause, if specified, provides such a test.

If the iteration specification provides a test, then the iterations will stop if either of the tests indicates stopping.

Here are some examples:

1.
   ```
   J=0;
   DO I = 1 TO 10 WHILE(J < 17);
   J = J + I;
   END;
   ```

   The group will be executed with I equal to 1, 2, 3, 4, 5 and 6. Then I will be assigned the value 7, but J will have the value 21 (=1+2+3+4+5+6), and so control will pass to the statement following the END statement. (The reader should work through this short program and verify for himself that J will end up with the value 21.)

2. `DO I = 1 REPEAT(2*I) WHILE(I<=32);   ...;   END;`

   I will be assigned the values 1, 2, 4, 8, 16, and 32. When I is assigned the value 64, the test in the WHILE clause will fail, and the loop will be terminated.

3. `DO I = 23 WHILE(A=B);   ...;   END;`

   I is assigned the value 23. If A=B, then the statement group will be executed once and only once. If A and B are unequal, then control will pass immediately to the statement following the END statement.

9.3.4 Multiple Specifications (B)

The format for this type of DO statement allows more than one specification to be given.

If you enter more than one specification, then CPL will complete each specification in turn, and then move on to the next one. When the last specification is completed, then control will pass to the statement following the END statement.

Here are some examples:

1. `DO I = 2,3,15,25,-3,66;  ...;  END;`

   The DO-variable I will assume the values shown in the order shown. This is a particularly convenient form of the DO statement when there is no simple formula for the sequence of values which you wish the DO-variable to assume.

2. `DO I = 1 TO 5, 3 TO 8 BY 2, 32 REPEAT(.5*I) WHILE(I>=1); ...; END`

   The statement group will be executed 14 times, with I assuming the values 1, 2, 3, 4, 5, 3, 5, 7, 32, 16, 8, 4, 2 and 1.

9.3.5 DO-variable With Non-arithmetic Data Type (C)

It is legal for the DO-variable to have any of the data types FIXED, FLOAT, CHARACTER, BIT or POINTER.

However, there is a restriction if the data type is not FIXED or FLOAT. In this case, the TO clause and the BY clause are illegal. The REPEAT and WHILE clauses are legal.

EXAMPLE:

```
DECLARE C CHAR(30) VARYING;
DO C='123' REPEAT(C!!'ABC') WHILE (LENGTH(C)<10);
...; END;
```

Here the DO-variable is the CHARACTER variable C. C will assume the values '123', '123ABC', and '123ABCABC'. Then C will be assigned the value '123ABCABCABC', but since the WHILE clause will not be satisfied (since the LENGTH(C) is 12), control will then pass to the statement following the END statement.

9.4 EXPRESSION EVALUATION IN DO-STATEMENT CLAUSES (D)

It may be of interest to some programmers to understand how and when the various expressions in the DO statement are evaluated. The following rules apply:

1. Each "spec" is fully terminated before any expression in a following "spec" is evaluated. The following rules discuss order of expression evaluation within a single "spec".

2. First, the "initial expression" is evaluated, and the value assigned to the DO-variable.

3. If the spec contains a to-expression or a by-expression, then that (those) expression(s) is (are) evaluated, and the values saved. They are not again evaluated for as long as that DO group is active.

4. If the spec contains a REPEAT clause, then that expression is evaluated each time the value of the DO-group variable is reassigned.

5. If the spec contains a WHILE clause, then that expression is evaluated each time that the DO-variable is assigned or reassigned, whether by means of the initial expression, by means of the by-expression, or by means of the REPEAT option.

Here is an example:

```
J = 10;
DO I = 1 TO J;
...  ;  J = 5 ;  ...;  END;
```

When the DO statement is first executed, the value of J is 10. This value is computed and saved, so the fact that the value of J changes inside the DO group will have no effect on the number of iterations.

9.5 NORMAL AND ABNORMAL TERMINATION OF DO GROUPS (C)

An active DO group can terminate in two ways:

1. The specifications in the clauses of the DO statement can be satisified. For example, in the DO statement "DO I=1 TO 10" the DO group will terminate when the value of I reaches 11.

2. The execution of a GO TO statement can transfer control out of the DO group.

The first of these terminations is called "normal," and the second "abnormal."

9.6 THE END STATEMENT (COLLECT ONLY) (B)

Format: END [identifier];

This statement is used to terminate DO groups in the manner described in this chapter. It is also used to terminate BEGIN blocks, PROCEDURE blocks and ON-units, in a similar manner, as will be described in later chapters.

9.6.1 Multiple Closure Of DO Groups (C)

The Format of the END statement given just above shows that an identifier may follow the END keyword. This format may be used when you wish to use one END to terminate several DO groups.

Consider the following program statements:

```
A:      DO I = 1 TO 10;
        DO J = 1 TO 10;
        ....
        END A;
```

The END statement specifies an identifier A. CPL will search all preceding unmatched DO statements for one with the label A. CPL will terminate not only that DO group, but also all unmatched DO groups following that one. CPL does this by inserting dummy END statements in your program.

The same method can be used to close multiple BEGIN and PROCEDURE statements, but it is not recommended that you do this.

9.6.2 GOTO To Multiple Closure END Statement (D)

If a multiple closure END statement has a statement label, then all dummy END statements are inserted before the statement label. This means that if a GOTO statement in one of the inner DO groups transfers control to the multiple closure END statement, then CPL terminates the inner DO groups abnormally, and transfers to the END statement for the outer DO group.

EXAMPLE: Consider the program segment

```
A:      DO I = 1 TO 10;
        DO J = 1 TO 10;
        ....
        GO TO DOEND;
        ....
DOEND:  END A;
```

When CPL executes the GOTO statement, it terminates the inner DO group abnormally, transfers to the END statement of the outer group.

CHAPTER 10

ARRAYS (DIMENSIONED VARIABLES) (B)

Normally a CPL identifier stands for only a single data value -- a single FIXED or FLOAT number, or a single CHARACTER or BIT string.

However, you may also use an identifier to stand for more than one data value. You would need this capability if, for example, you wished to use a variable to represent a mathematical vector or matrix.

10.1 ONE-DIMENSIONAL ARRAYS (VECTORS) (B)

Suppose your program contains the following declaration:

```
          DECLARE A(10) FLOAT;
```

This declaration specifies that the identifier array is to stand for 10 FLOAT data values, rather than the usual 1.

You refer to these 10 values individually by means of a "subscript." A subscript is a number or expression enclosed in parentheses following the dimensioned identifier. In the case of the the identifier A described above, you can refer to the 10 elements of the array A with A(1), A(2), A(3), ..., A(9), A(10). For example, the statement A(2)=5 sets the second array element to 5, and the statement PUT LIST(A(3)) types out the third element of the array.

Any expression may be used as a subscript. For example, if I has the value 5, then the statement A(I+2)=-10 will set A(7) to the value -10.

EXAMPLE: The sequence 1, 1, 2, 3, 5, 8, 13, ... is called the sequence of Fibonacci numbers. After the first two, each number is the sum of the preceding two.

Here is a program, using arrays, which prints out the first 50 Fibonacci numbers:

```
  10.     DECLARE FIB(50) FIXED;
  20.     FIB(1)=1; FIB(2)=1;
  30.     PUT LIST(FIB(1),FIB(2));
  40.     DO I = 3 TO 50;
  50.     FIB(I) = FIB(I-1) + FIB(I-2);
  60.     PUT LIST(FIB(I));
  70.     END
```

At the completion of the above program, the array FIB will have been assigned the values of the first 50 Fibonacci numbers.

10.2 LOWER BOUNDS OTHER THAN 1 (B)

In the declaration

```
          DECLARE A(10) FIXED;
```

the subscript values extend from 1 to 10. That is, the lower bound of the subscript values is 1, and the upper bound is 10.

The upper bound value 10 was specified in the declaration. There is also a way to specify the lower bound, if you do not wish it to be 1.

For example, in the declaration,

```
          DECLARE B(-3:12) FIXED;
```

the subscripts may range from -3 to +12; that is, the lower bound is -3 and the upper bound is 12.

10.3 OTHER DATA TYPES FOR ARRAYS (B)

Any computational data type may be used for an array. For example, the declaration

```
          DECLARE ST(10) CHAR(23) VAR;
```

defines the identifier ST to stand for 10 strings of the type CHAR(23) VARYING.

Any of the data types FIXED, FLOAT, CHAR [VARYING] and BIT [VARYING] may be used.

10.4 TWO-DIMENSIONAL ARRAYS (MATRICES) (B)

Consider the declaration,

```
          DECLARE CC(4,5) FIXED;
```

This declaration specifies that CC is a two-dimensional array. There are 20 (=4x5) elements in the array. You refer to the individual array elements by using two subscript values in parentheses: CC(1,1), CC(1,2), CC(1,3), CC(1,4), CC(1,5), CC(2,1), ... CC(2,5), CC(3,1), ... CC(5,5).

EXAMPLE: Suppose A is a 3x5 matrix, B is a 5x6 matrix, and C is a 3x6 matrix. Suppose you wish to set the matrix C to the matrix product of A and B. Here is a program segment which will do that:

```
10.     DECLARE A(3,5), B(5,6), C(3,6);
20.     DO I = 1 TO 3;
30.     DO J = 1 TO 6;
40.     C(I,J)=0 /* INITIALIZE C(I,J) */;
50.     DO K = 1 TO 5;
60.     C(I,J)=C(I,J)+A(I,K)*B(K,J) /*ADD NEXT TERM*/;
70.     END;
80.     END;
90.     END;
```

10.5 ARRAYS OF MORE THAN TWO DIMENSIONS (B)

The declaration

```
DECLARE D(3,4,8,3:5) FIXED;
```

specifies a four-dimensional array. The subscripts in the first position run from 1 to 3, in the second position from 1 to 4, in the third position from 1 to 8, and in the fourth position from 3 to 5.

Only in exceptional situations would you use more than 4 dimensions. However, if needed, CPL allows up to 256 dimensions.

CHAPTER 11

GET LIST AND PUT LIST TO TERMINAL (B-C)

11.1 THE PUT STATEMENT (COLLECT OR DIRECT) (B)

FORMAT: PUT [SKIP] LIST(exp1,exp2,...);

You may put one or more expressions in the list of expressions. CPL computes each expression, and prints its value on the terminal.

If you specify the SKIP option, then CPL prints a carriage return and line feed before printing the values of the expressions.

EXAMPLE: For PUT LIST('THE ANSWER IS',2+2); CPL will type "THE ANSWER IS 4" on the terminal.

11.2 THE ? STATEMENT (COLLECT OR DIRECT) (B)

FORMAT: ?exp1,exp2,...;

This statement is a short form of the PUT LIST statement.

11.3 THE GET STATEMENT (COLLECT OR DIRECT) (B)

FORMAT: GET LIST(vble1,vble2,...);

When CPL executes this statement, it stops computing and waitings for you to type in values for the specified variables.

When you type in values, use the same format as that of a constant. You may precede a FIXED or FLOAT numeric constant by an optional + or - sign, with no intervening blanks.

You may terminate the constant by one or more blanks, a comma, a comma with one or more blanks, or a carriage return - line feed.

11.3.1 Data Type Conversions (C)

The constant which is entered in response to a GET LIST statement should have the same data type as that of the variable to which the constant is to be assigned. If the constant has a different data type then a conversion will be attempted. For example, a FIXED constant will be converted to a FLOAT constant, and vice-versa.

The full rules for data type conversions among strings in these circumstances are described in a later section.

11.3.2 Omitted Data Values (C)

When CPL stops computing to wait for you to enter a value in response to a GET LIST statement, you may just type in a comma or a carriage return-line feed for any variable in the GET LIST statement. This will indicate to CPL that you wish to leave the current value of that variable unchanged.

11.4 VARIABLE FORMAT FOR PUT LIST (C)

In the default case, a CPL PUT LIST statement produces output for FIXED and FLOAT expressions in a variable format. That is, the format of the output depends upon the value of the quantity being typed out.

This variable format does not comply with the ANSI PL/I standard.

If you would like your program output to be fully ANSI standard, then you should add the following statement to the beginning of your program:

```
 1.        DECLARE SYSPRINT PRINT;
```

This statement will cause CPL to override the variable format option and use the ANSI standard option.

CHAPTER 12

GET AND PUT WITH STRING OPTION (C)

12.1 PUT WITH "STRING" OPTION (C)

```
Format:     PUT STRING(vble) LIST(exp-list);
or          PUT STRING(vble) EDIT(exp-list)(format);
```

where "vble" is an unsubscripted identifier which is scalar (i.e., not an array) and of CHARACTER or CHARACTER VARYING data type.

CPL computes and converts the expressions in the same manner as for the PUT statement to the terminal. Then it places the output into the specified CHARACTER string variabe instead of to the terminal.

12.2 GET WITH "STRING" OPTION (C)

```
Format:     GET STRING(expr) LIST(vble-list);
```

The expression "expr" is computed and converted to CHARACTER if necessary. Input is taken from the character string rather than from the terminal.

A frequent error in GET LIST with the STRING option is to forget to leave a blank or a comma in the source character string after the last data item in the string. For example, if CPL executes the statement

```
GET STRING('2 23 32') LIST(I,J,K);
```

then CPL will stop execution with an error message indicating that the terminating blank or comma for the data item "32" was not found.

CHAPTER 13

THE PUT EDIT STATEMENT (C-D)

13.1 INTRODUCTION (C)

The PUT LIST statement, already discussed, gives you very little control over the format of output. The output format follows rules set up by CPL.

PUT EDIT lets you control the format. You can control exactly where on the line the output will appear. You can also control the position of the decimal point and whether an "E" type exponent will appear.

If you do not need formatted output, then you may skip this chapter.

13.1.1 Basic Format (C)

The simplest format is:

```
Format:    PUT EDIT(output-list)(format-list) ;
```

The "output-list" is a list of expressions. The format is the same as the list of expressions in PUT LIST.

The "format-list" is a list of format items, separated by commas, which specifies how the data is to be printed.

13.1.2 Example (C)

Consider the statement

```
           PUT EDIT(23,45)(F(5),E(12,3)) ;
```

There are two expressions (23 and 45) and two format items (F(5) and E(12,3)). 23 is printed in the format F(5) and 45 is printed in the format E(12,3).

F(5) format specifies that the number is to be printed as an integer in a field 5 characters wide. Therefore, the characters bbb23 will be printed, where "b" stands for a blank.

E(12,3) format specifies the E-type format. There will be 3 digits following the decimal point. The total field width is 12 characters. Thus, the output would be bbb4.500E+01.

The total output from this statement will consist of the following 17 characters:

```
bbb23bbb4.500E+01
```

where "b" stands for a blank.

13.2 THE MOST COMMON FORMAT ITEMS (C)

Here are the data format items that you will use most often:

1. F(w). Print the data as an integer right-justified in a field w characters wide.

2. F(w,d). Print the data as a decimal number with d digits following the decimal point, right-justified in a field w characters wide.

3. E(w,d). Print the data as a floating point number in the format

   ```
   [-]9.99...9Es99
   ```

   where each of the 9's stands for a digit. There are d digits following the decimal point. "s" is a sign, + or -, depending upon the sign of the exponent. The exponent field always contains two digits.

4. A or A(w). Print the data as a character string. The width of the field will be w, if w is specified. If w is not specified, then the length of the data item will be used. The data is converted to character, and is left-justified in the output field.

Another group of format items is called the collect of "control" format items. These items do not specify data formats. They specify where in the record or on the page the data is to appear. Here are the most commonly used control format items:

1. X(w). Print w blanks.

2. SKIP[(n)]. If n is not specified, the 1 is assumed. The specfied number of carriage returns and line feeds are printed.

3. PAGE. Skip to the top of a new page.

13.3 FURTHER PUT EDIT EXAMPLES (C)

Let us consider some variation of the preceding example.

1. PUT EDIT(23,45)(F(5),X(4),E(12,3)); will produce the output

   ```
   bbb23bbbbbbb4.5E+01
   ```

   There are four extra blanks, due to the X(4) control format item.

2. PUT EDIT(23,45)(F(5),SKIP,E(12,3)); will produce the output

```
bbb23
bbb4.500E+01
```

13.4 ITERATION FACTORS (C)

If your format list specifies the same format item several times, you may be able to save some effort by using iteration factors. Iteration factors allow you to specify that a format item or group of format items is to be used by CPL more than once.

The simplest format is:

```
Format:   integer format-item
or        integer (format-list)
```

The "integer" specifies the number of times the format item or format-list is to be repeated.

EXAMPLE: PUT EDIT(1,2,3)(2 F(4),F(5,2)); will produce the output

```
bbb1bbb2b3.00
```

where "b" stands for blank.

Note that in the above PUT EDIT statement, there is a blank after the iteration factor "2". This blank is required.

13.5 HOW PUT EDIT IS EXECUTED (C)

If you plan to use PUT EDIT extensively, then you should read the following, which describes exactly how it is executed.

1. Initialize the "format pointer" to point to the beginning of the format list.

2. Compute the first (or next) data item and save its value.

3. Starting from the position of the current "format pointer," choose the next format item and update the format pointer. Take account of iteration factors in making the choice of the next format item. Also take remote formats (described later in this chapter) into account.

4. If the format item is a control format (like X or SKIP), then perform the specified control operation, then go back to the preceding step.

5. If the format item is a data format item, then print the data value in the specified format, and go back to step 2.

13.6 DETAILED SPECIFICATION OF THE FORMAT LIST (C)

This section supplants the previous informal descriptions with a formal description of the format list.

A format-list consists of one or more specifications, separated by commas. Each of these specifications has one of the following formats:

1. A format-item
2. iteration-factor format-item
3. iteration-factor (format-list)

The iteration factor is in one of the following two formats:

1. integer
2. (expression)

Note that the iteration factor may consist of a parenthesized expression. This expression may contain any variables or function calls. CPL will evaluate the expression and convert it to an integer, if necessary. The value may be zero or positive.

The "format-item" is either a data format item, a control format item, or a remote format item. These are described in the following sections.

13.7 DATA FORMAT ITEMS (C)

Each data item from the data list of the PUT EDIT statement is printed out in one of the formats specified by a data format item.

The data format item specifies both (1) the data type to which the data item is to be converted, and (2) the form in which the data type is to be printed.

13.7.1 The F Format Item (C)

Format: F(w [,d [,p]])

where w, d and p may be any CPL expressions. The values of the expressions are computed and converted to integers. The value of w must be positive, and the value of d may not be negative. If d is not specified, then it defaults to 0. If p is not specified then it defaults to 0.

The data item is converted to FLOAT.

The data item is printed right-justified in a field w characters wide.

The printed data is the character representation of a decimal number, possibly with a decimal point, rounded if necessary, and right-adjusted in the specified field.

If d=0, then only the integer portion of the number is written; no decimal point appears.

If d>0, then d fractional digits are printed, with a decimal point inserted before the fractional digits. Trailing zeroes are supplied if necessary to fill out d fractional digits. Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

If the value of the data item is less than zero, a minus sign is prefixed to the external character representation; if it is greater than or equal to zero, no sign appears. Therefore, for negative values of the data value, the field width specifiecation (w) must include a count of both the sign and decimal point.

The scaling factor, p, is usually useless. CPL multiplies the data value by 10**p before printing out the value.

13.7.2 The E Format Item (C)

Format: E(w [,d [,s]])

where w, d and s may be any CPL expressions. The values of w, d and s are computed and converted to integers. If d is not specified, it defaults to 7. If s is not specified, it defaults to the value of (d+1).

The argument w stands for the field width, and specifies the total number of characters in the field. The argument d stands for the number of fractional digits, and specifies the number of digits in the mantissa that follow the decimal point. The argument s stands for the number of significant digits and specifies the number of digits that must appear in the mantissa.

The values of d and s must satisfy d>=0 and s>=d. The minimum value of w can be computed from a formula given later in this section.

The internal data is converted to FLOAT and is printed in the following general form:

```
   [-] (s-d) digits .  d digits E [+!-] exponent
```

The value is rounded if necessary.

The exponent is a two-digit decimal integer constant, which may be two zeros. The exponent is automatically adjusted so that the leading digit of the mantissa is nonzero.

If the above form of the number does not fill the specified field, the number is right-adjusted and extended on the left with blanks.

Here is how to compute the minimum value of w:

1. Start with the quantity (s+4).
2. If s=d, then add 1.
3. If d>0 then add 1.
4. If the data value is negative, then add 1.

13.7.3 The A Format Item (C)

Format: A [(w)]

The argument w, if specified, is any CPL expression. It will be computed and converted to an integer which must be non-negative.

The data value is converted, if necessary, to CHARACTER and is truncated or extended with blanks on the right to the specified field width (w) before being printed. If w=0, then no characters are printed. Enclosing quotation marks are never printed. If the field width is not specified, it is assumed to be equal to the character-string length of the converted data element.

13.7.4 The B Format Item (C)

Format: B [(w)]

The argument w, if specified, is any CPL expression. It will be computed and converted to an integer which must be non-negative.

The data value is converted, if necessary, to BIT. The BIT string is then converted to a CHARACTER string of the same length. The CHARACTER string is truncated or extended with blanks on the left to the specified field width (w) before being printed. If the field width is not specified, it is assumed to be equal to the number of bits in the BIT string.

The resulting CHARACTER string is printed. No quotation marks are printed, nor is the identifying letter B.

13.7.5 The B1 Format Item (C)

Format: B1 [(w)]

This is identical to the B format item.

13.7.6 The B3 Format Item (C)

Format: B3 [(w)]

The argument w, if specified, is any CPL expression, It will be computed and converted to an integer, which must be non-negative.

The data value is converted, if necessary, to BIT. If the number of bits in the string is not precisely divisible by 3, then the BIT string is extended from the left with 1 or 2 0-bits, so that the length is divisible by 3.

The BIT string is converted to a CHARACTER string containing one octal digit for each 3 bits in the bit string. The characters in the CHARACTER string will be digits in the range 0 through 7. The length of the CHARACTER string will be 1/3 the length of the BIT string.

The CHARACTER string is then truncated or extended with blanks on the left to the specified field with (w) before being printed. If the field width is not specified, then it is assumed to be equal to the number of characters in the CHARACTER string.

The resulting CHARACTER string is printed. No quotation marks are printed, nor is the identifying radix B3.

13.7.7 The B4 Format Item (C)

Format: B4 [(w)]

The argument w, if specified, is any CPL expression. It will be computed and converted to an integer, which must be non-negative.

The data value is converted, if necessary, to BIT. If the number of bits in the BIT string is not precisely divisible by 4, then the BIT string is extended from the left with 1, 2 or 3 0-bits, so that the length is divisible by 4.

The BIT string is converted to a CHARACTER string containing one hexadecimal (base 16) digit for each 4 bits in the BIT string. The characters in the CHARACTER string will be digits in the range 0-9 or letters in the range A-F. The length of the CHARACTER string will be 1/4 the length of the BIT string.

The character string is then truncated or extended with blanks on the left to the specified field width (w) before being printed. If the field width is not specified, then it is assumed to be equal to the number of characters in the CHARACTER string.

The resulting CHARACTER string is print. No quotation marks are printed, nor is the identifying radix B4.

13.7.8 The B2 Format Item (C)

Format: B2 [(w)]

This is like the B3 and B4 format items, except that the bit string is converted to the base 4 number system.

13.8 CONTROL FORMAT ITEMS (C)

There is no data output associated with a control format item. The control format items specify the layout of the data.

A control format item has no effect unless it is encountered before the data list is exhausted.

13.8.1 The X Format Item (C)

Format: X(w)

The argument w is any CPL expression. It will be computed and converted to an integer, which must be non-negative.

CPL prints out w blank characters.

13.8.2 The COLUMN Format Item (C)

Abbrev: COL for COLUMN

Format: COLUMN(n)

The argument n is any CPL expression. It will be computed and converted to an integer, which must be non-negative. If n=0, then n=1 will be assumed.

CPL prints sufficient spaces so that the following output will be printed starting at the column number specified by n. (The column number is the position of the character on the line. The first character on the line is in column 1.)

If printing has already passed that column on the current line, then CPL will simulate an implied SKIP option, and then move to the specified column.

The current column number is computed to be 1 plus the number of characters which have been printed for this FILE identifier since the last carriage-return character was printed. Note that CPL does not take into account any other special characters (such as tab or other control characters) which may physically affect the position of the printing element, other than to count each such character as one character.

13.8.3 The SKIP Format Item (C)

Format: SKIP [(n)]

The argument n, if specified, is any CPL expression. It will be computed and converted to an integer, which must be non-negative. If n is not specified, then n=1 is assumed. If n is positive, then CPL prints carriage return-line feed pairs. The value n=0 is permitted only for PRINT files; in this case, CPL prints a carriage return, so that the output which follows will overprint the preceding line.

13.8.4 The PAGE Format Item (C)

Format: PAGE

You may use this format item only with PRINT files. CPL prints a form feed and carriage return, so that subsequent output will be on a new page.

13.9 THE REMOTE FORMAT ITEM (C)

Format: R(label)

The remote format item allows format items in a FORMAT statement (described below) to replace the remote format item.

The "label" is the label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item.

The specified FORMAT statement must be internal to the same block as the PUT EDIT statement.

13.10 THE FORMAT STATEMENT (COLLECT ONLY) (C)

Format: label: FORMAT (format-list) ;

A FORMAT statement must have a statement label.

The "format-list" is the same as is specified for the PUT EDIT statement.

The format-list in the FORMAT statement can be referenced by means of a remote format item appearing in a PUT EDIT statement format list.

The format-list in a FORMAT statement may contain a remote format item referring to other FORMAT statements.

13.11 OTHER OPTIONS OF THE PUT EDIT STATEMENT (C)

13.11.1 Multiple Lists (C)

There may be several data and format lists in a single FORMAT statement. The format is:

```
Format:    PUT EDIT(data-list)(format-list)

                 [(data-list)(format-list)...] ;
```

13.11.2 Other Options (C)

The STRING, SKIP, FILE and other options, which are legal with the PUT LIST statement, are also legal with the PUT EDIT statement.

13.12 THE FORMAT OF PUT LIST OUTPUT (D)

This section presents a brief specification of the output produced by the PUT LIST statement. Most users should skip this section.

13.12.1 Format Without The VFORM Attribute (D)

This is the output as specified by the ANSI PL/I standard PUT LIST.

For FIXED data, the format is F(13).

For FLOAT data, the format is E(14,7,8).

13.12.2 Format With VFORM Attribute (D)

The VFORM option is a non-standard FILE option which causes CPL to use a variable output format when printing FIXED or FLOAT items.

Implementors of other PL/I systems may find it useful to have a precise definition of the variable format rules. This section gives those rules.

In addition, in the chapter entitled "CPL Programming Examples," you will find the variable format rules coded as a CPL PROCEDURE.

13.12.2.1 FIXED With VFORM Attribute (D) - If the value of the FIXED quantity is 0, then print "0". Otherwise, define the following quantities:

1. Let v = the value of the FIXED quantity

2. Let e = the number of digits necessary to represent v. This can be defined by e=FLOOR(LOG10(ABS(v))).

3. Let s=1 if v<0, 0 if v>=0.

Then the VFORM output of v is given by F(e+s).

13.12.2.2 FLOAT With VFORM Attribute (D) - If the value of the FLOAT quantity is 0, then print "0". Otherwise, define the following quantities:

1. Let v = the value of the FLOAT quantity.

2. Let e=FLOOR(LOG10(ABS(v))). e represents the position of the first significant digit in the representation of v in an F format -- positive if the first significant digit is to the left of the decimal point, and negative if it is to the right.

3. Represent v as x*10**e, where 0.1<=x<1. Represent ABS(x) as 0.dd..ddd, where the last digit "d" is non-zero. Let n = the number of digits "d" in this representation.

4. Let s=1 if v<0, s=0 if v>=0.

Given the above definitions, we can define the output format for a variable format PUT LIST as follows:

Case 1. e>8 or e<-8. The format depends upon two subcases:

1. If n>1 then use E(s+6,n-1,n).

2. If n=1 then use E(s+5,0,1).

Case 2. e<=8 and e>=1. The format depends upon two subcases:

1. If n<=e then use F(e+s).

2. If n>e then use F(n+s+1,n-e).

Case 3. e>=-8 and e<=0. Use the format F(n-e+s+1,n-e), where this is a special F format with no 0 preceding the decimal point.

CHAPTER 14

FILE I/O TO ARBITRARY FILES (C-D)

14.1 SIMPLE USAGE OF GET AND PUT TO ARBITRARY FILES (C)

Before discussing the input and output of CPL files, this chapter shows some simple examples.

14.1.1 PUT To Arbitrary Files (C)

Here is a sample program segment showing output to an arbitrary file:

```
  10.     DECLARE F FILE;
  20.     OPEN FILE(F) OUTPUT TITLE('PROG.DT');
  30.     PUT FILE(F) LIST(X,Y,X+Y);
          .......
 960.     PUT FILE(F) SKIP(3) LIST(I,S,T**2);
 970.     PUT FILE(F) SKIP;
 980.     CLOSE FILE(F);
```

The TITLE option of the OPEN statement above specifies that output is to be to the file PROG.DT on disk. Each PUT statement with the FILE option causes the expressions to be computed and the values stored into the specified file. The SKIP(3) option of the PUT statement causes three carriage returns and line feeds to be inserted into the file. The CLOSE statement causes the file to be closed.

14.1.2 GET To Arbitrary Files (C)

```
  10.     DECLARE G FILE;
  20.     OPEN FILE(G) INPUT TITLE('PP.DT');
  30.     GET FILE(G) LIST(X,Y,Z);
  40.     GET FILE(G) SKIP LIST(A,B);
          ...
 960.     CLOSE FILE(G);
```

This OPEN statement is similar to the one in the preceding example, except that the file is being opened for INPUT rather than OUTPUT. Each GET statement with the FILE option specifies that data values are to be taken from the specified file rather than the terminal. The SKIP option specifies that CPL should skip to the beginning of the next line, which begins after the next line feed character.

We now turn our attention to the general description of file input/output in CPL.

14.2 DEFAULT FILE OPTIONS FOR GET AND PUT STATEMENTS (C)

Each GET and PUT statement must have either a FILE or a STRING option. If you supply neither, then CPL supplies a default FILE option. For a GET statement, the default FILE option is FILE(SYSIN), and for a PUT statement (including a ? statement), the default FILE option is FILE(SYSPRINT).

14.2.1 Special Properties Of SYSIN And SYSPRINT Identifiers (C)

These two identifiers are, for all practical purposes, reserved words in the CPL language. (This is not completely true since you could actually declare them to be other than files, but then the default GET and PUT statements would no longer work.)

Here are the special rules which apply to these two identifiers:

1. Instead of defaulting to FIXED or FLOAT arithmetic variables, SYSIN and SYSPRINT have the default attributes of FILE. In addition, SYSPRINT has the attributes PRINT and VFORM.

2. If a file is opened, whether by an explicit OPEN statement or implicitly with a GET, PUT, READ or WRITE statement, the default device name will be TTY rather than DSK.

14.3 FILE ATTRIBUTES (D)

If an identifier is declared to have the FILE attribute, then it will have a number of additional attributes, either by explicit declaration or by default. These attributes are discussed in the following paragraphs.

14.3.1 STREAM Versus RECORD Attribute (D)

There are two sets of input/output instructions in CPL. The first set, previously described, consists of GET and PUT, and the second set, described below, consists of READ and WRITE.

The STREAM attribute specifies that the GET and PUT statements will be used with the file. The RECORD attribute specifies that the READ and WRITE statements will be used.

These two sets of statements provide completely different methods for accessing files. The principal difference is that STREAM input/output provides conversions of data between internal (binary) format and external (ASCII) format in the file; RECORD input/output generally provides no such conversion.

14.3.2 INPUT Versus OUTPUT Attributes (D)

The INPUT attribute specifies that the GET or the READ statement will be used to access the file.

The OUTPUT attribute specifies that either the PUT or the WRITE statement will be used to access the file.

If none of the above is specified, then the default attribute INPUT will apply.

14.3.3 The PRINT Attribute (D)

This attribute implies, and is only legal with, the attributes OUTPUT and STREAM.

It indicates that the file is intended to be printed on a printer or typed on a terminal, and is not intended to be read by a program.

Recall that the default file option for the PUT and ? statements is FILE(SYSPRINT). Now CPL assigns to SYSPRINT the default attributes FILE and PRINT. Thus, the default output for the PUT LIST and ? statements follows the conventions of PRINT files.

The differences between STREAM OUTPUT files with and without the PRINT attribute are as follows:

1. You may use the PAGE option of the PUT statement with PRINT files, and you may not use it with non-PRINT files.

2. You may use SKIP(0) with PRINT files only, to force overprinting of lines.

3. Suppose the expression in your PUT LIST statement has the CHARACTER data type. If the output file does not have the PRINT attribute, then CPL will enclose the character string value in single quotes. If the output file is PRINT, then the single quotes will be omitted.

14.3.4 The ENVIRONMENT Attributes (D)

Abbrev: ENV for ENVIRONMENT

The ANSI PL/I standard permits an implementation to provide "implementation-defined" file attributes which are not in the standard itself. An implementation will use these attributes when it wishes to take advantage of some special file features of the operating system.

CPL provides three such options:

1. VFORM

2. APPEND

3. NOPAGE

If you wish to use any of these attributes in the DECLARE or OPEN statement, they you must use them with the ENVIRONMENT attribute.

For example, suppose you wish a file to have the PRINT, VFORM and APPEND attributes. Then you may use the following file declaration:

```
          DECLARE F FILE PRINT ENV(VFORM APPEND);
```

The use of each of these options is described in the following sections.

14.3.5 The VFORM Attribute (D)

Format: ENV(VFORM)

The VFORM attribute is not permitted by the ANSI PL/I standard, and so its use should be avoided for programs intended to be ANSI standard.

The VFORM attribute specifies that the format for FIXED and FLOAT expressions in a PUT LIST statement is variable. The output format will depend upon the value of the expression.

14.3.6 The APPEND Attribute (D)

Format: ENV(APPEND)

Since APPEND is an "implementation-defined" attribute, it must be specified with the ENVIRONMENT attribute.

The APPEND attribute may be used only for OUTPUT files. It specifies that if the output disk file already exists, then it is not to be erased; instead, all further output is to be appended to the end of the existing file.

14.3.7 The NOPAGE Attribute (D)

Format: ENV(NOPAGE)

Since NOPAGE is an "implementation-defined" attribute, it must be specified with the ENVIRONMENT attribute.

The NOPAGE attribute may be used only for PRINT files.

The NOPAGE attribute is useful, for example, when your program is producing a diagram which is going to run over a page. Usually, your diagram will have a split in it, which results from an automatic page eject taking place when the printer approaches the bottom of the page.

If the NOPAGE option is specified for your PRINT output file, however, then those page ejects will be suppressed when you print out the file. This means that your output will continue to be printed right over the fold in the paper.

NOTE: When the NOPAGE option is specified, CPL will output the DC3 character (octal 23) whenever it would normally output a line feed character (octal 12). This means that your file will be suitable for printing, but it will not be suitable for typing at a terminal, or for being used as an input file to CPL or any other language.

14.4 THE FILE DECLARATION (D)

FORMAT: DECLARE filename FILE [file-attributes];

where the file attributes are those described in the preceding section.

Of course, attribute factoring and other properties of the DECLARE statement apply here.

EXAMPLE: DECLARE (F INPUT, G OUTPUT) STREAM FILE; declares the two files F and G.

If any of the attributes VFORM, APPEND or NOPAGE are to be used, then they must be declared using the ENVIRONMENT keyword.

EXAMPLE: DECLARE F FILE OUTPUT ENV(VFORM APPEND);

14.5 THE OPEN STATEMENT (DIRECT OR COLLECT) (D)

FORMAT: OPEN open-spec [, open-spec ...] ;

where each "open-spec" consists of a list of options and attributes separated by blanks. The following may appear in this list:

1. FILE(filename) -- this option must appear. The "filename" must be DECLAREd to have the FILE attribute.

2. File attributes, as described above: STREAM, RECORD, INPUT, OUTPUT, PRINT, and the ENVIRONMENT attributes VFORM, APPEND and NOPAGE.

3. TITLE(expression). If specified, then the "expression" must be (or will be converted to) a CHARACTER value. This character string will be interpreted as an operating system file-specification, and the file so specified will be opened.

   The complete format of the file-specification depends upon the operating system you are using. For more information, please refer to the chapter entitled "Running CPL under TOPS-10" or to the chapter entitled "Running CPL under TOPS-20."

   The most commonly used parts of the file-specification are the same in both of these operating systems. Thus, in either system, you may use the following format:

   ```
   dev:name.typ
   ```

   where the following rules hold:

1. The "dev" is the device name. If it is not specified, then it will default to DSK (the disk). EXCEPTION: If the "filename" specified by the FILE option is either SYSIN or SYSPRINT, then the device will default to TTY (your terminal).

2. The "name" is the file-name. If you do not specify one, then there is no default. If the operating system requires a filename (as it does, for example, for disk devices,) then CPL will stop execution with an error message. If the operating system does not require a file-name (as is the case, for example, for TTY and LPT,) then there will be no error.

3. The "typ" is the file-type. If none is specified, then the default extension of DT will be used.

If no TITLE option is specified, then CPL will create a default TITLE option consisting of the file name appearing in the FILE option. Thus for example, the statement OPEN FILE(F); will cause the file "DSK:F.DT" to be opened.

If more than one "open-spec" is specified, then CPL will open more than one file.

If the file identifier specified by the FILE option of the OPEN statement is already open, then CPL will simply ignore the OPEN statement and go on to the next statement. This can lead to some confusion if, for example, a file identifier is open for INPUT, and an OPEN statement attempts to open the same file identifier for OUTPUT. In this case, the OPEN statement will perform no operation, and the program will continue with no error indication. (This convention may seem rather surprising, but it is precisely what the ANSI PL/I standard specifies.)

14.6 "OPEN" ATTRIBUTE MERGING (D)

The preceding discussion makes it clear that you may specify file attributes both by means of the DECLARE statement for the file identifier and by means of the OPEN statement.

When the OPEN statement is executed, all attributes specified by the DECLARE statement are merged with those specified in the OPEN statement to form a complete attribute set.

For example, if the declaration contains the attribute RECORD and the OPEN statement statement contains the attribute SEQUENTIAL, then the open file will have both of these attributes, and the default attribute INPUT will be added.

If any attributes specified in or implied by the DECLARE statement conflict with any attributes specified in or implied by the OPEN statement, then the error will be detected during the attribute merge and the open will fail. In this case execution will stop with an error message.

14.7 IMPLICIT FILE OPEN (D)

If CPL must execute a GET, PUT, READ or WRITE statement that specifies a file identifier that is not open, then CPL simulates an OPEN statement with attributes that depend upon the statement causing the implicit open.

If the statement is GET, for example, CPL simulates an OPEN with the attributes STREAM and INPUT. If it is PUT, the attributes are STREAM and OUTPUT. If it is READ, they are RECORD and INPUT; if it is write, they are RECORD and OUTPUT.

If CPL opens a file implicitly in this way, you cannot specify a TITLE option; thus, the default rules will apply. Under these rules, CPL will open the file "name.DT", where "name" is the file identifier name.

14.8 THE GET STATEMENT (COLLECT OR DIRECT) (D)

Format: GET option-list;

The "option-list" consists of one or more options separated by blanks. The options may be specified in any order. They are as follows:

1. FILE(filename). Your program must contain a DECLARE statement giving the "filename" the FILE attribute. The file must be opened with the INPUT and STREAM attributes. If the file is not open, then CPL will open it implicitly with these attributes. If there is no FILE option or STRING option, then CPL will assume FILE(SYSIN). (No declaration is needed for SYSIN.)

2. STRING(expression). This option conflicts with the FILE option. If you use the STRING option, then CPL will not take its input from a file. Instead, it will evaluate the character expression specified in parentheses by the STRING option, and will use that character string to take the values requested in the input list.

3. SKIP[(expression)]. If you omit the "(expression)," then SKIP(1) is assumed. CPL evaluates the expression to obtain an integer value, n, which must be positive. CPL then reads characters from the input file until n line feeds have been read.

   You may not use the SKIP option with the STRING option.

4. LIST(vble1 [,vble2 ...]). You specify a list of one or more variables whose values CPL is to read from the specified file or string. The variables must be scalars, or must be subscripted array elements.

You must specify either the LIST option or the SKIP option. You may specify both.

14.9 THE PUT STATEMENT (COLLECT OR DIRECT) (D)

Format: PUT option-list;

where the "option-list" contains one or more options separated by blanks. The options can appear in any order. They are as follows:

1. FILE(filename). Your program must contain a DECLARE statement giving the "filename" the FILE attribute. The file must be opened with the OUTPUT and STREAM attributes. If the file is not open, then CPL will open it implicitly with these attributes. If there is no FILE option or STRING option, then CPL will assume FILE(SYSPRINT). (No declaration is needed for SYSPRINT.)

2. STRING(identifier). The "identifier" must be a scalar with the CHARACTER attribute (either VARYING or NONVARYING). If you specify this option, then CPL will not transmit the output to a file. Instead, it will store the output into the character string variable specified by the identifier.

3. SKIP[(expression)]. If you omit the "(expression)", then CPL will assume SKIP(1). CPL evaluates the "expression" to obtain an integer value, n, which must be non-negative. If n is positive, then CPL places n carriage return-line feed pairs into the output file. The value n=0 is permitted only for PRINT files; in this case, CPL places a carriage return into the output file, so that the output which follows will overprint the preceding line.

4. PAGE. You may use this option only with PRINT files. CPL places the characters form feed and carriage return into the output file. If the file is then printed on a line printer, the printer will skip to a new page at this point.

5. LIST(expr [,expr ...]). CPL computes each expression appearing in the list. CPL converts the value to character format, and outputs the result to the specified string or file.

6. EDIT edit-spec, where the edit-spec has the format

   ```
   EDIT(output-spec)(format) [(output-spec)(format)]...
   ```

   EDIT and LIST are conflicting keywords. The full description of this option is given in the chapter entitled "THE PUT EDIT STATEMENT."

You must specify at least one of the options SKIP, PAGE, EDIT, or LIST. You may use SKIP or PAGE with LIST or EDIT. You may not use both SKIP and PAGE in the same PUT statement. Nor may you use both LIST and EDIT in the same PUT statement.

14.10 THE READ STATEMENT (COLLECT OR DIRECT) (D)

Format: READ option-list;

where the "option-list" consists of two or more options separated by blanks. The options may be in any order. They are as follows:

1. FILE(filename). Your program must contain a DECLARE statement giving the "filename" the FILE attribute. The file must be opened with the INPUT and RECORD attributes. If the file is not open, then CPL will open it implicitly with these attributes.

2. INTO(identifier). The "identifier" must be a scalar with the CHARACTER attribute, either VARYING or NONVARYING. CPL will read one record from the file. (That is, CPL will read all characters in the file up to an including the next line feed. CPL will discard the carriage return and line feed.) The characters in the record are stored as the value of the specified identifier.

3. IGNORE(expression). CPL evaluates the "expression" to obtain an integer value, n, which must be non-negative. If n is positive, then CPL reads and ignores n records from the file.

You must specify either the INTO option or the IGNORE option.

For the purposes of the READ statement, a "record" consists of all characters up to and including the next line feed. CPL ignores all carriage return characters. CPL stores all characters, up to but not including the line feed character, into the storage occupied by the INTO variable.

14.11 THE WRITE STATEMENT (COLLECT OR DIRECT) (D)

Format: WRITE option-list;

where the "option-list" consists of options separated by blanks. The options may be in any order. They are as follows:

1. FILE(filename). Your program must contain a DECLARE statement giving the "filename" the FILE attribute. The file must be opened with the INPUT and RECORD attributes. If the file is not open, then CPL will open it implicitly with these attributes.

2. FROM(identifier). The "identifier" must be a scalar with the CHARACTER attribute (either VARYING or NONVARYING). CPL transmits the current value of the CHARACTER variable to the file as a single record.

CPL transmits the characters to the output file, and then terminates the transmission with a carriage return and line feed.

14.12 THE CLOSE STATEMENT (COLLECT OR DIRECT) (D)

Format: CLOSE FILE(filename) [, FILE(filename) ...];

Your program must contain a DECLARE statement specifying the FILE attribute for the "filename."

CPL closes the specified file or files. If the file is not open, then CPL simply moves on to the next operation.

14.13 THE CLOSE FILES STATEMENT (COLLECT OR DIRECT) (D)

Format: CLOSE FILES ;

WARNING: This statement is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

CPL will close all open files.

CHAPTER 15

BLOCK STRUCTURE AND DECLARATION SCOPE RULES (C-D)

This and the following chapters contain a complete treatment of the way in which CPL handles program blocks.

This chapter is an introductory chapter in the sense that it contains a great deal of technical information that is common to all types of blocks. Therefore, the technical portions of the later chapters will depend upon the material given here.

Nonetheless, if you are a beginning CPL programmer, you are encouraged to skip this chapter entirely, or to skim it for an overview.

If you want to know how to code a subroutine, then start with the chapter entitled "Subroutine and Function PROCEDUREs." That chapter contains enough introductory material to get you started. And when you become an expert, you can always come back and learn the complexities.

Similarly, if you want to handle some error conditions, such as end of file, then start reading the chapter entitled "ON-conditions and error handling," learn the subleties as you need them.

This chapter also describes the SNAP statement. You use the SNAP statement when you wish CPL to type out a listing of all active blocks.

15.1 <u>THE BEGIN STATEMENT (COLLECT ONLY) (C)</u>

Format: BEGIN; ... END;

The BEGIN and END statements enclose a group of statements. You may put the BEGIN and END statements, and the statements they enclose, on separate lines of your program.

The BEGIN and END statements, as well as the statements they enclose, form what is called a program "block."

15.2 <u>PROGRAM BLOCKS WITH PROCEDURE STATEMENT (D)</u>

A PROCEDURE statement, its corresponding END statement, and the statements they enclose, also form a program block. The usage of PROCEDURE blocks is discussed in the chapter entitled "Subroutine and Function PROCEDUREs."

15.3 SCOPE RULES FOR DECLARATIONS IN BLOCKS (D)

Suppose a DECLARE statement appears within a BEGIN/END or PROC/END block in your program. Then that declaration will apply within that program block. Outside of that block, it will be as if that declaration had never been made.

The "scope" of a declaration is the range of statements over which the declaration applies. If a declaration is made inside a block, then its scope will include, at most, only the remaining statements within that block. If a declaration does not lie within any interior block, then its scope may be the whole program.

Consider this program segment:

```
  10.     A=5;
  20.     BEGIN;
  30.     DECLARE A(10);
  40.     A(1)=5;
  50.     END;
  60.     A=10;
```

Statement 30 declares A to be an array. The scope of this declaration is the BEGIN block which begins at statement 20 and ends at statement 50. Thus, the reference to A in statement 40 is to the array A.

However, the declaration of A in statement 30 does not apply at all outside the block. Thus, the references to A in statements 10 and 60 are to an entirely different variable A. In the absence of any declaration for A in the program outside the BEGIN/END block, the references to A in statements 10 and 60 are to a default scalar variable having nothing in common (except the identifier name) with the variable A declared in statement 30.

15.3.1 Default Vs. Explicit Declarations (D)

There are several kinds of explicit declarations in CPL:

1. The most common kind of explicit declaration is that of an identifier appearing in a DECLARE statement.
2. If you use a statement label, then the identifier appearing in the statement label is explicitly declared in this way.
3. In a PROCEDURE statement (to be described in a later chapter) all identifiers appearing in the parameter list are explicitly declared to be parameters.

All of these are explicit declarations. The discussions about scope rules in preceding and following sections apply to all such explicit declarations.

There is also a default declaration for each identifier. This default declaration is that one which is given in the chapter on the DEFAULT statement.

15.3.2 Scope Of Default And Explicit Declarations (D)

Consider the following program segment:

```
 10.     DECLARE A FIXED;
 20.     A,B=5;
 30.     BEGIN;
 40.     DECLARE (A,B) FLOAT;
 50.     A,B=5;
 60.     BEGIN;
 70.     DECLARE B FLOAT;
 80.     A,B=5;
 90.     END;
100.     END;
```

Let us consider the scope of each of the declarations of A and B.

Since the first declaration of A is not enclosed in a block, the default declaration of A has no scope at all. The declaration of A in statement 10 applies only to the statements outside the block at statement 30, since there is another declaration of A inside that block. The declaration of A in statement 40 has a scope equal to the entire block from statements 30 through 100. Thus the reference to A in statement 20 is to the declaration in statement 10, while the references to A in statements 50 and 80 are to the declaration in statement 40.

The reference to B in statement 20 is to the default declaration of B. The scope of the default declaration of B is all statements outside the block beginning at statement 30. The scope of the declaration of B in statement 40 is all statements inside the block beginning at statement 40, excluding all statements inside the block beginning at statement 60. The scope of the declaration of B in statement 70 is the block beginning at statement 60.

Thus the general rules are as follows:

The default declaration of an identifier has, as a scope, all parts of the program, except those parts in the scope of an explicit declaration of the same identifier.

An explicit declaration of an identifier applies to all statements inside the block in which the declaration lies (or to the whole program, if the declaration is not inside a block), with the exception of the statements inside any block which, itself, contains an explicit declaration of the same identifier.

15.3.3 Labels On PROC And BEGIN Statements (D)

A label appearing on a PROCEDURE or BEGIN statement is an explicit declaration of the statement label identifier. This declaration is not considered to be inside the BEGIN or PROC block; instead it is in the block encompassing the BEGIN or PROC statement.

15.4 BLOCK INVOCATION OR TERMINATION (D)

This sections describes when CPL invokes or terminates a program block.

15.4.1 How A Block Is Invoked (D)

The way in which a block is invoked depends upon the type of block. The possibles ways are as follows:

1. A BEGIN/END block which is not an ON-unit is invoked when CPL executes the BEGIN statement.

2. A PROCEDURE/END block is invoked by means of a CALL statement to the specified statement, or by means of an appropriate function reference. This subject is discussed in the chapter on PROCEDUREs.

3. A BEGIN/END block which is an ON-unit is invoked when the specified condition is raised. This topic will be discussed in detail in the chapter entitled "Error Handling and ON-conditions."

15.4.2 Normal Termination Of A Block (D)

The use of the terms "normal termination" and "abnormal termination" for blocks is similar to their use for DO/END groups.

How a block is terminated depends upon the type of block:

1. A BEGIN/END block which is not an ON-unit is terminated when the END statement is executed. Control passes to the statement following the END statement.

2. A PROCEDURE/END block is terminated normally by execution of a RETURN statement or of the END statement at the end of the block. When this happens, control returns to the point where the PROCEDURE was invoked (the CALL statement or the function reference).

3. A BEGIN/END block which is an ON-unit is terminated normally by execution of the END statement. In most cases, normal termination of an ON-unit is illegal; when it is legal, then control returns to the point at which the condition was raised. Please refer to the chapter on error handling for more information.

15.4.3 Abnormal Termination Of A Block (D)

CPL terminates a block abnormally when it executes a GOTO statement which transfers control outside of the block. When this happens, execution continues with the statement to which the GOTO was made.

15.5 THE BLOCK PROLOGUE AND EPILOGUE (D)

CPL performs special additional operations when a block is invoked or terminated. These special operations are called the block prologue and epilogue. This section describes these special operations.

15.5.1 The Block Prologue (D)

Whenever a new block is invoked, as described in the above rules, CPL performs a prologue. The block prologue performs the following:

1. If the block is a PROCEDURE/END block, and the PROCEDURE statement specifies parameters, then the parameters are matched to the arguments, in the manner described in the chapter on PROCEDUREs.

2. CPL allocates storage for each variable explicitly declared in the block with the AUTOMATIC attribute. (This is the default, unless modified by a DEFAULT statement.)

15.5.2 The Block Epilogue (D)

An epilogue is performed for each block which is terminated, whether normally or abnormally. All AUTOMATIC storage which was allocated by the prologue is freed.

15.6 RECURSIVE BLOCKS (D)

PROCEDURE and ON-unit blocks may be recursive. This means that there may be several invocations of the same block active at the same time. (This happens, for example, when a PROCEDURE calls itself. See the chapter on PROCEDUREs for examples of this.)

When a block is invoked recursively, separate copies are made of all the AUTOMATIC variables declared in that block. In this way, each AUTOMATIC variable in a block invocation is unique to that block invocation, and there are as many copies of it as there are additional invocations of the same block.

15.7 THE SNAP STATEMENT (DIRECT OR COLLECT) (C)

Format: SNAP ;

The SNAP statement causes CPL to type out on your terminal a "snap dump" of the current status of all active blocks in the execution of your program. You will find this statement useful for debugging your program when you are confused about which program blocks are active.

The blocks are listed in reverse order of invocation, with the most recently activated listed first.

Here is a typical SNAP dump:

```
STMT#   BLOCK   TYPE
85.+3   70.     PROC
50.     40.+1   ON
95.     90.     BEGIN
90.     70.     PROC
10.
```

Each line of this dump contains the following information about one active block:

1. STMT# is the statement number of the "current" statement active in that block.

2. BLOCK is the statement number of the first statement in the BLOCK. (This is a BEGIN statement or a PROCEDURE statement.)

3. TYPE is the type of block -- PROC for PROCEDURE/END block, BEGIN for BEGIN/END block which is not an ON-unit, and ON for an ON unit.

The blocks are listed in reverse order of invocation.

Thus, the above example gives the following information:

The current statement in the current block has statement number 85.+3. (This means that it is the fourth statement on line 85 of your program.) This statement is in the PROCEDURE block which begins at line 70.

That PROCEDURE was invoked at line 50, in the ON-unit which begins in line 40.+1.

The ON-unit was invoked by an error in statement 95 in the BEGIN block which begins at statement 90.

This BEGIN block was invoked at statement 90 (the BEGIN statement), which is in the PROCEDURE which begins at statement 70. This is a recursive invocation of that PROCEDURE, since the first line also shows the same PROCEDURE active.

Finally, this PROCEDURE was invoked by the statement at line 10.

CHAPTER 16

STORAGE CLASSES (D)

Most users will not need to understand this chapter to write programs. For this reason, you may skip this chapter on first reading.

Each CPL variable has a storage class. Either you declare the storage class explicitly (as in a DECLARE statement) or else CPL assigns a default storage class.

16.1 DEFAULT STORAGE CLASS RULES (D)

A default storage class may only be STATIC or AUTOMATIC. It will always be AUTOMATIC, unless your program contains a DEFAULT statement specifying a default attribute of STATIC. For more information on doing this, refer to the chapter entitled "The DECLARE and DEFAULT Statements."

16.2 DIFFERENCES AMONG THE STORAGE CLASSES (D)

The different storage classes are listed below. For each storage class, the description below will give the following information:

1. How the storage class is declared or otherwise specified.
2. Whether the storage class specifies that the identifier is a constant or a variable.
3. If it is a variable, when storage for the variable is allocated and freed.

16.3 LIST OF STORAGE CLASSES (D)

Here is a list of all the storage classes that CPL supports:

16.3.1 AUTOMATIC Storage Class (D)

Abbrev: AUTO for AUTOMATIC

Every CPL identifier which is declared in a DECLARE statement is given a storage class of AUTOMATIC, unless:

1. Your program contains a DEFAULT statement specifying that the default storage class should be STATIC.

2. The declaration specifies a different storage class (STATIC, CONTROLLED, BASED, or PARAMETER).

3. The attribute BUILTIN is declared.

4. The FILE attribute is declared.

If a variable has the AUTOMATIC storage class attribute, then storage for it is allocated when the program block in which it is declared is entered. It is freed when the block is terminated.

EXAMPLE: Consider the program segment

```
BEGIN ;
DECLARE A(1000) FLOAT AUTOMATIC ;
.....
END ;
```

The array A, when allocated, requires 1000 words. This storage is not actually allocated until the BEGIN statement is executed.

EXAMPLE: You may use variables in the extent expressions. These extent expressions are evaluated at the time the block is entered, before the storage is allocated.

Consider the following program segment:

```
N=10 ;
BEGIN ;
DECLARE A(N,N+1) AUTO ;
...
END ;
```

When the BEGIN statement is executed, the values of N and N+1 are computed using the current value (10) of N. The array bounds are set to those values. Thus, the array is allocated with dimensions 10 and 11.

The above example uses the heretofore undefined term "extent expression." This term refers to any expression, appearing in a declaration, which specifies a bound or size. There are three places where extent expressions occur:

1. A dimension bound expression, either an upper or lower bound, is an extent expression. The preceding example shows variable extent expressions for the upper bound of a dimension expression. Another EXAMPLE: DECLARE A(I:J+K); contains variable extent expressions for both the lower bound ("I") and the upper bound ("J+K").

2. The parenthesized expression following the CHARACTER attribute is an extent expression. EXAMPLE: CHAR(23) and CHARACTER(N+M) VARYING contain the extent expressions "23" and "N+M".

3. Same for the BIT attribute.

16.3.2 STATIC Storage Class (D)

An identifier will be given the STATIC storage class attribute if either a DEFAULT statement specifies a STATIC default, or the declaration of the identifier specifies STATIC.

STATIC storage is allocated at the time you enter the DECLARE statement. It is never freed (unless you erase the DECLARE statement).

Extent expressions in declarations for STATIC storage may not contain any variables.

16.3.3 CONTROLLED Storage Class (D)

Abbrev: CTL for CONTROLLED

If you wish a variable to have the CONTROLLED attribute you must specify it in the DECLARE statement.

CPL never automatically allocates or frees CONTROLLED storage. Instead, you must explicitly allocate storage for a CONTROLLED indentifier with an ALLOCATE statement, and free it with a FREE statement.

The ALLOCATE and FREE statement are described later in this chapter.

16.3.4 BASED Storage Class (D)

The BASED storage class is significantly different from all the others, and so it is described in a later chapter, "BASED Storage and POINTERs."

Please refer to that chapter for more information.

16.3.5 PARAMETER Storage Class (D)

Abbrev: PARM for PARAMETER

An identifier is declared to be a PARAMETER by virtue of its appearance in the parameter list of a PROCEDURE statement.

Your program may also contain a separate declaration of the same identifier in the same block for the purpose of specifying the data type of the parameter. In that case, the declaration may specify the PARAMETER attribute.

Parameters are discussed fully in the chapter entitled "Subroutine and Function PROCEDUREs."

16.3.6 NAMED CONSTANT Storage Class (D)

Any identifier with the NAMED CONSTANT is a constant, not a variable. It may never appear on the left hand side of an assignment statement.

A NAMED CONSTANT is declared in the following ways:

1. Any identifier declared with the FILE attribute is a NAMED CONSTANT.
2. A statement label is a NAMED CONSTANT.

16.4 THE ALLOCATE STATEMENT (DIRECT OR COLLECT) (D)

Abbrev: ALLOC for ALLOCATE

Format: ALLOCATE ident [,ident]...

The "ident" must be an identifier with the CONTROLLED storage class attribute.

CPL allocates storage for the specified identifier. If storage has already been allocated, then CPL allocates an additional copy.

16.5 THE FREE STATEMENT (DIRECT OR COLLECT) (D)

Format: FREE ident [,ident]...

The "ident" must be an identifier with the CONTROLLED storage class attribute.

There must exist at least one allocation of the specified identifier. CPL releases the storage occupied by the most recent allocation.

16.6 EXAMPLES OF CONTROLLED STORAGE (D)

Consider the following program segment:

```
DECLARE A CONTROLLED;
ALLOCATE A;
A = 2;
ALLOCATE A;
A = 12;
PUT LIST(A,ALLOCATION(A))/* RESULT IS "12 2" */;
FREE A;
PUT LIST(A,ALLOCATION(A))/* RESULT IS "2 1"*/;
```

The first ALLOCATE statement creates the first allocation of A. The second ALLOCATE statement creates a second allocation. The FREE statement frees the most recent allocation.

The ALLOCATION built-in function returns the current number of allocations of the specified CONTROLLED identifier.

It is also legal for a CONTROLLED declaration to ccontain variable extent expressions (array bounds and string lengths) as in the following example:

```
DECLARE A(N,N+1) CONTROLLED;
N = 12;
ALLOCATE A;
```

When the ALLOCATE statement is executed, the value of N is 12. Therefore, the array will be allocated with the dimensions (12,13).

CHAPTER 17

SUBROUTINE AND FUNCTION PROCEDURES (C-D)

17.1 WHY DO YOU NEED PROCEDURES? (C)

There are two reasons why you will use PROCEDUREs.

1. To save space. Suppose your program contains a section of code performing a specific operation, and suppose that this operation must be performed in several different places in the program. Instead of repeating the same section of code in several different sections of your program, you can save a lot of space by writing the section of code once as a subroutine PROCEDURE and then calling that procedure each time you want to execute the section of code.

   See the example below.

2. To provide program "modularity." A program is "modular" if it is broken up into small PROCEDUREs each of which performs a simple function.

   This use for PROCEDUREs is different from the preceding one. You may have a large complex section of code in your program. You may remove a section of this code which performs a simple function and place it into a subroutine. You would do this even though that subroutine would be called only once. In this way, you have simplified a large section of code, and you have made it easier to understand, debug and maintain.

   For more information on this concept, please refer to the chapter entitled "Structured and GOTO-less Program."

EXAMPLE: Suppose there are several points in your program where you wish to type out a table of values. You code a subroutine TABLEOUT which types out the the table. Then, at each point in your program where you wish to type out the table, you insert the statement "CALL TABLEOUT;".

Your program may end up looking something like this:

```
...
CALL TABLEOUT ;
...
CALL TABLEOUT ;
...
CALL TABLEOUT ;
...
```

```
TABLEOUT: PROCEDURE ;
        /* THIS PROC TYPES OUT THE TABLE */
        ...
        RETURN ;
        END TABLEOUT;
```

17.2 PROCEDURE ARGUMENTS AND PARAMETERS (C)

In the preceding example, we assumed that the subroutine PROCEDURE was to do exactly the same thing each time it was called.

But suppose you wish a PROCEDURE to operate in the same general way each time it's called, but with some minor differences, and you wish to specify the difference when you call it.

For example, suppose you wish to write a subroutine which prints out the n'th through m'th records of a file, with n and m to be specified at the time that the subroutine is called. The following program segment illustrates such a subroutine:

```
        ...
        CALL TYPEREC(30,40);
        ...
        CALL TYPEREC(50,70);
        ...
        GET LIST(I,J);
        CALL TYPEREC(I,J)
        ...
TYPEREC:        PROC(N,M);
        DECLARE F FILE, C CHAR(100) VAR;
        OPEN FILE(F) RECORD;
        READ FILE(F) IGNORE(N-1);
        DO I=N TO M;
        READ FILE(F) INTO(C);
        PUT SKIP LIST(C);
        END;
        CLOSE FILE(F);
        RETURN;
        END TYPEREC;
```

The first call to TYPEREC will cause records 30 through 40 to be typed out. The second call will cause records 50 through 70 to be typed. And the third call specifies variable limits, determined by the values of I and J which are typed in as a result of a GET LIST statement.

Each of the CALL statements in the above example specifies two "arguments," 30 and 40 in the first CALL, 50 and 70 in the second CALL, and I and J in the third. These arguments may be any expressions. They are evaluated before control passes to the subroutine.

Now look at the PROCEDURE statement in the above example. This statement specifies two "parameters," n and m. When the PROCEDURE is invoked, the two parameters will be given values equal to the two arguments (30 and 40 or 40 and 50 or I and J in the above example). Thus, in the first CALL, n will equal 30 inside the procedure, and m will equal 40. In the second CALL, n will be given the value 50 and m the value 60. And in the third call, n and m will be given whatever values I and J have.

17.2.1 Difference Between Argument And Parameter (C)

Be certain that you understand the difference between an "argument" and a "parameter." An argument appears in a CALL statement and a parameter appears in a PROCEDURE statement. The CALL statement must contain exactly the same number of arguments as there are parameters in the PROCEDURE statement. When the CALL statement is executed, each of the arguments is evaluated, and the parameters take on these values during execution of the statements in the PROCEDURE.

17.2.2 Real Versus "dummy" Arguments (C)

What happens if a PROCEDURE assigns a value to one of the parameters? Does the value of the argument get changed? The answer is "yes," unless a dummy argument is created.

For example, consider the following program:

```
  10.              A,B=5;
  20.              CALL CH(A,B+1);
  30.              ?A,B;
                   ...
 100.     CH:      PROCEDURE(X,Y);
 110.              ?X,Y
 120.              X,Y=10;
 130.              RETURN;
 140.              END CH;
```

Let us consider what happens in the above program. In statement 20, the arguments A and B+1 are evaluated and the values, 5 and 6 respectively, are given to the parameters X and Y. Therefore, when CH is entered statement 110 will print out the values 5 and 6. Then statement 120 assigns the value 10 to the parameters X and Y. The assignment to X will cause the value of the argument A to be changed to 10. But the assignment to Y will leave the value of B unchanged for the following reason: When the argument B+1 was evaluated, a "dummy" argument is created, and the value 6 is assigned to it. When an assignment is made to the parameter Y, the value of the dummy argument is changed, but the value of B is not modified.

Therefore, when the RETURN statement in statement 130 is executed, control will return to statement 30, and this statement will print the values 10 and 5.

17.2.3 Data Types Of Arguments And Parameters (C)

Arguments and parameters may have any computational data type. Consider, for example, the following program:

```
 10.               DECLARE C CHAR(5);
 20.               C='ABCDE';
 30.               CALL DE(C);
 40.               ?C;
                   ...
100.     DE:       PROCEDURE(P);
110.               DECLARE P CHAR(5);
120.               P = '12345';
130.               RETURN;
140.               END DE;
```

Notice that the PROCEDURE DE contains a separate declaration of P for the attributes CHAR(5). In the CALL in statement 30, the argument C also is CHAR(5), and so no dummy argument is created. When P is assigned a value in statement 120, the value of C is changed, so that statement 40 will print the value '12345'.

If the data type of the argument does not precisely match the data type of the parameter, then the argument will be converted to the data type of the parameter and a dummy argument with that data type and value will be created.

The complete rules on argument and parameter matching are given below in this chapter.

17.3 FUNCTION PROCEDURES (C)

A procedure can be invoked either as subroutine, using a CALL statement, or as a function, by referencing the PROCEDURE name in an expression.

The PROCEDURE may simply compute a complicated expression. Consider, for example, the following PROCEDURE:

```
          HYP:     PROCEDURE(A,B);
                   RETURN(SQRT(A**2+B**2));
                   END HYP;
```

This PROCEDURE computes the length of the hypotenuse of a right triangle with legs A and B.

For example, the statement "X=HYP(3,4);" will assign to X the value 5. In executing this statement, the parameters A and B take on the values of the arguments 3 and 4, respectively. The formula SQRT(A**2+B**2) is computed and substituted into the expression which invoked the procedure HYP.

Note that a different form of the RETURN statement was used. In this case, the RETURN statement specified an expression which was to be computed and returned to the caller.

The rules for arguments and parameters presented elsewhere in this chapter apply to function PROCEDUREs just as they apply to subroutine PROCEDUREs.

17.3.1 Further Examples Of Function Procedures (C)

Here is a PROCEDURE which computes and returns the factorial of the argument:

```
FAC:    PROCEDURE(NUM);
        PROD = 1;
        DO I = 1 TO NUM;
        PROD = PROD*I;
        END;
        RETURN(PROD);
        END FAC;
```

If you typed "?FAC(6)", the value 720 would be typed out.

A PROCEDURE can return any computational data type. You specify with the RETURNS option of the PROCEDURE statement what the data type to be returned should be.

Here, for example, is a PROCEDURE which has no parameters and which returns a CHARACTER string containing the time of day in the format hh:mm. It uses the TIME built-in function which returns the time of day as a character string in the format hhmmssddd (hours, minutes, seconds, thousandth's of a second -- see the description of the TIME built-in function in the chapter on built-in functions).

```
T:      PROCEDURE RETURNS(CHAR(5));
        DECLARE TEMP CHAR(9);
        DECLARE TIME BUILTIN;
        TEMP = TIME();
        RETURN(SUBSTR(TEMP,1,2)!!':'!!SUBSTR(TEMP),3,2));
        END T;
```

For example, if you invoke this PROCEDURE at 2:30 in the afternoon, say, by means of the statement "?T()", then it will return the character string '14:30'.

17.4 THE PROCEDURE STATEMENT (COLLECT ONLY) (C)

Abbrev: PROC for PROCEDURE

```
Format:    label:  PROCEDURE [(parameter-list)] [returns-option];
                   ...  ;  END;
```

where the optional "parameter-list" has the format

```
                   (identifier [,identifier ...])
```

and the optional returns-option has the format:

```
                   RETURNS (data type)
```

The data type in this case is one of the following, where "integer" stands for any positive integer: FIXED, FLOAT, CHAR(integer), CHAR(integer) VARYING, BIT(integer), or BIT(integer) VARYING.

If you omit the returns-option, then CPL will assume one. CPL will assume a returns-option of either FIXED or FLOAT, depending upon the default attributes of the "label" of the procedure. If there is no

DEFAULT statement in your program, then RETURNS(FLOAT) will be assumed. If you have a DEFAULT statement, then the default data type of the RETURNS will be exactly the same as it would be for the identifier "label," if "label" were being used as an ordinary variable.

When the PROCEDURE is invoked, either as a subroutine with a CALL statement, or as a function, then the parameters are assigned the values of the arguments, the point at which the PROCEDURE was invoked is saved, and control passes to the statement following the PROCEDURE statement. When a RETURN statement is encountered, then control passes back to the point of invocation.

The exact rules for matching arguments and parameters are discussed later in this chapter.

17.5 <u>THE CALL STATEMENT (DIRECT OR COLLECT) (C)</u>

Format: CALL name [(argument-list)];

The optional argument-list has the format:

```
(expression [,expression]...)
```

The "name" must be the label of a PROCEDURE statement appearing in your program. The number of arguments in the argument list must be exactly equal to the number of parameters appearing in the PROCEDURE statement.

CPL takes the following steps to execute a CALL statement:

1. Each of the expressions in the argument list is evaluated. Each argument is matched with the corresponding parameter in a manner described fully later in this chapter.

2. CPL saves and remembers the exact place from which the CALL took place. This is called the "point of invocation" of the PROCEDURE, since it is the place where the PROCEDURE was invoked.

3. CPL transfers control to the statement following the PROCEDURE statement.

17.6 <u>INVOCATION OF A PROCEDURE AS A FUNCTION (C)</u>

Exactly the same steps are taken if the PROCEDURE is invoked as a function. The "point of invocation," however, is a reference to the PROCEDURE appearing in an expression.

In this case, you must return from the PROCEDURE with a RETURN statement specifying an expression. The value of this expression will be substituted for the function reference in the expression at the point of invocation.

Although it is unusual, the same PROCEDURE can be used as both a function and a subroutine PROCEDURE. The only rule is that if a PROCEDURE is invoked as a subroutine (with a CALL statement) then you must return with a simple RETURN statement with no expression; but for a function invocation, you must return with a RETURN statement in the format RETURN(exp).

17.7 THE RETURN STATEMENT (DIRECT OR COLLECT) (C)

Format: RETURN [(expression)];

The form without the expression must be used if the PROCEDURE was invoked as a subroutine (with a CALL statement.)

The form with the expression must be used if the PROCEDURE was invoked as a function. In this case, the expression is evaluated and converted to the data type of the returns-option, if any, or of the default returns-option if you didn't specify one in your PROCEDURE statement.

In either case, control returns to the "point of invocation"; that is, the point where the PROCEDURE was originally invoked.

The RETURN statement must appear inside a PROCEDURE. It may appear inside a BEGIN/END block nested inside a PROCEDURE, as long as the BEGIN/END block is not an ON-unit. In that case, the BEGIN/END block is terminated normally before a RETURN is made from the PROCEDURE.

A RETURN statement may not be executed from an ON-unit.

17.7.1 Executing The End Statement Of A Procedure (C)

If you execute the END statement of a PROCEDURE, then a RETURN statement with no expression is simulated. This is legal only in a PROCEDURE invoked as a subroutine (with a CALL statement).

17.8 MATCHING ARGUMENTS TO PARAMETERS (D)

When a PROCEDURE is invoked, the CALL statement or function reference must specify exactly the same number of arguments as there are parameters in the PROCEDURE statement.

17.8.1 Data Attributes Of The Parameters (D)

Each parameter in your PROCEDURE has specific data attributes. You may specify these data attributes in a separate DECLARE statement in the PROCEDURE. The rules for such a declaration are given below.

If you do not have a separate declaration for the parameter, then CPL assigns default attributes to the parameter in the same way that it assigns them to an ordinary identifier. That is, if there is no DEFAULT statement in your program, then the parameter will be a FLOAT scalar. If you have a DEFAULT statement, then the parameter will be a

FIXED or FLOAT scalar, depending upon the first letter of the parameter name as specified in your DEFAULT statement.

17.8.2 Rules For The Separate Declaration For The Parameter (D)

Here are the rules for the attributes which may be specified for a parameter:

1. Any variable data type may be specified -- FIXED, FLOAT, CHAR [VAR], BIT [VAR, POINTER]. If no data type is specified, then the usual default rules apply.

2. The storage class PARAMETER may be specified. No other storage class may be specified. If no storage class is specified, then the storage class will default to PARAMETER. (CPL knows that the identifier is a parameter since it appeared in the parameter list of a PROCEDURE statement.)

3. You may specify that the parameter is an array. You do this by specifying array bounds in the usual manner.

4. You may not use variables in the length specification of a BIT or CHAR attribute, or in the dimension bounds for an array specification. (That is, you may not use variable extent expressions in your declaration.)

5. However, you may code an asterisk in place of a length specification or an array bound. If you do this, then you must specify PARAMETER. In that event, the corresponding values will be taken from the arguments.

   For example, the following declarations are legal:

   ```
   DECLARE A CHAR(*) PARAMETER;
   DECLARE B(*,*) PARAMETER;
   DECLARE C(*) BIT(*) VAR PARAMETER;
   ```

17.8.3 Rules For Matching Arguments And Parameters (D)

For each argument/parameter pair in a PROCEDURE invocation, there are three things that CPL can do:

1. CPL can match the argument directly to the parameter without creating a dummy. In this case, any change to the parameter will change the value of the argument.

2. CPL can create a dummy argument, and match that to the parameter. In that case, any change to the parameter will change the value of the dummy, but will not change the value of the original argument.

3. CPL can take an error stop. This will happen if the argument and parameter are incompatible.

The following sections give the rules when each of these happens.

17.8.3.1 Case 1 -- No Dummy Is Created (D) - This case applies if all the following are true:

1. The argument must be simple identifier or subscripted identifier.

2. The data type of the argument must be identical to the data type of the parameter. Note that CHARACTER VARYING is a different data type from CHARACTER NONVARYING.

3. If the argument is a scalar or a subscripted array element, then the parameter must be a scalar. If the argument is an array, then the parameter must be an array with the same number of dimensions.

4. For each length value or array bound value among the attributes of the argument, the corresponding extent expression must be either (a) an asterisk or (b) a constant expression equal to that value in the argument.

17.8.3.2 Case 2 -- A Dummy Argument Is Created (D) - A dummy argument is created when all the following happen:

1. The argument and the parameter are both scalar.

2. One of the conditions of case 1 does not hold. This means that the argument is an expression or constant or it has a different data type or length value from the parameter.

17.8.3.3 Case 3 -- Match Cannot Be Made (D) - This is the error case. This case holds whenever any of the following hold:

1. The argument has a non-computational data type.

2. The argument is a scalar and the parameter is an array.

3. The argument is an array and the parameter is a scalar.

4. The argument and parameter are both arrays, but they have a different number of dimensions.

5. The argument and parameter are both arrays, but they have different data types.

6. The argument and parameter are both arrays with the same data type, but either the length expression or the array bounds for the parameter are different from those for the argument. This cannot happen for any extent expression in the parameter declaration for which an asterisk is coded.

17.8.4 Some Examples Of Argument Matching (D)

Consider the following program:

```
 10.             DECLARE D CHAR(100), E(10) FLOAT ;
 20.             CALL PR(A,B,(C),D,E(3));
                 ....
100.     PR:     PROCEDURE(V,W,X,Y,Z) ;
110.             DECLARE V FIXED ;
120.             DECLARE Y CHAR(*) ;
                 .....
200.             END PR ;
```

Let us consider each of the four argument/parameter pairings.

1. The argument A is FLOAT and the parameter V is FIXED. A will be converted to FIXED and a dummy argument will be created.

2. The argument B and the parameter W are both FLOAT, so that no dummy will be created.

3. Enclosing the argument C in parentheses is a signal to CPL to create a dummy argument. This happens because (C) is considered to be an expression.

4. The argument D is CHAR(100) and the parameter Y is declared to be CHAR(*); No dummy argument is created. If you reference LENGTH(Y) inside the PROCEDURE, then CPL will return 100.

5. E(3) is a FLOAT scalar argument and since the parameter Z is FLOAT, no dummy will be created. If you change the value of Z inside the PROCEDURE, then the value of E(3) will be changed.

17.9 RECURSIVE PROCEDURES (D)

It is possible for you to invoke a procedure which is already active. If this happens, then the invocation is called a "recursive" invocation.

If you invoke a PROCEDURE recursively, then CPL makes new copies of the parameters and allocates new copies of all AUTOMATIC storage. All STATIC variables are the same in all invocations of the PROCEDURE.

Storage allocation has been described in detail in a preceding chapter, entitled "Block Structure and Declaration Scope Rules."

Generally if you have a choice between using a recursive procedure and coding a DO-loop, you should choose the latter since it is more efficient.

The FAC PROCEDURE shown near the beginning of this chapter is coded with a loop. Here is another way to code the same function using a recursive PROCEDURE.

This PROCEDURE is based on the following "recursive" definition of factorial:

```
FAC(0) = 0
FAC(N) = N*FAC(N-1)
```

If you have never seen this recursive definition of factorial, you should try to use the formulas in the definition to compute FAC(3), to see how it works.

Here is the translation of this definition into a recursive PROCEDURE:

```
FAC:     PROCEDURE (NUM) ;
         IF NUM<1 THEN RETURN(1);
         ELSE RETURN(NUM*FAC(NUM-1));
         END FAC ;
```

17.10 USE OF THE SNAP STATEMENT (C)

If your program stops in the middle of a PROCEDURE and you are wondering how it got there, then type the "SNAP" statement.

The format of the snap dump produced by the SNAP statement has been described in the chapter on block structures.

CHAPTER 18

ON-CONDITIONS AND ERROR HANDLING (C-D)

18.1 WHY DO YOU NEED ON-CONDITIONS? (C)

Suppose you are writing a program which is to be used by other people. You would like this program to continue executing under all circumstances, even in case of error. You may use ON-units to guarantee this.

There are a number of reasons that such a program might stop executing. Here are some possibilities:

1. The person running the program may type in illegal input.
2. The program may be unable to open an output file due to a lack of file space.
3. The program may reach end-of-file on an input file.
4. The program may contain a hitherto unknown bug.

By means of ON-units, you may write your program so that it can analyze all of the above conditions and decide whether and how execution is to continue.

18.2 SOME INTRODUCTORY EXAMPLES (C)

18.2.1 Example Of An ERROR ON-unit (C)

Suppose you are writing a program for others to use, and the program contains a GET LIST statement. You would like the program to continue even if the person using the program types invalid input in response to the GET LIST.

Consider the following program segment:

```
        ....
        ON ERROR BEGIN;
        PUT LIST('ILLEGAL INPUT TRY AGAIN'); PUT SKIP;
        GO TO GET;
        END;
        PUT LIST('ENTER VALUE:');
GET:    GET LIST(VALUE);
        ON ERROR SYSTEM;
        ....
```

Let us suppose that this program is executed. When control reaches the first ON statement, here is what happens:

1. CPL remembers the fact that you have executed an ON statement here, in case an error later occurs.

2. CPL skips over all statements between the BEGIN sub-statement of the ON statement and its corresponding END statement (3 statements down).

3. CPL transfers control to the PUT LIST statement following the END statement.

Now, when the GET LIST statement is executed, if there is no error in the input, then execution will simply continue. But if there is an error, then CPL will "raise" the "established ON-unit"; that is, CPL will execute the statements between the BEGIN and END statements which appear in conjunction with the ON statement.

Therefore, the function of the ON statement is: To specify a statement, or group of statements, which is to be executed in the event an error occurs.

The last statement in the segment above, ON ERROR SYSTEM, reverses the effect of the preceding ON statement. When the SYSTEM option is used, it tells CPL that, for any subsequent errors, the standard system action is to be taken. The standard system action is usually to type an error message and return to command level. It is a good precautionary measure to include this statement in the program. If you do not include it, then if any unexpected error occurs (say, due to an error in the program), then the preceding ON ERROR statement will still be in effect, and the results will be confusing.

18.2.2 Example Of An ENDFILE ON-unit (C)

Let us consider a different type of "error," reaching the end of an input file. This, of course, is not really an error. But it has one feature of an error in that it is unpredictable -- when you are writing the program, you cannot usually predict when an end of file will occur.

Here, for example, is a program segment which adds up all the numbers in a file :

```
        DECLARE F FILE ;
        OPEN FILE(F) INPUT ;
        ON ENDFILE(F) GO TO EOF;
        SUM = 0;
LOOP:   GET FILE(F) LIST(X) ;
        SUM = SUM + X;
        GO TO LOOP;
EOF:    CLOSE FILE(F);
        ....
```

In this program segment, the third statement specifies the following: If an end of file occurs in file F, then GOTO the statement with statement label EOF.

In this example, the "ON-unit" consists of just a single statement -- GOTO EOF -- rather than a BEGIN/END block containing a group of statements. In this case, CPL places a dummy BEGIN statement before the GOTO statement, and a dummy END statement after; CPL does this without any indication to you that it has done so.

18.3 THE ON STATEMENT (COLLECT ONLY) (C)

```
Format:   ON condition-list [SNAP] on-unit;
or        ON condition-list SYSTEM ;
```

The "condition-list" is either a single condition-name (these are described below) or a list of condition-names, separated by commas.

The "ON-unit" is either a single statement, usually a GOTO statement, or else a BEGIN/END block containing a group of statements. If it is a single statement, it may not be an IF, DO, END, RETURN, FORMAT, PROCEDURE, DECLARE or DEFAULT statement. An ON-unit, whether a single statement or a BEGIN/END block, may not have a statement label. (This restriction is as specified by the ANSI PL/I standard.)

In the first of the two formats specified above, the ON statement specifies to CPL what action is to be taken when an error or other exceptional condition occurs. CPL "remembers" the location of the ON-unit to be executed when the exceptional condition occurs and transfers control to the statement following the ON-unit. If the exceptional condition subsequently occurs, then CPL "invokes" the ON-unit, and executes the statement or statements specified therein. The ON-unit statement or statements are not executed until the exceptional condition occurs.

If you specify the SNAP option, then, before the ON-unit statements are executed, CPL will type out the following:

1. The error message which would have been typed if no ON-unit had been established.
2. A snap dump, identical in format to the snap dump produced by the SNAP statement. Refer to the description of that statement in the chapter on "Block Structure and Declaration Scope Rules."

In the second of the formats, the format using the SYSTEM option, the ON statement specifies that CPL is to take the "standard system action" for the exceptional condition. For most errors, this means that CPL will type an error message and terminate execution.

18.4 GOOD PROGRAMMING PRACTICES WITH ON-UNITS (C)

18.4.1 Normal Termination Of An On-unit (C)

In the examples given above, all of the ON-units contained a GOTO statement, and so when the ON-units were invoked, the invocations were always terminated "abnormally", by means of a GOTO statement.

If your program executes the END statement of an ON-unit, then you are attempting to terminate the ON-unit normally. This is usually an error.

It is good programming practice to make certain that your ON-units always terminate abnormally, with a GOTO statement which transfers control out of the ON-unit.

18.4.2 Avoiding ON-unit Recursion Loops (C)

Suppose you executed an ON ERROR statement, specifying an ON-unit to be invoked when any error occurs.

Now let us suppose that an error occurs and your ON-unit is invoked. Let us suppose further that lightning strikes again and an error occurs inside your ON-unit. Then the ERROR ON-unit will be invoked again. And if the same error occurs again, you will get into an infinite loop raising the ON-unit over and over, recursively. This will continue until you have so many active invocations of your ON-unit that you run out of core storage.

To guard against this possibility, you should use the SNAP option, so that you will be able to see your program entering this loop. If you do not wish to use the SNAP option, then you should put a statement in the beginning of your ON-unit whose execution will permit the looping process to stop. For example, you may use the following as the first statement of your ON-unit:

1. ON ERROR SYSTEM; If an error occurs while the ON-unit is active, then execution will stop and control will return to the terminal. No loop will occur. (NOTE: As will be explained below under "Scope of an ON-unit," when the ON-unit terminates without an error, say by a GOTO statement, then the ON ERROR SYSTEM will be cancelled.)

2. PUT LIST something. If the first statement of your ON-unit types something out, then you will be able to recognize when the loop described above occurs, an you can interrupt it with Control-C. A good choice for something to type out is the current value of the ONMSG built-in function, described below.

18.4.3 The ONMSG Built-in Function (C)

This built-in function returns a CHARACTER string containing the error message which would have been typed out if no ON-unit had been specified for the error.

The ONMSG built-in function takes no arguments. If you wish to type out the value returned by ONMSG, use the statement

```
?ONMSG();
```

18.5 LIST OF CONDITION NAMES (C-D)

These are the condition names that may appear in the ON statement.

18.5.1 The SUBSCRIPTRANGE Condition (C)

Abbrev: SUBRG for SUBSCRIPTRANGE

Format: SUBSCRIPTRANGE

This condition is raised whenever a subscript is evaluated and found to lie outside its specified bounds.

18.5.2 The STRINGRANGE Condition (C)

Abbrev: STRG for STRINGRANGE

Format: STRINGRANGE

The STRINGRANGE condition is raised whenever the second and third arguments to the SUBSTR built-in function or pseudo-variable are out of range.

Let k = the length of the first argument, let i = the value of the second argument, and let j = the value of the third argument, if it is specified, or else let j=k-i+1. Then the STRINGRANGE condition is raised if any of these inequailities are not satisfied:

1. i <= i <= k+1
2. 0 <= j <= k-i+1

18.5.3 The ZERODIVIDE Condition (C)

Abbrev: ZDIV for ZERODIVIDE

Format: ZERODIVIDE

The ZERODIVIDE condition occurs when an attempt is made to divide by zero.

18.5.4 The ENDFILE Condition (C)

Format: ENDFILE (ident)

where "ident" is an identifier which must be declared to be a FILE.

This condition is raised whenever a GET or READ statement fails for the specified FILE identifier because an end of file is reached.

If the file is not closed after ENDFILE occurs, then any subsequent GET or READ statement for that file immediately raises the ENDFILE condition again.

18.5.5 The UNDEFINEDFILE Condition (C)

Abbrev: UNDF for UNDEFINEDFILE

Format: UNDEFINEDFILE (ident)

where "ident" is an identifier which must be declared to be a FILE.

The UNDEFINEDFILE condition is raised whenever an attempt to open a file is unsuccessful for any reason.

Some of the reasons for which the UNDEFINEDFILE condition might be raised are the following:

1. A conflict in file attributes exists.
2. An input file does not exist.
3. A syntax error occurs in the file-specification given by the TITLE option of the OPEN statement.
4. An output file cannot be opened because there is no more disk space.

The UNDEFINEDFILE condition can be raised as the result of either an explicit open, with the OPEN statement, or an implicit open resulting from a GET, PUT, READ or WRITE statement.

18.5.6 The RECORD Condition (D)

Format: RECORD (ident)

where "ident" is an identifier which must be declared to be a FILE.

The RECORD condition is raised as a result of a READ operation, in the following two circumstances:

1. You have specified a READ statement and the next input record in the file is longer than the INTO character string.
2. You have specified a READ statement with an INTO variable which is a CHARACTER NONVARYING scalar or array, and the next input record from the file is shorter than the length of the CHARACTER string.

Before CPL raises the RECORD condition, it reads the input record anyway, and places characters from the record into the INTO variable storage. In the first of the above two cases, CPL stops storing characters when the CHARACTER string storage is filled up; however, CPL still reads to the end of the record. In the second case, the remaining characters in the INTO variable are left unchanged from their values before the READ statement was executed.

The statements of the preceding paragraph imply that after the RECORD condition is raised, it is meaningful for your program to examine the contents of the INTO variable to determine some information about the input record.

18.5.7 The ERROR Condition (C)

Format: ERROR

All of the conditions listed above are raised only for specific types of errors. The ERROR condition can be raised for any type of program error, including those errors covered by the above named conditions.

Specifically, the ERROR condition can be raised under the following circumstances:

1. An error occurs, and CPL does not have a specific ON-condition name for that type of error.

2. An error occurs, and CPL has a specific ON-condition name for that type of error, but your program has not established an ON-unit for that condition name with the ON statement.

18.5.8 The CONDITION Condition (C)

Abbrev: COND for CONDITION

Format: CONDITION (ident)

This is a programmer-defined condition. "ident" is any CPL identifier.

This condition can be raised only by means of the SIGNAL statement.

18.5.9 The ATTENTION Condition (D)

Abbrev: ATTN for ATTENTION

Format: ATTENTION

This condition is raised when you type a Control-C while your program is executing.

CAUTION: If your program establishes an ATTENTION ON-unit, then you will have lost the ability to stop a looping program by typing Control-C. This means that you will have to ask the system operator to stop your program for you.

18.6 THE SIGNAL STATEMENT (COLLECT ONLY) (C)

Format: SIGNAL condition-name ;

The SIGNAL statement is used to raise an ON-condition artificially.

For example, if your program executes the statement "SIGNAL ENDFILE(F);", then CPL will raise the ENDFILE(F) condition, just as if an end of file had occurred on a GET or READ statement for file F.

If your program has not established an ON-unit for the specified condition-name, then CPL will type a message of the form "CONDITION condition-name SIGNALLED" and will continue executing your program with the statement following the SIGNAL statement.

If your program has an established ON-unit for the specified condition, then the ON-unit will be invoked. If the ON-unit terminates normally (that is, the END statement is executed), then execution continues with the statement following the SIGNAL statement.

18.7 SCOPE OF AN ON-UNIT (D)

The ON statement associates an ON-unit with a specific condition. Once this association is established, it remains so until one of the following occurs:

1. It is overridden by another ON statement specifying the same condition. The overriding ON statement can specify a different ON-unit, or it can specify the SYSTEM option.

2. The block in which the ON statement was executed is terminated. When that happens, all ON-units which had been established before the block was entered, but which had been overridden inside the block, are automatically reestablished.

3. A REVERT statement, specifying the same condition, is executed. The REVERT statement is described below.

An established interrupt action (either an ON-unit or a SYSTEM option for a specified condition) passes from a block to any block that it activates, and the action remains in force for all subsequently activated blocks, unless it is overidden by the execution of another ON statement for the same condition. If it is overriden, the new action remains in formce only until that block is terminated or until a REVERT statement is executed cancelling the effect of the overriding ON statement. When control returns to the activating block, all established interrupt actions that existed at that point are re-established. This makes if impossible for a subroutine to alter the interrupt action established for the block that invoked the subroutine.

18.8 THE REVERT STATEMENT (DIRECT OR COLLECT) (D)

Format: REVERT condition-name [,condition-name]...

If you execute a REVERT statement for a specified condition-name, then any ON statement executed inside the current block invocation, whether it specifies an ON-unit action or the SYSTEM option, is cancelled. The action to be taken for the specified condition reverts to what it was when the current block was first invoked.

CHAPTER 19

BASED STORAGE AND POINTERS (D)

This chapter deals with a very sophisticated programming concept, and should be skipped or skimmed on the first reading of the CPL manual.

In a preceding chapter entitled "STORAGE CLASSES," the storage classes AUTOMATIC, STATIC and CONTROLLED were discussed.

In this chapter, we contrast those three storage classes with the BASED storage class.

We also introduce a new data type, the POINTER.

Finally, this chapter introduces the ALLOCATE and FREE statement for BASED storage. (The chapter on "STORAGE CLASSES" gives the format of these statements for CONTROLLED storage.)

19.1 <u>INTRODUCTION TO BASED STORAGE (D)</u>

Consider the following CPL statement:

```
          A = B + 1.5;
```

When CPL executes this statement, it automatically knows the following two things about the variables A and B:

1. The data type of A and B (presumably, in this example, FLOAT)
2. The location of the storage word containing the value of A and the location of the storage word containing the value of B.

These two statements are true whether the storage classes of A and B are AUTOMATIC, STATIC or CONTROLLED.

Even if there are multiple allocations of a variable, as would be the case, for example, if the storage class was CONTROLLED, then there is still always a unique "current" allocation and storage address for the variables.

In the case of BASED storage, however, the above two things are separated. The BASED variable specifies only the first of these (a data type) and you must specify the storage address separately.

If A anb B had been BASED storage, then the above statement would have to have been written something like the following:

```
          P1 -> A = P2 -> B + 1.5;
```

this form of statement representation introduces some new concepts.

The operator "->" is called the "pointer qualifier" operator. P1 and P2 would have to be DECLAREd somewhere to be POINTER variables. A POINTER variable specifies the location of the storage to be used in an operation.

A BASED identifier itself does not specify any storage. A BASED identifier specifies only a data type. It may not be used in an expression unless it is preceded by a POINTER value and the operator "->".

19.2 DECLARATION OF BASED STORAGE (D)

An identifier is DECLAREd to be BASED as in the following examples:

```
  10.     DECLARE IB FIXED BASED;
  20.     DECLARE (AB1,AB2) FLOAT BASED;
  30.     DECLARE BB BIT(36) BASED;
  40.     DECLARE CBV(5) CHAR(20) VAR BASED;
```

None of the identifiers DECLAREd in the above examples can be used alone. Each one must be preceded by a POINTER value and the operator "->" in order to specify the storage location. The BASED identifier alone cannot specify a storage location.

19.3 POINTER DATA TYPE (D)

Abbrev: PTR for POINTER

The following examples show how a POINTER variable is DECLAREd:

```
  10.     DECLARE P POINTER;
  20.     DECLARE PAC(5) POINTER CONTROLLED;
```

19.4 THE ADDR BUILT-IN FUNCTION (D)

The ADDR built-in function is the main tool which you will use to obtain POINTER values. ADDR takes one argument, an identifier or a subscripted identifier, and it returns the "address" of that argument.

For example, if P is a POINTER variable, and you execute the statement

```
          P = ADDR(J) ;
```

then P will be set to the address of the identifier J. That is, P will "point" to the storage location occupied by the variable J.

IMPLEMENTATION NOTE

ADDR does not return the "address" in the usual sense. A CPL POINTER value consists of a DECsystem-10/20 word address and a bit displacement within that 36-bit word. This means that a POINTER can point to any CPL data item, whether it be a full word or a character or bit within a word.

19.5 USE OF POINTERS AND BASED STORAGE (D)

Consider the following short program:

```
  10.      DECLARE P POINTER;
  20.      DECLARE IB BASED;
  30.      P = ADDR(J);
  40.      J = 10;
  50.      PUT LIST( P -> IB );
  60.      P->IB = 20;
  70.      PUT LIST(J);
```

Statement 30 sets P to the address of J, and statement 40 sets J to 10.

Statement 50 uses the "->" operator with a POINTER P and a BASED variable IB. Since P points to J, the reference to P->IB is the same as a reference to J. Therefore, statement 50 types out the value 10.

Statement 60, by the same reasoning, sets J to 20. Thus statement 60 will set J to 20. Thus, statement 70 will type out the value 20.

19.6 ADDITIONAL EXAMPLES (D)

19.6.1 Two POINTERs With Same BASED Variable (D)

Here is an example of a short program which uses two POINTERs with the same BASED variable.

```
  10.      DECLARE (P,Q) POINTER;
  20.      DECLARE IB BASED;
  30.      P = ADDR(I);
  40.      Q = ADDR(J);
  50.      P -> IB =5 /*SAME AS I=5 */;
  60.      Q->IB=P->IB+Q->IB /*SAME AS J=I+J*/;
```

In this example, there are two POINTERs, P and Q. Both of these POINTERs are used to qualify the same BASED identifier, IB. When P qualifies IB, the result is a reference to I; when Q qualifies IB, the result is a reference to J.

19.6.2 BASED CHARACTER Strings (D)

A POINTER variable may point to a character string, or to a particular character in a character string or character string array.

For example, consider the following short program:

```
  10.     DECLARE P POINTER;
  20.     DECLARE CA(5) CHAR(2);
  30.     DECLARE CB CHAR(3) BASED;
  40.     STRING(CA) = 'ABCDEFGHIJ';
  50.     P = ADDR(CA);
  60.     PUT LIST(P->CB) /* RESULT IS 'ABC' */;
  70.     P = ADDR(A(3));
  80.     PUT LIST(P->CB) /* RESULT IS 'EFG' */;
  90.     P = ADDR(CA(4));
 100.     PUT LIST(P->CB) /* RESULT IS 'GHI' */;
```

Statement 40 of this program uses the STRING pseudo-variable, which is described in the chapter on Built-in Functions and Pseudo-variables. This particular statement sets CA(1) to 'AB', CA(2) to 'CD', CA(3) to 'EF', CA(4) to 'GH', and CA(5) to 'IJ'.

CPL stores the characters of NONVARYING CHARACTER arrays one after the other in core storage. That is, the two characters in CA(2) directly follow the characters in CA(1) in core.

This means that if P is set to ADDR(CA), as in statement 50, and if P is used to qualify a CHAR(3) BASED string in the program, then the result will be the first three characters ('ABC') in the aggregate array.

Simlarly, when P points to CA(3), then P->CB refers to the two characters in CA(3) and to the first character in CA(4).

Note that this method will not work for CHARACTER VARYING arrays. The method fails because for ,character VARYING, CPL places before each element of the array a word containing the current length of that element of the array. Therefore, the characters in one element of the array are not directly adjacent to the characters in the next element.

19.6.3 BASED BIT Arrays (D)

The same technique can be used with BIT NONVARYING arrays.

If P is set to the address of a BIT in the string, then if P qualifies a BASED BIT string, you can refer to bits in adjacent elements of the array.

19.6.4 Mixing Data Types (D)

It is possible to let P point to data of one data type, and then use P to qualify a BASED variable of a different data type.

This type of operation is always machine dependent, since the results depend upon the particular internal bit format of the data on the particular machine.

For example, here is a program which types out the characters 'ABCD' in BIT(7) format:

```
  10.     DECLARE P POINTER;
  20.     DECLARE CA(4) CHAR(1);
  30.     STRING(CA) = 'ABCD';
  40.     DECLARE BB BIT(7) BASED;
  50.     DO I = 1 TO 4;
  60.     P = ADDR(CA(I));
  70.     PUT LIST(P->BB);
  80.     END;
```

In this program, statement 30 uses the STRING pseudo-variable to set the four CHAR(1) elements of the CA array to 'A', 'B', 'C' and 'D', respectively.

Statement 60 sets P to the address of CA(I), and statement 70 types out that character in BIT(7) format. Therefore, this program will produce the following output:

```
          '1000001'B '1000010'B '1000011'B '1000100'B
```

These are the representations of the first four characters in ASCII BIT(7) format.

19.7 COMPARISON OF POINTERS (D)

You may wish to compare two POINTER variables, for example as a test in an IF statement or in the WHILE option of the DO statement.

You may use the operators "=" and "^=" to compare two POINTER values for equality or inequality, respectively.

The other comparison operators (e.g., greater than) may not be used to compare POINTER values.

For example, in the statement

```
          IF P1=P2 THEN GO TO XYZ;
```

where P1 and P2 are POINTERs, control will transfer to XYZ if P1 and P2 point to the same data item (word or character or bit).

IMPLEMENTATION NOTE

For P1 and P2 to be considered equal, they must point to data at the same word address and same bit displacement within the word.

19.8 BASED ARRAYS (D)

An array may have the BASED storage class attribute. Here are some examples of such arrays.

19.8.1 Example Of BASED Array (D)

Here is a program segment which uses a BASED array called BARR:

```
  10.     DECLARE P POINTER;
  20.     DECLARE BARR(2) BASED;
  30.     DECLARE A(50);
          ....
 120.     P = ADDR(A);
 130.     X = P -> BARR(1) /* SAME AS X=A(1) */;
 140.     Y = P -> BARR(2) /* SAME AS Y=A(2) */;
          ....
 240.     P = ADDR(A(15));
 250.     X = P -> BARR(1) /* SAME AS X=A(15) */;
 260.     Y = P -> BARR(2) /* SAME AS Y=A(16) */;
```

Statement 20 DECLAREs a BASED array BARR. Statement 120 sets the POINTER P to the address of the array A. Thus, a reference to the array P->BARR is a reference to the array consisting of the first two elements of the array A. Therefore, P->BARR(1) in statement 130 refers to A(1), and P->BARR(2) in statement 140 refers to A(2).

Statement 240 shows a different kind of example. P is set to point to A(15), the 15'th element of the array A. Therefore, a reference to the array P->BARR is now a reference to the array consisting of the two elements A(15) and A(16). Therefore, P->BARR(1) refers to A(15) in statement 250, and P->BARR(2) refers to A(16) in statement 260.

19.8.2 Example Of BASED CHARACTER Array (D)

By using BASED CHARACTER arrays, it is possible to treat a string of characters as a scalar character string in one context and as an array in another context.

Consider the following program segment:

```
  10.     DECLARE S CHAR(4);
  20.     DECLARE SB(4) CHAR(1);
  30.     DECLARE P POINTER;
  40.     P = ADDR(S);
  50.     S = 'ABCD';
  60.     PUT LIST(P->SB(1)) /* RESULT IS 'A' */;
  70.     PUT LIST(P->SB(3)) /* RESULT IS 'C' */;
```

Statement 40 sets P to point to the character string S, which statement 50 sets to 'ABCD'.

Thus, a reference to P->SB is a reference to an array consisting of the four characters in the scalar string S. Therefore, P->SB(1) has the value 'A' in statement 60, and P->SB(3) refers to 'C' in statement 70.

19.9 THE ADDR AND NULL BUILT-IN FUNCTIONS (D)

The ADDR built-in function was defined in a previous section. This section gives more information and examples of this function, and introduces a new built-in function, NULL.

19.9.1 The NULL Built-in Function (D)

The NULL built-in function takes no arguments. It returns a "null" POINTER value.

When a POINTER variable is allocated, it does not point to any data and is said to be a "null" POINTER value.

If you wish to set a POINTER to an explicit null value, then you may use a statement like

```
          P = NULL() ;
```

to set the POINTER P to a null value.

Another use for the NULL built-in function is to compare with a POINTER variable to see whether the variable has a non-null value. The list processing example at the end of this chapter shows some examples of this use of the NULL built-in function.

19.9.2 Use Of ADDR As POINTER Qualifier (D)

The ADDR built-in function can be used directly as a POINTER qualifier of a BASED variable without going through an intermediate POINTER variable. For example, consider the following program segment:

```
  10.     DECLARE S(4) CHAR(1);
  20.     DECLARE BB BIT(7) BASED;
          ....
 140.     PUT LIST( ADDR(S(3)) -> BB );
```

Statement 140 uses the POINTER value ADDR(S(3)) directly as a POINTER qualifier for the BASED identifier BB. Thus, this statement will type out the character S(3) in BIT(7) format.

19.9.3 Use Of ADDR To "Increment" A POINTER (D)

It is illegal to perform ordinary arithmetic operations on a POINTER variable. (For example, if P is a POINTER, then you may not compute P+1 to, say, increase the address in P by 1 word.)

However, you may use the ADDR built-in function with a BASED argument to "increment" a POINTER variable to point to the next word or character or bit.

Here is an example of the technique that is used:

```
  10.     DECLARE P POINTER;
  20.     DECLARE S CHAR(100);
  25.     DECLARE SBA(2) CHAR(1) BASED;
  30.     P = ADDR(S);
  40.     DO I = 2 TO 100;
  50.     P = ADDR( P -> SBA(2) );
          ....
 120.     END;
```

Statement 30 sets P to point to the first character in the string S. Each time through the loop, statement 50 "increments" the pointer P to point to the next character in the string. Therefore, P will point in turn to the 2'nd through 100'th characters of the string S.

19.10 POINTERS WITH DO STATEMENTS (D)

There are two special topics to discuss under this general heading:

1. Use of a POINTER-qualified BASED DO-loop variable

2. DO-variable with POINTER data type

These topics are discussed below.

19.10.1 POINTER-qualified BASED DO-loop Variable (D)

A DO-loop variable can be BASED, with a POINTER-qualifier. Consider, for example, the following DO-group:

```
  10.     DECLARE P POINTER, IB BASED;
          ....
 100.     P = ADDR(J);
 110.     DO P ->IB = 1 TO 10;
          ....
 150.     END;
```

In this simple example, statement 110 is equivalent to

```
          DO J = 1 to 10;
```

since P points to the variable J.

Note the following: If you change the value of P inside the DO group, you will not change the location of the DO variable. Once the location of a DO variable has been established, it cannot be changed for the duration of the DO loop.

19.10.2 DO Variable With POINTER Data Type (D)

The DO variable may have the POINTER data type. In this case, the TO and BY clauses are illegal, but the REPEAT and WHILE clauses are legal.

19.10.2.1 Simple Example (D)

- Here is a simple example illustrating the use of a POINTER DO-variable:

```
  10.     DECLARE P POINTER, IB BASED;
          ....
 110.     DO P = ADDR(I), ADDR(J), ADDR(K);
 120.     PUT LIST(P->IB);
 130.     END;
```

The DO loop will be iterated three times, with P pointing to I, J and K, respectively, during each iteration. Therefore, statement 120 will type the value of I the first time through the loop, the value of J the second time, and the value of K the third time.

19.10.2.2 Example With REPEAT Clause (D)

- Here is an example of a POINTER DO variable. An example was given a page or two back in the sub-section entitled "Use of ADDR to "Increment" a POINTER." In that example, the POINTER variable was "incremented" as the first statement of the DO loop so that it pointed to successive characters in the string S, starting with the 2'nd and continuing to the 100'th.

The following example is a variation of that example. The DO-loop variable is initialized to point to the first character in the array for the first iteration of the loop. (This is a difference with the last example -- there the POINTER pointed to the second character for the first iteration.) The REPEAT clause "increments" the DO variable by one character.

```
  10.     DECLARE P POINTER;
  20.     DECLARE S CHAR(100);
  25.     DECLARE SBA(2) CHAR(1) BASED;
  30.     P = ADDR(S);
  40.     DO P = ADDR(S) REPEAT(ADDR(P->SBA(2)))
                    WHILE (P->SBA(1) ^= 'Z');
          ....
 120.     END;
```

This example is similar to the one referenced above, but the "incrementing" is done in the REPEAT clause rather than in a separate statement. The WHILE clause specifies that the loop is to be terminated as soon as P points to the character 'Z'.

19.11 POINTERS WITH PROCEDURES (D)

There are three special topics to discuss under this general heading:

1. PROCEDURE invocations with BASED arguments (dummy and non-dummy)
2. PROCEDURE invocations with POINTER arguments
3. Function PROCEDURE invocations with RETURNS(POINTER)

These topics are discussed below.

19.11.1 PROCEDURE Invocations With BASED Arguments (D)

You may invoke a PROCEDURE, either by means of a CALL statement or by a function reference, and you may pass a BASED argument. Consider the following program segment:

```
 10.                DECLARE P POINTER, IB BASED;
 20.                J = 5;
 30.                P = ADDR(J);
 40.                CALL PR( P->IB );
 50.                CALL PR( (P->IB) );
                    ....
250.        PR:     PROCEDURE(IARG);
                    ....
300.                END PR;
```

Statement 40 calls PR passing the real argument P->IB. Since P has been set to the address of J, this statement is equivalent to "CALL PR(J);" Statement 40 could also have been written:

```
CALL PR( ADDR(J)->IB );
```

In this form, the intermediate POINTER variable P would not have been needed at all.

If PROCEDURE PR changes the value of its argument, IARG, inside the PROCEDURE, then the value of J will automatically change.

Statement 50 is similar to statement 40, except that the extra set of parentheses causes a dummy argument to be created. In this case, if the PROCEDURE PR changes the value of IARG, then the dummy will change but J will not.

19.11.2 PROCEDURE Invocations With POINTER Arguments (D)

You may invoke a PROCEDURE, either by means of a CALL statement or by means of a function invocation, passing a POINTER argument.

Here is an example of a subroutine which is passed a POINTER, and which types out the value of the bit pointed to by the POINTER:

```
 10.              DECLARE P POINTER;
                  ....
110.              CALL TYPEBIT(P);
120.              CALL TYPEBIT(ADDR(J));
                  ....
720.     TYPEBIT: PROCEDURE(PTR);
730.              DECLARE PTR POINTER;
740.              DECLARE BB BASED BIT(1);
750.              PUT LIST(PTR->BB);
760.              END TYPEBIT;
```

Note that if you pass PROCEDURE TYPEBIT a POINTER argument, then the parameter must be DECLAREd to be a POINTER (as in statement 730 of the above example). Statement 750 takes the POINTER parameter and uses it to qualify a BASED identifier BB. Statements 110 and 120 call TYPEBIT, passing P and ADDR(J) as arguments, respectively.

The rules for dummy and real arguments are the same for POINTERs as for other data types. If you enclose the argument in an extra set of parentheses, then a dummy POINTER argument is created.

19.11.3 Function PROCEDURE Invocations With RETURNS(POINTER) (D)

You may use the RETURNS(POINTER) option of the PROCEDURE statement to specify that a function PROCEDURE is to return a POINTER value. A reference to such a function may be assigned to a POINTER variable by means of an assignment statement, or it may be used directly to qualify a BASED variable. Both of these uses are illustrated in the example below.

In this example, INCR is a PROCEDURE function which "increments" a POINTER argument by one character. It takes a POINTER argument and returns a POINTER value pointing one character beyond.

```
 10.              DECLARE P POINTER;
 20.              DECLARE S CHAR(5);
 30.              DECLARE CB CHAR(1) BASED;
 40.              S = 'ABCDE';
 50.              PUT LIST( ADDR(S) -> CB) /* RESULT IS 'A' */;
 60.              P = INCR(ADDR(S));
 70.              PUT LIST(P->CB) /* RESULT IS 'B' */;
 80.              PUT LIST( INCR(P)->CB ) /* RESULT IS 'C' */;
 90.              PUT LIST( INCR(INCR(P))->CB ) /* RESULT IS 'D' */;
                  ....
620.     INCR:    PROCEDURE(PP) RETURNS (POINTER);
630.              DECLARE PP POINTER;
640.              DECLARE CBA(2) CHAR(1) BASED;
640.              RETURN (ADDR(PP->CBA(2)));
650.              END INCR;
```

Study the example carefully to see how references to the INCR function are used in statements 60 through 90. Note statement 90 in particular, where INCR is called twice to increment the POINTER argument twice.

19.12 BASED AND DIMENSIONED POINTERS (D)

A POINTER variable may be dimensioned, BASED, or both. A BASED POINTER qualifier may itself be qualified by another POINTER. For example, if your program contains the following DECLARE statements:

```
  10.      DECLARE P POINTER;
  20.      DECLARE PBA(5) POINTER BASED;
```

then your program may later contain a construct which looks something like the following:

```
          P -> PBA(I) -> IB;
```

19.13 OTHER PLACES WHERE BASED VARIABLES ARE USED (D)

Generally, CPL permits a POINTER-qualifier BASED variable wherever it permits a variable. For example, your program might contain the following kinds of statements:

```
          PUT STRING(P->SB) LIST(X,Y);

          UNSPEC(P->IB)='1001'B;

          READ FILE(F) INTO(P->CB);
```

As these statements illustrate, a POINTER-qualified BASED variable can be used in the STRING option, as a pseudo-variable argument, and in the FROM or INTO option of the READ or WRITE statements.

19.14 VARIABLES IN BASED DECLARATIONS (D)

Extent expressions (string lengths and array bounds) in the declarations of BASED variables may contain variables. In that case, CPL will recompute the values of extent expressions each time that the BASED variable is referenced.

For example, if you have the declaration:

```
          DECLARE IAB(N,M+K) BASED;
```

in your program, and if you later reference

```
          P->IAB(I,J)
```

in a statement, then CPL will compare the subscript values (I and J) with the array bounds (N and M+K) determined by the current values of N, M and K.

19.15 THE ALLOCATE STATEMENT (DIRECT OR COLLECT) (D)

```
Format:   ALLOCATE ident [SET(pointer)];
or        ALLOCATE ident [SET(PTR)],IDENT [SET(PTR)] ...;

Abbrev:   ALLOC for ALLOCATE
```

The "ident" must have the CONTROLLED attribut (in which case the SET option may not be specified) or the BASED attribute (in which case the SET option must be specified). If the SET option is specified, it is followed in parentheses by a POINTER variable.

We have previously discussed the ALLOCATE statement for CONTROLLED storage only. We now give the general format, including the format for BASED storage.

19.15.1 ALLOCATE For CONTROLLED Storage (D)

If the "ident" has the CONTROLLED attribute, then the SET option may not be used..

CPL allocates storage for the specified identifier. If storage has already been allocated, then CPL allocates an additional copy.

If there are several allocations of a CONTROLLED variable in existence, then any statement which references the CONTROLLED variable will reference the most recent allocation of it.

See chapter entitled "Storage Classes" for an example of the ALLOCATE statement for CONTROLLED storage.

19.15.2 ALLOCATE For BASED Storage (D)

If the "ident" has the BASED attribute, then the SET option must be used. The "pointer" is a reference to a POINTER variable.

CPL allocates storage for the BASED variable, and sets the specified POINTER variable to point to it.

This type of storage allocation is different from other types in the following sense: You can only reference this storage by means of the POINTER variable. If your program "loses trac" of the value of the POINTER variable (for example, the POINTER variable is immediately assigned a different value), then the storage block is lost to your program completely, and there is no way to reference it again.

Examples of the ALLOCATE statement for BASED storage are given in the next chapter.

19.16 THE FREE STATEMENT (DIRECT OR COLLECT) (D)

```
Format:   FREE [pointer ->] ident
or        FREE [pointer ->] ident, [pointer->]ident ...
```

The "ident" must have the CONTROLLED attribute or the BASED attribute. The pointer qualifier must be used with a BASED identifier, and may not be used with a CONTROLLED identifier.

19.16.1 FREE For CONTROLLED Storage (D)

If the identifier has the CONTROLLED attribute, then there must exist at least one allocation of the specified identifier. CPL releases the storage occupied by the most recent allocation.

See the chapter entitled "Storage Classes" for an example of the FREE statement for CONTROLLED storage.

19.16.2 FREE For BASED Storage (D)

If the identifier has the BASED attribute, then a POINTER qualifier must be specified. The storage block pointed to by the POINTER qualifier is freed.

CPL imposes the following rule on this operation: The storage block being freed must be precisely a storage block which was allocated by a BASED ALLOCATE statement. No partial blocks may be freed. Therefore, the POINTER qualifier must point to the beginning of a storage block, and the size of the BASED variable must be identical to the size of the storage block.

Examples of the FREE statement for BASED storage are given in the next chapter.

19.17 RESTRICTIONS ON BASED STORAGE (D)

In order to guarantee the integrity of the CPL system, CPL imposes some restrictions on the use of BASED variables and POINTERs.

CPL will not permit you to use a POINTER to reference storage which is not inside a legitimate data area. This means that a POINTER variable can never point anywhere but to the inside of such a block. This means the following:

1. If you try to change the value of a POINTER by means of the UNSPEC pseudo-variable, you will not change the value of the POINTER.

2. If you try to "increment" a POINTER beyond the end of a data block, you will not be permitted to.

3. If a pointer-qualifier points to a data block which is too small for the the BASED variable being qualified, then CPL will consider the reference to be illegal.

Furthermore, as discussed above with the FREE statement, you must FREE a BASED storage block exactly as it was allocated by a BASED ALLOCATE statement.

CHAPTER 20

A LIST PROCESSING EXAMPLE (D)

This chapter gives a detailed example of BASED storage and POINTERs. This chapter depends heavily on the preceding chapter, and so may be skipped or skimmed on first reading.

20.1 DESCRIPTION OF APPLICATION (D)

The program we will describe is a simple accounting program. It reads data containing information about individuals, including name, address, salary and age, and performs the following processing:

1. Creates a "record" of the data in BASED storage
2. Sorts these "records" in alphabetical order by name
3. Prints out the sorted list.

We will not present the entire program which does all this; rather, we will give the crucial coding segments which perform the functions which are most important in an illustration of BASED storage and POINTERs.

20.2 THE BASIC DECLARATIONS (D)

The following declarations will be used throughout the example:

```
  10.     DECLARE PBASE POINTER;
  20.     DECLARE PAB(3) POINTER BASED;
  30.     DECLARE CAB(3) CHAR(20) VAR BASED;
  40.     DECLARE IAB(2) FIXED BASED;
  50.     DECLARE P POINTER;
```

As discussed above, we will need a "record" for each individual for whom we read a data card.

Each such "record" will actually consist of three BASED allocations, one for each of the BASED arrays PAB, CAB and IAB. These three BASED storage blocks will form a single "record" in the following manner:

1. PAB is a three-element POINTER array. The three elements in each allocation point to the following:

   1. The next "record" in alphabetical order (i.e., the address of the PAB block for the next "record")

   2. The CAB block for the current record

   3. The IAB block for the current record

2. CAB is a three-element CHARACTER array. It will contain the name, address and city of the person described by the current "record."

3. IAB is a two-element FIXED array. It will contain the salary and age of the person described by the current record.

It is through the use of the three POINTERs in the PAB array that the three components of the "record" can be considered as a single unit.

(Note: Other implementations of PL/I permit the use to declare a "structure" containing several different data types; with such an implementation, you would be able to combine PAB, CAB and IAB into a single BASED "structure." However, since CPL does not permit such declarations, this example uses three different declarations, one for each data type in the "record.")

The POINTER variable PBASE will point to the first PAB block, and each PAB(1) will point to the next PAB block. The last PAB(1) will have a "null" POINTER value.

20.3 INITIALIZING THE RECORD CHAIN (D)

In order to make processing easier, we will begin by allocating two dummy "records." These dummy records will contain the names "AAAAAAAAAAAAAAAAAAAA" and "ZZZZZZZZZZZZZZZZZZZZ", respectively. Later we will be adding records to the chain in alphabetical order, and we will always be able to assume that each new record will be inserted somewhere in the middle of the chain, never at the beginning or end of the chain.

We create the first dummy record as follows:

```
 60.     ALLOCATE PAB SET(PBASE);
 70.     ALLOCATE CAB SET(P);
 80.     P->CAB(1) = (20)'A';
 90.     PBASE->PAB(2) = P;
100.     ALLOCATE IAB SET(P);
110.     PBASE->PAB(3) = P;
```

The preceding six statements perform the following steps:

1. Statement 60 allocates the first PAB block, and sets PBASE to point to it.

2. Statement 70 allocates the first CAB block, and sets the pointer P to point to the new block. Statement 80 sets the name field to 'AAAAAAAAAAAAAAAAAAAA'.

3. Statement 90 causes PAB(2) to point to the new CAB block.

4. Statement 100 allocates the first IAB block, and causes PAB(3) to point to it.

We now allocate the dummy record which will terminate the chain. In the following code we take some shortcuts we didn't take in the above code.

```
 120.      ALLOCATE PAB SET(P);
 130.      PBASE->PAB(1) = P;
 140.      ALLOCATE CAB SET(P->PAB(2));
 150.      P->PAB(2)->CAB(1)=(20)'Z';
 160.      ALLOCATE IAB SET(P->PAB(3));
 170.      P->PAB(1)=NULL();
```

This sequence of code is not too different from the last sequence. But note that the ALLOCATE statements in lines 140 and 160 use a SET option which directly assigns the desired POINTER field, rather than going through an intermediate POINTER variable. In fact, lines 120, 140 and 160 could have all been replaced with a single ALLOCATE statement:

```
           ALLOCATE PAB SET(P), CAB SET(P->PAB(2)),
                        IAB SET(P->PAB(3));
```

20.4 PROCEDURE TO ADD A "RECORD" TO THE CHAIN (D)

After all the above code has been executed, the current chain will consist of the two dummy records.

Now we present a PROCEDURE which can be called to add the next record to the chain. It will insert the chain into the proper place in alphabetical order.

```
 700.      ADDNAM:   PROC(NAME,STREET,CITY,SALARY,AGE) RETURNS(PTR);
 710.              DECLARE (NAME,STREET,CITY) CHAR(20) VAR;
 720.              DECLARE (SALARY, AGE) FIXED; /*DECLARE PARMS*/
 730.              DECLARE (P,PNEW,PPREV) POINTER;
 740.              DO P=PBASE REPEAT(P->PAB(1))
                     WHILE (P->PAB(2)->CAB(1) < NAME);
 750.              PPREV = P;
 760.              END;
 770.              ALLOCATE PAB SET(PNEW), CAB SET(PNEW->PAB(2)),
                     IAB SET(NEW->PAB(3));
 780.              PNEW->PAB(2)->CAB(1) = NAME;
 790.              PNEW->PAB(2)->CAB(2) = STREET;
 800.              PNEW->PAB(2)->CAB(3) = CITY;
 810.              PNEW->PAB(3)->IAB(1) = SALARY;
 820.              PNEW->PAB(3)->IAB(2) = AGE;
 830.              PPREV->PAB(1) = PNEW;
 840.              PNEW->PAB(1) = P;
 850.              END ADDNAM;
```

Here is what this PROCEDURE does:

1. The arguments to the PROCEDURE pass all the information needed to create the new data record. (We assume that some preceding code read the data from a file and called this procedure with the proper arguments. We also assume that that code did some checking to see that the name contained legal characters.)

2. Statements 710 and 720 declare the parameters to have the correct attributes.

3. Line 730 declares three POINTER variables to be used locally inside this PROCEDURE.

4. Lines 740 through 760 search through the existing chain for the correct position of the new name (in alphabetical order). When this loop is completed, the new record will belong between the records pointed to by PPREV and P.

5. Line 770 allocates all the components of the new record, and lines 780 through 820 fill in all the fields with the values passed in the parameters to the PROCEDURE.

6. Lines 830 and 840 place the new record into the chain between PPREV and P.

20.5 <u>LISTING THE CHAIN (D)</u>

As a final piece of code, we show how a PROCEDURE can be called to list out all the names in the chain.

This PROCEDURE also shows how simple the REPEAT clause of the DO makes the process of following a linked chain.

```
900.     LISTNAMES:    PROCEDURE;
910.             DECLARE P POINTER;
920.             DO P = PBASE->PAB(1) REPEAT(P->PAB(1)
                   WHILE (P->PAB(1)^=NULL());
930.             PUT SKIP LIST(P->PAB(2)->CAB(1));
940.             END;
950.             END LISTNAMES;
```

Line 920 incorporates into a single DO statement all the logic that is needed to loop through the entire chain, skipping the first dummy record and stopping at the last dummy record.

Line 930 types out the name in the record pointed to by P. Additional statements could be added to type out the other fields in the record.

20.6 <u>A FINAL EXERCISE (D)</u>

If you would like to practice with the example given above, then you may wish to try the following exercise:

A LIST PROCESSING EXAMPLE (D)

Add a subroutine which does a simple bubble or interchange sort on the list by salary, and print out the list in increasing order by salary. Do this in the following manner: Change the declaration of PAB so that there is a fourth element, PAB(4). Invoke your PROCEDURE after the entire chain has been set up in alphabetical order, and then perform the sort so that the same chain is sorted in order by increasing order by salary through the PAB(4) field. That is, for each record, while PAB(1) continue to point to the next one in alphabetical order, PAB(4) will point to the next record in order of increasing salary.

CHAPTER 21

OTHER STATEMENTS (B)

21.1 THE NULL STATEMENT (DIRECT OR COLLECT) (B)

Format: `;`

This statement gives the PL/I language that aesthetically satisfying feeling of mathematical completeness that only a null statement can provide.

This statement takes no action.

EXAMPLE: `IF A>B THEN ; ELSE GO TO XYZ;`

If A>B then no action is taken; otherwise, control passes to the statement with label XYZ.

EXAMPLE: `/* THIS IS A COMMENT */;`

Comments may appear in any CPL statement. Sometimes, however, you may wish a comment to appear alone on a single line. When this happens, the statement is really a null statement.

21.2 THE STOP STATEMENT (DIRECT OR COLLECT) (B)

Format: `STOP;`

CPL will halt execution and return to command level.

21.3 THE DELAY STATEMENT (DIRECT OR COLLECT) (B)

Format: `DELAY(expression);`

CPL evaluates the "expression" and converts it to FIXED, if necessary, to obtain an integer value, n. CPL interprets n to be a number of milliseconds (thousandths of a second). CPL goes to sleep for the specified number of milliseconds.

WARNING: DELAY is not in the ANSI PL/I standard, and so may not be available in other PL/I implementations.

EXAMPLE: `DELAY(2000);` causes CPL to go to sleep for 2 seconds.

CHAPTER 22

STRUCTURED AND GOTO-LESS PROGRAMMING (C)

During the last few years, new programming techniques have been developed to make programming faster and easier and to make programs more reliable and easier to maintain and modify.

These techniques are usually given the name "structured programming," although that phrase has been given different meanings by different authors. In this section, we give a set of structured programming rules for CPL, including a modified rule for GOTO-less programming.

22.1 TRADITIONAL PROGRAMMING WITH GOTOs (C)

Many people feel that sloppy use of the GOTO statement is the greatest factor in making programs difficult to maintain and modify. If you have ever had the job of trying to figure out a program which had been written and modified by several people, you will probably agree. It is almost impossible to trace logic when a program has GOTOs which jump back and forth all over the place.

22.2 STRUCTURED LOOPS (C)

All program loops must be done by DO statement loops. You may never program a loop by means of a statement of the form IF...THEN GOTO....

The use of DO statements for program loops has been the subject of this chapter.

22.3 TESTING CASES (C)

Many programs have statement groups of the following type:

```
IF A=1 THEN GO TO L1;
ELSE IF A=2 THEN GO TO L2;
ELSE IF A=10 THEN GO TO L3;
ELSE GO TO L4;
```

There are four GOTO statements in this group. They specify transfers to statements which are remote from the tests and which may appear anywhere in the program -- near the beginning of the program, or near the end.

The same types of tests can be made in a structured in the following manner:

```
IF A=1 THEN DO;
...
END;
ELSE IF A=2 THEN DO;
...
END;
ELSE IF A=10 THEN DO;
...
END;
ELSE DO;
...
END;
```

When you are trying to understand code written in this manner, you can easily see what code goes with what test. You always know what block of code goes with each test, and you always know that there is only one way to get to each block of code.

22.4 RESTRICTIONS ON GOTO STATEMENTS (C)

Some practitioners of structured programming claim that the GOTO statement should not be used at all.

This writer has experimented with GOTO-less programming techniques and concluded that a restricted GOTO rule makes programs easier to write, modify and maintain.

What is needed is a construct known as a "LEAVE" statement in other languages. This contruct permits you to terminate a loop early, and to transfer to the statement following the end of the loop.

In terms of CPL, this means that the GOTO statement may be used only in the following circumstances:

1. A GOTO statement may specify only a forward transfer; you may never move backwards in your program except by means of a DO loop.

2. A GOTO statement may transfer to the statement following the END statement of a DO loop in which the GOTO statement lies.

Here is an illustration of this type of GOTO statement:

```
          DO I = 1 TO 100;
          ....
          IF A>B THEN GO TO E3;
          ....
          END;
E3:       ....
```

In this example, the test "A>B" is used to determine whether the loop should be terminated abnormally, by transferring to the first statement following the END statement.

22.5 USE OF GOTO WITH ON-UNITS (C)

Since most ON-units must terminate with a GOTO statement, it is necessary to use a GOTO statement with error control logic.

This is not usually a serious problem, since there are usually few ON-units in a program.

Nonetheless, the programmer should follow the rule that an ON-unit may only cause a transfer in the forward direction.

22.6 MODULARITY (C)

An important facet of well-structured programs is modularity.

Suppose you are writing a program of several hundred statements. Such a program will usually be very difficult to understand because, for example, there will be DO loops where the END statement is far from the DO statement. Understanding the loop strucutre of a program is essential to understanding the program.

For this reason, you should take large blocks of code and place them into separate PROCEDUREs. You should strive to make the main program, and each of the PROCEDUREs, no more than 100 statements long.

For more information on the use of PROCEDUREs, please see the chanpter on "Subroutine and Function PROCEDUREs."

CHAPTER 23

BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES (B-D,R)

23.1 WHAT ARE BUILT-IN FUNCTIONS? (B)

Certain identifiers have special meaning when used in expressions. These identifiers can be used to specify that certain special functions are to be invoked. When used in this way, these identifiers are called "built-in functions."

For example, in the statement

```
B = LOG(X);
```

the reference to "LOG(X)" causes CPL to compute the natural logarithm of the current value of X and assign its value to B.

Some functions take more than one argument. For example, the reference to SUBSTR('ABCDEF',3,2) returns the character string 'CD'. (This is the substring of 'ABCDEF' starting from the third character and continuing for 2 characters.)

23.2 HOW BUILT-IN FUNCTIONS ARE RECOGNIZED (B)

Consider the identifier LOG. If there is no declaration for LOG, then it will normally be an ordinary FIXED variable. However, in the absence of any array declaration, a reference to LOG(X) is illegal due to the appearance of the argument, X. When CPL detects such an illegal reference, it checks to see whether LOG is in its list of special built-in functions. If so, then the illegal array reference is interpreted as a legal built-in function reference.

Once LOG has been used as a built-in function in this manner, it may never again be used as a variable without a declaration.

If you wish, for purposes of program clarity, to specify that LOG is a built-in function, then you may insert the declaration:

```
DECLARE LOG BUILTIN;
```

23.3 BUILT-IN FUNCTIONS WITH NO ARGUMENTS (C)

The DATE built-in function takes no arguments, and it returns a character string of length 6 (CHAR(6)) containing the date in the format yymmdd, where yy=the last two digits of the year, mm=the number of the month, and dd=the day of the month.

Functions such as DATE which take no arguments present special recognition problems. In the statement "A=DATE;", CPL has no way of determining whether the programmer intended that the variable or the built-in function DATE be used.

CPL makes the following convention: A simple reference to "DATE" will be interpreted as a reference to the variable "DATE." To refer to the built-in function DATE, you must do one of two things:

1. Refer to "DATE()". The parentheses with no argument list signal a reference to the built-in function DATE.

2. Use an explicit declaration,

   ```
   DECLARE DATE BUILTIN;
   ```

   If this declaration appears, then any reference to DATE will be interpreted as a reference to the built-in function.

23.4 WHAT IS A PSEUDO-VARIABLE? (C)

Certain built-in functions may appear to the left of an equal sign in an assignment statement. When they appear in such contexts, these built-in functions are called "pseudo-variables".

Here is an example of the use of SUBSTR as an pseudo-variable:

```
DECLARE C CHAR(8);
C='ABCDEFGH';
SUBSTR(C,3,2) = 'XY';
```

After the last statement has executed, the value of C will be 'ABXYEFGH'.

23.5 USE OF BASED ARGUMENTS (D)

Several of the built-in functions and pseudo-variables in the list below require the first argument to be an identifier or a subscripted identifier.

In such cases, if the argument has the BASED attribute, then it may be qualified by a POINTER variable in the way described in the chapter entitled "BASED Storage and POINTERs."

23.6 ALPHABETICAL LIST OF BUILT-IN FUNCTIONS AND PSEUDO-VBLES (R)

23.6.1 ABS Built-in Function

ARGUMENT: X, either FIXED or FLOAT. If not arithmetic, x will be converted to FIXED.

RESULT: CPL returns the absolute value of X. If X is FIXED, then ABS(X) is FIXED; if x is FLOAT, then ABS(X) is FLOAT.

23.6.2 ACOS Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so may not be available in other PL/I implementations.

Arguments: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

Result: The result, V, will be the arccos of X in radians, such that 0<=V<pi.

23.6.3 ADDR Built-in Function

ARGUMENT: A, where A is any subscripted or unsubscripted variable of any data type, either scalar or array.

RESULT: The result is the "address" of the variable. This address can be used to qualify a BASED identifier by means of the "->" operator, or it can be assigned to a POINTER variable for later use.

For more information, please see the chapter entitled "BASED storage and POINTERs."

23.6.4 AFTER Built-in Function

ARGUMENTS: A,B, where both arguments are either both CHARACTER or BIT strings. If both arguments are BIT strings, then no conversion is done; otherwise, both are converted to CHARACTER strings if necessary.

RESULT: AFTER returns a CHARACTER string if the arguments are CHARACTER, and a BIT string if the arguments are BIT. The string returned is determined by the following rules:

1. If A is a null string, then the result is a null string.
2. If B is a null string, then the result is the string A.
3. If B is not a null string, and B is not a substring of the string A, then the result is the null string.
4. If B is not a null string, and B is a substring of the string A, then the result is the string of all characters or bits in A which follow the first occurrence of B in A.

EXAMPLE: A reference to AFTER('ABCDABCD','CD') yields the result 'ABCD'.

23.6.5 ALLOCATION Built-in Function

Abbrev: ALLOCN for ALLOCATION

ARGUMENTS: A, where A is an identifier having the CONTROLLED storage class attribute.

RESULT: The result is an integer. If A is not allocated, then the result is 0. If A is allocated, then the result is the current number of allocations.

23.6.6 ASIN Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so may not be available in other PL/I implementations.

Argument: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

Result: The result, V, will be the arcsine of X in radians, such that -pi/2 <=V<=+pi/2.

23.6.7 ATAN Built-in Function

ARGUMENTS: Y[,X] -- the second argument is optional. Both arguments must be FLOAT. A non-FLOAT argument will be converted to FLOAT.

RESULT: The result V will be a FLOAT value computed according to the following rules:

1. If the second argument (X) is not specified, then V will be the arctangent of Y, specified in radians, such that -pi/2<V<pi/2.

2. If both arguments are specified, then V will be the arctangent of (Y/X), specified in radians whose angle lies in the quadrant of the coordinates (X,Y). The mathematical rules are as follows:

    1. If X>=0, then 0<=V<=pi.

    2. If X<0 then -pi<V<0.

23.6.8 ATAND Built-in Function

This function is the same as the ATAN built-in function, except that the value returned is in radians. The value of ATAND is computed by multiplying the value of ATAN by 180/pi.

23.6.9 BEFORE Built-in Function

ARGUMENTS: A,B, where the arguments are either both CHARACTER strings or both BIT strings. If both arguments are BIT strings, then no conversion is done; otherwise, both are converted to CHARACTER strings if necessary.

Result: BEFORE returns a CHARACTER string if the arguments are CHARACTER, or a BIT string if the arguments are BIT. The string returned is determined as follows:

1. If A is a null string, then the result is a null-string.
2. If B is a null string, then the result is a null string.
3. If B is not a null string and it is not a substring of A, then the result is the string A.
4. If B is not a null string and B is a substring of A, then the result is the string of all bits or characters which precede the first occurrence of the substring B in A.

EXAMPLE: A reference to BEFORE('ABCDABCD','CD') yields the result 'AB'.

23.6.10 CEIL Built-in Function

ARGUMENTS: X, either FIXED or FLOAT. If X is not arithmetic, it will be converted to FIXED.

RESULT: If X is FLOAT, the result is FLOAT; otherwise, the result is FIXED. In either case, the result equals the smallest integer that is greater than or equal to the value of X.

23.6.11 COLLATE Built-in Function

ARGUMENTS: none

RESULT: COLLATE returns a CHAR(128) string containing all characters in the ASCII collating sequence in increasing order. The first character in the result has octal value 0 (the null character), and the last character has octal value 177.

23.6.12 COPY Built-in Function

ARGUMENTS: S,N, where S is a CHARACTER or BIT string and N is a non-negative integer. If S is not a BIT string, then S will be converted to a CHARACTER string, if necessary. N will be converted to FIXED, if necessary.

RESULT: If the string S is a BIT string, then the result is a BIT string; otherwise the result is a CHARACTER string.

If N=0, then the result is the null BIT or CHARACTER string.

If N>0, then the result is a string of length N*LENGTH(S) containing N copies of the string S.

EXAMPLE: COPY('ABCD',3) returns 'ABCDABCDABCD'.

23.6.13 COS Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: X is an angle assumed to be given in radians. The result is the cosine of X.

23.6.14 COSD Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: X is an angle assumed to be given in degrees. The result is the cosine of X.

23.6.15 COSH Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so may not be available in other PL/I implementations.

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: The hyperbolic cosine of X is returned as a FLOAT result.

23.6.16 DATE Built-in Function

ARGUMENTS: none

RESULT: The result is a CHARACTER string of length 6, in the format 'yymmdd', where yy represents the year, in the range 00 to 99, mm represents the month, in the range 01 to 12, and dd represents the day, in the range 01 to 31.

23.6.17 DIMENSION Built-in Function

Abbrev: DIM for DIMENSION

ARGUMENTS: X,N. The first argument must be a single unsubscripted identifier, and the identifier must be DECLAREd to be an array. N will be converted to FIXED, if necessary.

RESULT: N must be a positive integer less than the number of dimensions in the array X. Let lb and ub be the lower bound and upper bound, respectively, of the N'th dimension in X. Then the result is the FIXED value (ub-lb+1).

EXAMPLE: If we have DECLARE A(10,2:6), then DIMENSION(A,1) returns 10, and DIMENSION(A,2) returns 5.

23.6.18 DIVI Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

ARGUMENTS: X,Y, FIXED. If either argument is not FIXED, it will be converted to FIXED.

RESULT: The truncated value of X/Y is returned as a FIXED value.

NOTE: This function and the DIVF function are intended to make it easier for you to do without integer division. DIVI takes two FIXED arguments, and returns the truncated quotient. If you move your program to other implementations of PL/I, occurrences of DIVI(X,Y) should be replaced by occurrences of TRUNC(X/Y).

23.6.19 DIVF Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

ARGUMENTS: X,Y, FLOAT. If either argument is not FLOAT, it will be converted to FLOAT.

RESULT: The FLOAT value of X/Y is returned.

NOTE: This function and the DIVI function are inteded to make it easier for you to do without integer division. DIVF takes two arguments (usually FIXED), and it converts them the FLOAT and returns the FLOAT quotient. If you move your program to another PL/I implementation, occurrences of DIVF(X,Y) should be replaced by occurrences of FLOAT(X/Y).

23.6.20 EVERY Built-in Function

ARGUMENTS: X, a BIT string. If X is not a BIT string, it will be converted to BIT.

RESULT: EVERY returns a BIT(1) value. If X is the null BIT string or if every bit of X is '1'B, then EVERY returns '1'B; otherwise, if X contains at least one '0'B bit, then EVERY returns '0'B.

EXAMPLE: EVERY('111011'B) returns '0'B, while EVERY('111'B) returns '1'B.

23.6.21 EXP Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it is converted to FLOAT.

RESULT: This function returns the FLOAT value e**X, where e is base of the natural logarithm system.

23.6.22 FLOOR Built-in Function

ARGUMENTS: X, either FIXED or FLOAT. If X is not FLOAT, then it is converted to FIXED, if necessary.

RESULT: If X is FIXED, then the result is X. If X is FLOAT, then the result is the FLOAT value of the greatest integer less than or equal to the value of X.

23.6.23 FLTED Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

ARGUMENTS: V, F, W [,D [,S]] -- between 3 and 5 arguments are specified. V must be FLOAT, or will be converted to FLOAT if necessary. All other arguments are FIXED, and will be converted to FIXED if necessary.

RESULT: FLTED returns a variable length CHARACTER string. It formats the value V in the same way that the PUT LIST or PUT EDIT statements would do so.

If F=0, then V is returned in the format which would be given by a PUT LIST statement to a file with the VFORM attribute. (This non-standard file attribute specifies that the output is to have a variable format which depends on the value of the number being printed.)

If F=1, then V is returned in the format specified by the PUT EDIT format item

```
          E(w [,d [,s]])
```

If F=2, then V is returned in the format specified by the PUT EDIT format item

```
          F(w [,d [,s]])
```

23.6.24 HBOUND Built-in Function

ARGUMENTS: X,N. The first argument must be a single unsubscripted identifier, and the identifier must be DECLAREd to be an array. N will be converted to FIXED, if necessary.

RESULT: N must be a positive integer less than the number of dimensions in the array X. The result is the FIXED value equal to the upper bound of the N'th dimension of X.

EXAMPLE: If we have DECLARE A(10,2:6), then HBOUND(A,1) returns 10 and HBOUND(A,2) returns 6.

23.6.25 HIGH Built-in Function

ARGUMENT: N, FIXED. If N is not FIXED, then it will be converted to FIXED.

RESULT: N must be a non-negative integer. The result is a CHARACTER string of length N. If N=0, then the result is the null string. If N>0, then the result is a CHAR(N) string containing N occurrences of the highest character in the ASCII collating sequence, octal value 177.

23.6.26 INDEX Built-in Function

ARGUMENTS: S,C, either both CHARACTER strings or both BIT strings. If either of S or C is not a BIT string, then both arguments are converted to CHARACTER.

RESULT: INDEX returns a FIXED value. If C is not a null string, and if C is a substring of S, then INDEX returns the position in S of the leftmost occurrence of C. If C is a null string or it is not a substring of S, then INDEX returns 0.

EXAMPLE: INDEX('ABCDABCD','CD') returns 3, while INDEX('ABCDABCD','XY') returns 0.

23.6.27 LBOUND Built-in Function

ARGUMENTS: X,N. The first argument must be a single unsubscripted identifier, and the identifier must be DECLAREd to be an array. N will be converted to FIXED, if necessary.

RESULT: N must be a positive integer less than the number of dimensions in the array X. The result is the FIXED value equal to the lower bound of the N'th dimension of X.

EXAMPLE: If we have DECLARE A(10,2:6), then LBOUND(A,1) returns 1, and LBOUND(A,2) returns 2.

23.6.28 LENGTH Built-in Function

ARGUMENTS: S, which must be CHAR or BIT. If it is not BIT, then it is converted to CHAR, if necessary.

RESULT: LENGTH returns a FIXED value equal to the length of the string S.

EXAMPLE: LENGTH('ABC') returns 3, LENGTH('1101'B) returns 4, and LENGTH('267'B3) returns 9. LENGTH('') returns 0.

23.6.29 LOG Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it is converted to FLOAT.

RESULT: LOG returns the natural logarithm of X, as a FLOAT result. X must be positive.

23.6.30 LOG10 Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it is converted to FLOAT.

RESULT: LOG10 returns the common logarithm of X, as a FLOAT result. X must be positive.

23.6.31 LOG2 Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it is converted to FLOAT.

RESULT: X must be positive. LOG2 returns the base 2 logarithm of X.

23.6.32 LOW Built-in Function

ARGUMENT: N, FIXED. If N is not FIXED, then it will be converted to FIXED.

RESULT: N must be a non-negative integer. The result is a CHARACTER string of length N. If N=0, then the result is is the null string. If N>0, then the result is a CHAR(N) string containing N occurrences of the lowest character in the ASCII collating sequence, value 0 (this is the "null" character).

23.6.33 MAX Built-in Function

ARGUMENTS: X1,X2,...XN. This function takes a varying number of arguments, either all FIXED or all FLOAT. If at least one argument is FLOAT, then all will be converted to FLOAT, if necessary; otherwise, all are converted to FIXED, if necessary.

RESULT: If the arguments are FLOAT, then a FLOAT result is returned; otherwise a FIXED result is returned. In any case, the result is the numerical maximum of the arguments.

EXAMPLE: MAX(2,3,200,-5,4) returns 200.

23.6.34 MIN Built-in Function

ARGUMENTS: X1,X2,...,XN. This function takes a varying number of arguments, either all FIXED or all FLOAT. If at least one argument is FLOAT, then all will be converted to FLOAT, if necessary; otherwise all are converted to FIXED, if necessary.

RESULT: If the arguments are FLOAT, then a FLOAT result is returned; otherwise, a FIXED result is returned. In any case, the result is the numerical minimum of the arguments.

23.6.35 MOD Built-in Function

ARGUMENTS: X,Y, either both FIXED or both FLOAT. If one argument is FLOAT, then the other is converted to FLOAT, if necessary; otherwise, both are converted to FIXED, if necessary.

RESULT: If the arguments are FLOAT, then the result is FLOAT; otherwise, the result is FIXED. The result is the remainder obtained by dividing X by Y. The actual result is computed in the following two cases:

1. If Y=0, then the result is X.

2. If Y^=0, then the result is given by:
```
                    Y-X*FLOOR(Y/X)
```

23.6.36 NULL Built-in Function

ARGUMENT: None

RESULT: A reference to NULL() returns a "null" POINTER value.

For more information on the NULL built-in function, please refer to the chapter entitled "BASED storage and POINTERs."

23.6.37 ONMSG Built-in Function

WARNING: ONMSG is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

ARGUMENTS: None

RESULT: You use ONMSG in an ON-unit when, for example, you wish to type out a message indicating why the ON-unit was taken. ONMSG returns a character string value containing the error message which would have been typed if the ON-unit had not been taken.

For further information, please see the chapter entitled "Error Handling and ON-conditions."

23.6.38 RANDOM Built-in Function

WARNING: RANDOM is not in the ANSI PL/I standard, and so will not be available in other PL/I implementations.

ARGUMENTS: none

RESULT: RANDOM returns a FLOAT value, V, in the range 0<V<1. Each call to RANDOM returns a new random number, such that multiple calls to RANDOM produce a set of numbers uniformly distributed over the open interval (0,1).

EXAMPLE: To write a CPL program which plays a card game, you may wish to generate a random hand. To return a random integer between 1 and 52, use TRUNC(1+52*RANDOM()).

NOTE: Of course, RANDOM does not return random numbers in the true mathematical sense. Each random number is computed from the preceding one by an algorithm which garbles the bits. The result is that for a few million calls to RANDOM, the sequence will seem to be perfectly random. However, there is a cycle length of the random number sequence, probably on the order of 10**6 numbers.

23.6.39 RANDOM Pseudo-variable

ARGUMENTS: none

RESULT: The RANDOM pseudo-variable is used to initialize the RANDOM random number generator. As stated above, each new random number is generated by an algorithm from the old one. The RANDOM pseudo-variable can be used to initialize the random number generator.

You initialize the random number generator by executing the statement

```
RANDOM()=value;
```

where "value" is a FLOAT value or expression. (If it is not FLOAT, it will be converted to FLOAT.)

If "value" is non-zero, then that value will be used to initialize the random number generator. For example, if you execute RANDOM()=2, then subsequent calls to the RANDOM built-in function will generate a sequence of random numbers based on 2 as the initializing value. If you then rerun the same program, specifying RANDOM()=2, then you will get the same sequence of random numbers. This is a useful feature if you wish to debug a program using the same sequence of numbers. Executing RANDOM()=3 will cause a different sequence of random numbers to be generated.

If you execute RANDOM()=0, then CPL will use the time of day to create a unique generator. After you have debugged a program, you should begin your program with this statement so that each invocation of the program will generate a different sequence of random numbers.

23.6.40 REVERSE Built-in Function

ARGUMENT: S, either a BIT string or a CHAR string. If S is neither BIT nor CHAR, then it will be converted to CHAR.

RESULT: The result is a CHARACTER or BIT string of the same length as the CHARACTER or BIT string argument, S. The string contains all the characters or bits of S, but in reverse order.

EXAMPLE: REVERSE('ABCD') returns 'DCBA'.

23.6.41 SIGN Built-in Function

ARGUMENTS: X, where X is either FIXED or FLOAT. If it is not FIXED or FLOAT, X will be converted to FIXED.

RESULT: SIGN returns a FIXED value +1, 0, or -1, depending upon whether the argument is greater than 0, equal to 0, or less than 0, respectively.

EXAMPLE: SIGN(-5) returns -1, while SIGN(23.4E16) returns +1.

23.6.42 SIN Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: X is an angle assumed to be given in radians. The result is the FLOAT value of the sine of X.

23.6.43 SIND Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: X is an angle assumed to be given in degrees. The result is the FLOAT value of the sine of X.

23.6.44 SINH Built-in Function

WARNING: This function is not in the ANSI PL/I standard, and so may not be available in other PL/I implementations.

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: The hyperbolic sine of X is returned as a FLOAT result.

23.6.45 SOME Built-in Function

ARGUMENTS: X, a BIT string. If X is not a BIT string, then it will be converted to BIT.

RESULT: SOME returns a BIT(1) value. If X contains at least one bit with value '1'B, then the value of SOME(X) is '1'B; otherwise, the value is '0'B.

EXAMPLE: SOME(''B) returns '0'B. SOME('000000'B) returns '0'B. SOME('000000100'B) returns '1'B.

23.6.46 SQRT Built-in Function

ARGUMENTS: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: X must have non-negative value. CPL will return the positive square root of X.

23.6.47 STRING Built-in Function

ARGUMENTS: A, where A is an array whose elements are either CHAR NONVARYING or BIT NONVARYING.

RESULT: All the elements of array A are concatentated into one long BIT or CHARACTER string, and the result is returned.

EXAMPLE:

```
        DECLARE A(3) CHAR(5);
        A(1)='ABCDE'; A(2)='12345'; A(3)='VWXYZ';
```

After the above statements have been executed, an invocation of STRING(A) will return the CHAR(15) string, 'ABCDE12345VWXYZ'.

23.6.48 STRING Pseudo-variable

ARGUMENT: A, where A is an array whose elements are either CHARACTER NONVARYING or BIT NONVARYING.

RESULT: If you assign a value to STRING(A), then CPL treats the elements of the array A as one long CHARACTER or BIT string, and makes the assignment to that long string.

EXAMPLE:

```
        DECLARE A(3) CHAR(1);
        STRING(A) = 'ABC';
```

After the last statement is executed, A(1) will equal 'A', A(2) will equal 'B', and A(3) will equal 'C'.

23.6.49 SUBSTR Built-in Function

ARGUMENTS: S,I[,J] -- the third argument is optional. The first argument must be either CHARACTER or BIT; if it is not a BIT string, then it is converted to CHARACTER, if necessary. The arguments I and J will be converted to FIXED if they are not already FIXED.

RESULT: SUBSTR returns the substring of string S beginning at the bit or character in position I, and continuing for J bits or characters (if J is specified), or to the end of string S (if J is omitted).

EXAMPLE: SUBSTR('ABCDEF',2,3) returns 'BCD', and SUBSTR('ABCDEF',4) returns 'DEF'.

23.6.50 SUBSTR Pseudo-variable

ARGUMENTS: S,I[,J] -- the third argument is optional. The first argument, S, must be a scalar identifier or array element with either BIT or CHARACTER type, either VARYING or NONVARYING. The arguments I and J must be FIXED; they will be converted to FIXED if they are not.

RESULT: If you assign a value to SUBSTR(S,I,J), then CPL wil assign the value to the substring of S starting with the I'th character and continuing for J characters. If you assign a value to SUBSTR(S,I), then CPL will assign the value to the substring of S starting with the I'th character and contuing to the end of the string.

NOTE: If S is a VARYING string, the assignment to the SUBSTR pseudo-variable will not change the length of S.

EXAMPLE:

```
          DECLARE A CHAR(6);
          A='ABCDEF';
          SUBSTR(A,2,4)='1234';
```

The last statement causes A to have the value 'A1234F'. If the last statement had been SUBSTR(A,4)='1', then A would equal 'ABC1 '.

23.6.51 TANH Built-in Function

ARGUMENT: X, FLOAT. If X is not FLOAT, then it will be converted to FLOAT.

RESULT: The FLOAT value of the hyperbolic tangent of X is returned.

23.6.52 TIME Built-in Function

ARGUMENTS: none

RESULT: TIME returns a CHARACTER string of length 9 which contains the time of day in the format 'hhmmssddd', where hh is hours in the range 00 to 23, mm is minutes in the range 00 to 59, ss is seconds in the range 00 to 59, and ddd is milliseconds in the range 000 to 999.

23.6.53 TRANSLATE Built-in Function

ARGUMENTS: S,R[,P] -- the third argument is optional. All arguments will be converted to CHARACTER, if they are not already CHARACTER.

RESULT: The result is a CHARACTER string the same length as the argument S. CPL obtains the result from the string S by translating some of the characters, as specified by the other argument(s), according to the following rules:

1. If you have not specified the argument P, then CPL supplies an argument P equal to the value returned by the COLLATE built-in function.

2. The argument R must be the same length as the argument P. If it is not, then CPL will either truncate it (by removing characters from the end) or pad it (by adding blanks to the end) so that it is the same length as P.

3. For each character in the string S, if any, CPL performs the following operations:

    1. CPL finds the leftmost occurrence, if any, of this character in the string P.

    2. If this character does not occur in string P, then CPL inserts this character into the result string.

    3. If this character occurs in the string P, and the leftmost occurrence is at position i, then CPL finds the character in the i'th position of string R and inserts that character into the result string.

EXAMPLE: TRANSLATE('ABCDEFG','123','DGCBCC') returns 'A 31EF2'.

23.6.54 TRUNC Built-in Function

ARGUMENT: X, either FIXED or FLOAT. If X is not FLOAT, then it is converted to FIXED, if necessary.

RESULT: If X is FIXED, then the result is X. If X is FLOAT, then the result is the FLOAT value of the integer obtained by "truncating" the value of X -- by removing the fractional part.

If X is positive, then TRUNC(X) = FLOOR(X). If X is negative, then TRUNC(X) = CEIL(X).

EXAMPLE: TRUNC(2.8) returns 2, while TRUNC(-2.8) returns -2.

23.6.55 UNSPEC Built-in Function

ARGUMENT: X, where X is any unsubscripted identifier of any computational data type, either scalar or array.

RESULT: The result is a BIT string representing the entire data area occupied by the variable X. Thus, UNSPEC allows you to examine the BIT representation of CPL data. All fullwords in the data area are included in the BIT representation, so that the length of the BIT string returned by UNSPEC will always be an exact multiple of 36.

Note that the entire data area is included. This includes bit 35 of character string words, as well as bits at the end of the data area which are used as filler.

It is legal to take UNSPEC of a POINTER variable, but the result will always be a BIT(36) string of zeroes.

EXAMPLE: If I=5; has been executed, then UNSPEC(I) will return the BIT string '000000000000000000000000000000000101'B.

23.6.56 UNSPEC Pseudo-variable

ARGUMENT: X, same as for UNSPEC built-in function.

RESULT: If you assign a BIT string to UNSPEC(X), then the data area will be set to that BIT string, regardless of whether the bit string is meaningful for that data type. The data area includes all 36-bit words, as in the case of the UNSPEC built-in function.

It is legal to apply the UNSPEC pseudo-variable to a POINTER variable, but doing so will have no effect upon the value of the POINTER.

EXAMPLE: If I is FIXED, and UNSPEC(I)=(33)'1'B; is executed, then I will assume the value -8.

23.6.57 VERIFY Built-in Function

ARGUMENTS: S,C. Both arguments must be CHARACTER strings. They will be converted to CHARACTER strings, if necessary.

RESULT: The result is a FIXED value, computed as follows:

1. If S is the null CHARACTER string, then the result is 0.
2. If each of the characters of the string S occurs in the string C, then the result is 0.
3. Otherwise, suppose the character in the i'th position of the string S is the first character in S which does not also appear in the string C. Then return the value i.

EXAMPLE: VERIFY can be used to verify that all characters in a given string are valid for a particular context. The second argument to VERIFY should contain all the "legal" characters, so that the position of the first "illegal" character will be returned.

For example, if you wish to find the first non-numeric character in '729B788F9', then VERIFY('729B788F9','0123456789') will return the position, 4, of the character 'B'.

CHAPTER 24

CONVERSIONS AMONG COMPUTATIONAL DATA TYPES (D)

Except for the conversions from FIXED to FLOAT and FLOAT to FIXED, the PL/I conversion rules are quite complicated. For this reason, you should use conversions to and from CHARACTER and BIT data types with great care.

This chapter gives all the rules for conversions.

24.1 CONVERSIONS FROM FIXED (D)

24.1.1 FIXED To FLOAT (D)

This conversion presents no problem. No errors can occur.

EXAMPLE: 2 is converted to 2.0E0.

24.1.2 FIXED To CHARACTER (D)

The result is CHAR(13). The integer value is represented in character format as a decimal integer, and the value is right adjusted in the 13 character field. All leading zeros are suppressed. If the integer is negative, then the number is preceded by a minus sign. All other characters are blank.

No conversion errors can occur.

EXAMPLE: The integer 23 is converted to 'bbbbbbbbbbb23', where "b" is a blank character. The integer -345 is converted to 'bbbbbbbbb-345'.

24.1.3 FIXED To BIT (D)

The result is BIT(35). If the source integer is negative, then the absolute value is taken. The integer is then represented in binary notation, with 35 binary digits. The BIT(35) result consists of these 35 binary digits.

An error occurs if the source is the minimum negative number.

EXAMPLE: The value 18 is converted to '00000000000000000000000000000010010'B.

24.2 CONVERSIONS FROM FLOAT (D)

24.2.1 FLOAT To FIXED (D)

The FLOAT value is converted to FIXED by "truncating" any fractional part. For positive numbers, the result is the greatest integer not greater than the FLOAT source value, and for negative numbers the result is the smallest integer not less than the FLOAT source value.

An error occurs if the absolute value of the FLOAT value is greater than 2 to the power 35.

EXAMPLE: 2.35 is converted to 2, while -2.35 is converted to -2.

24.2.2 FLOAT To CHARACTER (D)

The result is a character string of length 14. The format is:

```
sd.dddddddEsdd
```

where the symbols have the following meanings:

1. The first "s" indicates the sign of the number. It will be "-" for a negative number, and blank for a positive number.
2. The first "d" is the first significant digit of the number. It is always non-zero, unless the number is zero.
3. The next 7 "d" characters are the next seven significant digits of the number.
4. The next "s" indicates the sign of the exponent; it will be "-" if the exponent is negative, and "+" otherwise.
5. The last two "d" characters are the two digits of the exponent. The exponent is the power of ten which must be multiplied by the mantissa shown to get the actual number.

No error can occur in this conversion.

EXAMPLE: 23.3 is converted to ' 2.3300000E+01'. -0.234 is converted to '-2.3400000E-01'.

24.2.3 FLOAT To BIT (D)

The result is BIT(35). The conversion is made by converting the FLOAT value to FIXED, and then the FIXED value to BIT, according to the rules already given.

An error will occur if the absolute value of the FLOAT value is greater than 2 to the power 35.

EXAMPLE: -9.3 is converted to '00000000000000000000000000000001001'B.

24.3 CONVERSIONS FROM CHARACTER (D)

24.3.1 CHARACTER To FIXED Or FLOAT (D)

The source character string must consist of an integer or floating point constant, optionally preceded by and followed by one or more blanks.

The constant is evaluated and converted to FIXED or FLOAT, as the target requires.

The source character string may be the null string or may contain only blanks. In this case, the target FIXED or FLOAT value is set to zero.

An error occurs if the source character string contains a character which makes it an illegal FIXED or FLOAT constant.

EXAMPLE of CHAR to FIXED conversion: '   -23   ' is converted to -23. '   23.4E3   ' is converted to 23400.

EXAMPLE of CHAR to FLOAT conversion: '   -23   ' is converted to -2.3E1, and '   23.4E3   ' is converted to 2.34E4.

24.3.2 CHARACTER To BIT (D)

The only characters which may appear in the source character string are 0's and 1's. The target BIT string has the same length as the source CHAR string. Each bit in the target BIT string is obtained from the corresponding character in the source CHAR string by changing a 0 character to a 0-bit, and a 1 character to a 1-bit.

EXAMPLE: '00101' is converted to '00101'B.

24.4 CONVERSIONS FROM BIT (D)

24.4.1 BIT To FIXED (D)

The bits in the source BIT string are interpreted as binary digits. The string of binary digits is then interpreted as an integer. The resulting integer is always non-negative.

An error occurs if the resulting integer is greater than 2 to the power 35.

EXAMPLE: '1100011'B is converted to 99.

24.4.2 BIT To FLOAT (D)

The source BIT string is first converted to FIXED, and the FIXED result is converted to FLOAT, according to the rules already described.

An error occurs if there is an error in the BIT to FIXED conversion.

24.4.3 BIT To CHARACTER (D)

The target character string has the same length as the source BIT string. Each character of the target string is obtained from the corresponding bit in the BIT string by converting a 0-bit to a 0 character, and a 1-bit to a 1 character.

No error can occur.

EXAMPLE: '1100101'B is converted to '1100101'.

CHAPTER 25

CPL ERROR MESSAGES (R)

This chapter contains an alphabetical list of CPL error messages, together with explanatory material.

The messages are listed in the following order:

1. Those messages beginning with the character " are listed first.

2. Those beginning with the character # are second.

3. Those beginning with the left parenthesis are next.

4. Those beginning with digits are next.

5. Those beginning with letters are last.

"keyword" AND "keyword" ARE CONFLICTING KEYWORDS

Explanation: The two specified keywords may not be specified as options in the same statement or as attributes for the same identifier in a DECLARE statement.

"keyword" AND "keyword" CONFLICT IN "OPEN" ATTRIBUTE MERGE

Explanation: When you open a file, either with an explicit OPEN statement or implicitly with a GET, PUT, READ or WRITE statement, CPL must merge file attributes from the open operation with those in the FILE declaration. Two of these attributes have been found to conflict.

User Response: You must change the attributes specified in either the DECLARE statement or the OPEN statement.

This message can also appear as the result of an implicit open due to a GET, PUT, READ, or WRITE. These statements open the file with certain implied attributes. These implied attributes are:

Statement	Implied attributes
GET	STREAM and INPUT
PUT	STREAM and OUTPUT
READ	RECORD and INPUT
WRITE	RECORD and OUTPUT

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

"keyword" IS ILLEGAL FOR DEVICE "device"

Explanation: (TOPS-10 only) The specified device cannot be used for the specified type of operation. For example, the device LPT: (the line printer) is illegal for input, while the device CDR: (the card reader) is illegal for output.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

"ident" IS NOT ALLOCATED

Explanation: Either storage has not been allocated for the specified variable, or storage was once allocated but has been freed.

There are several circumstances which can give rise to this or a similar message:

1. You are referencing a CONTROLLED variable, but you have never executed an ALLOCATE statement for the variable.

2. You are referencing a CONTROLLED variable for which you have executed an ALLOCATE statement, but you have also executed a FREE statement releasing the storage.

3. You have initiated a DO-group with a DO-loop variable which occupies CONTROLLED or BASED storage, but during execution of the DO-group the storage occupied by the DO-loop variable has been freed. This error will be detected when the END is reached and CPL attempts to iterate the DO-group.

4. You have invoked a function or subroutine, passing a real (non-dummy) argument occupying CONTROLLED or BASED storage, you have freed the storage, and you are attempting to access the corresponding parameter. CPL establishes the rule that if the storage occupied by an argument is freed, then the corresponding parameter is no longer available for access.

5. You have specified the SUBSTR pseudo-variable in such a way that the first argument occupies CONTROLLED or BASED storage, and this storage was released while CPL was evaluating the second or third argument. (This could happen if the second or third argument involves a call to a function PROCEDURE.)

Note also the following: If you type an XEQ statement rather than a CONTINUE statement, they all BASED and CONTROLLED storage is freed.

User Response: You should make sure that your CONTROLLED and BASED storage is allocated before it is needed, and is not freed until your program is finished referencing it.

If the logic of your program demands that storage be freed even though you still need the value of the variable, then you should copy that variable to an AUTOMATIC or STATIC variable which will not be freed.

Note: If this or a similar error message occurs with reference to AUTOMATIC or STATIC storage, then the message is a CPL system error. Please save all relevant output and report the problem to Digital Equipment Corporation.

"*" LEGAL FOR ident ONLY WITH "PARAMETER"

Explanation: In a declaration, you have specified an asterisk in an extent expression (a string length or an array bound), but you have not specified "parameter." An asterisk is permitted only with "parameter."

User Response: You should retype your DECLARE statement, specifiying the PARAMETER attribute.

"BY" OR "TO" CLAUSE ILLEGAL WITH NON-ARITHMETIC DO LOOP VBLE

Explanation: You have attempted to use a BY or TO clause with a DO-loop variable which is not FIXED or FLOAT, in violation of the CPL rules. The BY and TO clauses are illegal when the DO-loop variable is CHARACTER, BIT or POINTER.

User Response: You must respecify the DO-loop. You can try one of the following:

1. Try to respecify it so that you you can use an arithmetic DO-loop variable. If you were using a TO or BY clause, then this method will often succeed.

2. Respecify it using the REPEAT and WHILE options, both of which are legal with string-type loop variables.

"COLUMN" FORMAT ITEM ILLEGAL WITH PUT STRING

Explanation: You have specified a PUT EDIT statement with the STRING option, and one of the format items is a COLUMN format item. This is illegal.

User Response: Replace the COLUMN format item with the X format item, specifying the correct number of blanks.

"CURRENT STATEMENT" IS NOT DEFINED

Explanation: You have used a CONTINUE statement with no FROM option, and there is no "current statement" from which your program can continue.

User Response: Specify the XEQ statement to start the program from the beginning, or else use the CONTINUE statement with the FROM option to specify the statement from which you wish to continue executing.

Implementation Note: CPL defines the "current statement" whenever execution stops as the result of a program error or a breakpoint. The "current statement" is then defined to be the statement in which the error or breakpoint occurred.

CPL will make the "current statement" undefined if you erase or change the statement in which the error or breakpoint occurred.

"ELSE" CLAUSE IS COLLECT-ONLY -- LINE NUMBER REQUIRED

Explanation: You have used an ELSE clause in a statement which has no line number.

"END" STATEMENT MAY NOT BE AND "ELSE" CLAUSE

Explanation: You have typed in the statement "ELSE END;"., in violation of the CPL rules.

"GET" FILE filespec OPEN FOR keyword keyword RATHER THAN INPUT STREAM

Explanation: The file specified in a GET operation must be opened with the attributes INPUT and STREAM.

User Response: There are several ways in which this error can be made:

1. If you have written an output file, you must specify the CLOSE statement for the file before you can start to get input from it. Keep in mind that an OPEN statement is a "no-operation" if the filename is already open, even with different attributes.

2. It is illegal to perform READ and GET operations on the same input file. You may open the same input file twice with different file variables and then do GETs with one file variable and READs with the other.

3. The declaration for the file identifier may contain attributes which conflict with INPUT and STREAM. In this case you should change your file declaration.

4. It is legal to be doing GETs and PUTs to the same disk file provided that you are using different file identifiers in CPL. If you are doing GETs and PUTs to the same disk file then use two different identifiers and open them separately for INPUT and OUTPUT using the TITLE option of the OPEN statement.

"NONVARYING" SPECIFIED FOR ident WITHOUT "BIT" OR "CHAR"

Explanation: In a declaration, you have specified the NONVARYING attribute for an identifier for which neither BIT nor CHAR were specified.

"PAGE" INVALID FOR NON-PRINT FILE filespec

Explanation: You have used either the PAGE option of the PUT statement or a PAGE format item in a PUT EDIT statement, but the file does not have the PRINT attribute.

User Response: Change the FILE declaration by adding the PRINT attribute.

"RETURN" IN AN ON-UNIT IS ILLEGAL

Explanation: You have attempted to execute a RETURN statement from an ON-unit.

User Response: To terminate an ON-unit, you should use a GOTO statement, specifying the statement with which you wish to continue executing.

"RETURN" NOT INSIDE A PROCEDURE

Explanation: You have attempted to execute a RETURN statement, but the statement is not inside a PROCEDURE.

User Response: If you wish to halt execution of your program, then use a STOP statement.

"RETURN" WITH EXPRESSION FOR CALL STATEMENT

Explanation: A PROCEDURE has been invoked as a subroutine, using a CALL statement, but you are attempting to specify a return expression as if it were a function invocation.

User Response: Replace the RETURN statement with one which does not specify an expression.

If you are using the same PROCEDURE for both subroutine and function invocations, then you will need a way of testing how the PROCEDURE was invoked (perhaps by examining the value of one of the parameters), and then use an IF statement to specify the correct form of the RETURN statement.

"RETURN" WITH NO EXPRESSION FOR FUNCTION CALL

Explanation: A PROCEDURE has been invoked as a function, but you are attempting to execute a RETURN statement with no expression to specify what value the function is to return.

This message will also appear if you execute the END statement of a PROCEDURE invoked as a function, since a RETURN statement is simulated in such a case.

User Response: Replace the RETURN statement with one which does specify an expression value.

If you are using the same PROCEDURE for both subroutine and function invocations, then you will need a way of testing how the PROCEDURE was invoked (perhaps by examining the value of the one of the parameters), and then use an IF statement to execute the correct form of the RETURN statement.

"SET" VARIABLE ident IS NOT A POINTER

Explanation: The identifier appearing in the SET option of the ALLOCATE statement does not have the POINTER data type.

Note: Full language implementations of the PL/I language permit an "implicit" declaration as POINTER of any identifier appearing in the SET option of an ALLOCATE statement. CPL requires an explicit declaration.

User Response: Type a DECLARE statement for the identifier, specifying the POINTER attribute.

"SET" VARIABLE ident IS NOT A SCALAR

Explanation: The identifier appearing in the SET option of the ALLOCATE statement has been DECLAREd to be an array.

User Response: Retype the SET option so that it specifies either a scalar POINTER identifier or a single element of a POINTER array.

"SKIP(0)" INVALID FOR "GET" OPERATION

Explanation: You have specified the SKIP option with a zero argument in a GET statement. SKIP(0) is legal only in a PUT operation.

"SKIP(0)" IS INVALID FOR NON-PRINT FILE filespec

Explanation: You have specified SKIP(0) either as an option of the PUT statement or as a format item in a PUT EDIT statement, but the file does not have the PRINT attribute.

User Response: Change the FILE declaration by adding the PRINT attribute.

"STRING" OPTION VBLE ident IS NOT ALLOCATED

Explanation: The variable specified in the STRING option of a PUT statement has not been allocated. For further information, refer to the message "ident" IS NOT ALLOCATED.

"THEN" CLAUSE MAY NOT ALSO BE AN "ELSE" CLAUSE

Explanation: You have entered a statement in the form IF ... THEN ELSE .... This is illegal.

"THEN" CLAUSE MAY NOT BE "END"

Explanation: You have illegally entered a statement of the form IF ... THEN END; This is illegal.

"THRU" SPEC LARGER THAN "FROM" SPEC

Explanation: In an LIST or ERASE statement with a THRU clause, the second line number is smaller than the first.

"XEQ" OR "CONTINUE" TO INACTIVE BLOCK

Explanation: You have typed an XEQ or CONTINUE statement with the FROM option which attempts to begin execution from a statement inside a block which is not currently active.

User Response: This error may have occurred because you wish to test out a PROCEDURE in your program by starting to execute from a point inside. You may accomplish the same thing by invoking the PROCEDURE with a CALL statement after breakpointing the first statement in the PROCEDURE. Then when you reach the breakpoint you can continue executing from any point within the PROCEDURE.

If you are attempting to continue executing, but you are unsure which of your PROCEDUREs is active, then type the SNAP statement for a snap dump.

# OF DIMENSIONS FOR ARG ident DIFFERENT THAN FOR PARM ident

Explanation: You have invoked a PROCEDURE and you have specified an argument which is an array. This message will occur if the parameter is also an array, but the number of dimensions is different.

User Response: Usually this error simply indicates that you have made a typing error when you entered the call to the PROCEDURE.

However, sometimes you may wish to reference a storage area differently in a PROCEDURE than you do in your main program. (For example, you may wish to reference the storage occupied as a two-dimensional array by means of a one-dimensional array.) You may do this in CPL by means of BASED storage and a POINTER. You let a POINTER variable equal ADDR of the two-dimensional array, and you use it to qualify a BASED one-dimensional array. For more information on this technique, please read the chapter on BASED storage and POINTERs.

(RECORD) FILE filespec RECORD IS LONGER THAN ASCII INTO VARIABLE

Explanation: You have specified a READ statement, and the next input record in the file is longer than the INTO character string.

System Action: When CPL discovers this error, it reads the record anyway, and places characters into the INTO variable storage up to the length of the string. Only then does CPL begin entering the error handling logic which may result in stopping execution. For this reason, the part of the record up to the length of the INTO variable is available for examination.

User Response: If you wish execution to continue when this error occurs, then you may use the RECORD ON-condition to specify what action CPL should take.

(RECORD) FILE filespec RECORD IS SHORTER THAN ASCII INTO VARIABLE

Explanation: You have executed a READ statement with a CHARACTER NONVARYING INTO variable, and the next input record in the file is shorter than the character string.

System Action: When CPL discovers this error, it reads the entire record into the storage of the INTO variable, without modifying the characters following the end of the record. Only then does CPL begin entering the error handling logic which may result in stopping execution. For this reason, the record is available in the INTO variable.

User Response: If you wish execution to continue when this error occurs, then you may use the RECORD ON-condition to specify what action CPL should take.

(SIZE) BIT STRING TOO LONG FOR CONVERSION TO ARITHMETIC

Explanation: You have attempted to convert a BIT string longer than BIT(35) to FIXED or FLOAT.

Note: Such a conversion is legal if all but the last 35 bits of the bit string are zeroes.

(STRINGRANGE) ARGUMENTS TO ident OUT OF RANGE

Explanation: The second or third argument to the SUBSTR built-in function or pseudo-variable is out of range.

Let k=the legngth of the first argument, let i=the value of the second argument. Let j=the value of the third argument, if it is specified, or else let j=k-i+1. Then an error condition occurs if any of these inequalities are not satisfied:

1. 1 <= i <= k+1

2. 0 <= j <= k-i+1

User Response: If you wish execution to continue when this error occurs, then you may use the STRINGRANGE ON-condition to specify what action CPL should take.

(ZERODIVIDE) DIVISION BY ZERO

Explanation: The denominator of a floating point division operation is zero.

User Response:

If you wish execution to continue when this error occurs, then you may use the ZERODIVIDE ON-condition to specify what action CPL should take.

0**(NON-POSITIVE #) IS ILLEGAL

Explanation: You have attempted to raise 0 to a FIXED power which is less than or equal to 0.

2ND ARG TO FUNCTION ident IS OUT OF RANGE

Explanation: The second argument to the DIMENSION, LBOUND or HBOUND function is out of range. This argument is required to be positive and less than or equal to the number of dimensions in the array of the first argument.

2ND ARGUMENT TO FUNCTION ident IS NEGATIVE

Explanation: The second argument to the COPY built-in function must be zero or a positive number.

ALL BLOCKS RESET

Explanation: CPL has found it necessary to terminate all active BEGIN blocks, PROCEDURE blocks, and ON-units in your program.

CPL takes this action whenever any of the following conditions occur:

1. You have inserted or erased any explicit declaration in your program. Explicit declarations include the DECLARE statement and a statement label.

2. You have entered or erased any BEGIN or PROCEDURE statement in your program.

3. You have entered or erased any ON statement not specifying the SYSTEM option.

4. You have typed the XEQ statement.

5. You are attempting to continue execution of a program in which you have entered or erased a DO or END statement which has changed the block structure of the program. CPL detects this by finding a BEGIN or PROC statement which is not matched with the same END statement as before execution stopped.

6. You are attempting to continue execution of a program which contains a multiple-closure END statement such that a BEGIN or PROCEDURE block is closed by a dummy end statement generated as a result of the multiple closure END. This is a restriction in CPL as a result of which it is recommended that you not use multiple closure END statements in this fashion.

7. You have erased or changed a statement in your program which invoked one of the active blocks. For example, if a PROCEDURE is active, then this can happen if you erase or change the statement which invoked the PROCEDURE, whether by CALL or by function reference. If an ON-unit is active, then all blocks will be reset if you erase or change the statement which contains the error which caused the ON-unit to be raised.

ALL STORAGE RESET

Explanation: CPL has released and reset all program storage. All variables which have previously been assigned values no longer have these values.

CPL takes this action in two circumstances:

1. If you enter a LOAD statement.

2. If you enter an ERASE statement which erases all remaining statements in your program.

ALLOCATE/FREE VBLE ident IS NOT CONTROLLED OR BASED

Explanation: A variable appearing in an ALLOCATE or FREE statement does not have the CONTROLLED or BASED storage class attribute.

User Response: Either retype the ALLOCATE statement to specify a BASED or CONTROLLED identifier, or else use a DECLARE statement to make the identifer CONTROLLED or BASED.

AN ON-UNIT MAY NOT BE A "DO" STATEMENT

Explanation: You have specified ON ... DO;, in violation of the CPL rules.

User Response: An ON-unit can be a single GOTO statement. If you wish to specify an ON-unit containing a group of statements, then you should use a BEGIN block rather than a DO group.

AN ON-UNIT MAY NOT BE AN "IF" STATEMENT

Explanation: You have specified ON ... IF ..., in violation of CPL rules.

User Response: An ON-unit may be a single GOTO statement. If you wish to use an IF statement, then enclose the IF statement in a BEGIN/END block, where you may specify as many statements as you like.

AN ON-UNIT MAY NOT BE LABELED

Explanation: An ON-unit may not have a statement label. This is true of a single statement ON-unit as well as the BEGIN statement of a multi-statement ON-unit.

User Response: If you wish to execute a block of code consisting of an ON-unit, you will have to use a method other than a GOTO to the ON-unit. Here are some possible ways:

1. You may use the SIGNAL statement to raise the ON-condition artificially, and so enter the ON-unit.

2. You may place the group of statements into a PROCEDURE, and then call the PROCEDURE from the ON-unit and from anywhere else in your program that you like.

ARG ident TO ident PSEUDO-VBLE MAY NOT BE AN ARRAY

Explanation: The string argument to the SUBSTR pseudo-variable must be a scalar.

ARG ident TO ident PSEUDO-VBLE MUST BE BIT OR CHAR TYPE

Explanation: The first argument to the SUBSTR pseudo-variable must be a BIT or CHARACTER string.

ARG ident TO FCN ident IS NOT A CHAR OR BIT NONVARYING ARRAY

Explanation: The STRING built-in function and pseudo-variable requires the following attributes for its argument: An array, either BIT or CHARACTER, which is NONVARYING.

ARG ident TO FNC ident IS NOT ARITHMETIC OR STRING TYPE

Explanation: The argument of the UNSPEC built-in function or pseudo-variable must be FIXED, FLOAT, CHARACTER or BIT.

ARG ident TO FUNCTION ident MUST BE AN ARRAY

Explanation: The first argument to the DIMENSION, LBOUND and HBOUND built-in functions must be an array.

ARG ident TO FUNCTION ident MUST BE CONTROLLED

Explanation: The argument to the ALLOCATION built-in function must have the CONTROLLED storage class attribute.

ARGUMENT TO "COLUMN" OPTION IS NEGATIVE

Explanation: The expression appearing with the COLUMN format item was negative.

User Response: Usually you use a positive value with the COLUMN format item. You are permitted to use a zero value; in this case, the value 1 is assumed.

ARRAY ident IS NOT ALLOCATED

Explanation: The specified array has not been allocated. For more information, please refer to the error message "ident" IS NOT ALLOCATED.

ARRAY ident SUBSCRIPT GREATER THAN UPPER BOUND

Explanation: One the subscripts in an array reference is greater than the upper bound for the subscript specified in the array declaration.

user Response: Usually this message indicates a simple programming error, such as an error in a loop variable. When execution stops, use the ? statement to examine the value of the subscript variables. Also, you may use the HBOUND built-in function to find out the upper bound of the dimension range of the array.

If the identifier is for a BASED array which has variables in the dimension bounds in the DECLARE statement, then the dimension bounds are evaluated each time the array identifier is referenced. Therefore, in this case, there are two possible programming errors: You may have an error in the evaluation of the array bounds in the DECLARE statement, or you may have an error in the subscript in the array reference.

If you wish execution to continue when this error occurs, then you may use the SUBSCRIPTRANGE ON-condition to specify what action CPL should take.

ARRAY ident SUBSCRIPT LESS THAN LOWER BOUND

Explanation: One of the subscripts in an array reference is less than the lower bound for the subscript specified in the array declaration. Note that if no lower bound is specified in the array declaration, then 1 is assumed.

user Response: Usually this message indicates a simple programming error, such as an error in a loop variable. When execution stops, use the ? statement to examine the value of the subscript variables. Also, you may use the LBOUND built-in function to find out the lower bound of the dimension range of the array.

If the identifier is for a BASED array which has variables in the dimension bounds in the DECLARE statement, then the dimension bounds are evaluated each time the array identifier is referenced. Therefore, in this case, there are two possible programming errors: You may have an error in the evaluation of the array bounds in the DECLARE statement, or you may have an error in the subscript in the array reference.

If you wish execution to continue when this error occurs, then you may use the SUBSCRIPTRANGE ON-condition to specify what action CPL should take.

ARRAY ARG ident HAS DIFFERENT STRING LENGTH THAN PARM ident

Explanation: You have invoked a procedure, specifying an array argument with the BIT [VARYING] or CHARACTER [VARYING] data type. The corresponding parameter has also been declared to be an array, with the same number of dimensions and the same data type, but the string length is different.

user Response: Retype the declaration for the parameter, specifying an asterisk for the string length rather than an explicit expression. CPL will match an asterisk to any string length in the argument.

This error may have occurred because you wish to reference a string array in a manner different from the way in which the string array was specified in the DECLARE statement. For example, suppose you wish to the elements in a CHAR(10) array as individual characters in a CHAR(1) array. You may do this using BASED storage and a POINTER variable. You let the POINTER variable equal the ADDR of the CHAR(10) array, and then you use the POINTER to qualify a BASED CHAR(1) array. For more information on this technique, please read the chapter on BASED storage and POINTERs.

ASSIGNMENT TO ARRAY ident FORBIDDEN

Explanation: An attempt has been made to assign an expression to an array. Such an assignment is illegal in CPL.

User Response: Replace the assignment statement with a DO-loop which will make the assignment to the array.

ASSIGNMENT TO NAMED CONSTANT ident FORBIDDEN

Explanation: The variable appearing on the left hand side of an assignment is a NAMED CONSTANT. FILE identifiers and statement labels are NAMED CONSTANTs.

ATTENTION

Explanation: You have typed a Control-C while CPL was executing a program.

User Response: If you wish execution to continue when you type the Control-C character, then you may specify an ON-unit for the ATTENTION ON-condition. Extreme care must be taken with such ON-units, however, since careless use of an ATTENTION ON-unit may make it impossible for you to exit from your program.

AUTOMATIC LINE NUMBER EXCEEDS 9999.99

Explanation: A line number generated by CPL, either as a result of the NUMBER statement or as a result of the NUMBER option of the LOAD or WEAVE statement, exceeds 9999.99.

System Action: If the line number was generated as a result of a NUMBER statement, automatic line numbering will be terminated and CPL will type an asterisk to signify that it is ready for a new command.

If the line number was generated as a result of the NUMBER option of the LOAD or WEAVE statement, then loading will terminate immediately and the input file will be closed. CPL will type an asterisk to signify that it is ready for a new command.

BASED IDENT ident NOT ALLOWED IN DECLARATION

Explanation: CPL restriction. A BASED variable may not appear in the extent expressions (array bounds or string lengths) of a DECLARE statement.

This error is detected when CPL has occasion to evaluate the extent expressions being evaluated. If the variable being DECLAREd is AUTOMATIC, this occurs when the program block containing the declaration is invoked; if it is CONTROLLED, when the ALLOCATE statement is executed; and if it is BASED, whenever it is referenced.

User Response: Replace the reference to a BASED variable in the extent expression with a reference to a non-BASED variable.

If the logic of your program requires that an expression with a BASED variable be used, then you may accomplish the same thing by replacing the reference to the BASED variable by a reference to a function PROCEDURE. The function PROCEDURE can return the value of the BASED variable.

BASED IDENT ident NOT ALLOWED IN FORMAT STATEMENT

Explanation: CPL restriction. A reference to a BASED variable may not appear in an expression which appears in a FORMAT statement. (Note, however, that a BASED variable may appear in the format specification of a PUT EDIT statement.) This error is detected when the FORMAT statement is referenced by an R format item reached during execution of a PUT EDIT statement.

User Response: Replace the reference to a BASED variable in the FORMAT statement with a reference to a non-BASED variable, or else move the format specification to the PUT EDIT statement so that a FORMAT statement will not be needed.

If the logic of your program requires that an expression with a BASED variable be used in a FORMAT statement, then you may accomplish the same thing by replacing the reference to the BASED variable by a reference to a function PROCEDURE. The function PROCEDURE can return the value of the BASED variable.

BIT STRING ident NOT ALLOCATED

Explanation: The specified BIT string is not allocated. For more information, please refer to the message "ident" IS NOT ALLOCATED.

BREAKPOINT

Explanation: Your program has reached a statement for which a previous BREAK statement indicated that a breakpoint should be set.

System Action: When CPL reaches such a statement it does not execute it. Instead, it returns to command level with the BREAKPOINT error message.

user Response: To continue executing, type the CONTINUE command. If you wish to continue executing from a different statement, they use the FROM option of the CONTINUE statement to specify which statement.

CALL TARGET ident IS NOT AN ENTRY

Explanation: The target specified in a CALL statement is not the statement label on a PROCEDURE statement.

CAN'T ALLOCATE CHANNEL FOR FILE filespec -- ALL CHANNELS IN USE

Explanation: (TOPS-10 only) CPL cannot allocate a channel to open a new file, since you already have 15 files open.

user Response: You may use the CLOSE FILES command to close all your files.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

CAN'T RECOGNIZE STATEMENT BEGINNING WITH "string"

Explanation: You have entered a statement beginning with an illegal element which CPL does not recognize.

CHARACTER STRING ident NOT ALLOCATED

Explanation: The specified CHARACTER string has not been allocated. For more information, please refer to the message "ident" IS NOT ALLOCATED.

CHARACTER/BIT STRING NOT TERMINATED IN INPUT RECORD

Explanation: In a GET LIST operation, CPL has begun reading a data item which begins with a single quote ('), but has read a line feed character before finding the closing quote.

user Response: Correct the erroneous record in your data file.

CMG -- INVALID TYPE REQUEST

Explanation: System error. A .CORE GET or FREE request has specified an illegal storage type. The storage type must be less than or equal to 5.

user Response: Please save all relevant output and mail it to Digital Equipment Corporation.

COMPOUND STATEMENTS ("IF" AND "ON") ARE COLLECT ONLY -- USE LINE NUMBER

Explanation: You have typed an IF or ON statement without a line number. All compound statements must have a line number.

Note: The ON statement with the SYSTEM option may be used as a direct statement, since it is not a compound statement.

CONDITION keyword[(ident)] SIGNALLED

Explanation: The specified condition was raised by a SIGNAL statement.

System Action: If an ON-unit is specified for the named condition, then the ON-unit will be entered. If there is no ON-unit specified, then execution will continue with the next statement after the message has been typed.

DATA ERROR FOR FILE "filespec"

Explanation: (TOPS-20 only) System error. Data or device error occurred while attempting to read or write a file.

User Response: This message usually indicates a potentially serious hardware error. Notify your system operator of the problem at once.

DATATYPE OF DIMENSIONED ARG ident DIFFERENT FROM PARM ident

Explanation: You have invoked a PROCEDURE with an array argument. The corresponding parameter is also an array, but with a different data type.

Note: CHAR NONVARYING is a different data type from CHAR VARYING. Similarly, BIT NONVARYING is a different data type from BIT VARYING.

DDT IS NOT AVAILABLE

Explanation: (TOPS-10 only) You have executed the DDT command, but DDT is not linked into the CPL system you are using.

DECLARE AND DEFAULT STMTS MAY NOT BE "THEN" OR "ELSE" CLAUSES

Explanation: Self-explanatory.

DECLARE AND DEFAULT STMTS MAY NOT BE LABELED

Explanation: Self-explanatory.

DEFAULT RANGE ID "string" MUST HAVE ONLY ONE LETTER

Explanation: The format of the RANGE specification is RANGE(let) or RANGE(let:let), where "let" is a single letter.

DEVICE "device" IS ILLEGAL

Explanation: (TOPS-10 only) The device specified in the TITLE option of an OPEN statement, or in a LOAD, SAVE or WEAVE statement, is illegal.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DEVICE "device" IS NOT AVAILABLE TO THIS JOB

Explanation: (TOPS-10 only) The device specified in the TITLE option of an OPEN statement, or in a LOAD, SAVE or WEAVE statement, is not available to this job. This will happen if the device is not shareable and is currently assigned to another job.

User Response: Either specify a different device, or contact the system operator and ask him to make the device available to you.

DIMENSION BOUNDS OF ARG ident DO NOT MATCH THOSE OF PARM ident

Explanation: In a PROCEDURE invocation, you have specified as an argument an array. The corresponding parameter has also been declared to be an array, but the array bounds do not match.

User Response: You should change your parameter declaration so that an asterisk is specified for each array bound. If you do that, then the argument may have any dimension bounds.

It may be that you wish to reference a storage area differently in a PROCEDURE than you do in your main program. For example, you may wish to reference a 4x3 array as a 2x6 array in the PROCEDURE. You may do this in CPL by means of BASED storage and a POINTER variable. You let the POINTER variable equal ADDR of the 4x3 array, and you use it to qualify a BASED 2x6 array. For more information on this technique, please read the chapter on BASED storage and POINTERs.

DIMENSIONED ARG ident MAY NOT BE PASSED AS SCALAR PARM ident

Explanation: You have invoked a PROCEDURE specifying as an argument an array, but the matching parameter is a scalar.

User Response: If you wish to pass an array as an argument to a PROCEDURE, then you must supply an array declaration for the parameter inside the PROCEDURE.

It may be that you wish to reference a storage are differently in a PROCEDURE than you do in your main program. For example, you may wish to reference a CHAR(1) NONVARYING array as a single CHAR scalar in the PROCEDURE. You may do this in CPL by means of BASED storage and a POINTER variable. You let the POINTER variable equal the ADDR of the CHAR(1) array and you use it to qualify a BASED CHAR scalar. For more information on this technique, please read the chapter on BASED storage and POINTERs.

DO LOOP VBLE ident IS AN ARRAY

Explanation: A DO-loop variable must be a scalar. It may be a subscripted array element.

DO TARGET VBLE "ident" IS NO LONGER ALLOCATED

Explanation: Your program has executed a DO statement with a DO loop variable, but sometime since the location of the DO variable was established, the DO variable has been freed.

This can happen, for example, if your DO loop contains a FREE statement to free the storage occupied by the DO variable.

For more information, please refer to the message, "ident" IS NOT ALLOCATED.

DO VBLE ident IS NOT COMPUTATIONAL OR PTR DATA TYPE

Explanation: A DO-loop variable must be FIXED, FLOAT, CHARACTER BIT, or POINTER.

DOUBLE "ELSE" KEYWORD IS ILLEGAL

Explanation: Self-explanatory.

DOUBLE DEVICE NAME IN FILEID "string"

Explanation: (TOPS-10 only) The file-specification given by the TITLE option of the OPEN statement or in the WEAVE, SAVE or LOAD statement contains two device names. A device name is followed in the file-specification by the character colon (:). You may specify only one.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DOUBLE DOT IN FILEID "string"

Explanation: (TOPS-10 only) There are two dots in the file-specification appearing in the TITLE option of an OPEN statement or in the LOAD, SAVE or WEAVE statement. The file-specification may contain only a single dot. This dot appears between the filename and the filename extension.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DOUBLE FILENAME IN FILEID "string"

Explanation: (TOPS-10 only) Two filenames appeared in the file-specification in the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statement.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DOUBLE PPN SPECIFIED IN FILEID "string"

Explanation: (TOPS-10 only) Two project-programmer numbers appear in the file-specification in the TITLE option of the OPEN statement or in the WEAVE, SAVE or LOAD statement.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DTA PROTECTION FAILURE OR DIRECTORY FULL FOR filespec

Explanation: (TOPS-10 only) CPL is unable to open an OUTPUT file on the desired DECtape because either the directory is full or the device is write-protected.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

DUMMY I/O MODULE FILRW

Explanation: The REWRITE statement is not implemented.

DUMMY ROUTINE NAME

Explanation: The "name" is the name of a routine which has not been implemented in CPL. You have tried to use a feature of CPL which is only partially implemented.

User Response: Do not use unimplemented features of CPL.

DUPLICATE KEYWORD "keyword"

Explanation: Either the statement you have typed in uses the same keyword twice, or a declaration specifies the same attribute twice for the same identifier.

EKDECL -- INVALID DCLOP VALUE

Explanation: System error. The DCLOP field of the current BSB contains an invalid value.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

END LABEL ident CANNOT BE MATCHED

Explanation: Your program contains a statement of the form "END label," and CPL is unable to match the label. This can happen either if there is no DO, PROC or BEGIN statement with the specified label, or if all such statements are already matched by different END statements.

END OF FILE ON filespec

Explanation: A GET or READ operation has failed because the end of the file has been reached.

<u>User Response:</u> If you wish execution to continue when this error occurs, then you may use the ENDFILE(filename) ON-condition to specify what action CPL should take.

END OF PROGRAM

<u>Explanation:</u> Program execution has completed. You have executed the last statement of the program.

END OF STRING ON GET STRING OPERATION

<u>Explanation:</u> You have executed a GET LIST statement with the STRING option, and the end of the string was reached before the entire data list was satisfied.

<u>User Response:</u> This error will occur if you forget to insert a blank at the end of the input string. A GET LIST operation is not satisfied until the blank or comma following the end of the data item has been read. If this is the reason for the error, then concatenate the expression in the STRING option with a blank.

EOPIPL -- CAN'T HANDLE IO OPERATION

<u>Explanation:</u> System error.

<u>User Response:</u> Save all relevant output and mail it to Digital Equipment Corporation.

EOPSS -- CANNOT HANDLE ARRAY ident

<u>Explanation:</u> The SMB operator following the CCOSS operator and the subscript count is not recognized by the EOPSS routine.

<u>User Response:</u> This is a system error. Please save all relevant output and mail it to Digital Equipment Corporation.

ERROR IN **

<u>Explanation:</u> An error was detected in attempting to raise a number to a FLOAT power.

An error will be detected if you attempt to raise 0 to a value which is zero or negative.

ERROR IN FUNCTION ident

<u>Explanation:</u> An error has occurred in the computation of a mathematical built-in function.

The following errors will give rise to this error message:

1. ASIN or ACOS has an argument greater than 1 in magnitude.

2. SQRT has a negative argument.

3. The argument to SINH or COSH exceeds, in absolute value, the value of 88.029+LOG(2).

4. Argument of EXP exceeds 88.029 or is less than -89.41.

5. Argument of LOG, LOG2 or LOG10 is negative.

ERROR IN OPEN UUO FOR FILE filespec

Explanation: (TOPS-10 only) System error. An unexpected error occurred in an OPEN UUO.

User Response: Save all relevant output and send to Digital Equipment Corporation.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

ERROR READING UFD RIB OR FILE RIB FOR FILE filespec

Explanation: (TOPS-10 only) One of the following has occurred while CPL was trying to open a file:

1. A hardware-detected device or data error was detected while reading the UFD RIB or UFD data block.

2. A software-detected data inconsistency error was detected while reading the UFD RIB or file RIB.

EXCBIF -- FUNCTION ident IS NOT IMPLEMENTED

Explanation: The specified built-in function has not been implemented in CPL.

EXCLUP -- CAN'T HANDLE SMB OPERATOR

Explanation: The SMB contains an operator which the execution routines cannot handle. This will happen if you attempt to use a partially implemented feature which is not described in the documentation.

User Response: If you have used an undocumented feature, then you should remove that use from your program.

If you are using only documented features, then this message indicates a system error. Please save all relevant output and send it to Digital Equipment Corporation.

EXCLUP -- CAN'T HANDLE STATEMENT TYPE

Explanation: The statement type cannot be handled by the execution routines. This will happen if the statement type is only partially implemented.

User Response: Remove all uses of undocumented statements from your program.

EXCLUP -- INVALID ACTION CODE RETURNED FROM SUBROUTINE

Explanation: System error. A subroutine called by EXCLUP returned an invalid value in the AACT argument.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

EXCLUP -- INVALID VALUE IN DCLOP IN CALL TO EXCLUP

Explanation: System error. An invalid value was found in the DCLOP field of the current BSB.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

EXPONENT MUST BE ONE OR TWO DIGITS

Explanation: In entering an E-type floating point number, the exponent field must contain one or two digits. For example, 2.3E123 is illegal.

FIELD TOO NARROW IN F FORMAT ITEM

Explanation: Let w and d be the first two arguments to the F format item, and let v be the value being printed. Then the following inequalities must be satisfied:

1. If d=0 and v>=0 then w must satisfy w>0.
2. If d=0 and v<0 then w must satisfy w>1.
3. If d>0 and v>=0 then w must satisfy w>d+1.
4. If d>0 and v<0 then w must satisfy w>d+2.

This error message will appear if the appropriate inequality is not satisfied.

User Response: Increase the field size in the F format item.

FIELD WIDTH TOO NARROW IN E FORMAT ITEM

Explanation: The width argument to the E format item is too narrow.

Here is how to compute the minimum field width needed.

1. The minimum size is s+4, where s is the number of significant digits -- the value specified in the third argument.

2. If s=d, where d is the second argument, then add 1.

3. If d>0 then add 1.

4. If the data value is negative, then add 1.

If the first argument to E is smaller than the above quantity, then this error message will be typed.

User Response: Increase the size of the first argument to the E format item.

FILE filespec IS keyword keyword INSTEAD OF STREAM OUTPUT

Explanation: The file specified in a PUT operation must be opened with the attributes OUTPUT and STREAM.

User Response: There are several ways in which this error can be made:

1. If you have been reading a file using the same file identifier, you must specify the CLOSE statement for the file before you can start to write to it. Keep in mind that an OPEN statement is a "no-operation" if the file variable is already open, even with different attributes.

2. It is illegal to perform WRITE and PUT operations on the same output file.

3. The declaration for the file variable may contain attributes which conflict with OUTPUT and STREAM. In this case you should change your file declaration.

4. It is legal to be doing GETs and PUTs to the same disk file provided that you are using different file variables in CPL. If you are doing GETs and PUTs to the same disk file, then use two different variables and open them separately for INPUT and OUTPUT using the TITLE option of the OPEN statement.

FILE filespec IS ALREADY BEING MODIFIED

Explanation: (TOPS-10 only) A file cannot be opened for output if either this job or another job is already writing that file.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

FILE filespec NOT FOUND

Explanation: (TOPS-10 only) An attempt has been made to open a file for INPUT, but the file was not found.

If you do not open the file explicitly with the TITLE option of the OPEN statement, then CPL supplies the default file-specification of DSK:name.DT, with your own project-programmer number.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

FILE NAME MISSING FOR FILE filespec

Explanation: (TOPS-10 only) A file-specification given by the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statement does not specify a filename, and the device requires a LOOKUP or ENTER.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

FILE NOT "INPUT" OR "UPDATE"

Explanation: A READ operation was attempted on a file open for OUTPUT.

FILE NOT "OUTPUT" OR "UPDATE"

Explanation: A WRITE operation was attempted on a file open for OUTPUT.

FILE NOT OPEN IN "RECORD" MODE

Explanation: A READ or WRITE operation was attempted on a file opened with the STREAM attribute.

CPL does not allow you to perform RECORD and STREAM operations on the same file.

FILE VARIABLE ident IS NOT A FILE

Explanation: The identifier appearing in the FILE option of an OPEN, CLOSE, GET, PUT, READ, WRITE or CLOSE statement has not been declared to have the FILE attribute.

FILJFN -- JSYS ERROR: DEVICE FIELD NOT IN A VALID POSITION

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement or in the LOAD, WEAVE or SAVE statement contains a misplaced device field.

User Response: Respecify the file-specification so that the device field is given first.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DEVICE IS NOT AVAILABLE TO THIS JOB

Explanation: (TOPS-20 only)You are attempting to open a file, but the device you have specified is not shareable, and it is in use by another user or cannot be assigned.

User Response: Contact your system operator and arrange to have the device assigned to your job.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DEVICE IS NOT ON-LINE

Explanation: (TOPS-20 only) You are attempting to open a file, but the device you have specified is either off-line or not ready.

User Response: Ask the operator to put the device on-line.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DIRECTORY ACCESS PRIVILEGES REQUIRED

Explanation: (TOPS-20 only) You do not have the privileges necessary to access the directory specified in the TITLE option of the OPEN statement, or in the LOAD, SAVE or WEAVE statement.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DIRECTORY FIELD NOT IN A VALID POSITION

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement or in the LOAD, WEAVE or SAVE statement contains a misplaced directory field.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DIRECTORY FULL

Explanation: (TOPS-20 only) You have attempted to open a file for OUTPUT, either explicitly with an OPEN statement, or implicitly with a SAVE, PUT or WRITE statement, but the target directory has no room left.

User Response: You must use the "MONITOR" command to return to command level in the operating system, and then you must delete some files to make room in your directory.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: DIRECTORY TERMINATING DELIMITER IS NOT PRECEDED BY A VALID BEGINNING DELIMITER

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, is missing the left angle bracket which defines the start of a TOPS-20 directory field.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FIELD CANNOT BE LONGER THAN 39 CHARACTERS

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, contains a directory, name, or type field that is too long.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILE NAME MUST NOT EXCEED 6 CHARACTERS

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, SAVE or WEAVE statement, has a name field longer than six characters.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILE NOT FOUND BECAUSE OUTPUT-ONLY DEVICE WAS SPECIFIED

Explanation: (TOPS-20 only) You are attempting to open a file for INPUT, but the LOAD statement or the TITLE option of the OPEN statement specifies a device name which is valid for output only.

Note that in an OPEN statement, if you don't specify whether a file is INPUT or OUTPUT, then INPUT is assumed.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILE NOT FOUND

Explanation: (TOPS-20 only) You have attempted to open a file for INPUT, either explicitly with an OPEN statement, or implicitly with a GET, READ or LOAD statement, and the file does not exist.

Note that if your OPEN statement does not specify either INPUT or OUTPUT, then INPUT is assumed.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILE TYPE MUST NOT EXCEED 3 CHARACTERS

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, contains a type field which is longer than three characters.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILE WAS EXPUNGED

Explanation: (TOPS-20 only) You are attempting to open a file for input, either implicitly with the OPEN statement, or explicitly with the GET, READ, LOAD or WEAVE statement, but the file has been entirely deleted from the system.

User Response: You will have to find a way to recreate the file.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: FILENAME WAS NOT SPECIFIED

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, does not contain a name field. The name field may be omitted only for a non-directory device, such as the line printer or card reader.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: INTERNAL FORMAT OF DIRECTORY IS INCORRECT

Explanation: (TOPS-20 only) System error. In attempting to open a file, either explicitly with the OPEN statement, or explicitly with the GET, PUT, READ, WRITE, LOAD, SAVE, or WEAVE statements, the operating system discovered an illegal field in the disk directory.

User Response: Inform your system operator immediately, as this usually indicates a serious error.

FILJFN -- JSYS ERROR: INVALID CHARACTER IN FILENAME

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, SAVE or WEAVE statement, has a file name field which contains an illegal character.

User Response: Respecify the file-specification so that the name field contains only letters and digits.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: JOB STORAGE BLOCK FULL

Explanation: (TOPS-20 only) System error. You have attempted to open a file, but the your job could not obtain any free space. Normally this could happen only if you have more than 60 files open.

User Response: Bring this problem to the attention of your system analyst.

FILJFN -- JSYS ERROR: LOGICAL NAME LOOP DETECTED

Explanation: (TOPS-20 only) You have used two or more DEFINE commands to the TOPS-20 monitor to define in a circular manner the device name in the file you are trying to open.

User Response: Redefine the logical name using the appropriate monitor commands.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: MORE THAN ONE NAME FIELD IS NOT ALLOWED

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, contains more than one file name field.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: NO JFNS AVAILABLE

Explanation: (TOPS-20 only) You are attempting to open a file, but you have already opened the system-defined maximum number of files permitted for your job. (The maximum number of JFN's is greater than 60.)

User Response: You must issue CLOSE statements to close some of the files you have opened, before you can open another one. This will require modification to your program. You may use the CLOSE FILES command to close all your open files simultaneously.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: NO SUCH DEVICE

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, specifies a device name which does not exist in the system.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: NO SUCH DIRECTORY

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement or in the LOAD, WEAVE or SAVE statement, specifies a directory which does not exist.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: NO SUCH FILE TYPE

Explanation: (TOPS-20 only) You are attempting to open a file for INPUT, but the file type for the specified file does not exist.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: NO SUCH FILENAME

Explanation: (TOPS-20 only) You are attempting to open a file for INPUT, but there is no existing file with the specified filename.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILJFN -- JSYS ERROR: UNDEFINED ATTRIBUTE IN FILE SPECIFICATION

Explanation: (TOPS-20 only) The file-specification given by the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, contains an unrecognizable expression in the TOPS-20 file attribute field.

User Response: The following are the legal values for the file attribute field:

1. P ("protection"), followed by six octal digits (this is the default)

2. A ("account"), followed by an account ID

3. T ("temporary"), for a temporary file

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILRD -- NO DIRECT OR WORD

Explanation: DIRECT and WORD modes are not implemented.

FILSOO -- OPEN FAILURE: DEVICE ASSIGNED TO ANOTHER JOB

Explanation: (TOPS-20 only) The device specified in the file-specification in the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, is currently unavailable because it is not shareable and it is assigned to another user.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: DEVICE IS NOT ON-LINE

Explanation: (TOPS-20 only) The device specified in the TITLE option of the OPEN statement, or in the LOAD, WEAVE or SAVE statement, is off-line or not ready.

User Response: Ask the system operator to make the device online and ready.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: DEVICE IS WRITE-LOCKED

Explanation: (TOPS-20 only) You have attempted to open a file for OUTPUT, but the output device is "write-locked" and so you cannot write on it.

User Response: This error occurs most frequently with tape devices. You must ask the system operator to write-enable the device.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: DISK QUOTA EXCEEDED

Explanation: (TOPS-20 only) You are attempting to open a file for OUTPUT, but your disk file quota has been exceeded.

User Response: You must use the MONITOR command to return to monitor level, and delete some files before you can create any new files.

If you do not wish to delete any files, then you must contact your system administrator and ask him to increase your file space quota.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: ENTIRE PUBLIC DISK FULL

Explanation: (TOPS-20 only) You have attempted to open a file for OUTPUT, but there is no space left on the disk pack.

User Response: This is usually a serious system problem, since no users will be able to create any new files until more disk space can be found. Therefore, you should notify your system operator of the problem.

If you have several disk packs available to you, then you may be able to get around the problem by specifying a different disk for your output file.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: FILE DOES NOT EXIST

Explanation: (TOPS-20 only) You are attempting to open a file for INPUT, but the following problem has occurred: The file has been created, but has not yet been closed.

This can happen if your CPL program attempts to read a file that it has just created, but it fails to issue a CLOSE statement to close the file after creating it.

User Response: Modify your program so that it executes a CLOSE statement for the file identifier for the newly created file.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: FILE HAS BAD INDEX BLOCK

Explanation: (TOPS-20 only) System error. You have attempted to open a file, but your disk structure contains a bad index block.

User Response: This problem usually indicates a potentially serious disk hardware error. Report this problem to your system operator immediately.

FILSOO -- OPEN FAILURE: FILE IS ALREADY OPEN

Explanation: (TOPS-20 only) CPL system error. CPL issued a JSYS to open a control block which was already open.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: INVALID SIMULTANEOUS ACCESS

Explanation: (TOPS-20 only) One of the following has happened:

1. You have attempted to open a file for output, but the file is already open for output either by your job or by another job.
2. You have attempted to open a file on a non-directory device, such as a tape device, but there is already a file open on that device.

User Response: This message usually indicates that you have forgotten to execute a CLOSE statement before attempting to open the same file using a different file variable. If this is the case, then modify your program by adding such a CLOSE statement.

If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: NO ROOM IN JOB FOR LONG FILE PAGE TABLE

Explanation: (TOPS-20 only) System error. You have attempted to open a file, but the TOPS-20 operating system has insufficient free space for your job in the page table.

User Response: Notify your system analyst.

FILSOO -- OPEN FAILURE: READ ACCESS REQUIRED

Explanation: (TOPS-20 only) You are attempted to open a file for INPUT, but your job does not have the access privileges necessary to read the file you have specified.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILSOO -- OPEN FAILURE: WRITE ACCESS REQUIRED

Explanation: (TOPS-20 only) You have attempted to open a file for OUTPUT, but your job does not have the access privileges necessary to write-access the file you have specified.

User Response: If you wish execution to continue after this error occurs, then you may use the UNDEFINEDFILE ON-condition to specify what action CPL should take.

FILWR -- NO DIRECT OR WORD

Explanation: DIRECT and WORD modes are not implemented.

FIXEDOVERFLOW OR FLOAT OVERFLOW OR UNDERFLOW

Explanation: Your program is attempting to compute a value which lies outside the range of values permitted by the DECsystem-10/20 hardware. The terms used in the error message have the following meanings:

1. FIXEDOVERFLOW occurs when a computation on FIXED values exceeds, in absolute value, the maximum integer which the DECsystem-10/20 can compute. This value is equal to 2**35-1, or 34359738367.

2. FLOAT OVERFLOW occurs when a computation on FLOAT values exceeds, in absolute value, the maximum floating point value which the DECsystem-10/20 hardware can compute. This value is equal to 2**127, or 1.7014118E+38.

3. FLOAT UNDERFLOW occurs when a computation on FLOAT values causes the exponent fields of a floating point value to become less than the minimum value which the DECsystem-10/20 hardware permits. The minimum positive value which the hardware permits is 2**-129, or 1.4693679E-39.

This error can occur in a variety of circumstances.

If the message occurs when you type in a statement, then probably one of the following has happened:

1. You have typed in a FIXED or FLOAT constant which is out of the hardware's range, as described above.

2. You have typed in array bounds or string lengths which are so large that the computation of the required storage caused a computational error. (Of course, in this case, there would not have been enough storage to allocate the data block anyway.)

If the message occurs during execution of a program, then one of the following may have occurred:

1. Some numeric computation exceeded the hardware's range, as described above.

2. A constant which was converted as the result of a GET LIST statement was too large.

3. A constant appearing in a CHARACTER string, which was used in a CHARACTER to FIXED or CHARACTER to FLOAT conversion was too large.

4. Your program contains variables in the extent expressions (array bounds or string lengths) of a DECLARE statement, and, at the time that CPL attempts to evaluate the extent expressions (either because of block entry for AUTOMATIC storage identifier or because of an ALLOCATE statement for CONTROLLED storage or because of any reference for BASED storage) the computation of the data block size resulted in an excessively large intermediate value. (Of course, in this case, there would not have been enough storage to allocate the data block anyway.)

It is also possible that this message will occur if you pass an illegal argument to a built-in function; but usually an argument error to a built-in function will cause the message "ERROR IN FUNCTION ident."

FIRST ARG OF FCN ident MUST BE A SIMPLE OR SUBSCRIPTED ID

Explanation: The first argument of the specified built-in function or pseudo-variable must be a simple identifier or a subscripted identifier. It may not be an expression. For example, the SUBSTR pseudo-variable has this requirement.

FORMAT STATEMENT MUST BE LABELED

Explanation: A FORMAT statement must have a statement label.

FROM/INTO VBLE ident MAY NOT BE CHAR VAR ARRAY FOR ASCII READ/WRITE

Explanation: The variable specified in the FROM option of the WRITE statement or the INTO option of the READ statement may be a CHARACTER VARYING or NONVARYING scalar, or it may be a CHARACTER NONVARYING array. It may not be a CHARACTER VARYING array.

FROM/INTO VBLE ident MUST BE CHAR FOR ASCII READ/WRITE

Explanation: The variable specified in the FROM option of the WRITE statement or the INTO option of the READ statement must be CHARACTER or CHARACTER VARYING.

FUNCTION ident MUST HAVE 3 TO 5 ARGS

Explanation: The FLTED function requires between 3 and 5 arguments.

FUNCTION ident MUST HAVE AT LEAST 2 ARGUMENTS

Explanation: The MIN and MAX built-in functions must have at least two arguments.

GET LIST ITEM ident HAS NON-COMPUTATIONAL DATA TYPE

Explanation: Each item appearing in a GET LIST data list must be FIXED, FLOAT, CHAR or BIT.

GET LIST ITEM ident IS A NAMED CONSTANT

Explanation: An item appearing in a GET LIST data list may not be a NAMED CONSTANT. Statement label and FILE identifiers are examples of NAMED CONSTANTS.

GET LIST ITEM ident IS NOT A SCALAR

Explanation: It is illegal to specify an array as a GET LIST data item.

GOTO TARGET ident IS NOT A LABEL

Explanation: The target of the GOTO statement is not a statement label.

Note that you may not GOTO into a separate block from outside that block. Therefore, this message will also be typed if the target is a statement label for a statement which appears in a block which does not contain the GOTO statement.

HARD DEVICE-DETECTED ERROR FOR FILE filespec -- FILE STATUS code

Explanation: (TOPS-10 only) Hard device detected error (IO.DER), other than data parity error. This is a search, power supply, or channel memory parity error. The device is in error rather than the data on the medium. However, the data read into core or written on the device is probably incorrect.

ID ident DCL'D "PARAMETER" BUT NOT IN A PROC STATEMENT

Explanation: Your program contains a DECLARE statement which specifies the PARAMETER attribute for an identifier. Either the DECLARE statement in not immediately inside a PROCEDURE or else it is immediately inside a PROCEDURE but the identifier does not appear in the parameter list of the PROCEDURE statement.

ID ident HAS "VARYING" SPECIFIED WITHOUT "CHAR" OR "BIT"

Explanation: The VARYING attribute may be specified only if CHARACTER or BIT is also specified.

ID ident HAS VARIABLE EXTENT EXPRESSIONS IN OUTER BLOCK

Explanation: Your program contains a DECLARE statement for AUTOMATIC storage with variable extent expressions, but the declaration is not inside a a BEGIN/END block, a PROCEDURE/END block, or an ON-unit. (An extent expression is one which is a string length in the CHARACTER or BIT attribute, or which is an array bound.)

User Response: If you wish to use variable string lengths or array bounds, then you must specify them differently. There are several techniques you can use:

1. Use the identifier only inside some PROCEDURE or BEGIN block, by placing the DECLARE statement inside that block. Of course, you will not be able to access that variable outside the block.

2. If you want to use the variable in your whole program, then you can place your whole program inside a BEGIN/END block. Then the DECLARE statement will be legal.

3. A simpler technique is to give the variable the CONTROLLED storage class attribute. (AUTOMATIC is the default.) If you do that, then you may use variables in your extent expressions. You will have to explicitly allocate that storage for the variable by means of the ALLOCATE statement, and you will have to free it using the FREE statement.

4. You may also used BASED storage and POINTER variables to have the same effect, as described in the chapter on BASED storage and POINTER variables. BASED storage is functionally more powerful than CONTROLLED storage for this purpose, but it is less efficient.

ID ident IS NOT A BUILTIN FUNCTION

Explanation: You have attempted to DECLARE the specified identifier with the BUILTIN attribute, but that identifier is not a legal built-in function in CPL.

ID ident IS NOT A LEGAL PSEUDO-VARIABLE

Explanation: The specified built-in function may not appear as a pseudo-variable on the left-hand side of an assignment statement.

ID ident IS NOT AN ARRAY OR FUNCTION

Explanation: The specified identifier is followed by a parenthesized list of expressions, but it not an array or a legal CPL function.

IDENTIFIER LONGER THAN 31 CHARACTERS

Explanation: You have entered a statement containing an identifier longer than 31 characters.

ILLEGAL ARGUMENT TO IGNORE OPTION

Explanation: The argument of the IGNORE option of the READ statement may not be negative.

ILLEGAL CHAR "char" IN BIT STRING

Explanation: The specified character is illegal for this type of bit string constant.

A bit string constant has the format '...'radix, where the "radix" is one of B, B1, B2, B3 or B4. The characters that may appear between the single quotes depend upon the radix, as follows:

Radix	Number system	Legal digits
B or B1	Binary	0,1
B2	Base 4	0,1,2,3
B3	Octal	0-7
B4	Hexadecimal	0-9, A-F

ILLEGAL CHARACTER "char"

Explanation: The specified character may not appear in a CPL statement except in a character string constant.

ILLEGAL CHARACTER "char" IN FILEID "string"

Explanation: (TOPS-10 only) The specified character may not appear in a file-specification.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

ILLEGAL DATA TYPE FOR PUT LIST ITEM

Explanation: The data type of a PUT LIST or PUT EDIT expression must be FIXED, FLOAT, CHAR or BIT.

ILLEGAL DEBUG OPTION

Explanation: The first argument of the DEBUG statement must be in the range 1 to 5. (The DEBUG statement is not useful to most programmers. It is used by system programmers who are debugging the CPL system itself.)

ILLEGAL DECIMAL POINT IN EXPONENT

Explanation: In an E-type floating point constant, there may be only one decimal point, and that must appear before the E. EXAMPLE: 1.289E5 is a legal constant, equal to 128900.

ILLEGAL INSTRUCTION EXECUTED AT LOCATION "location"

Explanation: (TOPS-20 only) CPL system error. CPL has executed an illegal instruction at the specified location.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

IMPROPER MODE FOR FILE filespec -- FILE STATUS code

Explanation: (TOPS-10 only) An IN or OUT UUO failed, with the specified file status.

User Response: This is a system error. Save all relevant output and send to Digital Equipment Corporation.

INTEGER DIVISION NOT ALLOWED

Explanation: CPL restriction: You may not divide two variables or expressions with the FIXED attribute.

User Response: There are several methods you can use to accomplish the same result:

1. Make one of the expressions FLOAT. For example, replace I/2 with I/2.0. In this case, the FLOAT denominator will cause CPL to convert the numerator to FLOAT before performing the division.

2. Use the DIVI or DIVF built-in functions. These functions are not in the ANSI standard, but they have been provided in CPL to because integer division is not available.

   DIVI(I,J) takes the quotient of the FIXED values I and J, and returns the truncated quotient.

   DIVF(I,J) causes the two arguments to be converted to FLOAT. FLOAT division is performed on the converted arguments and the FLOAT quotient is returned.

INVALID "ELSE" CLAUSE

Explanation: An ELSE clause appears in your program in a position where it cannot be matched with an IF statement. An implementation restriction makes it impossible for CPL to execute even direct mode statements while you have such an error in your program.

INVALID "FROM" OR "INTO" VARIABLE ident

Explanation: The variable specified in the FROM option of the WRITE statement or the INTO optin of the READ statement must be either a CHARACTER VARYING or NONVARYING scalar or a CHARACTER NONVARYING array.

INVALID ARGUMENT TO keyword OPTION

Explanation: A negative argument is illegal.

INVALID ASSIGNMENT CONVERSION AMONG NON-COMPUTATIONAL DATA TYPES

Explanation: The computational data types, FIXED, FLOAT, CHAR and BIT. It is illegal to convert a non-computational data type (FILE, LABEL, POINTER, etc.) to a computational data type, or vice-versa.

INVALID ASSIGNMENT STATEMENT

Explanation: An assignment statement has the format

```
          target [,target]...  = expression;
```

where the "target" is either a simple identifier or a subscripted identifier.

INVALID CHAR "char" AT END OF STRING "string"

Explanation: A character string constant is of the form '...'. A bit string constant is of the form '...'radix, where "radix" is one of B, B1, B2, B3 and B4.

INVALID CHAR "char" IN BIT STRING TOKEN "string"

Explanation: The specified character is illegal for this type of bit string constant.

A bit string constant has the format '...'radix, where the "radix" is one of B, B1, B2, B3 or B4. The characters that may appear between the single quotes depend upon the radix, as follows:

Radix	Number system	Legal digits
B or B1	Binary	0,1
B2	Base 4	0,1,2,3
B3	Octal	0-7
B4	Hexadecimal	0-9, A-F

INVALID CHAR "char" IN CHAR TO BIT CONVERSION OF "string"

Explanation: In a CHARACTER to BIT conversion, the CHARACTER string source may not contain any other character besides 0 and 1.

INVALID CHAR "char" IN NUMERIC STRING "string"

Explanation: The only characters which are legal in numeric strings are the digits 0-9, the decimal point (.), a sign (+ or -), and the letter E in E-type FLOAT constants.

INVALID CHAR "char" IN PPN FOR FILEID "string"

Explanation: (TOPS-10 only) An illegal character appears in the project-programmer number in the file-specification in the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statement. A project-programmer number has the format [proj#,prog#], where each of the numbers is octal, containing only the digits 0-7.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

INVALID CHARACTER "'" IN CHARACTER TO ARITHMETIC CONVERSION

Explanation: A numeric string, used in a CHARACTER to arithmetic conversion, must contain an arithmetic constant, possibly with blanks preceding and following.

INVALID COMPARISON OF NON-COMPUTATIONAL DATA TYPES

Explanation: When comparing two POINTER values, you may test them only for equality or inequality (= or ^=). The other comparison operators (e.g., greater than) are valid only for computational data types.

INVALID CONVERSION -- NON-COMPUTATIONAL TO keyword

Explanation: An attempt has been made to convert a non-computational data type (FILE, LABEL, POINTER etc.) to a computational data type (FIXED, FLOAT, CHARACTER, BIT). Such conversions are illegal.

INVALID CONVERSION FROM keyword TO keyword

Explanation: An attempt has been made to convert a non-computational data type (FILE, LABEL, POINTER, etc.), to a computational data type (FIXED, FLOAT, CHARACTER or BIT), or vice-versa. Such conversions are illegal.

INVALID DIMENSION RANGE FOR ident

Explanation: In the declaration for the specified array, the lower bound for one of the dimensions exceeds the upper bound.

Note: If no lower bound is specified in a dimension range, then 1 is assumed.

INVALID DUPLICATE OR CONFLICTING DECLARATIONS OF ident IN SAME BLOCK

Explanation: A block contains two explicit declarations of the same identifier. All the following are treated by CPL as explicit declarations:

1. An identifier being declared in a DECLARE statement.
2. An identifier appearing in a statement label.
3. An identifier appearing in the parameter list of a PROCEDURE statement.

Note: There is one circumstance in which two explicit declarations of the same identifier may appear in the same block: If an identifier appears in the parameter list of a PROCEDURE statement, then the PROCEDURE block may contain a separate declaration of the parameter in a DECLARE statement giving the data type attributes.

INVALID IDENTIFIER IN EXTENT OR INITIAL EXPRESSION FOR STATIC STORAGE FOR ident

Explanation: A STATIC storage class may not have variable extent expressions. All expressions appearing in dimensions bounds or string lengths must contain only constants.

INVALID NEGATIVE ARGUMENT FOR DELAY

Explanation: The argument to the DELAY statement must be zero or positive.

INVALID NEGATIVE ARGUMENT TO FUNCTION ident

Explanation: The argument to the HIGH and LOW built-in functions must be 0 or positive.

INVALID OR DOUBLE DOT IN NUMERIC STRING "string"

Explanation: A numeric string may contain only a single decimal point, and it must appear before the E in an E-type FLOAT constant. EXAMPLE: 1289.45e7 is a legal constant, equal to 12894500000.

INVALID OR DOUBLE EXPONENT IN NUMERIC STRING "string"

Explanation: A numeric string may contain the character "E" to specify a power of 10 by which the mantissa is to be multiplied. The exponent field must be at the end of the number, and consist of the letter E, followed by an optional exponent sign, followed by one or two digits. EXAMPLE: 23E5 and 34.8E-4 are legal numeric constants, equal to 2300000 and .00348, respectively.

INVALID PTR QUALIFIER FOR NON-BASED IDENTIFIER ident

Explanation: An expression contains a pointer qualifier (POINTER variable followed by the operator "->") but the identifier being qualified does not have the BASED attribute.

This message will also be typed if you use the SET option of the ALLOCATE statement with an identifier which does not have the BASED attribute.

User Response: Add a DECLARE statement to your program, specifiying that the identifier being qualified has the BASED attribute.

INVALID RANGE SPEC IN DEFAULT STATEMENT

Explanation: The RANGE spec in the DEFAULT statement is in the format RANGE(let) or RANGE(let:let). In the second format, the first letter may not appear later in the alphabet than the second letter.

INVALID SIGN "char" IN NUMERIC STRING "string"

Explanation: In CHARACTER to FIXED or FLOAT conversion, a sign may appear at the beginning of the numeric string or it may appear after the letter E to specify the sign of the exponent in E-type FLOAT constants.

INVALID TRANSFER INTO ITERATIVE DO GROUP

Explanation: A GOTO statement may not make a transfer from outside a DO group into the DO group.

Note: There is one exception to this rule. If the DO statement is non-iterative (just "DO;"), then it is legal to transfer into it.

INVALID USE OF ARRAY ident

Explanation: An array may not be used in a CPL expression. This is a restriction in CPL.

User Response: You can usually replace an array expression with a DO loop. For example, the statements

```
DECLARE A(10);
A = A + 1;
```

can be replaced with

```
          DECLARE A(10);
          DO I = 1 TO 10;
          A(I) = A(I) + 1;
          END;
```

INVALID ZERO FIELD IN PPN IN FILEID "string"

Explanation: (TOPS-10 only) The project-programmer number field appearing in a file-specification in the TITLE option of the OPEN statement or in a WEAVE or LOAD or SAVE statement is illegal. The ppn field must have the format [proj#,prog#], where the two numbers are non-zero octal numbers.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

KEYCKI -- INVALID KEYWORD ARGUMENT keyword

Explanation: System error. The keyword code passed to the KEYCKI routine is invalid.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

KEYCKK -- TOO MANY KEYWORDS

Explanation: System error. Too may keywords were passed to KEYCKI.

Implementation Note: The KEYCKI routine is called to check for duplicate or conflicting keywords in several statements such as GET, PUT, READ, WRITE, OPEN and CLOSE. It is also used to check for duplicate or conflicting attributes for each identifier declared in a DECLARE statement. And it is used to check for conflicting attributes during the OPEN attribute merge.

The table allocated to hold the keywords contains 50 entries. If more than 50 keywords are passed to KEYCKK then the table will overflow. If necessary, the table size can be increased.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

KEYFNT -- INVALID ARGUMENT

Explanation: System error. An invalid built-in function code was passed to KEYFNT.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

LABELS PERMITTED ONLY FOR COLLECT STATEMENTS -- LINE NUMBER REQUIRED

Explanation: You have used a statement label on a statement which has no line number. Labels are not permitted in direct mode.

LABEL MAY NOT PRECEDE "ELSE" IN SAME STATEMENT

Explanation: You have entered a statement in the form "label: ELSE ...;". The ELSE keyword should precede the label.

LINDFI -- INVALID DEFAULT STATEMENT

Explanation: System error. A DEFAULT statement SMB is invalid.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

LINDLB -- INVALID LABEL SMB

Explanation: System error. A LABEL SMB has an illegal format.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

LINDPA -- INVALID VARIABLE IN DECLARATION

Explanation: System error. LINDPA has detected a variable in a DECLARE statement which does not lie inside an extent expression.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

LINDPF -- DON'T RECOGNIZE KEYWORD "keyword"

Explanation: A keyword has appeared in a DECLARE statement which the link routines do not recognized. This can happen if you use a partially implemented feature which is not described in the CPL documentation.

User Response: If you are using an undocumented feature then stop using it.

If you are using only documented features, then this is a system error. Please save all relevant output and mail it to Digital Equipment Corporation.

LINE NUMBER string EXCEEDS 9999.99

Explanation: CPL line numbers must be in the range 1.00 to 9999.99.

LINE NUMBER string HAS MORE THAN 2 FRACTIONAL DIGITS

Explanation: CPL line numbers must be in the range 1.00 to 9999.99. No more than two digits may appear after the decimal point.

LINE NUMBER string IS ZERO

Explanation: CPL line numbers must be in the range 1.00 to 9999.99. If the line number appears in the BY clause of the NUMBER statement or the NUMBER option of the LOAD or WEAVE statement then it must be in the range 0.01 thru 9999.99.

LINE NUMBER string MAY NOT BE LESS THAN 1

Explanation: CPL line numbers must be in the range 1.00 through 9999.99.

LINE NUMBER NOT FOUND

Explanation: There is no statement in your program with the line number specified in the ERASE statement.

LINFDC -- CAN'T FIND DCB FOR DECLARATION OF ident

Explanation: System error. CPL cannot erase an explicit declaration because it cannot find the declaration block (DCB) for the explicit declaration being erase.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

LINGDO -- REACHED CCOEOX IN DECLARATION

Explanation: System error. LINGDO has reached the CCOEOX operator in the DECLARE SMB.

Implementation Note: Any routine using LINGDO to defactor a DECLARE statement must stop processing at the CKDECLARE operator. This operator appears in the DECLARE SMB just before the CCOEOX operator.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

LINULB -- CAN'T FIND DCB FOR LABEL SMB

Explanation: System error. CPL is unable to erase a LABEL declaration since it cannot find the declaration block (DCB) for the LABEL declaration.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

LINULB -- INVALID LABEL BLOCK

Explanation: System error. The SMB for a statement label contains an illegal operator.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

MISSING PTR QUALIFIER FOR BASED IDENT ident

Explanation: The specified identifier has the BASED attribute, but there is no POINTER qualifier (POINTER value followed by "->" operator) for it.

This message will also appear if you execute an ALLOCATE statement, specifying a BASED identifier, but there is no SET option in the statement.

User Response: Respecify your statement so that all BASED variables have POINTER qualifiers.

MISSPS -- INVALID POSITION TO SET IN SAB

Explanation: System error. An illegal argument has been specified in either a ".SAB SET,POSITION" or ".SAB ADJUST" operation. The position specified must lie between 0 and the value in the LENGTH field of the SAB.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

MORE THAN 4 DIGITS IN LINE NUMBER string

Explanation: A CPL line number must lie in the range 1.00 to 9999.99. Only four digits may appear before the decimal point and two digits may appear after.

NEGATIVE # OF DIGITS IN E FORMAT ITEM

Explanation: The second argument to an E format item is negative. This argument may be 0 if you want no digits to follow the decimal point in the output, but it may not be negative.

NEGATIVE # OF DIGITS IN F FORMAT ITEM

Explanation: The second argument to an F format item is negative. This argument may be 0 if you want no digits to follow the decimal point in the output, but it may not be negative.

NEGATIVE SKIP OPTION IS ILLEGAL

Explanation: The argument of the SKIP option of the PUT or GET statement, or the SKIP format item of the PUT EDIT statement may not be negative.

NO DEVICE NAME PRECEDES COLON IN FILEID "string"

Explanation: (TOPS-10 only) In a file-specification appearing in the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statements, contains a colon with no device name preceding.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

NO DIGITS IN EXPONENT OF STRING "string"

Explanation: In a CHARACTER to FIXED or FLOAT conversion, the exponent of an E-type FLOAT constant contains no digits. EXAMPLE: 63.27E3 is a legal E-type constant equal to 63270.

NO DIGITS IN MANTISSA OF STRING "string"

Explanation: In a CHARACTER to FIXED or FLOAT conversion, the mantissa portion of the numeric string contains no digits.

NO DIGITS IN STRING "string"

Explanation: In a CHARACTER to FIXED or FLOAT conversion, the numeric string contains no digits.

NO FILENAME PRECEDES DOT IN FILEID "string"

Explanation: (TOPS-10 only) In the file-specification given by the TITLE option of an OPEN statement or in the LOAD, WEAVE or SAVE statement contains a dot with no filename preceding.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

NO MONITOR TABLE SPACE TO OPEN FILE filespec

Explanation: (TOPS-10 only) The TOPS-10 monitor is unable to open a file due to lack of table space.

User Response: Please report this problem to your system analyst.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

NO ROOM ON STRUCTURE OR QUOTA EXCEEDED FOR FILE filespec

Explanation: (TOPS-10 only) An ENTER UUO failed because either

1. There is no more room on the structure.

2. You have used up your personal quota on the output structure.

User Response: If you wish to create a new output file, then you must return to monitor level and delete some files

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

NO UFD FOR FILE filespec

Explanation: (TOPS-10 only) CPL was unable to open a file on the specified disk device because there is no User File Directory (UFD) for the ppn on that device.

User Response: If you have been assigned a quota on the specified device, and if you wish to create a file on that device, then return to monitor level and issue a TOPS-10 MOUNT command. This command will create a UFD for your ppn on that device.

If you have not been assigned a quota on the desired device, then contact your system administrator.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

NON-POSITIVE FIELD WIDTH FOR E OR F FORMAT ITEM

Explanation: The first argument to the E or F format item must not be negative or zero.

NORMAL TERMINATION OF keyword ON-UNIT

Explanation: Your program has executed the END statement of the specified ON-unit. This is usually an error.

User Response: You should change your ON-unit so that a GOTO statement will be executed out of the ON-unit. This "abnormal termination" of the ON-unit will permit your program to recover from the error.

ONLY ONE FIELD IN PPN IN FILEID "string"

Explanation: (TOPS-10 only) The project-programmer number field of a file-specification appearing in the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statement should be in the format [proj#,prog#], where the two numbers are octal.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

OUTPUT HAS EXCEEDED LENGTH OF STRING VBLE ident

Explanation: In a PUT STRING operator, the output has exceeded the length of the string variable specified in the STRING option.

System Action: All PUT processing stops. No change is made to the value in the STRING option variable. The value of this variable is exactly what it was before the PUT operator began.

User Response: There are several ways you can handle this problem:

1. Specify fewer items in the output list.

2. Change the declaration of the STRING variable so that it will be long enough to accomodate all the output.

3. Use PUT EDIT with the STRING option so that you can control the format of the output. You may be able to use less space in the output string by specify short field width's in the formats.

PAGESIZE OPTION REQUIRES A "PRINT" FILE

Explanation: Self-explanatory.

PARM ident DECLARED WITH INITIAL ATTRIBUTE

Explanation: Self-explanatory.

PARM ident DECLARED WITH VBLE EXTENT EXPRESSIONS

Explanation: The separate declaration for a parameter appearing in a PROCEDURE statement may not contain variables in the string lengths or in the array bounds.

User Response: Instead of using variables in the extent expressions, code an asterisk. The corresponding string length or array bound pair will be taken from the argument when the PROCEDURE is called.

PARPAT -- ILLEGAL INDEX INTO REDUCTION TABLE

Explanation: System error. An illegal argument was passed to PARPAT.

User Response: Save all relevant output and send to Digital Equipment Corporation.

PARPPT -- ILLEGAL INDEX INTO PARSER TABLE

Explanation: System error. An illegal argument was passed to PARPPT.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

PARSE ERROR -- CAN'T RECOGNIZE "string"

Explanation: CPL does not fully recognize the statement which has just been entered.

The "parse" phase of CPL is that phase which recognizes the statement which you have typed in and which translates that statement into an internal CPL format.

CPL examines the statement from left to right. If it finds an identifier, keyword, constant or operator which does not belong where it was found, then CPL types the above error message. The invalid identifier, keyword, constant or operator is typed out between the quotes.

If the invalid string is a ";", then the statement ended unexpectedly.

System Action: CPL ignores the entire line which contained the erroneous element. Even if there are several statements on the line and only one of them is incorrect, the whole line is ignored.

User Response: Correct the error and retype the line.

PARSEM -- ILLEGAL PARSER SEMANTIC CODE

Explanation: System error. An invalid argument was passed to the PARSEM routine.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

PARSER STACK OVERFLOW

Explanation: System error. The parser stack overflowed.

Implementation Note: The parser stack contains one entry for each token which has not yet been reduced. The current CPL grammar should not require more than about 10 stack entries.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

PARSER STACK UNDERFLOW

Explanation: System error. The parser stack has underflowed.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

POINTER FOR ident DOES NOT POINT TO BEGINNING OF DATA BLOCK FOR ident2

Explanation: You have attempted to execute a FREE statement for a BASED identifier (the first identifier appearing in the message), but the POINTER variable qualifying the BASED variable does not point to the beginning of a storage area allocated by an ALLOCATE statement. There is a CPL restriction that a FREE statement may free only an entire data block; a partial data block may not be freed.

The second identifier (ident2) appearing in the error message is for your information. It specifies the name of the BASED identifier which appeared in the ALLOCATE statement which allocated the storage to which the POINTER qualifier points.

User Response: You must change your program so that it releases only entire data blocks. Furthermore, your program must maintain a copy of the pointer which was returned by the SET option of the ALLOCATE statement which allocated the BASED storage, so that you can free the storage later.

POINTER FOR ident DOES NOT POINT TO POINTER STORAGE AREA

Explanation: The specified identifier is a POINTER with the BASED storage class attribute. It is qualified by another POINTER which points to a storage area for a data type other than POINTER.

Normally CPL permits you to qualify a BASED variable with a POINTER pointing to any data area, whether that data area was allocated with the same data type as the BASED identifier or not. But in the case of POINTER BASED identifiers, the data area must be for a POINTER.

User Response: Set the qualifying POINTER variable to the ADDR of an actual POINTER.

POINTER FOR ident HAS INVALID BIT ALIGNMENT

Explanation: The specified BASED identifier has a POINTER qualifier which has an invalid bit displacement for the data type of the BASED identifier. When a POINTER is assigned the ADDR (address) of a BIT or CHARACTER data element which is not left-justified in the 36-bit DECsystem-10/20 word, then the POINTER will have a non-zero bit alignment. Now if such a POINTER is used to qualify a BASED identifier, then the data type of the BASED identifier must permit the bit alignment of the POINTER.

If the BASED identifier is FIXED, FLOAT, CHARACTER VARYING or BIT VARYING, then the POINTER must have a bit displacement of zero (the POINTER value must be word-aligned). If the BASED identifier is CHARACTER NONVARYING, then the POINTER must have a bit displacement of 0, 7, 14, 21 or 28. If the BASED identifier is BIT NONVARYING, then the POINTER may have any bit displacement (0, 1, 2, ..., 35).

User Response: This error message almost always indicates a programming error. Usually you will find that you have not initialized the POINTER variable to the address you had thought.

However, it may be that you are attempting to do some machine-dependent arithmetic, such as to treat a CHARACTER string (in internal format) as an integer. If that is the case, you'll have to do it in such a way that your POINTER has the proper alignment. For example, you might copy the character string to another CHARACTER variable and use the ADDR of that variable as your POINTER value.

POINTER FOR ident IS NULL OR INVALID

Explanation: The specified BASED identifier is qualified by a null or invalid POINTER variable. A POINTER variable is null or invalid in the following circumstances:

1. When a POINTER is first allocated, it is set to a null value.
2. When a POINTER is assigned the value of the NULL built-in function, then the POINTER is given a null value.
3. When a storage area is freed for any reason, then all POINTER variables which point to an address in that storage area become invalid. A storage area may be freed for any of the following reasons:
    1. If a program block is terminated, then storage for all AUTOMATIC variables DECLAREd in the block is freed.
    2. A FREE statement can free a storage block for CONTROLLED or BASED storage.
    3. If you erase or replace a DECLARE statement in your program, then all data blocks allocated for identifiers DECLAREd in that statement are freed.
    4. If you type an XEQ statement, then CPL automatically frees all BASED and CONTROLLED is automatically freed, and all POINTERs pointing to such storage are automatically made invalid.

User Response: Usually this error message indicates that you have not intialized your POINTER variable. In that case, you should use the ADDR built-in function to set the value of the POINTER.

If the problem has arisen because you typed the XEQ command, then you should use the CONTINUE statement when you wish to continue execution.

POINTER FOR ident POINTS TO NON-BASED STORAGE FOR ident2

Explanation: A FREE statement is attempting to free the specified BASED identifier, but the POINTER value qualifying the BASED identifier does not point to storage which was allocated by an ALLOCATE statement for BASED storage.

The second identifier ("ident2") appearing in the error message is given for your information. It specifies the name of the identifier for which the storage area to which the POINTER value points was allocated.

User Response: This message usually indicates a simple programming error. Probably the POINTER value qualifying the BASED identifier was not properly initialized.

POINTER FOR ident POINTS TO STORAGE BLOCK SMALLER THAN ident2'S (integer WORDS)

Explanation: A FREE statement for BASED storage is attempted to release a partial block of storage. CPL requires that you free the entire data block. In this case, the POINTER value qualifying the BASED identifier in the FREE statement points to a storage block which is larger than the storage area indicated by the declaration for the BASED identifier specified in the FREE statement.

The integer given in the error message is for your information. It gives the actual size (in words) of the data block which you are attempting to free.

POINTER POINTS TO STORAGE AREA TOO SMALL FOR ident

Explanation: The specified BASED identifier is qualified by a POINTER which points to a storage area which is too small for the BASED identifier. It is illegal in CPL to reference any data beyond the end of a data block.

User Response: Occasionally this error situation occurs in what might be a valid data reference. For example, perhaps the BASED identifier is an array and you are referencing the first element of the array. The reference will be illegal if the data block is too small for the entire array. If this type of error occurs, then you will have to DECLARE an additional BASED identifier with smaller array bounds, and use that identifier instead.

If you require some flexibility in the size of the the based identifier, then you can use variables in the extent expressions for the declaration of the BASED variable. CPL will evaluate these extent expressions anew each time the BASED identifier is referenced.

POINTER VBLE DOES NOT HAVE POINTER DATA TYPE

Explanation: Your expression contains a reference preceding the operator "->" which does not have the POINTER data type.

User Response: If the reference is to a variable, make sure that it has been DECLAREd to have the POINTER data type.

If the reference is a function PROCEDURE reference, make sure that the PROCEDURE statement contains a RETURNS option specifying the POINTER data type.

PROC ident REQUIRES integer ARGUMENTS

Explanation: You have attempted to invoke a PROCEDURE, but the number of arguments in the reference does not equal the number of parameters in the PROCEDURE statement.

PROC AND ENTRY STATEMENTS MUST BE LABELED

Explanation: The PROCEDURE statement must have a label.

PROC OR ENTRY STMTS MAY NOT BE "THEN" OR "ELSE" CLAUSES

Explanation: A PROCEDURE statement may not be the THEN or ELSE clause of an IF statement.

PROGRAM CONTAINS NO STATEMENTS

Explanation: You have specified a line number in a LIST or ERASE statement, but your program contains no statements.

REMOTE FORMAT ident IS NOT IN SAME BLOCK

Explanation: You have specified an R format item in a PUT EDIT statement, but the specified FORMAT statement does not lie in the same block as the PUT EDIT statement.

User Response: Move the FORMAT statement into the same block as the PUT EDIT statement.

REMOTE TARGET ident IS NOT A FORMAT

Explanation: You have specified an R format item in a PUT EDIT statement, but the argument is not the label of a FORMAT statement.

Note also that you may not reference a FORMAT statement which is in a different block from the PUT EDIT statement.

RESTRICTION -- DFT STATEMENT MUST BE IN OUTER BLOCK

Explanation: A DEFAULT statement may not be inside a BEGIN/END block or a PROCEDURE/END block.

User Response: Move all your DEFAULT statements to the outer block. It is considered good programming practice to put all your DEFAULT statements at the beginning of your program.

S LESS THAN D IN E FORMAT ITEM

Explanation: An E format item appearing in a PUT EDIT statement has a third argument which is smaller than the second argument.

If the arguments to the E format item are w, d and s, then "d" stands for the number of digits to appear after the decimal point, and "s" stands for the number of significant digits to be printed. The value of s must be at least as large as the value of d.

SCALAR ARG ident MAY NOT BE PASSED AS DIMENSIONED PARM ident

Explanation: You have attempted to invoke a PROCEDURE, specifying a scalar argument where the corresponding PROCEDURE parameter is an array.

SCALAR EXPRESSION ARGUMENT MAY NOT BE PASSED AS DIMENSIONED PARM ident

Explanation: You have attempted to invoke a PROCEDURE, specifying an expression where the corresponding parameter is an array.

SEARCH LIST EMPTY -- CAN'T OPEN filespec

Explanation: (TOPS-10 only) You have attempted to open a file using the default device DSK:, but your search list is empty.

User Response: Either specify a specify disk in the TITLE option of the OPEN statement, or else return to monitor level and use the SETSRC system program to create a search list for your job.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

SEFBFR -- STACK ERROR FOR PSEUDO-VARIABLE

Explanation: System error. After semantic processing of a pseudo-variable reference had been completed, the semantic stack still had some entries in it.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEFTAB -- ARGUMENT DOES NOT HAVE A TABLE VALUE

Explanation: System error. An invalid argument was passed to the SEFTAB routine.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

SEFTAB -- ARGUMENT IS OUT OF RANGE

Explanation: System error. An invalid argument was passed to the SEFTAB routine.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

SEMDUP -- STACK ERROR

Explanation: System error. After handling the CCODUP operator, the stack did not have 2 entries on it.

SEML1 -- ILLEGAL VALUE OF AON OR AOFF

Explanation: System error. The ON and OFF values returned by SEMTAB for the current operator.

Implementation Note: The OFF value is the number of stack entries to be removed by the current operator. The ON value is the number of stack entries to be added. The SEML1 routine can handle values of ON less than or equal to 3 and values of OFF less than or equal to 1.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEML1 -- STACK UNDERFLOW

Explanation: System error. The semantic stack has underflowed.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEMLUP -- CAN'T HANDLE SMB OPERATOR

Explanation: The SMB contains an operator which the semantic routines cannot handle. This will happen if you attempt to use a partially implemented feature which is not descripbed in the documentation.

User Response: If you have used an undocumented feature, then you should remove that use from your program.

If you are using only document features, then this message indicates a system error. Please save all relevant output and send it to Digital Equipment Corporation.

SEMLUP -- INVALID ACTION CODE

Explanation: System error. The action code returned by SEMTAB was invalid.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEMLV1 -- STACK ERROR

Explanation: System error. After the CCOLVI operator had been processed in an assignment statement SMB, the stack was found to contain additional entries.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEMSS1 -- CAN'T HANDLE ARRAY OPERATOR

Explanation: System error. The SMB operator following the CCOSS operator and the subscript count is not recognized by the SEMSS1 routine.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SEMSS1 -- STACK UNDERFLOW

Explanation: System error. The subscript count found after the CCOSS operator in an SMB is larger than the number of items on the semantic stack.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

STACK OVERFLOW -- SIMPLIFY STATEMENT

Explanation: In attempting to execute a statement, the CPL semantic routines found an expression to be so complex that its internal "stack" overflowed.

The "complexity" of an expression has nothing to do with its length, but rather with the number of intermediate results which must be saved in order to evaluate it. Thus, for example, an expression containing a large depth of nested parentheses may cause this message to appear.

User Response: If the expression cannot be simplified, then the computation should be broken up into several separate statements.

STATEMENT HAS NO LINE NUMBER

Explanation: In loading a program with the LOAD or WEAVE statement with no NUMBER option, a line was found with no line number.

System Action: CPL ignores the unnumbered line and continues loading.

User Response: You must either edit the file so that all lines have line numbers, or you should specify the NUMBER option of the LOAD or WEAVE statement.

STATEMENT HAS OVERFLOWED STATEMENT BLOCK

Explanation: Your statement is too long to fit into an internal statement block.

Implementation Note: CPL "decomposes" statements into an internal form and stores the coded form in a "statement block." If your statement is too long, then the decomposed form will not fit into a default size statement block.

If this error gets to be a problem, it is possible for a system analyst to modify the internal CPL system to allow bigger statements.

User Response: Simplify the long statement. If you cannot simplify it, then try to split it up into several statements by assigning complex intermediate values to variables and then using the variables in the long statement.

STATEMENT NUMBER NOT FOUND

Explanation: The statement number specified in a direct statement (such as BREAK, NOBREAK, XEQ, etc.) cannot be found.

STATEMENT TOO LONG

Explanation: The statement text is too long to fit into a "statement text block."

Implementation Note: CPL stores the statement text in a statement text block with room for over 400 characters. If necessary, a system analyst can modify the internal CPL system to provide a larger statement text block.

User Response: Simplfy the long statement. If you canot simplify it, then try to split it up into several statements.

STATEMENT TYPE IS COLLECT-ONLY -- ENTER LINE NUMBER

Explanation: Certain CPL statements (such as SIGNAL and BEGIN) may not be executed in "desk-calculator" mode, but must have a line number.

User Response: Reenter the statement, specifying a line number.

STATEMENT TYPE IS DIRECT-ONLY -- NO LINE NUMBER ALLOWED

Explanation: The statement may not have a line number.

STRING ident IS NOT ALLOCATED

Explanation: Storage has not been allocated for the specified variable. For more information, refer to the message "ident" IS NOT ALLOCATED.

STRING LENGTH FOR ident MUST BE POSITIVE

Explanation: The string length expression specified as the argument to the CHARACTER or BIT attribute in a DECLARE statement is negative or zero.

STRING VARIABLE ident IS A NAMED CONSTANT

Explanation: The specified identifier is a NAMED CONSTANT (e.g., a FILE or LABEL).

STRING VARIABLE ident IS NOT A SCALAR

Explanation: The variable specified in the STRING option of a GET or PUT statement has been declared to be an array.

STRING VARIABLE ident IS NOT CHARACTER TYPE

Explanation: The variable specified in the STRING option of a GET or PUT statement is not the CHARACTER data type.

STRINGSIZE [FOR ident]

Explanation: One of the following has occurred:

1. In an assignment statement, the CHARACTER or BIT expression computed on the right hand size was longer than the length or maximum length of the string variable on the left hand side of the equal sign.

2. The CHARACTER or BIT expression specified in a RETURN statement is longer than the length specified with the CHARACTER or BIT attribute in the RETURNS option of the PROCEDURE statement.

System Action: After typing out the warning message, CPL continues executing the statement. No error exit is taken.

User Response: If you do not want this message to appear in your program, you may do the following:

1. For an assignment statement, change the declaration of the variable on the left-hand side of the statement so that it specifies a longer string.

2. For either an assignment statement or a RETURN statement, use the SUBSTR built-in function to remove the last few bits or characters from the CHARACTER or BIT string being computed.

STRUCTURE OR DTA IS FULL OR QUOTA EXCEEDED FOR FILE filespec -- FILE STATUS code

Explanation: (TOPS-10 only) Block too large. One of the following has occurred:

1. A block of data from a device is too large to fit in a buffer

2. A block is too large for the unit.

3. The file structure (DSK) or unit (DTA) has filled.

4. The user's quota on the file structure has been exceeded.

User Response: In the first two cases above, report the problem to your system analyst.

In the last two cases, you must delete some files before you can continue writing files on the file structure or DECtape. You do this by returning to monitor level and using the DELETE command.

If the problem is a full disk device, then you may petition your system administrator to increase your quota.

SUB-STATEMENT NUMBER DOES NOT EXIST

Explanation: In an XEQ, CONTINUE, BREAK or NOBREAK statement, the statement number specified does not exist, because the sub-statement number (the number specified after the "+" sign in the statement number specification) does not exist on the line.

SYSDEL -- ERROR IN DISMS JSYS

Explanation: (TOPS-20 only) System error. Your program has attempted to execute a DELAY statement, but an error occurred in the DISMS JSYS.

User Response: Save all relevant output and mail to Digital Equipment Corporation.

SYSDEL -- ERROR IN HIBER UUO

Explanation: (TOPS-10 only) The HIBER UUO took an error return. The HIBER UUO is used in the DELAY statement.

SYSTEM TABLE JOBTTY NOT FOUND

Explanation: (TOPS-20 only) CPL system error.

User Response: CPL system error. Save all relevant output and mail to Digital Equipment Corporation.

THE FIRST ARG OF FCN ident MUST BE A SIMPLE ID

Explanation: The first argument of the specified built-in function or pseudo-variable must be a simple identifier. It may not be an expression, nor may it be a subscripted identifier. For example, the following functions have such a requirement: ALLOCATION, DIMENSION, LBOUND, HBOUND, STRING and UNSPEC.

TOO MANY FIELDS IN PPN IN FILEID "string"

Explanation: (TOPS-10 only) The project-programmer number field of the file-specification in the TITLE option of the OPEN statement or in the LOAD, SAVE or WEAVE statements has more than two fields in it.

The format of the ppn field is [proj#,prog#], where the two numbers are octal. No more than two fields may be specified.

Note: CPL does not support SFDs (sub-file directories).

TTY UNIT NUMBER NOT FOUND IN SYSTEM TABLE JOBTTY

Explanation: (TOPS-20 only) CPL system error.

User Response: CPL system error. Save al relevant output and mail to Digital Equipment Corporation.

TWO DECIMAL POINTS IN NUMBER

Explanation: A numeric string may contain only a single decimal point, and it must appear before the E in an E-type FLOAT constant. EXAMPLE: 1289.45E7 is a legal constant, equal to 12894500000.

TWO EXPONENTS IN NUMBER

Explanation: A numeric string may contain the character "E" to specify a power of 10 by which the mantissa is to be multiplied. The exponent field must be at the end of the number, and consist of the letter E, followed by an optional exponent sign, followed by one or two digits. EXAMPLE: 23E5 and 34.8E-4 are legal numeric constants, equal to 2300000 and .00348, respectively.

UNBALANCED LEFT PARENTHESIS

Explanation: The statement contains an open (or left) parenthesis for which there is no matching close (or right) parenthesis.

UNBALANCED RIGHT PARENTHESIS

Explanation: The statement contains a close (or right) parenthesis for which there is no matching left (or open) parenthesis.

UNMATCHED "END" STATEMENT

Explanation: Your program contains an END statement for which there is no matching DO, PROCEDURE or BEGIN statement.

Note: An implementation restriction makes it impossible to execute many direct-mode statements while such an error exists in your program.

UNMATCHED DO PROC OR BEGIN STATEMENT

Explanation: You program contains a DO, PROCEDURE or BEGIN statement for which there is no matching END statement.

Note: An implementation restriction makes it impossible to execute many direct-mode statements while such an error exists in your program.

UNRECOGNIZED LOOKUP/ENTER CODE integer FOR FILE filespec

Explanation: (TOPS-10 only) System error. A LOOKUP or ENTER UUO failed with an unrecognized error code.

User Response: Save all relevant output and mail it to Digital Equipment Corporation.

If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

UNTERMINATED CHARACTER STRING

Explanation: You have entered a statement which contains an opening quote (') with no matching closing quote.

UNTERMINATED COMMENT

Explanation: You have entered a statement which contains an opening of a comment (/*) but there is no close (*/).

VARIABLE EXPRESSION IN STATIC DECLARATION OF ident

Explanation: A variable with the STATIC storage class may not have variable extent expressions. All expressions appearing in dimension bounds or string lengths must contain only constants.

WRITE-LOCK ERROR ATTEMPTING TO WRITE ON filespec

Explanation: (TOPS-10 only) An ENTER UUO failed because of a write-lock on the specified disk file structure.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

WRITE-LOCK OR NO-CREATE FOR ALL STRUCTURES IN SEARCH LIST FOR FILE filespec

Explanation: (TOPS-10 only) An ENTER UUO failed in an attempt to open an OUTPUT file because all structures in your search list have the no-create or write-lock bit set.

User Response: If you wish execution to continue when this error occurs, then you may use the UNDEFINEDFILE(filename) ON-condition to specify what action CPL should take.

WRONG NUMBER ARGUMENTS TO FUNCTION ident

Explanation: The specified built-in function or pseudo-variable has the wrong number of arguments.

WRONG NUMBER OF SUBSCRIPTS FOR DIMENSIONED VARIABLE

Explanation: The number of subscripts specified for an array reference is different from the number of dimensions specified in the DECLARE statement.

XEQ "STOP"

Explanation: Your program has executed a STOP statement.

CHAPTER 26

QUESTIONS AND ANSWERS ABOUT CPL (R)

This chapter answers some of the questions most often asked about CPL.

26.1 <u>IS CPL THE SAME AS PL/I?</u>

CPL implements a subset of the ANSI PL/I language standard.

The full PL/I language is enormous. Only two vendors have implemented the full language, and both of those implementations required hundreds of man-years of effort. CPL makes no claim to have matched that level of development.

On the other hand, the CPL subset is a very powerful programming language. All the PL/I program control constructs (IF, DO, PROCEDUREs, and ON for error handling) have been implemented. There are six computational data types (FIXED, FLOAT, CHARACTER, CHARACTER VARYING, BIT, BIT VARYING) and one non-computational variable data type (POINTER). There are all four PL/I storage classes (AUTOMATIC, STATIC, CONTROLLED and BASED). And almost all PL/I statements are implemented or partially implemented. Read the chapter entitled "Comparison of CPL with ANSI PL/I Standard."

26.2 <u>WHAT IS AN INTERPRETER?</u>

A "raw" computer cannot handle a complex language like PL/I. A computer can "think" only in the sense of executing very simple instructions (such as "add these two numbers and store the result there"). A complex programming language like PL/I, with its IF and DO statements, its PROCEDUREs and its input/output, is much too complex to be handled directly by the computer. The computer requires some intermediate program, a "translator" of some sort, which will make the complex statements of the language accessible to the simple computer. The statement "I=J+1;" means nothing to a computer unless some intermediate program tells the computer where to find I and J, that "+" means add, and that "=" means "store the result in."

The most commonly used intermediate program of this type is called a "compiler." A compiler is a program which translates a high-level computer language (like PL/I) into a actual computer instructions. The symbols appearing in the high-level language program are turned into simple individual instructions at the hands of the compiler. FORTRAN-10 is an example of a compiler of the FORTRAN language on the DECsystem-10.

When you use a compiler, you usually have to go through several steps: First, the compiler translates the program into an "object" format, then this "object" format must be loaded into core storage, and finally the computer is told to executed the loaded program.

An "interpreter" works somewhat differently. It is not a "translator" in the same sense. A true interpreter executes the program directly from the high-level source statements. It takes a statement like "I=J+1;" and it "interprets" the statement by causing it to be executed immediately. If the statement is in a loop, then a true interpreter will re-interpret the source each time that it is executed.

CPL is actually a "semi-interpreter." CPL maintains the statement in a form known as "reverse Polish notation." This notation represents a source statement in a compact internal form which can be executed quickly. This translation to internal form is done in two stages:

1. When you type the statement in, CPL "parses" the statement into a form of this internal notation which does not take into account the data types of the variables appearing in the expression.

2. The first time that the statement is executed, the internal format is translated further into a form which takes into account the data types of the variables. This translation will not have to be repeated unless you change some declarations in your program.

CPL executes the statement by interpreting this final internal form of the statement. This method is not as fast as having the computer execute the translated instructions, but it is not nearly as slow as executing directly from the source code each time.

26.3 <u>WHAT ARE THE ADVANTAGES OF AN INTERPRETER?</u>

There are a number of advantages to an interpreter which offset the disadvantage of slower execution speed.

CPL provides true source-level debugging for PL/I programs. When CPL executes a program, it checks for a number of execution errors which are not checked by language systems with a compiler. An example is the validity of POINTER variables; CPL guarantees that a POINTER points to only a valid storage area, while a compiler system may permit a program to wipe out important data in storage inadvertently.

When CPL discovers an error in execution, a detailed error message is typed out. The user then has several options. He can find out the values of variables to find out where his program went wrong. He can modify program statements or insert additional statements, save his modifications on disk, and then continue execution of his program without having to start from the beginning. Source level breakpoints are also available.

26.4 WHO CAN USE CPL?

The high level of execution support make CPL uniquely suited for certain types of users.

26.4.1 Beginning Or Infrequent Programmers

A person who needs to write an occasional program to get an answer will find CPL a useful tool. As the first chapter of the User's Manual shows, the beginner with no previous experience in using computers can begin getting answers as soon as he begins using CPL. The first chapter will show him how to write simple programs quickly.

26.4.2 PL/I Program Developers

Users who are developing production PL/I programs to be run on PL/I compilers on other computers can use CPL as a program checkout tool.

26.4.3 Students Learning Programming

The advanced features in the CPL subset, particularly the block structuring and BASED storage, make it a useful educational tool. In addition, the thorough error-checking features of the CPL interpreter aid the learning process.

26.5 HOW WAS CPL WRITTEN?

CPL was written as a programming experiment to prove that modern programming techniques could be used to substantially reduce software development costs and improve programming quality and reliability. CPL was written by one person (John Xenakis) in an eight month period, during which he also wrote the program documentation and the quality assurance test system.

26.6 HOW DOES CPL WORK INTERNALLY?

A module called DRIVER invokes the various CPL phases. These phases are as follows:

1. ACCEPT -- DRIVER calls this phase to input the statement from the terminal (or the LOAD file) and break up the statement into statement tokens.

2. PARSE -- this phase parses the source text into reverse Polish notation, an internal format which represents the statement in compact form.

3. LINK -- DRIVER calls this phase if the statement is collect. This phase causes the collect statement to be linked in with the other statements in the proper position. This phase also processes DECLARE statements and other declarative statements.

4. EXECUTION -- DRIVER calls this phase if the statement is direct. This phase executes the statement and returns, unless the direct statement causes other statements to be executed (as an EXECUTE statement would).

5. SEMANTIC RECOMPOSITION -- the EXECUTION phase calls this phase the first time that a statement is executed. This phase modifies the reverse Polish notation to take into account the data types of the variables in the statement.

26.7 WHY IS THE LIST COMMAND IMPLEMENTED AS IT IS?

When you type a statement into CPL, CPL removes all leading blanks and tabs. When typing it out as a result of the LIST command, CPL indents two tabs if the statement is unlabeled, and one tab if the statement is labeled.

This format was chosen after many discussions with a number of people. It is the only one which satisfied a number of constraints:

1. The listing should not look sloppy if the statements were entered sloppily.

2. The output format for the LIST statement must be the same as the output format for the SAVE statement.

3. If statements are typed in, the LIST format should be identical to the format produced if the same statements were read from a file as the result of a LOAD or WEAVE statement. The same must be true of statements typed in with the NUMBER statement and read with the NUMBER option of the LOAD or WEAVE statement.

26.8 WHY ISN'T INTEGER DIVISION PERMITTED?

Full ANSI PL/I permits division of FIXED quantities, but specifies it in such a way that it cannot be implemented in any system that does not support scaled data types.

As a compromise, CPL has implemented the non-standard functions DIVI and DIVF.

26.9 WHY IS "OPEN" A NOP IF THE FILE IS ALREADY OPEN?

The ANSI PL/I standard specifies that if the file specified by the FILE option of an OPEN statement has already been opened (and hasn't been closed with a CLOSE statement), then the OPEN statement will be a "no-operation," even if different file attributes are specified.

26.10 WHY AREN'T PL/I STRUCTURES IMPLEMENTED?

Although CPL was designed to permit structures to be implemented in the future, it was not possible to implement that feature in the limited time when CPL was being developed.

26.11 WHY ARE LINE CONTINUATIONS SO AWKWARD?

In implementing a conversational version of PL/I, the following problem arises: Most users will type their programs with precisely one statement per line. They will require more than one line for a statement very rarely.

In ANSI PL/I, you indicate the end of a statement by typing a semicolon character. Therefore, if a line does not end in a semicolon, the language processor knows that the statement runs over onto another line.

But in CPL, we have provided a more logical way: The semicolon will be inserted for you since, in the vast majority of the cases, this is what the user wants. But in the exceptional case where you need several lines for a statement, then you type an additional character(&).

This is not an ideal solution. It is ugly and unpleasant to use. But is is consistent with the conversation nature of CPL, and will serve the needs of most users most of the time.

CHAPTER 27

COMPARISON OF CPL WITH ANSI PL/I STANDARD (R)

27.1 DESCRIPTION OF CPL SUBSET

The following is a fairly complete description of the ANSI features which are and are not supported by CPL.

27.1.1 Statements

1. The ALLOCATE and FREE statements are supported for CONTROLLED and BASED storage, but not for AREAs.
2. The assignment statement is supported, but no aggregate assignments are permitted.
3. The BEGIN statement is fully implemented.
4. The CALL statement with dummy and not-dummy arguments is implemented. So are function PROCEDURE invocations.
5. The CLOSE statement is implemented.
6. The DECLARE statement is implemented, permitting factored attributes and variables in the extent expression fields.
7. The DEFAULT statement is implemented with the following options only: RANGE, FIXED, FLOAT, AUTOMATIC and STATIC.
8. The DELETE statement is not implemented.
9. The DO statement is fully implemented, with the WHILE, BY, TO and REPEAT clauses, multiple specifications, and permitting non-arithmetic control variables.
10. The END statement is implemented, permitting full multiple closure.
11. The ENTRY statement is not implemented.
12. The FORMAT statement is implemented, but must be in the same block as the PUT EDIT statement referencing it.
13. FREE statement -- see ALLOCATE.

14. The GET statement is implemented, with the FILE, STRING, SKIP, and LIST options. The COPY, DATA and EDIT options are not implemented. Aggregates and DOs are not permitted in the input list.

15. The GOTO statement is implemented, permitting termination of a DO-group or a block.

16. The IF/THEN/ELSE statement is completely implemented, along with all interactions with the DO statement.

17. The LOCATE statement is not implemented.

18. The null statement is implemented.

19. The ON statement with the SNAP and SYSTEM options, and permitting arbitrary BEGIN/END block ON-units is implemented. All ON-units are recursive.

20. The OPEN statement is implemented, with the TITLE option and all supported file attributes including ENVIRONMENT. The TAB, LINESIZE and PAGESIZE options are not available for PRINT files.

21. The PROCEDURE statement is implemented, with the RETURNS option. All function and subroutine PROCEDURESs are recursive. Parameters may be declared to have any supported data type.

22. The PUT statement is implemented, with the FILE, STRING, SKIP, PAGE, EDIT and LIST options. The LINE and DATA options are not implemented. Aggregates and DOs are not allowed in the output list.

23. The READ statement with the INTO and IGNORE options is implemented. The SET, KEY and KEYTO options are not implemented.

24. The RETURN statement for subroutine and function PROCEDUREs is implemented.

25. The REVERT statement is implemented.

26. The REWRITE statement is not implemented.

27. The SIGNAL statement is implemented.

28. The STOP statement is implemented.

29. The WRITE statement is implemented. The KEYFROM option is not implemented.

27.1.2 Data Attributes

1. The following data type attributes are implemented: BINARY, BIT, BIT VARYING, CHARACTER, CHARACTER VARYING, ENTRY, FILE, FIXED, FLOAT, FORMAT, LABEL, POINTER, REAL, RETURNS. The following data type attributes are not implemented: COMPLEX,

DECIMAL, ENTRY VARIABLE, FILE VARIABLE, FORMAT VARIABLE, LABEL VARIABLE, OFFSET, PICTURE, GENERIC.

2. The following file attributes are implemented: INPUT, OUTPUT, PRINT, RECORD, SEQUENTIAL, STREAM, ENVIRONMENT. The following file attributes are not implemented: DIRECT, KEYED, UPDATE. CPL supports only ASCII files for INPUT or OUTPUT. No binary files are supported.

3. All four storage classes are supported: STATIC, AUTOMATIC, BASED and CONTROLLED. Also, PARAMETER is supported. EXTERNAL is not supported.

4. Arrays are supported. DEFINED, STRUCTURE, LIKE, POSITION and REFER are not supported.

5. The following additional attributes are not supported: ALIGNED, UNALIGNED, AREA, CONNECTED, INITIAL, REDUCIBLE and IRREDUCIBLE.

6. Arbitrary variables and expressions are permitted in attribute declarations.

27.1.3 Formats

Formats are supported for PUT EDIT only.

1. The following data format items are implemented: A, B, B1, B2, B3, B4, E and F. The C format item is not implemented.

2. The COLUMN, PAGE and SKIP control format items are implemented. The LINE and TAB format items are not.

3. The remote format item (R) is implemented.

4. Repetition factors are permitted.

5. Arbitrary expressions are permitted in format expressions and repetition factors.

27.1.4 ON Conditions

CPL does not support condition prefixes. All supported conditions are always enabled and cannot be disabled.

The following conditions are supported: CONDITION, ENDFILE, ERROR, RECORD, STRINGRANGE, SUBSCRIPTRANGE, UNDEFINEDFILE and ZERODIVIDE.

The following conditions are not supported, although the errors corresponding to these conditions can be processed by the ERROR condition: AREA, CONVERSION, ENDPAGE, FINISH, FIXEDOVERFLOW, KEY, NAME, OVERFLOW, SIZE, STRINGSIZE, STORAGE, TRANSMIT and UNDERFLOW.

27.1.5 Built-in Functions And Pseudo-variables

There are several non-ANSI built-in functions and pseudo-variables implemented. Refer to the "compatibility" section later in this chapter for a list of those.

1. The following ANSI arithmetic functions are implemented: ABS, CEIL, FLOOR, MAX, MIN, MOD, SIGN, TRUNC. The following are not implemented: ADD, BINARY, COMPLEX, CONJG, DECIMAL, DIVIDE, FIXED, FLOAT, IMAG, MULTIPLY, PRECISION, REAL, ROUND.

2. All the ANSI mathematical functions are implemented: ATAN, ATAND, COS, COSD, EXP, LOG, LOG10, LOG2, SIN, SIND, SQRT.

3. The following string-handling functions are implemented: AFTER, BEFORE, COLLATE, COPY, EVERY, HIGH, INDEX, LENGTH, LOW, REVERSE, SOME, STRING, SUBSTR, TRANSLATE, UNSPEC, VERIFY. The following are not implemented: BIT, CHARACTER, DECAT, VALID.

4. The following array functions are implemented: DIMENSION, HBOUND, LBOUND. The following are not implemented: DOT, PROD, SUM.

5. The following storage control functions are implemented: ADDR, ALLOCATION, NULL. The following are not implemented: EMPTY, OFFSET, POINTER.

6. The following ON-condition built-in functions are not implemented: ONCHAR, ONFIELD, ONFILE, ONCODE, ONKEY, ONLOC, ONSOURCE.

7. The following functions are implemented: DATE and TIME.

8. The following functions are not implemented: BOOL, LINENO, PAGENO.

9. The following pseudo-variables are implemented: STRING, SUBSTR and UNSPEC. The following are not implemented: IMAG, ONCHAR, ONSOURCE, PAGENO, REAL.

27.2 COMPATIBILITY WITH ANSI STANDARD

27.2.1 Compatibility Philosophy

CPL was implemented with the philosophy that it must be a true subset of the ANSI PL/I standard. This meant that any extensions to the standard must be clearly documented as extensions, must be easy to avoid by any programmer who wants to, and must not be easy to confuse with a feature of the standard language. The last criterion is very important, in that it means that, for example, a new option of the DO statement would be unacceptable since it would be too easy to confuse it with existing options.

The last criterion was, however, broken in two very important cases -- the default FLOAT rules and the use of the VFORM file option. These incompatibilities were implemented only after many hours of agonizing on the part of the implementor, and only after deciding that CPL's ease of use for the beginning user would be overwhelmingly improved. In fact, the first chapter of this manual would simply have been impossible to write under the default rules defined by the standard.

Of course, the first two criteria above are still valid, and a user wishing to be fully ANSI standard can simply use the methods described in the next section. In fact, CPL will be fully ANSI standard if the user starts his session by inserting the statement:

```
1.    DECLARE SYSPRINT PRINT;  DEFAULT(RANGE(*)) FIXED;
```

into his program.

27.2.2 List Of Incompatibilities

The following is a complete list of places where CPL is incompatible with the ansi PL/I standard. To write a fully standard program, you must avoid these features.

1. All CPL variables default to FLOAT rather than FIXED, as the standard specifies. To make your program completely standard, insert a default statement which specifies defaults for all variables. For example, you may insert:

   ```
   1.    DEFAULT (RANGE(*)) FIXED;
   ```

   to force CPL to make all your variables default to FIXED. Or, you may insert

   ```
   1.    DEFAULT (RANGE(*)) FLOAT;
   ```

   so that when your program is moved to other systems, the default of FLOAT will be carried over.

   On some early implementations of PL/I, the "I through N rule" was used. This rule specifies that all variables default to FLOAT except those whole first letters begin with I-N; these default to FIXED. If you would like to be fully ANSI standard and still be compatible with this rule, then you should insert these statements in your program:

   ```
   1.    DEFAULT (RANGE(I:N)) FIXED;
   2.    DEFAULT (RANGE(A:H)!RANGE(O:Z)) FLOAT;
   ```

2. Default PUT LIST output to the SYSPRINT file is in "variable format." Furthermore, the VFORM option of the ENVIRONMENT of the FILE declaration permits the programmer to specify this option for other output files. This option specifies that when a FIXED or FLOAT quantity is printed by means of a PUT LIST statement, the format used depends upon the value of the quantity being printed. (For example, 2.3E0 will be printed as 2.3, rather than as 2.30000000E+00).

   The user wishing to write a fully standard program should avoid the VFORM option, and should prevent the VFORM option default for SYSPRINT by inserting this statement at the beginning of his program:

   ```
   3.    DECLARE SYSPRINT FILE PRINT;
   ```

Note, however, that one of the sample programs given in the chapter entitled "CPL Programming Examples" shows how to implement the variable format as a PL/I PROCEDURE.

3. The CLOSE FILES statement is not in the ANSI standard. If you wish to write standard programs, then you should restrict use of this statement to direct mode.

4. The DELAY statement is not in the ANSI standard, although it is available in IBM implementations of PL/I.

5. The SNAP statement is not in the ANSI standard. However, the SNAP option of the ON statement is standard.

6. The following mathematical built-in functions are not in the ANSI standard, although they are available in some other PL/I implementations: ACOS, ASIN, COSH, SINH.

7. The following built-in functions are not available to any other PL/I implementation known to this implementor: DIVI, DIVF, FLTED, ONMSG, RANDOM.

8. The ATTENTION ON-condition is not in the ANSI standard.

9. The "?" statement should be replaced by the PUT LIST statement.

CHAPTER 28

RUNNING CPL UNDER TOPS-10 (R)

This chapter sows you how to run CPL if you are running under the TOPS-10 operating system. Please refer to the next chapter for information on running under the TOPS-20 operating system.

28.1 HOW TO LOG ON AND RUN CPL

This section shows you how to log on to the DECsystem-10. If you already now how to log on, you may skip to the next section.

28.1.1 How To Log On

In order to begin using the system, do the following:

1. After you turn the terminal on, press the CTRL key. While it is still in the down position, press the C key. This is called "typing a Control-C."

2. After you see the period (.), type the word LOGIN and press the RETURN key.

3. After you see the #, type your project-programmer number and press the RETURN key.

4. After you see the word PASSWORD:, type your password and press the RETURN key. Your password will not be printed.

Steps 1 through 4 above are called the logging-in procedure. Below is an example of the logging-in procedure, where the underlined portion is what you type:

```
.LOGIN <RETURN>
JOB 32     R5725D SYS #40/2 TTY73
#27,4072 <RETURN>
PASSWORD: <RETURN>
1131    21-JAN-75       TUE
```

28.1.2 How To Use A Telephone Connection

If the terminal you are using is not a local terminal (hooked directly to the computer), you must make connection with the computer through the use of a telephone. The procedure to establish this connection is:

1. Turn on the terminal.
2. Check the speed setting of the terminal (most terminals are labeled either 10 CPS (characters per second), 15 CPS and 30 CPS or 110 baud, 150 baud and 300 baud.) It should initially be set to either 10 CPS or 110 baud.
3. Dial the phone number to the computer.
4. Wait for a steady tone (sometimes this is a high pitched beep).
5. Lay the receiver in the slots provided for it either on the terminal or on the acoustic coupler.
6. Wait for the carrier detect light to come on.
7. If you see nothing printed on the terminal, type two Control-C characters until you see either a period or the words PLEASE LOGIN OR ATTACH.
8. Follow the logging-in procedure listed above.

28.1.3 Running CPL

After you have logged in and the system has typed a "." to indicate that you are DECsystem-10 monitor level, you may enter CPL by typing the command:

```
.R CPL <RETURN>
CONVERSATIONAL PROGRAMMING LANGUAGE
*
```

After CPL has typed out the asterisk shown on the last line, CPL is ready for commands.

28.2 HOW TO LOG OFF

This section shows you how to leave CPL and log off the DECsystem-10.

28.2.1 How To Leave CPL

To leave CPL, type the command "MONITOR", or its abbreviation, "MON".

28.2.2 How To Log Off

When you are finished using the system, do the following: Type K/F and press the RETURN key.

28.3 FORMAT OF A TOPS-10 FILE-SPECIFICATION

A file-specification is used in several CPL commands:

1. The LOAD, WEAVE and SAVE statements
2. The TITLE option of the OPEN statement

The format of a TOPS-10 file-specification is as follows:

```
dev:name.type[project-number,programmer-number]
```

Usually, any of these file fields may be omitted, and CPL will supply a default.

The meanings of these fields and their default values are as follows:

1. The "dev" is the file device name, which specifies which physical or logical device the file is on. If it is not specified, then the default device is DSK, the disk.

   EXCEPTION: In the TITLE option of the OPEN statement, if the FILE option specifies either SYSIN or SYSPRINT, then the default device is TTY, your terminal.

2. The "name" field is the file-name. Not all devices require a file-name; however, if one is required then you must specify it, since there is no default.

3. The "type" field is the file-type (also known as the "filename extension" field). It may consist of up to 3 letters or digits. If no file extension is specified, then the default file-type is the following:

   1. In the case of the TITLE option of the OPEN statement, the default file-type is DT (for "data").

   2. In the case of the LOAD, SAVE or WEAVE statement, the default file-type is CPL.

4. The "project-number" and "programmer-number" fields each consist of one to six octal digits. If you do not specified a ppn (project-programmer-number), then your own ppn will be used as the default.

CHAPTER 29

RUNNING CPL UNDER TOPS-20 (R)

This chapter shows you how to run CPL if you are running under the TOPS-20 operating system. Please refer to the preceding chapter for information on running under the TOPS-10 operating system.

29.1 HOW TO LOG ON AND RUN CPL

This section shows you how to log on to the DECsystem-20. If you already know how to log on, you may skip to the next section.

29.1.1 How To Log On

In order to begin using the system, do the following:

1. Get someone to show you how to turn on the computer terminal.

2. After you turn on the terminal, press the key labeled CTRL and, at the same time, type the letter C. (This is known as typing Control-C.)

3. After you see the "@" character, type LOGIN, and press the key labeled ESC. (On some terminals, this key is labeled ESCAPE or ALT or ALTMODE.) In the example displayed later in this section, the ESC key is shown as a dollar sign ($).

4. After the system prints (USER), type your user name and press the ESC key again.

5. After the system prints (PASSWORD), type your password and press the ESC key.

6. After the system prints (ACCOUNT), type your account number and press the key labeled RETURN (or CR).

Below is an example of how you would log in if your user name were PORADA, your password were WILKS, and your account number were 10300. The portions that you type are underlined; all other parts are typed by the operating system.

```
 V 1.02.36, TOPS-20 DEVELOPMENT MONITOR, 19-JAN-72. TOPS-20 1(100)
@LOGIN$ (USER) PORADA$ (PASSWORD) $ (ACCOUNT) 10300
 JOB 17 ON TTY20 19-JAN-76 13:00
@
```

29.1.2 HOW TO USE A TELEPHONE CONNECTION

If the terminal you are using is not directly connected to the computer, you must connect to the computer via a telephone. The procedure for obtaining this type of connection is:

1. Turn on the computer terminal.
2. Check the speed setting of the terminal. Most terminals are labeled in either characters per second (CPS) or bits per second (baud). If your terminal is labeled in CPS, set it to 30 CPS. If your terminal is labeled in baud, set it to 300 baud.
3. Dial the phone number to the computer.
4. Wait for a steady ton. Sometimes the tone is a high-pitched beep.
5. Place the telephone receiver in the slots provided for it in either the terminal or the acoustic coupler.
6. Wait for the carrier detect light to come on.
7. At this point, if nothing has been printed on the terminal, press the key labeled CTRL (or CONTROL) and, at the same time, type a C.
8. Follow the logging-in procedure listed above.

29.1.3 Running Cpl

After you have logged in, and the system has typed a "@" to indicate that you are at the DECsystem-20 monitor level, you may enter CPL by typing the command:

```
          @CPL
          CONVERSATIONAL PROGRAMMING LANGUAGE
*
```

After CPL has typed out the asterisk shown on the last line, CPL is ready for commands.

29.2 HOW TO LOG OFF

This section shows you how to leave CPL and log off the DECsystem-20.

29.2.1 How To Leave CPL

To leave CPL, type the command "MONITOR," or its abbreviation "MON."

29.2.2 How To Log Off

When you are finished using the system, do the following: Type LOGOUT and press the RETURN key.

29.3 FORMAT OF A TOPS-20 FILE-SPECIFICATION

A file-specification is used in several CPL commands:

1. The LOAD, WEAVE and SAVE statements

2. The TITLE option of the OPEN statement

The format of a TOPS-20 file-specification is as follows:

```
dev:<dir>name.type.gen
```

Usually, any of these file fields may be omitted, and CPL will supply a default.

The meanings of these fields and their default values are as follows:

1. The "dev" is the file device name, which specifies which physical or logical device the file is on. If it is not specified, then the default device is DSK, the disk.

   EXCEPTION: In the TITLE option of the OPEN statement, if the FILE option specifies either SYSIN or SYSPRINT, then the default device is TTY, your terminal.

2. The "dir" field is the directory-name, consisting of up to 39 alphanumeric (including hyphen, dollar sign and underline) characters. The directory-name is enclosed in angle brackets. If you do not specify a directory-name, then the default is your own personal directory.

3. The "name" field is the file-name, consisting of up to six alphanumeric characters. Not all devices require a file-name; however, if one is required then you must specify it, since there is no default.

4. The "type" field is the file-type. It may consist of up to 3 letters or digits. If no file-type is specified, then the default file-type is the following:

   1. In the case of the TITLE option of the OPEN statement, the default file-type is DT (for "data").

   2. In the case of the LOAD, SAVE or WEAVE statement, the default file-type is CPL.

CHAPTER 30

CPL PROGRAMMING EXAMPLES (B-D)

This chapter gives some examples of complete programs, illustrating both simple and advanced features of CPL.

30.1 SIMPLE PROGRAM TO TYPE PRIME NUMBERS (B)

```
 10.            /* THIS PROGRAM TYPES OUT PRIME NUMBERS */;
 20.            /* ALL PRIMES IN THE SPECIFIED RANGE ARE TYPED */;
 30.            /* COPYRIGHT 1976 BY DIGITAL EQUIPMENT CORPORATION */;
 40.            /* WRITTEN JANUARY 1976 BY JOHN XENAKIS */;
 50.            START = 3;
 60.            END=1000;
 70.            DO I = START TO END BY 2;
 80.            DO;
 90.            DO J = 3 TO SQRT(I) BY 2;
100.            IF MOD(I,J) = 0 THEN GO TO NO_PRIME;
110.            END;
120.            PUT LIST(I);
130.            END;
140.    NO_PRIME:       ;
150.            END;
```

This program illustrates the following features:

1. The MOD built-in function, which returns the remainder which results from dividing the two arguments.

2. DO-groups, IF statements, and structured programming.

For extra practice, you may try the following problems:

1. Change statements 50 and 60 so that a different range of prime number is typed out.

2. Use PUT SKIP to make each prime print out on a separate line.

30.2 TABLE OF SINES AND COSINES (B)

```
 10.          /* THIS PROGRAM TYPES OUT A TABLE OF SINES AND */;
 20.          /* COSINES IN THE RANGE 0 TO PI IN INCREMENTS */;
 30.          /* OF 0.2. */;
 40.          /* COPYRIGHT 1976 BY DIGITAL EQUIPMENT CORPORATION */;
 50.          /* WRITTEN JANUARY 1976 BY JOHN XENAKIS */;
 60.          PUT EDIT('X','SIN(X)','COS(X)')(COL(3),A(6),A(10),A);
 70.          PI = 3.14159;
 80.          DO X = 0 TO PI BY .2;
 90.          PUT SKIP EDIT(X,SIN(X),COS(X))(F(4,1),2 F(10,6));
100.          END;
```

This program illustrates the following CPL features:

1. The SIN and COS mathematical built-in functions
2. The PUT EDIT statement with the "F" format item.

For extra practice, try the following:

1. Type out a table of other functions (LOG, EXP, etc.)
2. Change the F format item to E, and experiment with other forms of output.
3. Cause the output to be directed to a disk file, by inserting the statement

   ```
   55.         OPEN FILE(SYSPRINT) TITLE('DSK:X.DT');
   ```

   to cause output to go to the disk file X.DT.

30.3 MAKE A "CONCORDANCE OF LETTERS" (C)

```
  1.          /* THIS PROGRAM ACCEPTS AN INPUT LINE FROM THE */;
  2.          /* TERMINAL, AND TYPES OUT A "CONCORDANCE" OF THE */;
  3.          /* NUMBER OF TIMES EACH LETTER OF THE ALPHABET */;
  4.          /* APPEARS IN THAT LINE. */;
  5.          /* COPYRIGHT 1976 BY DIGITAL EQUIPMENT CORPORATION */;
  6.          /* WRITTEN FEBRUARY 1976 BY JOHN XENAKIS */;
 10.          DECLARE S CHAR(200) VAR;
 20.          DECLARE ALPH CHAR(26);
 30.          ALPH = SUBSTR(COLLATE(),66,26);
 40.          DECLARE COUNT(26);
 50.          DO WHILE(1=1);
 60.          PUT SKIP LIST('TYPE LINE:');
 70.          READ FILE(SYSIN) INTO(S);
 80.          UNSPEC(COUNT) = ''B;
 90.          DO J = 1 TO LENGTH(S);
100.          I = INDEX(ALPH,SUBSTR(S,J,1));
110.          IF I > 0 THEN COUNT(I) = COUNT(I) + 1;
120.          END;
130.          PUT SKIP;
140.          DO I = 1 TO 26;
150.          PUT EDIT(SUBSTR(ALPH,I,1))(X(1),A);
160.          END;
170.          PUT SKIP;
180.          DO I = 1 TO 26;
190.          PUT EDIT(COUNT(I))(F(2));
200.          END;
210.          END;
```

This program illustrates the following CPL features:

1. CHARACTER string declarations
2. Using the SUBSTR and COLLATE built-in functions to obtain the entire alphabet.
3. Arrays (the COUNT array is declared in statement 40)
4. The READ statement, which allows you to input a CHARACTER string which is not enclosed in quotes.
5. The A format item
6. The LENGTH and INDEX built-in functions.
7. The use of the UNSPEC pseudo-variable as a trick to zero out an entire array.

For additional practice, do the following:

1. Print out the letters in a different format.
2. Extend the "concordance" so that it handles additional characters (like punctuation, digits, etc.)
3. Change the program so that it does not use the COLLATE built-in function or the UNSPEC pseudo-variable

30.4 HEXADECIMAL ADDING MACHINE (C)

```
  10.          /* HEXADECIMAL ADDING MACHINE */;
  20.          /* COPYRIGHT 1976 BY DIGITAL EQUIPMENT CORPORATION */;
  30.          /* WRITTEN FEBRUARY, 1976, BY JOHN XENAKIS */;
  40.          DCL S CHAR(40) VAR;
  45.          DCL (SUM,SUMMAND) FIXED;
  46.          DCL BB BIT(36) ;
  50.          DO WHILE(1=1);
  55.          PUT SKIP LIST('TYPE FIRST SUMMAND');
  60.          SUM = 0;
  70.          READ FILE(SYSIN) INTO(S);
  80.          DO WHILE(LENGTH(S) > 0);
  81.          SIGN = 1;
  82.          IF SUBSTR(S,1,1)='-' THEN SIGN = -1;
  83.          IF INDEX('+-',SUBSTR(S,1,1))>0 THEN S=SUBSTR(S,2);
  90.          S = ''''!!COPY('0',9-LENGTH(S))!!S!!'''B4 ';
 100.          GET STRING(S) LIST(BB);
 110.          UNSPEC(SUMMAND) = BB;
 120.          SUM = SUM + SIGN * SUMMAND;
 125.          PUT LIST('ENTER NEXT SUMMAND');
 130.          READ FILE(SYSIN) INTO(S);
 140.          END;
 150.          IF SUM<0 THEN PUT EDIT('-')(A);
 160.          PUT EDIT(SUM)(B4);
 170.          END;
```

This program illustrates the following CPL features:

1. The B4 format item for hexadecimal output
2. The use of GET STRING
3. The use of B4 to indicate a hexadecimal bit string
4. The use of the UNSPEC built-in function and pseudo-variable to perform machine-dependent bit-string manipulations

You probably will not understand the above example unless you understand the 36-bit format of a DECsystem-10/20 word.

For additional practice, try the following:

1. Change each occurrence of "B4" to "B3" so that the program becomes an octal adding machine.
2. Use an ON ERROR statement so that if the person running the program types in an illegal character, the program will continue executing.

30.5 PROCEDURE TO SIMULATE "VFORM" FILE ATTRIBUTE (D)

```
   1.             /* PROGRAM WHICH IMPLEMENTS THE 'VFORM' ATTRIBUTE */;
   2.             /* COPYRIGHT 1976 BY DIGITAL EQUIPMENT CORPORATION */;
   3.             /* WRITTEN JANUARY 1976 BY JOHN XENAKIS */;
  10.     VFORM:  PROC(X) RETURNS(CHAR(40) VAR);
  20.             DECLARE (X,Y,Z) FLOAT;
  30.             DECLARE CC CHAR(40) VAR;
  35.             DECLARE SD FIXED;
  40.             DECLARE (N,I) FIXED;
  45.             DECLARE S CHAR(1) VAR;
  50.             IF X=0 THEN RETURN('0');
  60.             Y=ABS(X);
  70.             IF Y > 1E-29 THEN DO;
  75.             I = 8;
  80.             UNSPEC(Z)=SUBSTR(UNSPEC(Y),1,9)!!SUBSTR(UNSPEC(I),10);
  90.             Y=Y+Z;
 100.             END;
 101.             SD = 8; /* # SIGNIFICANT DIGITS */;
 105.             IF X<0 THEN S='-'; ELSE S='';
 110.             CC = SIGN(X)*Y;
 120.             N = SUBSTR(CC,SD+4)+1;
 130.             DO I = SD+1 TO 4 BY -1 WHILE(SUBSTR(CC,I,1)='0');
 140.             END;
 150.             I=I-2;
 160.             IF N>8!N<-8 THEN IF I=1 THEN DO;
 170.             RETURN(S!!SUBSTR(CC,2,1)!!SUBSTR(CC,SD+3));
 175.             END;
 180.             ELSE RETURN(S!!SUBSTR(CC,2,I+1)!!SUBSTR(CC,SD+3));
 200.             CC=SUBSTR(CC,2,1)!!SUBSTR(CC,4,SD-1);
 210.             CC=SUBSTR(CC,1,I);
 220.             IF N>=0 THEN DO;
 230.             IF I<N THEN RETURN(S!!CC!!COPY('0',N-I));
 240.             IF I=N THEN RETURN(S!!CC);
 250.             RETURN(S!!SUBSTR(CC,1,N)!!'.'!!SUBSTR(CC,N+1));
 260.             END;
 270.             RETURN(S !! '.' !! COPY('0',-N) !! CC);
 280.             END VFORM;
```

This program consists of a function PROCEDURE which returns a character string. If you type "XEQ" you will not get any output since there are no statements in the program which invoke the VFORM procedure.

To invoke this function, type something like the following:

```
?VFORM(234.5)
```

The VFORM function simulates the effect of the non-standard VFORM file attribute. It takes a floating point number as an argument and returns a "variable format" representation of that number as a character string.

This program illustrates the following CPL features:

1. A function PROCEDURE which returns a CHARACTER string
2. The SUBSTR built-in function for bit strings.

3. The UNSPEC built-in function and pseudo-variable
4. The COPY built-in function
5. A DO-clause with a negative "BY" increment

If you would like additional practice, try the following:

1. Change the format so that it types out a plus sign if the number is positive.
2. Change the format so that it types a maximum of 6 significant digits.
3. Change the format so that it switches to an "E" type format when the number is greater than 1E6 rather than 1E8.
4. If you have available to you an implementation of PL/I on another machine, convert this PROCEDURE to run on that machine. Statements 70 through 101 are machine dependent and will have to be changed. Statements 70 through 100 "round" the argument to prevent output of the form "1.9999999" rather than "2". Statement 101 sets SD to the number of significant digits in a floating-point number.

   If you do that, then you will have a way of printing variable formats on other implementations of PL/I.

30.6 FORMAT CPL PROGRAMS FOR OTHER PL/I IMPLEMENTATIONS (C)

```
 10.           DECLARE S CHAR(250) VAR;
 20.           DECLARE (IN INPUT, OUT OUTPUT) FILE RECORD;
 30.           DECLARE CTAB CHAR(1); CTAB=SUBSTR(COLLATE(),10,1);
 40.           DECLARE CCONT BIT(1);
 50.           DECLARE PNAME CHAR(31) VAR;
 60.           ON UNDEFINEDFILE(IN) BEGIN;?ONMSG(); GO TO GETIN; END;
 70.   GETIN:  PUT SKIP LIST('INPUT FILE:');
 80.           READ FILE(SYSIN) INTO(S);
 90.           OPEN FILE(IN) TITLE(S);
100.           ON UNDEFINEDFILE(OUT) BEGIN;?ONMSG(); GOTO GETOUT;END;
110.   GETOUT: PUT SKIP LIST('OUTPUT FILE:');
120.           READ FILE(SYSIN) INTO(S);
130.           OPEN FILE(OUT) TITLE(S);
140.           PUT SKIP LIST('PROGRAM NAME:');
150.           READ FILE(SYSIN) INTO(PNAME);
160.           S = ' ' !! PNAME !! ': PROC OPTIONS(MAIN);';
170.           WRITE FILE(OUT) FROM(S);
180.           S = ' DFT (RANGE(*)) FLOAT;';
190.           WRITE FILE(OUT) FROM(S);
200.           ON ENDFILE(IN) GO TO CLOSE;
210.           CCONT = '0'B;
220.           DO WHILE(1=1);
230.           READ FILE(IN) INTO(S);
240.           IF ^CCONT THEN S = AFTER(S,CTAB);
250.           CCONT = SUBSTR(S,LENGTH(S),1) = '&';
260.           IF CCONT THEN S = SUBSTR(S,1,LENGTH(S)-1);
270.           S = ' ' !! S;
280.           WRITE FILE(OUT) FROM(S);
290.           END;
300.   CLOSE:  S = ' END ' !! PNAME !! ';';
310.           WRITE FILE(OUT) FROM(S);
320.           CLOSE FILE(IN),FILE(OUT);
```

CPL programs contain line numbers and continuation characters which are not used in other PL/I implementations. This program reads a file containing a CPL program and writes out a disk file containing the program in a format which other implementations use.

This program illustrates the following CPL features:

1. A utility program which reads one file, modifies it, and writes a new file as output.
2. A factored FILE declaration in statement 20 for the identifiers IN and OUT.
3. The ON-conditions UNDEFINEDFILE and ENDFILE.
4. The READ and WRITE statements.
5. The OPEN statement for an arbitrary disk file.

For additional practice, try the following:

1. Modify the program so that it removes all tabs from the file.

2. Modify the program so that it will flag the use of those CPL built-in functions which are not in the ANSI standard.

3. Rewrite the program so that it goes in the other direction -- takes a PL/I program written on some other machine and transforms it to the CPL format.

30.7 PROGRAM WHICH PRINTS ITS OWN SOURCE (D)

Consider the following problem:

Write a program which prints its own source, but which does not read any input files.

This problem was posed to the author in 1968. He did not solve it until 2 years later.

The following is a solution for CPL. The LIST and XEQ statements will produce the same identical output, even to the comments.

If you type this program in, you must be certain that every line number and character is exactly as shown, even in the comments.

```
 100.          /* COPYRIGHT 1976 BY DIG EQUIPMENT CORP */;
 101.          /* WRITTEN 2/76 BY JOHN XENAKIS */;
 102.          DECLARE S(104:122) CHAR(60) VAR;
 103.          DECLARE TABS CHAR(2), DOL CHAR(1);
 104.          S(104)='/* COPYRIGHT 1976 BY DIG EQUIPMENT CORP */;';
 105.          S(105)='/* WRITTEN 2/76 BY JOHN XENAKIS */;';
 106.          S(106)='DECLARE S(104:122) CHAR(60) VAR;';
 107.          S(107)='DECLARE TABS CHAR(2), DOL CHAR(1);';
 108.          S(108)='TABS=COPY(SUBSTR(COLLATE(),10,1),2);';
 109.          S(109)='DOL=SUBSTR(COLLATE(),37,1) /* DOLLAR */;';
 110.          S(110)='C=104-1;';
 111.          S(111)='DO L=100 TO 137;';
 112.          S(112)='PUT SKIP EDIT(L,$.$,TABS)(F(4),2 A);';
 113.          S(113)='IF L>103&L<123 THEN DO;';
 114.          S(114)='PUT EDIT($S($,L,$)=$$$)(A,F(3),A);';
 115.          S(115)='PUT EDIT(S(L),$$$;$)(A);';
 116.          S(116)='END;';
 117.          S(117)='ELSE DO;';
 118.          S(118)='C=C+1;';
 119.          S(119)='PUT EDIT(TRANSLATE(S(C),$$$$,DOL))(A);';
 120.          S(120)='END;';
 121.          S(121)='END;';
 122.          S(122)='PUT SKIP;';
 123.          TABS=COPY(SUBSTR(COLLATE(),10,1),2);
 124.          DOL=SUBSTR(COLLATE(),37,1) /* DOLLAR */;
 125.          C=104-1;
 126.          DO L=100 TO 137;
 127.          PUT SKIP EDIT(L,'.',TABS)(F(4),2 A);
 128.          IF L>103&L<123 THEN DO;
 129.          PUT EDIT('S(',L,')=''')(A,F(3),A);
 130.          PUT EDIT(S(L),''';')(A);
 131.          END;
 132.          ELSE DO;
 133.          C=C+1;
 134.          PUT EDIT(TRANSLATE(S(C),'''',DOL))(A);
 135.          END;
 136.          END;
 137.          PUT SKIP;
```

This program illustrates the following CPL features:

CPL PROGRAMMING EXAMPLES (B-D)

1. CHARACTER string arrays

2. The use of SUBSTR and COLLATE built-in functions to obtain special characters (in this case, tab and dollar sign)

3. The COPY and TRANSLATE built-in functions

4. PUT EDIT with the A format item

For extra practice, try doing the same thing in FORTRAN.

CHAPTER 31

LIST OF CPL ABBREVIATIONS (R)

CPL provides abbreviations for certain keywords and built-in functions. The abbreviations themselves are keywords or built-in functions and CPLL will recognize them as synonymous in every respect with the full denotationss, except that in the case of built-in functions the abbreviations have separate declarations and name scopes.

ALLOCATE	ALLOC
ALLOCATION	ALLOCN
ATTENTION	ATTN
AUTOMATIC	AUTO
CHARACTER	CHAR
CONDITION	COND
CONTROLLED	CTL
CONTINUE	CONT
CONVERSION	CONV
DECLARE	DCL
DEFAULT	DFT
DIMENSION	DIM
ENVIRONMENT	ENV
EXECUTE	XEQ
FIXEDOVERFLOW	FOFL
INITIAL	INIT
MONITOR	MON
OVERFLOW	OFL
PARAMETER	PARM
POINTER	PTR
PROCEDURE	PROC
SEQUENTIAL	SEQL
STRINGRANGE	STRG
STRINGSIZE	STRZ
SUBSCRIPTRANGE	SUBRG
THROUGH	THRU
UNDEFINEDFILE	UNDF
UNDERFLOW	UFL
VARYING	VAR
ZERODIVIDE	ZDIV

CHAPTER 32

CPL SUMMARY (R)

This chapter is a short reference summary of the CPL language. It includes examples of all the program elements and statements.

32.1 PROGRAM ELEMENTS

Identifiers are 1 to 31 characters. The characters are all letters and digits and the "break" character, "_". The first character must be a letter.

FIXED constants: 2, 23, 34567.

FLOAT constants: 2.3, 4.5E10, 3.9876992E-20.

CHARACTER constants: 'ABC123', 'THAT''S OK'.

BIT constants: '01001'B, '03213'B2, '0374573'B3, '029CDFAB53'B4.

Repetition factors: (5)'AB' is same as 'ABABABABAB'. (2)'101'B is same as '101101'B.

Arithmetic operators: +, -, *, /, ** (exponentiation)

Comparison operators: =, ^=, >=, <=, >, <, ^>, ^<

Logical operators: ! (or), & (and), ^ (not)

Concatenation operator: !!

Comments: /* ... */

Line numbers: 1.00 to 9999.99

Direct statements: "A=5;" is executed immediately.

Collect statements: "10.3 A=5" is saved as part of the collect program.

32.2 DIRECT-ONLY STATEMENTS

LIST 10, 50 THRU 60, 1000 THRU ...; lists the indicated statements in your program.

ERASE 20.34; erase statement 20.34.

ERASE THRU ...; erase entire program, resetting all storage.

SAVE 'DSKC:PROG.C[10,3701]'; save program in specified file.

LOAD 'PROG'; load program from file PROG.CPL.

NUMBER 100 BY 5; generate automatic line numbers starting from 100 and continuing 105, 110, etc. To terminate automatic line numbering mode, type the single character "#", followed by a carriage return.

LOAD 'PROG' NUMBER 3000; load PROG.CPL, renumbering the statements starting from 3000.

WEAVE 'NEWPRG' NUMBER 2000; load 'NEWPRG.CPL' without erasing the existing program, so that the two programs will be combined.

XEQ; execute the program.

XEQ FROM 21.3; execute program starting from statement 21.3.

CONTINUE; continue after a breakpoint, cr reexecute the statement causing an error.

CONTINUE FROM 2310; continue executing from statement 2310.

BREAK 22+4; set breakpoint on fifth statement on line 22.

NOBREAK 22+4; remove this breakpoint.

NOBREAK; remove all breakpoints.

MONITOR; return to monitor.

32.3 DECLARATIVE AND STORAGE ALLOCATION STATEMENTS

DEFAULT (RANGE(I:N)) FIXED; sets the "I-N" rule: all identifiers beginning with letters I through N default to FIXED; all others default to FLOAT.

DECLARE A FIXED, (B,C) FLOAT; makes A FIXED, and B and C FLOAT.

DECLARE C1 CHAR(100), C2 CHAR(23) VAR; makes C1 CHARACTER with fixed length 100, and C2 CHARACTER with varying length less than or equal to 23.

DECLARE B1 BIT(100), B2 BIT(23) VAR; same for BIT.

DECLARE ARR(10,2:5); declares a two-dimensional array, with the first subscript ranging from 1 to 10 and the second subscript ranging from 2 to 5.

ALLOCATE A; allocates CONTROLLED identifier A.

FREE A; frees CONTROLLED identifier A.

32.4 ASSIGNMENT AND FLOW OF CONTROL STATEMENTS

A=B+LOG(F+2); assign to A the value of the expression to the right of the equal sign.

LAB1: A=5; the assignment statement "A=5" is given the statement label LAB1.

GO TO LAB1; transfers control to the statement with label LAB1.

IF I>5 THEN A=I+2; ELSE GO TO XYZ; specifies that the action to be taken depends upon whether I is greater than 5.

IF I>5 THEN DO; ...; END; ELSE DO; ...; END; allows the user to specify a group of statements for the THEN and ELSE clauses. (The statements in the group may be placed on individual lines.)

DO WHILE(A<=B+5); ...; END; Keep executing the group of statements as long as A remains less than or equal to B+5.

DO I=3,5,6 TO 9,15; ...; END; Execute the group of statements. The variable I will take on the values 3, 5, 6, 7, 8, 9 and 15.

DO I=1 REPEAT(I+I) WHILE(I<=32); ...; END; Execute the group of statements. The variable I will assume the values 1, 2, 4, 8, 16 and 32.

STOP; causes program execution to stop.

DELAY(2000); will cause CPL to go to sleep for 2000 milliseconds (2 seconds).

32.5 ON CONDITIONS AND ERROR HANDLING

ON ERROR SNAP GO TO RESTART; specifies that if any program error occurs, type out a snap dump and then transfer to the statement with label RESTART.

ON ENDFILE(F) BEGIN; CLOSE FILE(F); GO TO END_FILE; END; specifies that when end of file is reached in reading file F, then close file F and transfer to the statement with label END_FILE.

ON ERROR SYSTEM; specifies that if an error occurs from that point on, the standard system action (usually to type an error message and halt execution) is to take place.

SIGNAL ENDFILE(F); specifies that the named condition is to be raised artificially, so that the ON-unit for that condition can be invoked.

REVERT ERROR; specifies that any ON-unit or SYSTEM option specified for the ERROR condition in the current block is to be canceled, and the status of the ERROR condition is to be the same as it was when the current block was entered.

LIST OF ON-CONDITIONS

Here is a list of the condition names, and the occurences which cause the conditions to be raised:

1. SUBSCRIPTRANGE -- array subscript is out of range
2. STRINGRANGE -- argument to SUBSTR is out of range
3. ZERODIVIDE -- division by zero
4. ENDFILE(filename) -- end of file on input
5. UNDEFINEDFILE(filename) -- file cannot be opened
6. RECORD(filename) -- record length error in READ statement
7. ERROR -- error for which no other established ON-unit applies
8. CONDITION(ident) -- programmer-named condition; may only be raised by means of SIGNAL statement
9. ATTENTION condition -- Control-C typed while program is executing

32.6 INPUT/OUTPUT STATEMENTS

PUT LIST(A,B+C); type out the values of A and of B+C.

?A,B+C; ?... is an abbreviation for PUT LIST(...).

GET LIST(X,Y,Z); reads the values of the variables X, Y and Z from the terminal.

PUT EDIT(A,B+C)(F(2),E(12,3)); prints the values of A and B+C in the specified formats. A complete list of format items is given below.

PUT STRING(S) LIST(A,B+C); stores the values of A and of B+C into the CHARACTER string S.

GET STRING(S!!' 2 ') LIST(X,Y,Z); obtains the values of X, Y and Z from the CHARACTER string expression S!!' 2 '.

In the following, assume F has been DECLAREd to be a FILE.

PUT FILE(F) SKIP LIST(A,B); puts the values of A and B into the file F at the beginning of a new line.

GET FILE(F) LIST(X,Y); gets the values of X and Y from file F.

OPEN FILE(F) TITLE('XYZ.MAC'); opens file XYZ.MAC for input.

OPEN FILE(F) OUTPUT; opens file DSK:F.DT for output.

READ FILE(F) INTO(C); reads a record from RECORD file F into the CHARACTER scalar C.

READ FILE(F) IGNORE(5); skips 5 records in file F.

WRITE FILE(F) FROM(C); writes a record into file F from the CHARACTER scalar C.

CLOSE FILE(F); close file F.

CLOSE FILES; close all open files.

32.7 PUT EDIT FORMAT ITEMS

In all of these, w is the width of the field.

DATA FORMAT ITEMS

1. F(w) -- integer
2. F(w,d) -- d digits after the decimal point
3. E(w) -- E-type constant, with 7 digits after the decimal point and 8 significant digits
4. E(w,d) -- E-type constant, with d digits after the decimal point and (d+1) significant digits
5. E(w,d,s) -- E-type constant, with d digits after the decimal point and s significant digits.
6. A[(w)] -- CHARACTER
7. B[(w)] or B1[(w)] -- BIT
8. B3[(w)] -- BIT in octal format
9. B4[(w)] -- BIT in hexadecimal format
10. B2[(w)] -- BIT as base 4 integer format

CONTROL FORMAT ITEMS

1. X(w) -- print w blanks
2. SKIP[(n)] -- skip n blank lines
3. PAGE -- skip to new page
4. COLUMN(n) -- move to the n'th column on the page

REMOTE FORMAT ITEM

R(label) -- use the specified FORMAT statement

32.8 SUMMARY OF BUILT-IN FUNCTIONS

The following chart summarizes the built-in functions supported by CPL.

Function	#Args	Meaning
********** ARITHMETIC BUILT-IN FUNCTIONS		
ABS	1	Compute absolute value
CEIL	1	Compute least integer not less than argument
DIVF(*)	2	Take FLOAT quotient of two arguments
DIVI(*)	2	Take integer quotient of two arguments
FLOOR	1	Take greatest integer not exceeding argument
MAX	>=2	Compute maximum of arguments
MIN	>=2	Compute minimum of arguments
MOD	2	Compute remainder in dividing two arguments
SIGN	1	Return +1, 0 or -1 depending upon sign of arg
TRUNC	1	Truncate argument to integer
********** MATHEMATICAL BUILT-IN FUNCTIONS		
ACOS(*)	1	Compute arccosine in radians
ASIN(*)	1	Compute arcsine in radians
ATAN	1,2	Compute arctangent in radians
ATAND	1,2	Compute arctangent in degrees
COS	1	Compute cosine of argument in radians
COSD	1	Compute cosine of argument in degrees
COSH(*)	1	Compute hyperbolic cosine
EXP	1	Compute e (=2.71828...) to power of arg
LOG	1	Compute natural logarithm of argument
LOG10	1	Compute common logarithm of argument
LOG2	1	Compute logarithm to base 2
RANDOM(*)	0	Compute random number in interval (0,1)
SIN	1	Compute sine of argument in radians
SIND	1	Compute sine of argument in degrees
SINH(*)	1	Compute hyperbolic sine of argument
SQRT	1	Compute square root of argument
TANH(*)	1	Compute hyperbolic tangent of argument

(*) Function not in ANSI standard

Function	#Args	Meaning
**********		STRING-HANDLING BUILT-IN FUNCTIONS
AFTER	2	Return string following pattern string
BEFORE	2	Return string before pattern string
COLLATE	0	Return ASCII collating sequence
COPY	2	Perform concatenations of string with itself
EVERY	1	Test whether every bit is on
FLTED(*)	3-5	Format floating point number
HIGH	1	Return highest character in collating sequence
INDEX	2	Search for pattern string
LENGTH	1	Return length of string
LOW	1	Return lowest character in collating sequence
REVERSE	1	Reverse string
SOME		Test whether any bit is set
STRING	1	Concatenate elements of NONVARYING string array
SUBSTR	2,3	Take substring of string
TRANSLATE	2,3	Translate characters in string
UNSPEC	1	Return BIT-string representation of argument
VERIFY	2	Verify validity of characters in string
**********		ARRAY BUILT-IN FUNCTIONS
DIMENSION	2	Return size of array dimension
HBOUND	2	Return upper bound of array dimension
LBOUND	2	Return lower bound of array dimension
**********		STORAGE CONTROL BUILT-IN FUNCTIONS
ADDR	1	Return address (POINTER data type) of argument
ALLOCATION	1	Return # of allocations of CONTROLLED ident
NULL	0	Return null POINTER value
**********		MISCELLANEOUS BUILT-IN FUNCTIONS
DATE	0	Return current date as character string
ONMSG(*)	0	Return error message for error invoking ON-unit
TIME	0	Return current time as character string
**********		PSEUDO-VARIABLES
RANDOM(*)	0	Initialize random number generator
STRING	1	Assign to concatenated NONVARYING string array
SUBSTR	2,3	Assign to substring of string
UNSPEC	1	Assign to bit string representation of argument

(*) Function not in ANSI standard

INDEX