CP-6 Introduction to PL-6

D00

XP03-00

June 1988

Preface

This document is an introduction the the PL-6 system programming language used to create and maintain the CP-6 Operating System.

Computer Aided Publication (CAP) is an advanced electronic technical publishing system. The Honeywell Bull Los Angeles Development Center (Product Development Organization) Documentation Services group authors, edits, reviews, and creates laser print masters with integrated text and graphics using CAP. This manual is a product of CAP.

Readers of this document may report errors or suggest changes through a STAR on the CP-6 STARLOG system. Prompt response is made to any STAR against a CP-6 manual, and changes will be incorporated into subsequent releases and/or revisions of the manuals.

The information and specifications in this document are subject to change without notice. Consult your Honeywell Bull Marketing Representative for product or service availability.

Copyright (c) Honeywell Bull Inc., 1988

File No.:

Table of Contents

Section 1.	PL-6 IN	N A N	JTSH	ELL										•												1-1
Basic Form																										
Declaratio																										
Based Data																										
Assignment	Staten	nents	•	•	•										-	•										1-6
Control St																										
Procedure	Related	i Stai	eme	nts	3					•																1-9
Scope of V																										
Built-in F	unction	ns and	d Su	bro	out	in	es		•				•	•	•	•						•				1-11
Input and	0utput	• •		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	1-14
Appendix A.	Exampl	e Pro	gra	m	•	•	•	•	•	•	•	•		-	•	•	•	•			•	•	•	•	•	A-1
Appendix B.	PL-6	Reser	ved	Κe		or	ds		_			_														R_1

XP03-00 iii

About This Manual

This manual is a beginner's guide to the PL-6 programming language. It was originally prepared for college course-level instruction at IUP (Indiana University of Pennsylvania), a CP-6 educational site. It is currently in a form appropriate for that environment, but someday, the section on I/O at the end will be redone to reflect proper usage of CP-6 Monitor Services directly from the CP-6 system.

XP03-00

Syntax Notations

	Table 1. Notation Conventions Table
Notation	Description
Lowercase	
	Lowercase letters identify an element that must be replaced by a user-selected value. For example, in a TEXT macro the value:
	"figure title" indicates the user supplies a value for figure title.
CAPITAL LET	TERS
	Capital letters indicate a literal, to be entered as shown. For example:
	FIG "figure title" indicates that the value FIG must be entered as shown.
Special Cha	racters
	The character ^ has a special function and is to be entered as shown. Likewise the combination of characters: is used to introduce CAP macros and must be entered as shown. The period is used to introduce TEXT control words.
Carated Let	ters
	Letters inside carats (< >) identify physical keys on the terminal. Carats are not typed. The indicated keys are pressed.
	pi esseu.

XP03-00

Notation	Description
Brackets	
	Brackets are used to enclose an optional element. If more than one element is enclosed in brackets, the notation indicates an optional choice. Multiple elements in brackets can be stacked or separated with the OR bar. For example:
	[id] indicates that a value for id may be entered.
	<pre>[id] or [idIA] indicates that a value for id or [A] the literal A may be entered.</pre>
	When enclosing keywords, brackets signify that all or part of the bracketed portion may be entered.
	K[EY] indicates KEY can appear as K, KE, or KEY.
Braces	
	Braces are used to enclose a required choice: one of the elements enclosed in the braces must be selected. The elements can be stacked or separated with the OR bar. For example:
	<pre>{A } or {Alid} both mean that either the {id} literal A or a user-supplied</pre>
OR Bar	
	The OR bar separates elements enclosed in braces or brackets from which one may or must be chosen.
	<pre>{Alid} means that either the letter A or the value of id must be entered.</pre>

	Table 1. Notation Conv	ventions Table (cont)
Notation	Description	
Horizontal	Ellipsis	
	The horizontal ellipsis repeated. For example:	indicates that an element can be
	#data#data#data	indicates that the user can supply multiple data fields, each one separated from the preceding field by the #.
Vertical El	lipsis	
	The vertical ellipsis in omitted purposely. For	ndicates that something has been example:
	This is the first li	ine of text
	•	indicates that text lines have been omitted.
	This is the last lin	ne of text.

Section 1

PL-6 IN A NUTSHELL

PL-6 is the CP-6 system implementation language. It provides for efficient block structured programming of tasks that are closely related to the operation of the system, for example, compilers, assemblers, simulators, and communication packages. PL-6 has a variety of data types, flexible control statements, and a simple syntax. It also provides easy access to the monitor services of CP-6.

Basic Forms

A PL-6 program is organized into procedures. One of these procedures is designated as the MAIN procedure or program. Other procedures can be considered subroutines. These procedures may be declared inside or outside the main procedure. All procedures begin with a PROC statement and end with an END statement.

Identifiers in PL-6 may be formed using any of the following characters: A through Z, O through 9, #, \$, a, and _. The first character must be alphabetic and the number of characters less than 31. Each statement in a procedure must be ended with a semicolon (;). Each statement may have a label. A label has the same form as an identifier, must be to the left of the statement, and must end with a colon (:). Commas are used to separate portions of certain statements and the apostrophe is used to delimit string constants.

The arithmetic operators are:

- + (add)
- (subtract)
- * (multiply)
- / (divide)

There is NO exponentiation.
There are NO floating point values.

The relational operators are symbolic:

- = for equal
- < for less than
- > for greater than
- <= for less than or equal
- >= for greater than or equal
- ~= for not equal

The logical operators are:

```
~ for "not"
```

- I for "or"
- & for "and"
- \ for "exclusive or"

The keywords "NOT", "AND", and "OR" may also be used with truth-value expressions in IF and DO statements.

Logical operators work on bit-string values, performing bit-by-bit operations.

NOT, AND, and OR work on bit-strings and evaluate to truth values; if ANY bit of a named variable is 1, its truth value is TRUE.

Parentheses and the equal sign (=) have the usual meaning in assignment statements. The beginning and end of a comment are denoted by "/*" and "*/", respectively. The symbols "->" are used in associating a pointer with a based variable.

Declarations

All identifiers (names) must be declared using the DCL statement. All DCL statements for a procedure must appear at the beginning of the procedure. A declared identifier is known in all procedures contained within the procedure in which it is declared. Only ONE variable, array, or structure can be declared with each DCL statement. The forms of the DCL statement are shown below.

```
DCL id [attributelist]; To declare a variable named "id".
```

```
DCL id (0:b) [attributelist]; To declare the array "id" that has elements 0 to b.
```

DCL level1 id1 [attributelist1] [, leveln idn attributelistn];

To declare a structure. The top level (level1) is 1; other levels (leveln) may 2 through 10. This is similar to a COBOL record, except that levels may not be skipped. Also, * may be used in place of an identifier name for an "idn" in order to denote "filler".

There are four types of attributes that may appear in an attribute list in a DCL statement: data type, alignment, storage class, and special. A data type should be declared for each identifier (except for the level 1 name in a structure). The data type attributes are listed below.

Attribute	Meaning	Default Alignment
SBIN(n)	Signed binary integer	ALIGNED
UBIN(n)	Unsigned binary integer	ALIGNED
CHAR(c)	Character string	CALIGNED
BIT(b)	Bit string	UNAL
PTR	Pointer to another variable	ALIGNED

The "n" in parentheses after the attribute name is optional. If it is included, it represents the number of bits used to represent the integer. If "n" is not used, the default size is 36 bits. For SBIN and UBIN, a size may be specified by keyword instead of using "n". The form is "SBIN size" or "UBIN size" where "size" is WORD (same as n=36), HALF (same as n=18), or BYTE (same as n=9).

The "c" represents the number of characters in a character string. Each character of the string may be accessed individually through the use of several built-in subroutines. The "b" represents the number of bits in a bit string.

An alignment attribute may be specified for any identifier that has a data type. If an alignment other than the default is to be used, it must be specified in the DCL statement. The alignment attributes indicate how storage should be allocated for an identifier. The following list indicates the storage boundaries used for various alignments.

Attribute	Beginning and ending boundary unit
ALIGNED	Word
CALIGNED	Byte
DALIGNED	Even numbered word
HALIGNED	Half word
WALIGNED	Word (same as ALIGNED)
UNAL	Bit

XP03-00

The storage class of an identifier determines how permanent it is and where it is known. The storage class may be specified for any identifier. A list of most storage classes and their meanings follows.

Class	Meaning
AUTO	Storage is allocated for the identifier each time the procedure is called. Storage is released when the procedure is exited.
STATIC	Storage is allocated throughout the running of the program.
CONSTANT	Storage is allocated to contain a constant value. The value cannot be changed.
EXT	The identifier may be referenced outside the procedure in which it is declared.
DCB	Storage is allocated for a data control block. Other attributes cannot be used with this one.
SYMREF	Indicates that the identifier refers to the same location as the same identifier declared with SYMDEF in another procedure.
BASED	No storage is allocated for the identifier, only a storage template is made. The template is associated with storage through the use of a pointer variable.

The special attributes are used to indicate special properties associated with an identifier. The following is a list of the most important special attributes.

Attribute	Meaning						
SYMDEF	Specifies that the identifier may be referred to with a SYMREF in another procedure.						
REDEF	Specifies an alternate description for a previously declared storage area. Note: the sizes must match.						
ENTRY (n)	Specifies that the identifier is an external procedure name. The "n" represents the number of parameters the procedure has; "(n)" can be omitted if it is zero.						

ALTRET Specifies that an external procedure can have an ALTRETURN.

INIT Specifies starting values to be assigned to identifiers. An array may be initialized by listing individual elements or by specifying entries of the form "value*r" to repeat the "value" r times. Note: "value*0" repeats the value enough times to fill the unspecified portion of the array.

Based Data Structures

Most languages associate one or more specific memory locations with each named data item. This is also true of most data items in PL-6. However, PL-6 also provides the capability of defining simple or complex data structures that are not directly associated with any memory locations. Such data items are said to be "based" and have a declaration similar to the following one.

```
DCL ABC(0:9) CHAR(1) BASED;
```

ABC is declared to be an array of 10 single characters, but no storage is allocated to hold the array. Instead, only the template (or form) of the array is set up. The template is associated with memory locations through the use of a pointer variable. For example:

```
DCL P$ PTR;
DCL STR CHAR(10) STATIC;
```

declares P\$ to be a pointer and STR to be a string of 10 characters.

By assigning P\$ the address of STR, the ABC structure can be overlayed onto the locations allocated to STR. P\$ is assigned the address by a statement such as:

```
PS = ADDR(STR);
```

(See the Built-in Functions section for a description of ADDR.) Then the following statement shows a pointer-qualified reference to ABC which causes an asterisk to be put in the 4th element of ABC, that is, the 4th character of STR.

$$PS \rightarrow ABC(3) = '*'$$
;

Note: P\$ can also be used in association with other templates.

Pointer qualification can be explicit (as in the previous example) or implicit. Implicit qualification occurs when a pointer variable is associated with a based data item in the DCL statement, for example:

```
DCL XYZ(0:4) CHAR(2) BASED PS;
```

Here XYZ is a based variable that is automatically associated with the P\$ pointer, unless it is explicitly pointer qualified with another pointer. Thus, the following two statements are equivalent. Both set the 4th and 5th characters of STR to blank.

```
XYZ(2) = ' ';
P$ -> XYZ(2) = ' ';
```

Based data items and pointer variables are particularly useful in dealing with linked data structures.

Assignment Statements

The form of an assignment statement is:

target = expression;

The validity of such statements depends on the syntax of the expression and of the value types of the target and expressions. The following list describes the valid combinations of value types; where possible, the data type of the target is shown.

Target	Expression type	Special action					
SBIN or UBIN	Arithmetic	High order bits are lost if target field is too small.					
CHAR	Character string	If target shorter, truncate on the right. If target is longer, extend on the right with blanks.					
BIT or array or structure	Bit string	If target is shorter, truncate on the right. If target is longer, extend on the right with zero bits.					
PTR	Pointer	None, but value must be ADDR of a location.					

XP03-00

The following list indicates the order of precedence in evaluating an expression in an assignment statement. The list is in order from highest to lowest.

Order of precedence by operation symbol:

```
~, unary -, unary +
*, /
binary -, binary +
<, <=, >, >=, =, ~=
&
I, \
NOT
AND
OR
```

Control Statements

There are three statements of the control type in PL-6 that are of interest. They are IF, DO, and EXIT. Each of these is described below. The IF and DO have more than one form.

There are two forms for the IF statement:

```
IF exp THEN stmt;
          and
IF exp THEN stmt1; ELSE stmt2;
```

The "exp" is a bit-string or truth-value expression. The "stmt" in each IF may be almost any executable PL-6 statement or a DO block. Each "stmt" may have a label. For nested IF statements, each ELSE goes with the nearest unELSEd IF.

The DO statement has at least six forms that you may find useful. The simplest DO is not a control statement at all; it is a way of making a compound statement. Its form is:

DO:

Following the DO are a series of statements which are collectively treated as a single statement. The end of the series of statements is marked with an END statement. This collection of statements is referred to as a DO block and can usually appear anywhere a single statement can occur.

```
DO CASE (exp);
```

The "exp" is an arithmetic expression or variable, a UBIN value. This statement is followed with a series of CASE statements of the form:

CASE (const,const,...);

The "exp" in the DO CASE is evaluated and matched against each of the CASE constants ("const"). When a match is found, the statements associated with that CASE statement are performed.

An example is shown at the right.

Each "const" must be a constant at compile time. The series of CASE (M); CASE (M); J=J+1; CASE statements is terminated with an END statement. The "const" may be the word ELSE to cover all other cases. The example increments J by 1 if K is 0, by 2 if K is 1 or 3, and sets J to zero otherwise.

DO SELECT (exp);

The "exp" may be any type of expression, representing a UBIN, SBIN, CHAR, or BIT value. This statement is followed by a series of SELECT statements of the form:

SELECT (const,const,...);

Each of the constants ("const") must be of the same type as the "exp" in the DO SELECT statement. The "exp" is evaluated and matched against the constants. When a match is found, the statements associated with that SELECT statement are performed. As with the DO CASE, ELSE may be used in place of a constant in a SELECT statement.

DO var = exp1 TO/DOWNTO exp2 [BY exp3];

The "var" is a scalar variable (not part of a structure); "exp1" is the initial value for "var"; "exp2" is the limit for "var"; and "exp3" is the increment for "var". The default increment is +1 for TO and -1 for DOWNTO. Exp3 must be greater than 0 for the TO form and must be less than 0 for the DOWNTO form. This DO statement works like the FOR statement in FORTRAN, except that the test is made at the top of the loop. It is delimited by an END statement. (The DO statement in FORTRAN is implemented differently.)

DO WHILE (exp);

The "exp" may be a bit-string or truth-value expression. The DO WHILE statement is followed by a series of statements which are executed repeatedly as long as the "exp" is true. The series of statements is delimited by an END statement. The statements inside the DO WHILE loop may be any executable PL-6 statements. The check of the expression is made at the beginning of the loop.

DO UNTIL (exp);

The "exp" may be a bit-string or truth-value expression. The DO UNTIL statement is followed by a series of statements that are executed repeatedly until the "exp" is true. The series of statements is delimited by an END statement. The statements inside the DO UNTIL loop may be any executable PL-6 statements. The check of the expression is made at the end of the loop.

The EXIT statement has the form:

EXIT [label];

This statement is used to escape from a DO loop. The simple form of the statement (EXIT;) jumps to the statement after the END of the innermost DO which the EXIT statement is in. To get out of several levels of nested DO loops, the other form (EXIT label;) is used. The EXIT label; jumps to the statement after the END of the DO loop which has the specified label on its DO statement. The DO loops from which the EXIT escapes may be any of the forms of the DO described previously.

Procedure Related Statements

There are five procedure related statements that you will be using frequently. Each of them is described below after their format is shown.

entry: PROC MAIN/(parameterlist) [ALTRET];

Only one PROC in a program is the MAIN PROC. All other PROCs are subroutines and have parameter lists. The parameter list is a sequence of identifiers separated by commas. They correspond to the dummy parameters in a FORTRAN subroutine. The declarations of the dummy parameters in a procedure must match the declarations of the parameters in the procedure from which it is called. The "entry" is the name by which the procedure is called.

A PROC may be internal or external. An internal PROC is wholly contained within another procedure. Internal procedures are known only to the procedures they are contained within. Internal procedures cannot be recursive. An external procedure is not contained within any other PROC. To be performed, a PROC must be called; execution cannot "fall into" a PROC.

XP03-00 1-9

The use of "ALTRET" in the PROC statement enables the use of the ALTRETURN statement within the procedure to cause special action to be taken when control is returned from a call.

CALL ref [(parameterlist)] [WHENALTRETURN DO; ... END];

The CALL statement is used to invoke a procedure. The "ref" may be an entry point for a procedure or it may be the name of an EPTR variable (such variables are not described in this document). The arguments in the parameter list must match those in the definition of the PROC being called. The WHENALTRETURN clause specifies a DO block of statements that should be performed if the called procedure does an ALTRETURN. Both internal and external procedures can be invoked with the CALL statement. All parameters are passed by reference; that is, the main program variable and associated subroutine variable refer to the same location.

RETURN; and ALTRETURN;

These two statements are used to return control from a called PROC. For ALTRETURN, control is returned to the DO block specified with the WHENALTRETURN in the CALL statement and then continues with the statement after the CALL. For RETURN, control is returned to the statement after the CALL statement. If ALTRET is not specified in the PROC definition, ALTRETURN does the same thing as the RETURN. ALTRETURN is normally used to indicate an error condition within the called procedure.

END [id];

The END statement is used to mark the end of a procedure or D0 block. For a D0 block there is no "id". For a PROC, the "id" is the entry name of the procedure.

Scope of Variables

Variable names declared in a procedure are known in all procedures that are internal to the one in which the declaration is made. If the same variable name is redeclared in the internal procedure, the internal declaration overrides the outer declaration, but only within the internal procedure. Variables declared in procedures which are disjoint are unknown in the disjoint procedures. unless they are specially declared with the EXT or SYMDEF attributes.

The portion of a program in which a variable is known (is accessible) is referred to as the scope of the variable. As an example of scope, consider the following program:

FIRST: PROC MAIN; DCL A CHAR(1);

```
DCL B SBIN;

SECOND: PROC;
DCL A SBIN;
DCL C UBIN;

END SECOND;
THIRD: PROC;
DCL D CHAR(1);

END THIRD;
END FIRST;
```

Here, A refers to the same CHAR(1) location in FIRST and THIRD. In SECOND, A is a SBIN location that is unrelated to A in the other procedures. Variable C is known only in SECOND; variable D is known only in THIRD; variable B is known everywhere in the program.

Because of the way the scope of a variable can be used to pass values between nested procedures, internal procedures rarely have parameters.

Built-in Functions and Subroutines

There are many functions and subroutines built in to PL-6. These perform various tasks, such as string manipulation, base conversion, and pointer manipulation. The most important functions and subroutines that you will need are listed below and described briefly. Note: as with FORTRAN, functions are invoked by using their name and subroutines are invoked by the use of a CALL statement.

```
ADDR (ref)
```

ADDR is a function that returns a pointer value that identifies the storage location referred to by "ref". It is useful in assigning values to identifiers of type PTR. ADDR may be used in an INIT.

```
ASCBIN (char-exp)
```

ASCBIN is a function that converts a single character into a binary integer. The length of the "char-exp" must be one. The integer is returned as UBIN(36).

BINASC (exp)

XP03-00 1-11

BINASC is a function that converts the low order 9 bits of the value of the arithmetic "exp" to a single character. The "exp" may be UBIN or SBIN. The returned result is CHAR(1). Related function: BITASC which works with a bit expression rather than a numeric one.

MOD (div,dsr)

MOD is a function that returns the remainder when "div" is divided by "dsr". Both "div" and "dsr" may be arithmetic expressions. The remainder has the same sign as the "div".

PINCRW (ptr, nword)

PINCRW is a function that returns a pointer value equivalent to incrementing the pointer (ptr) by a specified number of words (nword). The value of nword may be positive or negative. Related functions: PINCRC to increment a pointer by a specified number of characters and PINCRB to increment a pointer by a specified number of bits.

POFFW (ptr1,ptr2)

POFFW is a function that returns a signed value that represents the positional relationship in words between two locations pointed to by ptr1 and ptr2. The pointers must point to locations in the same data segment and the bit and character offsets of the pointers must be the same. Related function: POFFC returns a value that represents the positional difference in characters between the two pointer values.

CALL BINCHAR (str, val);

BINCHAR is a subroutine that converts a binary integer to a character string with no spaces and no sign characters. The "str" is the resultant string and must be of type CHAR. Leading zeros are supplied to fill the field if necessary. The "val" may be of type SBIN or UBIN; it may also be an arithmetic expression that evaluates to one of these types. If the result does not fit in "str", it may be truncated or an error may occur. Related subroutine: BINXCHAR which puts a leading sign character on the character string.

CALL CHARBIN (ref, str);

1–12 XP03–00

CHARBIN is a subroutine that converts an unsigned character string of decimal digits to a positive binary value. The "ref" is the location to contain the binary value; the "str" is the CHAR variable containing the character string. If there is an illegal character in the string, an error occurs. If the binary value will not fit in the ref location, it may be truncated or an error may occur. Related subroutine: XCHARBIN which accepts a sign character at the beginning of the character string.

CALL CONCAT (tostr, str1 [, str2]...[, str6]);

CONCAT is a subroutine that concatenates up to six strings to form a single string. The "tostr" is the destination, the character location where the concatenated string is to be stored. Each of the "str"s is a character expression that is to be concatenated. If "tostr" is larger than the formed string, it is right-filled with blanks. If it is smaller, the concatenated string is truncated on the right. If truncation occurs, an ALTRETURN is used and may be handled with a WHENALTRETURN.

CALL INDEX (index, str1, str2, [, [start] [,mask]]);

The INDEX subroutine locates the first or next occurrence of a string within another string using a left to right scan. The "index" is the character position number at which the found string begins. The "str1" is the string which is to be found; "str2" is the string to be searched. If "start" is specified, it must be in the range 0 to the length of "str2". If "str1" is not found, the index is set to the length of "str2" and ALTRETURN is taken, if WHENALTRETURN is specified. A "mask" may be specified if only portions of the characters are to be compared. The "mask" is specified as a literal value and indicates which bits of the characters are to be considered. Related subroutine: INDEXR which uses a right to left scan to locate the occurrences of a substring.

SUBSTR (str1, start [,len])

SUBSTR is a function that extracts part of a character string for use in a character expression. The "str1" is the string from which the substring is to be extracted. The "start" is the character position of the first character in the substring. The "len" is the length of the substring. When "len" is not specified, the substring is assumed to begin at "start" and go to the end of "str1". SUBSTR may also be used on the left side of an assignment statement to replace a portion of a string with the value of a character expression.

CALL INSERT (tostr, [start], [len], str1 [,str2]...[,str6]);

XP03-00 1-13

The INSERT subroutine allows up to six strings to be concatenated and then inserts the resultant string into another string. Each of the "str"s is used in the same way as for the CONCAT subroutine. The "tostr" is a reference to the string into which the insertion is to be made. The "start" indicates where in "tostr" the insertion is to be made and "len" indicates how many characters are to be inserted. The sum of the values of "start" and "len" must be less than the length of "tostr". Truncation and blank filling occur based on the size of "len" and the concatenated expression in the same way as for CONCAT.

Input and Output

Input and output in PL-6 must be done with the aid of monitor service calls. All monitor service calls are done with the CAll statement and each of them has a name that begins with M\$. The principal monitor services that you will need are involved with opening, reading, writing, and closing a file.

Many of the monitor service calls require the set up of a rather elaborate data structure. There are macros defined to make the creation of these data structures as simple as possible. The macros are kept in several files under the account :LIBRARY. To gain access to the macros all you have to do is place the following statement in your PL-6 program:

%INCLUDE CP_6;

1-14

Because the macros contain a number of declarations, you should place the statement among the DCL statements in your program, perhaps right at the end of your own DCLs.

Each of the macros is invoked by using a statement of the form:

%macroname [(refid=string[,refid=string]...)];

Many of the macros that you will need have names of the form %FPT_name, where FPT stands for Function Parameter Table, and "name" is the name of the associated function. For example, %FPT_OPEN is used in conjunction with opening a file. The function parameter table is the data structure the monitor service call requires to perform the function. Each monitor service call takes one parameter, the name of the FPT that is associated with the call.

The "refid"s in each invocation of the macros depend on the function to be performed. One "refid" that can be used with each FPT macro is FPTN which is used to specify the name of the FPT structure. This name can be used in other references to the structure. The following list shows some of the most common macros, the associated monitor service call and the "refid"s that are usually specified. Nearly all "refid"s have default values which are used if they are not specified.

Macro	M. Service	Refids and strings
FPT_OPEN	MSOPEN	DCB=dcbname, NAME=variable holding the file name, ASN=FILE for disk files, FUN=function (IN, UPDATE or CREATE), ACCT=variable holding account name, EXIST=status (NEWFILE or OLDFILE)
FPT_READ	MSREAD	DCB=dcbname, BUF=variable used as an input buffer
FPT_WRITE	M\$WRITE	DCB=dcbname, BUF=variable used as an output buffer
FPT_CLOSE	M\$CLOSE	DCB=dcbname

Examples of each of these macros and the associated monitor service calls are given in the attached sample program. An additional macro and monitor service call is used in the sample program to change the prompt used when the system attempts to get input from a terminal See the PROMPT PROC.

Certain "refid"s take variables as values, for example NAME and ACCT above. There are macros available to be sure that such variables are in the proper form. Some of these are referred to as VLPs (Vector located parameters). The macro used to handle the NAME refid is %VLP_NAME. There is an example of its use in the sample program.

Because of the complexity of I/O in PL-6, you may wish to use FORTRAN subroutines to handle the I/O in your programs. It is not difficult to do this; however, there are a few special things that need to be done. The following statements must be put in the PL-6 main program.

```
DCL XFF_7INITL ENTRY CONV(0) ALTRET;
```

CALL XFF_7INITL;

XFF_7INITL is a FORTRAN initialization library routine. The CALL should be placed in the main program so that it is executed before a call to any FORTRAN subroutine.

All FORTRAN subroutine names should be declared using the ENTRY attribute, in the form:

DCL subr-name ENTRY (no-of-parameters);

FORTRAN functions cannot be used. The "no-of-parameters" is a maximum; the actual FORTRAN subroutine may have fewer. The attributes of the PL-6 identifiers must match the FORTRAN variable declarations. The following list shows the matching attributes.

XP03-00 1-15

FORTRAN PL-6

_____ _____

SBIN WORD

INTEGER REAL BIT(36) ALIGNED REAL DOUBLE PRECISION BIT(72) DALIGNED LOGICAL BIT(1) ALIGNED CHARACTER*n CHAR(n)

The second sample program illustrates the use of FORTRAN subroutines to do file I/O. Terminal I/O and printer output can also be done with FORTRAN subroutines.

NOTE:

FORTRAN subroutine names must be no more than 8 characters long. Longer names are not handled properly when linking.

1-16 XP03-00

Appendix A

Example Program

```
EXAMPLE: PROC MAIN;
       /* This program does a linear search of a file whose
       name is read from the terminal. The file is in the form
       of a symbol table with mnemonic, opcode and type. The
       file is read in before the search begins. The prompt is
       changed for each bit of information that is requested from
       the terminal */
       DCL 1 OPREC (0:400) STATIC,
            DCL F$10 DCB;
       DCL M$SI DCB;
       DCL M$LO DCB;
       DCL COMMAND STATIC CHAR(31); /* terminal buffer */
       DCL INREC STATIC CHAR (12); /* file buffer */
       DCL DIGITS CHAR(3);
       DCL HOLD STATIC CHAR(6);
       DCL OUTLINE STATIC CHAR(30); /* output buffer */
       DCL NFOUND CHAR(9) CONSTANT INIT('NOT FOUND');
                    /* number of records */
       DCL N SBIN:
                         /* location of dot in file name */
       DCL MARK SBIN; /* location of dot in f DCL FINI SBIN; /* end of input line */
       DCL LOOK STATIC SBIN INIT(-1); /* trial location */
       /* Setting up structures to be used by the monitor calls */
       %INCLUDE CP 6;
       %VLP_NAME(FPTN=SOURCE);
       DCL SACCT CHAR(8) STATIC;
       %FPT_OPEN (FPTN=OPSRCE,FUN=IN,ACCT=SACCT,ASN=FILE,
               NAME=SOURCE, DCB=F$10);
       %FPT_READ (FPTN=OPFILE, BUF=INREC, DCB=F$10);
       %FPT READ (FPTN=TTY, BUF=COMMAND, DCB=M$SI);
       %FPT_WRITE (FPTN=TERM, BUF=OUTLINE, DCB=M$LO);
       %FPT CLOSE (FPTN=OPFILEC,DCB=F$10);
```

xP03-00 A-1

```
/* Set the prompt; read the file name; find the dot separator
       and the trailing blank. Then separate the COMMAND string
       into its parts and open the file */
       CALL PROMPT('Enter file name: '):
       COMMAND = ' ':
       CALL MSREAD(TTY);
       CALL INDEX (MARK, '.', COMMAND);
       CALL INDEX (FINI, ', COMMAND);
       /* Set length of name string (.L#); put name string in the
       structure (.NAME#); and put account string in SACCT. */
       SOURCE.L#=MARK;
       SOURCE.NAME# = SUBSTR(COMMAND,0,MARK);
       SACCT = SUBSTR(COMMAND, MARK+1, FINI-MARK-1);
       CALL MSOPEN(OPSRCE);
       /* Read in all the records of the file and keep track of
       how many there were in N. Each record part is moved to the
       appropriate place in OPREC. ★/
       CALL MSREAD (OPFILE);
       N = 0;
       /* N counts the number of records in the file. The (1=1) test
       a dodge to create an infinite loop. */
READ: DO WHILE (1=1);
           OPREC.MNE(N) = SUBSTR(INREC,0,6); /* note start at 0 */
           DIGITS = SUBSTR(INREC,7,3);
           CALL CHARBIN(OPREC.OPCODE(N), DIGITS);
           OPREC.ITYPE(N) = ASCBIN(SUBSTR(INREC, 11, 1));
           N = N + 1;
           CALL MSREAD (OPFILE) WHENALTRETURN DO; EXIT READ; END;
           END:
        CALL M&CLOSE(OPFILEC);
       /* Change the prompt and loop through getting instruction
       names to look up */
       CALL PROMPT ('Enter mnemonic: ');
       DO WHILE (FINI > 0 AND LOOK ~= 0);
         COMMAND = ' ';
         CALL MSREAD(TTY);
         CALL INDEX(FINI, ', COMMAND);
         HOLD = SUBSTR(COMMAND,O,FINI);
         LOOK = 0;
```

A-2

```
/★ Search through the entries and print a response whether found
      or not */
        DO WHILE (OPREC.MNE(LOOK) ~= HOLD AND LOOK < N);
          LOOK = LOOK + 1;
          END;
         IF OPREC.MNE(LOOK) = HOLD THEN
             CALL BINCHAR(DIGITS, OPREC. OPCODE(LOOK));
             CALL CONCAT (OUTLINE, 'The code is ', DIGITS,' The type is '
                   BINASC(OPREC.ITYPE(LOOK)));
             END;
         ELSE OUTLINE = NFOUND;
         CALL MSWRITE(TERM);
       /* A search for an ADD instruction ends the loop because
       it is the first mnemonic in the OPREC table. Its entry
       number is zero. */
       END;
       /* Procedure to change the prompt on reads from the
       terminal. The prompt string is expected to be passed
       as a parameter to this procedure. */
PROMPT: PROC(DIS);
       DCL DIS CHAR(20);
       DCL DAT CHAR(20) STATIC;
       DCL SIZ SBIN STATIC;
       %FPT PROMPT(FPTN=DOIT, DCB=M$UC, PROMPT=DAT);
       CALL INDEX (SIZ, ': ', DIS);
       DAT = SUBSTR(DIS, 0, SIZ+1);
       CALL MSPROMPT(DOIT);
       RETURN;
       END PROMPT;
END EXAMPLE;
```

XP03-00 A-3

EX_WITH_FOR: PROC MAIN; /* This program does a linear search of a file whose name is read from the terminal. The file is in the form of a symbol table with mnemonic, opcode and type. The file is read in before the search begins. The prompt is changed for each bit of information that is requested from the terminal. The file is opened, read, and closed using FORTRAN subroutines. */ DCL 1 OPREC (0:400) STATIC, 2 MNE CHAR(6), /* Mnemonic for instruction */ /* Opcode for instruction */ 2 OPCODE UBIN, 2 ITYPE UBIN; /* Type for instruction */ DCL HOLD STATIC CHAR(6): DCL NFOUND CHAR(9) CONSTANT INIT('NOT FOUND'); DCL N SBIN; /* number of records */ /* end of input line */ DCL FINI SBIN; DCL LOOK STATIC SBIN INIT(-1); /* trial location */ DCL RDTTY ENTRY(1); DCL WRITELN ENTRY(4); DCL OPENF ENTRY; DCL RDFILE ENTRY (4); DCL CLOSEF ENTRY; DCL XFF 7INITL ENTRY CONV(0) ALTRET; DCL EOF SBIN STATIC INIT(0); /* end of file indicator */ /★ Set the prompt and read in the file name. Then, open the file with the subroutine. */ /* Initialize FORTRAN routines */ CALL XFF 7INITL; CALL PROMPT('Enter file name: '); CALL OPENF; /* Read in all the records of the file and keep track of how many there were in N. Each record part is moved to the appropriate place in OPREC. */ N = -1;/* N counts the number of records in the file. */ DO WHILE (EOF=0); N = N + 1;CALL RDFILE(OPREC.MNE(N),OPREC.OPCODE(N),OPREC.ITYPE(N),EOF);

CALL CLOSEF;

```
/★ Change the prompt and loop through getting instruction
      names to look up */
      CALL PROMPT ('Enter mnemonic: ');
      DO WHILE (FINI > 0 AND LOOK ~= 0);
        CALL RDTTY(HOLD);
        LOOK = 0:
      /\star Search through the entries and print a response whether found
      or not */
        DO WHILE (OPREC.MNE(LOOK) ~= HOLD AND LOOK < N);
          LOOK = LOOK + 1;
          END:
         IF OPREC.MNE(LOOK) = HOLD THEN
             CALL WRITELN(OPREC.OPCODE(LOOK), 0, OPREC.ITYPE(LOOK), 0);
         ELSE CALL WRITELN(0,NFOUND,0,1);
       /\star A search for an ADD instruction ends the loop because
       it is the first mnemonic in the OPREC table. Its entry
       number is zero. */
      END:
       /* Procedure to change the prompt on reads from the
       terminal. The prompt string is expected to be passed
       as a parameter to this procedure. */
PROMPT: PROC(DIS);
       DCL DIS CHAR(20);
       DCL DAT CHAR(20) STATIC;
       DCL SIZ SBIN STATIC;
       %INCLUDE CP_6;
       %FPT_PROMPT(FPTN=DOIT,DCB=M$UC,PROMPT=DAT);
       CALL INDEX (SIZ,':',DIS);
       DAT = SUBSTR(DIS, 0, SIZ+1);
       CALL MSPROMPT(DOIT);
       RETURN;
       END PROMPT;
END EX_WITH_FOR;
```

```
C
C
       Subroutine to open the file
C
       SUBROUTINE OPENF
       CHARACTER*30 FNAME
       READ 31, FNAME
31
       FORMAT(A30)
       OPEN(UNIT=10, FILE=FNAME)
       RETURN
       END
C
C
       Subroutine to read a record from the file
C
         STATUS is set negative on end of file.
C
       SUBROUTINE RDFILE(MNE, OP, TYPE, STATUS);
       INTEGER STATUS, OP, TYPE
       CHARACTER*6 MNE
       STATUS = 0
       READ (10,21,END=19) MNE, OP, TYPE
21
       FORMAT(A6, 1x, 13, 1x, 11)
       RETURN
19
       STATUS = -1
       RETURN
       END
C
C
       Subroutine to close the file
C
       SUBROUTINE CLOSEF
       CLOSE(10)
       RETURN
       END
C
C
       Subroutine to read a mnemonic from the terminal
C
       SUBROUTINE RDTTY(COM)
       CHARACTER*6 COM
       READ 20, COM
20
       FORMAT(A6)
       RETURN
       END
C
C
       Subroutine to print the results
C
       IND indicates whether to print MSG or CODE and TYPE
       SUBROUTINE WRITELN(CODE, MSG, TYPE, IND)
       CHARACTER*9 MSG
       INTEGER CODE, TYPE, IND
       IF (IND .EQ. 0) THEN
```

```
PRINT 20, CODE, TYPE

FORMAT(' The code is ',13,' The type is ',11/)

ELSE

PRINT 21, MSG

FORMAT(' ',A9/)

ENDIF

RETURN
```

XP03-00 A-7

Appendix B

PL-6 Reserved Keywords

ADDR	DALIGNED	IF	PALIGND	UBIN
ALIGNED	DBASE	IN	PALIGNW	UNAL
ALTRET	DBOUND	INDEX	PINCRB	UNINHIBIT
ALTRETURN	DCB	INDEX1	PINCRC	UNTIL
AND	DCBADDR	INDEX1R	PINCRW	UNWIND
AREADEF	DCBNUM	INDEX2	POFFC	USE
AREAREF	DCL	INDEX2R	POFFW	
ASCBIN	DO	INHIBIT	PRIVY	VBASE
ASCBIT	DOMAIN	INIT	PROC	VBOUND
ASYNC	DOWNTO	INSERT	PSTART*	VFLAGS
AUTO	DSCR		PTR	VECTOR
AVOID	DSTART*	LENGTHB		VTYPE
	DTYPE	LENGTHC	READONLY	
BASED		LENGTHV	REDEF	WALIGNED
BINASC	EDITCHAR	LENGTHW	REMEMBER	WHENALTRETURN
BINBIT	EDITSTR		RETURN	WHENRETURN
BINCHAR	EDITXCHAR	MAIN		WHILE
BINXCHAR	ELSE	MATERIALIZE	SBIN	WORD
BIT	END	MAXIMUM	SEARCH	WSTART*
BITASC	ENTADDR	MINIMUM	SEARCHR	
BITBIN	ENTRY	MOD	SELECT	XCHARBIN
BITDSCR	EPTR		SIZEB	XLATE
BITVECT	ERASE	NIL	SIZEC	XLATE_6_TO_9
BY	ESTART*	NOAUTO	SIZEV	XLATE_9_T0_6
BYTE	EXIT	NOT	SIZEW	
	EXT		SPOIL	
CALIGNED	EXTROOT	OFF	STATIC	\$PR5
CALL		ON	SUBSTR	\$PR6
CASE	FOR	OPTIONAL	SYMDEF	\$PR7
CHAR		OR	SYMREF	\$X5
CHARBIN	GOTO			\$X6
CONCAT			THEN	\$X7
CONSTANT	HALF		TO	
CONV	HALIGNED		THRU*	
CPTR	HSTART*			
CSTART*				

 $[\]star$ These keywords are not part of PL-6 yet; but they will probably be adopted in the near future.

XP03-00 B-1

Index

```
Assignment Statements - 1-6
В
  Based Data Structures - 1-5
  Basic Forms - 1-1
  Braces - viii
  Brackets - viii
  Built-in Functions and Subroutines - 1-11
С
  Capital Letters - vii
  Control Statements - 1-7
D
  Declarations - 1-2
Н
  Horizontal Ellipsis - ix
I
  Input and Output - 1-14
0
  OR Bar - viii
```

XP03-00

```
P
Procedure Related Statements - 1-9

S
Scope of Variables - 1-10
Special Characters - vii

V
Vertical Ellipsis - ix
```

i-2