The Q1/LMC Systems Software Manual

Preliminary Edition

Table of Contents

PL/1

Entering, Altering, and Running Your Program

Entering a Program

To enter a program, turn on the machine and insert a PL/1 programmers disk into the machine. The display will read: "Q1/LMC AT YOUR SERVICE." To tell the machine that you want to enter a program, type:

```
EDIT,N FILEA
```

If you make a mistake, back up the cursor (the bright little square) by pushing the COR key. There are many other keys that allow for correction of mistakes and efficient entry of data. Refer to the section on the editor for a description of these keys.

EDIT is the program which takes text (your program) from the keyboard and puts it on the disk. FILEA is the name of the file on which the program will be put. ",N" after EDIT means that it will be a new program. That means that any old program which happened to be in FILEA will be discarded.

Now press the return key. The command will disappear and you can begin entering your program. Type in:

```
GET SKIP LIST(X);
PUT SKIP LIST(X);
END;
```

Push the return key after each line.

This is a simple program to read a number from the keyboard and print it back on the printer. You will learn the exact meaning of these statements later. When you have entered all three lines, push the F7 key to indicate that you are finished entering the program. The system will respond: "Q1/LMC AT YOUR SERVICE," meaning it wants more work to do. Now type:

```
PRINT FILEA
```

Push return. The system will print what has just been entered and you may inspect it.

Running a Program

Type:

```
PL1 FILEA
```

Now your program will be compiled and put into execution. You may type in any number and it will be printed by the printer as soon as you push the return key. At the same time, the display will again say: "Q1/LMC AT YOUR SERVICE," because the program is finished. Try this repeatedly, beginning at PL1 FILEA.

Altering a Program

Type:

```
EDIT FILEA
```

Note the absence of ",N" after EDIT this time. The first line will be displayed: "GET SKIP LIST(X);." Push the return key again to get the next line: "PUT SKIP LIST(X);." The cursor will appear over the "P." Now push the CHAR ADV key until the cursor appears over the ")" and type:

```
+X);
```

To insert a line after the one just modified, push F2 and type:

```
PUT SKIP LIST(X*X);
```

followed by F1 to indicate that you are done with insertion and F7 to indicate that you are done modifying the program. Now print the modified program by entering:

```
PRINT FILEA
```

It should read:

```
GET SKIP LIST(X);
PUT SKIP LIST(X+X);
PUT SKIP LIST(X*X);
END;
```

Now let's try it: PL1 FILEA. When you enter a number, two-times-it and its square will be printed.

You now have the skills necessary to enter and alter a simple program.

Basic Arithmetic and Input/Output

A Simple Program

Let's take another look at the program we entered in the last section.

```
GET SKIP LIST(X);
PUT SKIP LIST(X+X);
PUT SKIP LIST(X*X);
END;
```

There are four statements here; a statement being the text between two semicolons. These statements are executed sequentially by the system starting with the GET statement, which is the statement executed when a number is being typed into the system. X is a variable; it is used by the program to store a numerical value. A variable may be made up of letters, numbers, $, _, or   . The following are valid variable names:

digits

```
AMOUNT
GROSS_INCOME
Q1
ACCOUNT
X
```

We will avoid variable names that begin with I, J, K, L, M, or N for the time being because PL/1 will assign a special meaning to them.

The PUT statement is used to output data. In the program above, X+X and X*X are arithmetic expressions. "*" is used for multiplication and "/" is used for division. The following are valid arithmetic expressions:

```
A*(B+C)
2 – VALUE
STUFF + PROFIT/2.78
PRICE/EARNINGS
```

The expressions are evaluated by the ordinary rules of algebra. Thus, in the first expression, B and C are added and the result is multiplied by A.

Assignment Statements

As we have seen, the results of an arithmetic expression can be printed by use of a PUT statement. A programmer often has occasion to save the results of an expression in a variable for later use. This can be done with an assignment statement, which consists of a variable, an "=" sign, an arithmetic expression, and a semicolon. Consider the following sequence of assignment statements:

```
AMOUNT = 6.3;
G = AMOUNT – 5;
Z = G*(AMOUNT/3);
```

The first gives AMOUNT the value of 6.3. The second subtracts 5 from 6.3 to get 1.3 which it puts in G. The last divides 6.3 by 3 to get 2.1, which is multiplied by 1.3 to get 2.73, which is stored in Z.

The following is a program that uses assignment statements:

```
GET SKIP LIST (SPEED, TIME, GASMILEAGE);
DISTANCE = SPEED*TIME;
GAS = DISTANCE/GASMILEAGE;
PUT SKIP LIST (GAS, DISTANCE);
END;
```

More about GET and PUT

In our first program, the two PUT statements, PUT SKIP LIST (X+X); and PUT SKIP LIST (X*X); printed results on two lines. If we wanted to print them on the same line, we could write:

```
PUT SKIP LIST (X+X);
PUT LIST (X*X);
```

or else

```
PUT SKIP LIST (X+X, X*X);
```

This is the case because PUT SKIP means advance to a new line on the printer. GET SKIP means advance to a new line of input. The data typed before the return key is pushed is considered a line. SKIP (17) means advance 17 lines.

Non-numerical data may be printed by enclosing it in single quotes.

```
PUT SKIP LIST ('THE NUMBER YOU TYPED WAS' ,X);
```

Data may be output to the display instead of the printer as follows:

```
PUT FILE (DISPLAY) LIST ('FASTER THAN LIGHT');
```

Program Comments

Programmers often put in notes about how their programs work so that they or someone else can understand what the program is doing. These are called comments. Comments are ignored by the computer, which has no idea what the program is supposed to be doing and doesn't really care.

A comment is written between statements with a /* before the comment and a */ after it.

Some additions can now be made to the gas mileage program:

```
GET SKIP LIST (SPEED, TIME, GASMILEAGE);
   /* all 3 numbers may be on the same line */
DISTANCE = SPEED*TIME; /* in any physics course */
GAS = DISTANCE/GASMILEAGE;
PUT SKIP LIST ('YOU DROVE', DISTANCE, 'MILES AND USED',
   GAS, 'GALLONS');
END;
```

Note that the PUT statement is on two lines. A statement may occupy many lines or many statements may be on one line. The semicolons determine where the statements begin and end.

Control Statements

GO TO Statements

All of the programs that appeared in previous sections would go through the statements once and then "Q1/LMC AT YOUR SERVICE" would appear on the display. Suppose we wanted to modify one of these programs so that it would execute repeatedly. The program to get a number from the keyboard and print it would then be:

```
LOOP:          GET SKIP LIST (X);
               PUT SKIP LIST (X);
               GO TO LOOP;
               END;
```

Loop is a label. A label is used to identify a statement for reference by a GO TO statement. A colon must appear between the label and the statement it identifies. After the machine is finished with the PUT statement, it will execute the GO TO statement, which will cause it to go back to the first statement, that is to the statement identified by LOOP. Then the program will accept a new number from the keyboard. The program will never get to the END statement, so the operator must push the "restart" button in order to get out of this program.

IF Statements

Using the IF statement, it is possible to modify the previous program so that it will only print numbers greater than 22:

```
LABL:        GET SKIP LIST (X);
             IF (X> 22) THEN PUT SKIP LIST (X);
             GO TO LABL;
             END;
```

The symbol ">" means greater than. Arithmetic expressions may be used with it as well as numbers and variables, as in (X> Y/2). Note that THEN must be followed by a complete statement. ">" is called a comparison operator. The comparison operators are:

>	greater than
<	less than
=	equal
¬=	not equal
<=	less than or equal
>=	greater than or equal

ELSE Statements

ELSE is used where one action is to be taken when the relation after IF is true and another is to be taken if it is false. The following program will get two numbers from the keyboard and print the largest:

```
GET SKIP LIST (A,B); /*read 2 numbers from keyboard*/
IF (A>=B) THEN PUT SKIP LIST (A);
ELSE PUT SKIP LIST (B);
END;
```

The statement after THEN is executed only if the first number typed (A) is greater than or equal to the second number typed (B). The statement after ELSE is executed only if the statement after THEN is not executed. In this case, the statement after ELSE is executed only if A is less than B.

The GO TO statement is often used in conjunction with the IF statement. The above program can be made to repeat itself when A and B are not equal.

```
ANOTHER:    GET SKIP LIST (A,B);
            IF (A>=B) THEN PUT SKIP LIST (A);
            ELSE PUT SKIP LIST (B);
            IF (A¬=B) THEN GO TO ANOTHER;
            /*repeat if unequal*/
            END;
```

Character Data

Character Variables

In the section above, "More about GET and PUT," character data was printed by enclosing it in single quotes. This would not be adequate if we wanted to read in some character data from the keyboard and print it. For this sort of operation, we would need to define a variable in which the character data could be stored. This is done with the DECLARE statement:

```
DECLARE ALOTTADATA CHARACTER (20);
```

This statement will enable the program to use ALOTTADATA to store up to twenty characters. The position of the DECLARE statement does not affect the execution of the program, but it must appear before the first use of ALOTTADATA. The following abbreviation is allowed:

```
DCL ALOTTADATA CHAR(20);
```

The following program will read character data from the keyboard and put it on the printer twice:

```
DCL A CHAR(5);
GET SKIP LIST (A);
PUT SKIP LIST (A,A);
END;
```

If this program is run, and the characters AB$3 are typed, followed by the return key, the output will read: AB$3 AB$3
Blanks are added on the right if not enough data is typed-in to fill the character variable.

Comparison of Character Data

Character data may be compared in an IF statement the same way that numerical data was compared. For example, the previous program can be modified to end when STOP is typed in:

```
                DCL A CHAR(4);
LUNCH:          GET SKIP LIST (A);
                PUT SKIP LIST (A,A);
                IF (A¬='STOP') THEN GO TO LUNCH;
                END;
```

One string of characters is considered less than another if it would occur first in an alphabetical list.

Assignment Statements with Character Strings

Character data may be transferred from one variable to another with an assignment statement in much the same way as numerical values. If a character variable is not large enough to hold all of the characters transferred to it by the assignment statement, some characters will be lost from the end of the string. For example:

```
DCL A CHAR(9), B CHAR(6);
A = 'MANY MORE';
B = A;
```

will result in B receiving only the characters MANY M. Note that the blank between MANY and MORE is considered a character. Two variables are declared in one statement above by separating the two declarations by a comma.

Concatination

The concatination operator may be written with two vertical bars, ||, or CAT. It is used with an assignment statement to put several character strings together into one.

```
DCL A CHAR(2), C CHAR(6);
A = 'BB';
C = A CAT 'QU' CAT '4';
```

will give C the characters BBQU4. In a concatination statement, do not put the same variable on the left and right sides of the "=" sign. The results might be confusing.

SUBSTR

It is possible to use only a portion of a character string as a variable. This is called a substring. It is described by the word SUBSTR followed by a parenthesized list containing a variable name, the number of the first character to be used, and the number of characters to be used. Thus, SUBSTR (A,3,2) can be used to describe the third and fourth characters of A. For example:

```
DCL X CHAR(4), Y CHAR(9);
Y = 'ABCDEFGHI';
X = SUBSTR (Y,5,3);
```

will put EFG in X.

SUBSTR may also be used on the left side of the "=" sign:

```
SUBSTR (Y,6,2) = '$$';
```

will change Y from ABCDEFGHI to ABCDE$$HI. Make sure that there are at least as many characters on the right as on the left of the "=" sign when using this feature.

LENGTH

LENGTH is a function which is used to determine how many characters are in a character variable:

```
DCL A CHAR(92);
A = '4Q2';
X = LENGTH (A);
```

X will receive the value 3.

INDEX

INDEX is used to locate one string within another. For example:

```
DCL A CHAR(10);
A = '7$QMZC';
B = INDEX (A,'MZ');
```

The variable A will be searched for an M followed immediately by a Z. This occurs starting at the fourth character, so B will receive the value 4. B would receive the value zero if MZ were not found.

VERIFY

VERIFY is used to determine if one string is composed entirely of the characters in the second. It has the value -1 if so and 0 if not.

```
IF (VERIFY(A,'0123456789') = 0) THEN GO TO
   BADNUMBER;
```

could be used to determine if the character variable A contained all numerical characters.

Automatic Conversions between Numerical and Character Data

A character variable may be used in place of a numerical one and vice versa. The appropriate conversions will be made automatically.

The following program reads in a number, converts the decimal point to a comma, and prints out the results and the numerical value.

```
DCL A CHAR(10);
GET SKIP LIST (A);
Y = A; /*convert to number*/
X = INDEX (A,'.'); /*locate decimal point*/
SUBSTR (A,X,1) = ','; /*change to comma*/
PUT SKIP LIST (A,Y);
END;
```

If the input were "3.00" the output would be "3,00 3".

Files

Getting Ready to Use a File

In order to use a file, it is first necessary to name it. This is done with the DECLARE statement:

```
DCL MASTER FILE;
```

Recall that the DECLARE statement may appear anywhere in the program as long as it appears before the first use of the symbol.

Before the file is used, the program must execute an OPEN statement:

```
OPEN MASTER;
```

When this statement is executed, the system should have a disk with a file name MASTER in one of the drives or the following message will appear on the display: "MASTER NOT FOUND." For a description of how to put new files on the disk, refer to the section on the Disk Utility program.

The names of the files which are on the disk can be determined by giving a PRINT INDEX command to the system.

The READ Statement

The following program will read itself from the disk and print itself out, provided the program is in FILEA.

```
                DCL LINE CHAR(70);
                DCL FILEA FILE;
                OPEN FILEA;
LOOP:           READ FILE (FILEA) INTO (LINE);
                /*read one line from program*/
                PUT SKIP LIST (LINE);
                /*print the line*/
                IF (INDEX(LINE, 'END') = 0) THEN GO TO
                   LOOP; /*end of program?*/
                END;
```

The first time a READ statement is executed after an OPEN, the first record is read into the variable after INTO. The next execution of a READ gets the second record and so on. If the file is opened again, it will go back to the first record. For the time being, reading a file should be done into a character variable.

Access to File by KEY

Records in the middle of the file sometimes must be accessed without reading all of the records before it. This can be done with the READ KEY statement. For now, we will only consider keys at the beginning of the record. For example:

```
    READ KEY ('LOOP:') FILE(FILEA) INTO (LINE);
```

would get the fourth record of the program in the previous example.

When using character variables to specify keys, it is best to be sure that the key string has its maximum (declared)length.

Altering a Record

A record which has just been read may be modified using a REWRITE statement. For example, suppose we wanted to change the label LOOP to HATS when FILEA contains the preceding program:

```
READ KEY ('LOOP') FILE (FILEA) INTO (LINE);
SUBSTR (LINE,1,4) = 'HATS';
/*change the first 4 characters*/
REWRITE FILE (FILEA) FROM (LINE);
```

Generating a New File

If all of the data which is in a file is to be discarded and only new data is to be put into a file, the OPEN statement should be followed by the execution of successive WRITE statements. After the last WRITE, a CLOSE statement should be executed. The following program will write the numbers 1 to 20 into FILEB:

```
                DCL FILEB FILE, Q CHAR(10);
                OPEN FILEB;
                Q = 0;
LOOP:           Q = Q+1;
                WRITE FILE (FILEB) FROM (Q);
                IF (Q¬=20) THEN GO TO LOOP;
                CLOSE FILEB;
                END;
```

Adding Records to a File

In order to add records to an existing file, open the file, execute CALL SEOF (filename); statement and a WRITE statement for each record to be added and a CLOSE statement. Thus, if the records generated by the previous program were to be added to an existing file, it would be only necessary to add the statement:

```
CALL SEOF (FILEB);
```

after the OPEN statement.

The ON Statement

The program used to illustrate the READ statement had a rather awkward way of finding the end of a file which was dependent on the content of the last record. Using an ON statement will make this program a little more general-purpose.

```
          DCL LINE CHAR(80);
          DCL FILEA FILE;
          OPEN FILEA;
LOOP:     ON ENDFILE GO TO DONE;
          READ FILE (FILEA) INTO (LINE);
          PUT SKIP LIST (LINE);
          GO TO LOOP;
DONE:     END;
```

ON must always be followed by a GO TO statement. The ON statement must be executed before each input or output statement to which it will apply. There is one other type of ON statement besides ON ENDFILE: the ON ERROR statement. If an ON ERROR statement is executed before an input or output statement and an error occurs, a number indicating the type of error will be put into a predefined variable called ONCODE. These values are:

1 = Disk format error

2 = Read error

3 = Write error

4 = Key not found

5 = File not opened or disk removed since file was opened

6 = Program tried to access a record after the end of the file

7 = Program tried to write on a protected file

If a disk error occurs without an ON ERROR statement, the system will report the error to the operator and wait for a response. For this reason, it is not ordinarily useful for the programmer to provide for any error checks.

The following program will provide response to READ errors. The program allows the operator to enter a four-digit number and add to the account balance.

```
                DCL A CHAR(24);  /*The first 4 characters
                   are the account name followed by 2
                   blanks and the account balance*/
                DCL MASTER FILE, AK CHAR(4), AC
                   CHAR(4);
                OPEN MASTER;
ACCOUNT:        GET SKIP LIST (AC); /*Get account name*/
                PUT FILE(DISPLAY) SKIP;
                /*Clear display of error message*/
                ON ERROR GO TO BAD;
                READ KEY (AK) FILE (MASTER) INTO
                   (A);
                GET LIST (X); /*Change in balance*/
                X=X + SUBSTR (A,6,18); /*New balance*/
                SUBSTR (A,6,18) = X;
                REWRITE FILE (MASTER) FROM (A);
                GO TO ACCOUNT;
BAD:            IF (ONCODE=4) THEN GO TO KEYER;
                /*Key not found?*/
                PUT FILE (DISPLAY) SKIP LIST
                ('BAD RECORD');
                GO TO ACCOUNT;
KEYER:          PUT FILE (DISPLAY) SKIP LIST (AK,
                'IS NOT A VALID ACCOUNT');
                GO TO ACCOUNT;
                END;
```

Edited Output

PUT EDIT

When printing numbers, it is often desirable to print in formats other than the one offered by the PUT LIST statement. This can be done by converting the number to a character string and using the character string functions, but this is clumsy. For this reason, the PUT EDIT statement is offered. Its format is:

```
PUT SKIP EDIT(X) (P'99V.99') (Y) (P'99');
```

Each number is followed by a P and a character string in parentheses. This is known as a picture. The character V indicates the assumed location of the decimal point in the picture. If there is no V, it will be assumed to be at the right of the picture. The "." indicates where the decimal point is to be printed. Commas and blanks can appear also. Here are some examples:

Picture	Value	Output
99V.99	93.7	93.70
9,999V.9	1234.44	1,234.4
9V99.9	.123	012.3
9V,99	8.73	8,73
9999	3	0003

The number of character printed will always be the number of characters in the picture excluding V. You may have as many as 6 digits after the V or as many as 14 before. A picture may contain up to 32 characters.

Zero Suppression Character

In the last example, the leading zeros may be undesirable. The characters Z and * may be used in the picture to suppress the leading zeros. Z will replace leading zeros with spaces and * will replace the zeros with *'s. Zeros will not be suppressed to the right of V. Commas and periods will not print unless an unsuppressed number or V is to the left of it (see examples below).

Drifting Characters

A sequence of $'s may be used in a picture to suppress leading zeros and print a $ to the left of the first printed character. Similarly, a sequence of –'s will print a – to the left of the first printed characters if the value is negative. Examples are:

Picture	Value	Output
Z,ZZZ,ZZZV.99	1123	bbbb1,123.00
Z.V99	.01	bb01
ZV.99	.03	b.03
$$$9V.99	.1	bb$0.10
––––V.99	–8.2	b–8.20
––––V.99	.03	bbb.03
***V,99	7	**7,00

The "b" is used to indicate a blank.

Extra spaces may be printed before and after a number by using X followed by the number of spaces in parentheses before or after the picture:

```
PUT EDIT(A) (X(5), P'999', X(3));
```

Edited Output with Character Strings

Each character string should be followed by an A with the number of print positions in parentheses. The string will be padded on the right or truncated depending on its length:

```
DCL GRADES CHAR(4);
GRADES = 'FFFF';
PUT EDIT (GRADES) (A(6));
```

would output FFFFbb. If the A has no number after it, the entire string will be printed.

Column Headings and Page Headings

Column headings on records may be printed with PUT LIST statements. The decision to print a page heading may be made by counting the number of lines printed. The following sample program prints four numbers per line and 200 items per page. It prints headings every time it uses up an eleven-inch (66 line) page.

```
            LINES = 49; /*line counter*/
            ITEM = 3; /*count of items on line*/
            PAGE = 1; /*page count*/
```

```
PAGE    1 DATA BY QUARTER


FIRST   SECOND  THIRD   LAST
QUARTER QUARTER QUARTER QUARTER

   0.00    1.00    2.00    3.00
 456.23   89.70    1.10    0.01
   9.00    5.00    5.00    4.00
  36.00    5.00   85.00    6.00
  56.00   69.00    6.00   56.00
  85.00    5.00    5.00   25.00
  25.00    5.00   52.00   55.00
  52.00   52.00  451.00   63.00
   3.20
```

```
GETONE:    GET LIST(DATA);


           ITEM = ITEM + 1; /*count numbers on line*/
           IF ITEM¬=4 THEN GO TO NOHEAD;
              /*line full?*/
           PUT SKIP; /*next line*/
           ITEM = 0;
           LINES = LINES + 1; /*count lines on page*/
           IF LINES¬=50 THEN GO TO NOHEAD;
              /*end of page?*/
           PUT SKIP(10) EDIT ('PAGE') (A(6)) (PAGE)
              (P'Z9') (' DATA BY QUARTER') (A);
           PAGE = PAGE + 1;    LINES = 0;
           PUT SKIP(4) LIST
('  FIRST    SECOND     THIRD       LAST        ');
           PUT SKIP  LIST
('  QUARTER  QUARTER    QUARTER     QUARTER     ');
           PUT SKIP(2);

NOHEAD:    PUT EDIT(DATA) (P  'ZZZ9V.99');
              /*print one number*/
           GO TO GETONE;
           END; /*of program*/
```

The DECLARE Statement

Arrays

All of the programs in previous sections of this chapter have a variable name for each number or character string which is to be stored. This would be very awkward for a program that input 100 numbers before processing them. A group of 100 numbers can be defined under one variable name as follows:

```
        DECLARE X(100);
```

Which one of the 100 variables in the array (X) is specified by an expression in parentheses after the array name. This is called a subscript. To get a value for each element of X from the keyboard, the following program segment could be used:

```
                     Z = 1;
        HUNGRY:      GET SKIP LIST (X(Z));
                     Z = Z+1;
                     IF (Z¬=100) THEN GO TO HUNGRY;
```

Note that elements of X are numbered from 1 to 100. X could be given 100 elements numbered from 19 to 118 as follows:

```
        DCL X (19:118);
```

An array of character variables can be declared as follows:

```
        DCL CH(23) CHAR(5);
```

5 refers to the maximum length and 23 is the number of elements in the array. An element of an array may be selected by two or three subscripts if it is declared that way:

```
        DECLARE FROG (10,20);
        FROG (X,Y) = 79.3;
```

FROG will have 200 elements. The first subscript can vary from 1 to 10 and the second from 1 to 20.

Precision

Here is a program which will test the accuracy of the numbers we have been using:

```
        X = 1.234567891234
        PUT SKIP LIST (X); END;
```

It only prints the first nine digits: 1.23456789. That is because only nine decimal digits are reserved in memory for X. X is said to have a precision of nine. You may define a variable with a precision of 1 to 13 as follows:

```
DCL Q FLOAT(13);
```

FIXED and FLOAT Variables

In the above example, Q will be used to contain the thirteen most significant digits of whatever is stored into it. This is the meaning of the word FLOAT. The digits may occupy any position in relation to the decimal point. Variables with a fixed number of digits before and after the decimal point are declared FIXED.

```
DCL RICK FIXED(8,2), H(5) FIXED(4), E FIXED;
```

RICK will be a variable with 8 digits, two of which are after the decimal point. H will be an array with each element having four digits before the decimal point and none after, so it can only have integer values. E will be a nine-digit integer. FIXED variables are often used in accounting applications where fractions of cents are not desirable. They are also useful in representing numbers in files because they are more compact, not needing the extra memory to store the relative position of the decimal point.

BINARY Variables

In the section, "Basic Arithmetic and Input/Output", the programmer was advised against using variable names that begin with the letters I, J, K, L, M, and N. That is because they will be assumed to be BINARY unless they are declared otherwise. BINARY variables are faster and use less memory than decimal (FIXED or FLOAT) but they can only contain integers from –32767 to 32767. Therefore, they are excellent for subscripts, counters, etc. A variable that does not begin with the letters I through N may be declared BINARY:

```
DCL Z BINARY;
```

INITIAL

It is often desirable to have a variable have a certain value when the program is loaded. This can be done with INITIAL:

```
DECLARE X FIXED(4) INITIAL(325);
```

Arrays are initialized with a list of values:

```
DCL C(4) CHAR(10) INITIAL ('BOB','CAROL', 'TED','ALICE');
```

In initializing arrays, one of the elements in the list may be repeated by enclosing the number of repetitions in parentheses before the element:

```
DCL A(29) BINARY INITIAL (5,6,(27)0);
```

DO Statements

Indexed DO Statements

A DO Statement is used to group several statements together. This group may be executed repeatedly by DO statements with the following form:

```
DO I=1 TO 5;
X=X*X;
Y=Y*Y;
END;
```

X and Y will be squared 5 times. The END statement is used to designate the end of the DO group and another END will be required to end the program. The variable I has the value 1 for the first execution of the group and it is increased by 1 each time the group is executed. If we wanted to increase it by 2 each time, we would write:

```
DO J=1 TO 5 BY 2;
```

Then J would assume the values 1, 3, and 5. The numbers 1, 5, and 2 may be replaced by arithmetic expressions.

Nesting DO Statements

One DO may be put inside of another. For example, consider the following program to print a multiplication table:

indent

```
DO I=0 TO 9;
DO J=0 TO I;
PUT EDIT (I*J) (P'ZZ9');
END; /*of group for second (inside) DO*/
PUT SKIP; /*new line*/
END; /*of group for first DO*/
END; /*of program*/
```

The first DO statement specifies that the statements associated with it are to be executed ten times. Each of these ten includes 1 to 10 (depending on the value of I) repetitions of the PUT statement. The output will be:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
```

and so on.

Using DO Statements with IF Statements

The IF statement allows the conditional execution of one statement. If more than one statement is to be conditional, they may be grouped by DO. Thus we can halve a number until it is less than one as follows:

```
HALVE:      IF X¬=1 THEN DO;
            X=X/2;
            GO TO HALVE;
            END;
```

The GO TO will only be executed if X=X/2; is executed.

Functions and Subroutines

Mathematical Functions

The square root of a variable may be specified by the word SQRT followed by the variable in parentheses:

```
PUT LIST (SQRT(X));
```

SQRT is one of many mathematical functions which are automatically loaded with the program if it is used:

Name	Function
SQRT	square root
SIN	sine (radians)
COS	cosine
TAN	tangent
ASIN	arc sine
ATAN	arc tangent
ABS	absolute value
LOG	log base 10
LN	natural log
SINH	hyperbolic sine
COSH	hyperbolic cosine
ERF	error function
EXP	e^x

The accuracy of these functions is eight digits.

Calling Subroutines from the Library

One of the files on the PL/1 programmers disk (PL1LIB) contains subroutines which may be invoked by the CALL statement. For example, we have already used the SEOF subroutine to set a file to its end of file position:

```
CALL SEOF(FILEA);
```

If this statement appears in the program, the subroutine SEOF will automatically be loaded with the program and executed whenever the CALL statement is executed.

Subroutines Included in the Program

A subroutine may be written before the start of the main program and called by the program in the same way that subroutines may be called from the library file. The following subroutine will take two values and exchange them if the first is greater. The main program uses this subroutine repeatedly to put three numbers in order.

```
ORDER:      PROCEDURE(A,B);
            IF (A>B) THEN DO;
            C=A; A=B; B=C; END; /*exchange values*/
            RETURN;
            END;
MAIN:       GET SKIP LIST (X,Y,Z);
            CALL ORDER(X,Y);
            CALL ORDER(Y,Z); /*Z now has largest*/
            CALL ORDER(X,Y);
              /*in case last value entered was smallest*/
            PUT SKIP LIST(X,Y,Z);
            GO TO MAIN;
            END;
```

The first statement of any subroutine is a PROCEDURE statement. The label on the procedure statement (in this case ORDER) is used to refer to the subroutine in a CALL statement. The list of variables(A,B) following the word PROCEDURE is called parameters. The CALL ORDER(X,Y); statement will cause the subroutine to be executed using X and Y for the parameters A and B. The parameter names (A and B) may not be used outside the subroutine. The RETURN statement causes the computer to leave the subroutine and resume executing the calling program at the statement after the CALL which stated the subroutine execution.

Note that there are three end statements: one to end the DO group, one to end the subroutine, and one to end the program.

Attributes of Parameters

All of the variables and parameters in the previous example use the FLOAT data type and the 9-digit precision which are automatically assigned by the compiler.

In general, all attributes of the variables used in the CALL statement must be the same as the corresponding parameters in PROCEDURE statement for the subroutine. For example, suppose we want to have a subroutine which takes two fixed point numbers and puts them into a character string separated by a "?"

```
CRAM:     PROCEDURE(X,Y,C);
          DCL C CHAR(5), X FIXED(2), Y FIXED(2);
          C=X CAT '?' CAT Y;
          RETURN;
          END;
MAIN:     DCL CH CHAR(5), B FIXED(2), Q FIXED(2);
          CALL CRAM(B,Q,CH);
          .
          .
          .
          .
          END;
```

CH must be declared CHAR(5) and B and Q must have exactly two digits with none after the decimal point. The compiler does not check to see that the attributes of the variables used with the CALL match the parameters of the subroutine, so caution should be exercised here. It is not advisable to use constants in a CALL because it is difficult to determine what attributes the compiler will assign them.

Defining Functions in the Program

A function, which has one FLOAT parameter and returns a FLOAT result can be written as a procedure: :

```
COSH:          PRECEDURE(X);
               Y = EXP(X) + EXP(–X);
               RETURN (Y/2);
                  /*value of function is after RETURN*/
               END;
/*main program starts here*/
               DO Z = .1 TO 10 by .1;
               PUT SKIP LIST(COSH(Z));
               END;
               END;
```

Logical Operators

Comparison Operators

The operators >, <, ¬=, >=, =, and <= have been used with the IF statement. These operators actually may be used in expressions the same way that +, –, *, and / are used. The results of these operators is zero if the condition is false and –1 if it is true.

```
        PUT SKIP (3<2);
```

would print "0" since 3 is not less than 2.

These operators worked with the IF statement because the statement after THEN is executed if and only if the expression after IF is not zero. Thus we could say:

```
        A=3.2; IF (A) THEN GO TO TACO;
```

The GO TO would be executed.

And, Or, and Not

The symbols &, |, and ¬ correspond to the logical operations and, or, and not. These operations operate on each bit of the binary operands independently. For example, to determine the result of the expression 10&7, we would write each number in binary and form the result bit by bit keeping in mind that the result of an "and" is one only if both bits are ones:

```
        1 0 1 0          10
        0 1 1 1        &  7
        -------        ----
        0 0 1 0           2
```

The result is two.

Similarly, the ¬ operator placed before an expression complements each bit and the | operator "ors" each bit of two binary numbers.

Since the –1 generated by the comparison operators is represented by all ones in binary and the zeros by all zeros, all bits in the operations will behave the same and the comparison operators may be used in conjunction with logical operators such as IF (A > B) & (B >C) THEN . . . .

For example, A> B | A = B is the equivalent of A>=B and (A¬=B) is the same as ¬(A=B).

Structures

Declaring a Structure

A structure is an accumulation of variables, which are not necessarily the same type. For example, we could define a structure, ST, that contains a character variable and a number.

```
DCL 1 ST, 2 C CHAR(10), 2 X FLOAT;
```

C is a character variable which may be used like any other character variable, and X is a floating decimal number. ST is a name which can be used to refer to X and C together.

The numbers before each name are called levels. Levels must be single digit numbers. As in the above, when a structure is being declared which does not contain other structures, only levels 1 and 2 need be used. However, more levels are necessary for structures containing structures as in the following:

```
DECLARE 1 BIG,
             2 S,
                3 X FIXED,
                3 Y FIXED,
             2 T,
                3 C CHAR(5)
                3 Z(10);
```

In this case, BIG is a structure containing two other structures: one containing two fixed integers and one containing a character variable and a ten element floating decimal array.

Assignment Statements Using Structures

If two structures have identical layout:

```
DCL 1  S1,2  X,2  Y;
DCL 1  S2,2  A,2  B;
```

the information can be transferred from one to the other by a single assignment statement:

```
S1=S2;
```

Care should be taken that both structures contain variables with the same attributes in the same order.

Arrays of Structures

An array of structures is declared like other arrays:

```
DECLARE 1 STA(5),
          2X,
          2Y;
```

A reference to X or Y will be interpreted as X or Y of STA(1). In order to obtain X for STA(4), we could write:

```
STA(1)=STA(4); PUT LIST(X);
```

The more important uses of structures will be covered in the following sections.

Advanced Use of Files

Use of Structures with Files

Up to now, files with more than one data item in a record have been handled by reading records into character variables and separating the items with SUBSTR. This can be done more efficiently using structures.

Consider again the program on page ? which allowed the operator to type in an account name followed by a change in the account balance:

```
               DCL 1 A, /*Structure for records*/
               2 NAME CHAR(4), /*Account name*/
               2 BAL FIXED(10,2); /*Account balance*/
               DCL MASTER FILE,
               AK CHAR(4);
               OPEN MASTER;
ACCOUNT:       GET SKIP LIST(AK); /*Get account name*/
               READ KEY(AK) FILE(MASTER) INTO(A);
                  /*Put record into structure*/
               GET LIST(X); /*Change in account balance*/
               BAL = BAL+X;
               REWRITE FILE(MASTER) FROM(A);
                  /*Update record*/
               GO TO ACCOUNT;
               END;
```

The structure into which a record is read should have the same layout as the structure from which it was written.

The information is written into the file exactly the same way that it is represented in memory. Therefore, using the FIXED data type for the number is the most efficient use of disk space.

Use of UNSPEC with Files

UNSPEC is a function which allows the programmer to operate on the first two bytes of a variable as a binary number. When used with a file name, it refers to record number (starting with zero).

The following sequence would access the hundredth record of FILEA:

```
UNSPEC(FILEA) = 99;
READ FILE(FILEA) INTO (WIDGET);
```

Structures and Numbers as Keys

All previous examples used only character strings as keys. Numbers may also be used as keys, provided the corresponding number in the structure from which the record was written has exactly the same attributes.

Similarly, a structure may be used as a key, provided a corresponding structure is part of the structure from which the records were written.

An example of this usage appears in the next section.

Offset Keys

Until now, all keys have referred to the first variable in a record. A different position in the record is referred to by the word KEYTO followed by the variable in the structure which corresponds to the key in the record.

In the following example the records contain a first name, a last t name, and a number. By entering a number from 1 to 4, the operator specifies what variable the file is to be accessed by:

1	first name
2	last name
3	both names
4	number

```
             DCL 1 REC, /*Structure for record*/
             2 NAME, /*Structure within a structure*/
                3 FIRST CHAR(10),
                3 LAST CHAR(10),
             2 NUM FIXED;
             DCL MAIN FILE, X FIXED; OPEN MAIN;
REPEAT:      GET SKIP LIST(N); /*Type of access*/
             IF N<3 THEN GO TO ANAME;
             IF N=3 THEN GO TO BOTH;
             GET LIST(X);
```

```
                    READ KEY(X) FILE(MAIN) INTO(REC)
                       KEYTO(NUM);  /*Access by number*/
OUT:                PUT SKIP LIST(FIRST, LAST, NUM);
                    GO TO REPEAT;
ANAME:              GET LIST(FIRST);
                    IF (N=1) THEN READ KEY(FIRST) FILE
                       (MAIN) INTO (REC);
                       /*If it is a first name*/
                    ELSE READ KEY(FIRST) FILE(MAIN)
                       INTO (REC) KEYTO(LAST);
                    GO TO OUT;
BOTH:               GET LIST(FIRST, LAST);
                    READ KEY(NAME) FILE(MAIN) INTO
                       (REC);  /*Both names must match*/
                    GO TO OUT;
                    END;
```

Reading and Writing Arrays of Structures

If a READ or a WRITE statement uses an array of N structures, N records will be read or written. This is much faster than doing the operation in a loop, which would require a revolution of the disk for each record.

BASED and POINTER Variables

POINTER Variables

A POINTER variable is a variable which contains the address of another variable. It is declared as follows:

```
DCL P POINTER;
```

The ADDR Function

After it is so declared, a POINTER variable can be assigned the address of another variable using the ADDR function:

```
    P = ADDR(X);
```

BASED Variables

A BASED variable is not assigned any memory, but it uses whatever memory its associated POINTER variable happens to be pointing to at the time.

A BASED variable is declared by putting the word BASED after the variable name, followed by its associated POINTER variable in parentheses:

```
    DCL P POINTER, B BASED(P) FLOAT(8);
    P = ADDR(Y);
    B = 2; /*Y gets the value 2*/
    X = B+7;
    P = ADDR(X); /*B becomes X*/
    PUT LIST(B); /*result is 9*/
    END;
```

BASED Structures

If a structure is BASED, all of its elements will be BASED and they will move around in memory when the POINTER variable associated with the structure is changed.

A linked list is an ordered group of structures in which each structure contains a pointer to the next structure. The following partial program contains a linked list of single-character variables with a sequence of statements that will delete a character from the beginning of the list and another that will print the Nth character:

```
    /*Area in memory for linked list*/
    DCL 1 LIST(100),
       2 PX POINTER,
       2 CX CHAR(1);
    DCL PS POINTER; /*Pointer to start of linked list*/
    DCL Q POINTER; /*Pointer for THAT*/
```

```
DCL 1 THAT BASED(Q),
   2 P POINTER,
   2 C CHAR(1); /*Structure currently being accessed*/
      .
      .
      .
/*Delete one element of the list*/
Q = PS; /*THAT is now first structure of the list*/
PS = P; /*PS now points to what was second structure*/
      .
      .
      .
/*Print Nth character*/
Q = PS; /*To first of list*/
DO I=2 TO N;
Q = P; /*Move THAT to next structure of list*/
END;
      .
      .
      .
END;
```

Use of UNSPEC with Pointers

The memory location referenced by a pointer can be incremented by 2 bytes as follows:

```
UNSPEC(P) = UNSPEC(P)+2;
```

A pointer can be made to point to a fixed address as follows:

```
UNSPEC(P) = 23192;
```

Q1 Systems Software

Disk Utility

To Initialize a Disk

Insert a disk with the required file formats in disk drive no. 1 (the left-hand drive). Insert a new disk in drive no. 2 (the right-hand drive).

```
Key-in:   DISK INIT
```

All of the file names, track numbers, record lengths, sector headers, and the index from the disk in drive no. 1 will be copied onto the disk in drive no. 2. Note that the contents of the files are not copied.

To Copy a Disk

```
Key-in:   DISK COPY
```

This will copy the contents of all files that are in common from the disk in drive no. 1 to the disk in drive no. 2.

To Create a File

```
Key-in:   DISK ALLOCATE t FILE1
```

't' is the number of tracks assigned to this file. File names should be eight or less characters. The system will then ask you to: "ENTER RECORD SIZE."

The following information will be useful in selecting a record size:

* Programs, such as compiler or assembler output should be stored in a file with a 255 byte record size.
* The editor can handle record sizes up to 127 bytes.
* The PL/1 compiler only looks at the first 80 characters of each record. For longer records, the end of the record will be ignored.
* The assembler looks at the first 63 characters of each record. Records in files being read or written by PL/1 generated programs should be at least as long as the structure or character string or some data may be lost.

To Change a File Name

```
Key-in:   DISK RENAME FILE1 FILE2
```

To Change a File Size

```
Key-in:   DISK ALLOCATE t FILE1
```

To Delete a File

```
Key-in:   DISK ALLOCATE 0 FILE1
```

To Copy a File

```
Key-in:   DISK COPY FILE1 FILE2
```

The system will start to look for these files on disk no.1, then disk no. 2, etc. If the file names are the same, the system will look for the first file only on drive 1 and will start looking for the 'second' file on disk no. 2, etc. If either file is not found on the disks, "FILE1 NOT FOUND ON INDEX" will be displayed.

To Prevent Changes to a File

```
Key-in:   DISK PROTECT FILE1
```

To Remove File Protection

```
Key-in:   DISK FREE FILE1
```

The Editor

The purpose of the editor is:

1. To record an ASCII file on a disk unit
2. To change an existing file, and
3. To display an existing file.

Use of the Editor

If the operating system is ready for a command, one may load the editor by typing EDIT followed by the name of the file to be edited. Use EDIT,N file when using a new file or to discard all the data in an old one.

After the editor has been loaded, the first line of the file will appear on the display, except if you entered EDIT,N, only the cursor will appear. You may use any of the operating system keys which are designed to help you enter data. These keys are not exclusively for use with the editor and may be used any time the operator is entering data into the system:

COR ( )
Moves the cursor one position to the left without affecting displayed data.

CHAR ADV ( )
Moves the cursor one position to the right without affecting displayed data.

INS MODE
After this key is pushed, each character typed will be inserted in the line and all data to the right of the cursor will be moved right. To terminate this mode, push this key again.

TAB
Moves the cursor one tab to the right.

REV TAB
Moves the cursor one tab to the left.

HEX
The next two characters will be interpreted as the hexidecimal value of the ASCII code for a character. This allows the entry of codes for which there are no keys.

(Q1/Lite Only)

Moves the cursor down one line.

(Q1/Lite Only)

Moves the cursor up one line.

GO
Causes the operating system to proceed after reporting an error. The erroneous data will be accepted by the system and may produce unusual results.

The following keys will be displayed as non-printable characters, but will have the effect listed below when the file is printed:

INDEX
Advances paper to the next line without moving the carriage.

HALF REV INDEX
Moves paper one-half line backwards.

BACKSPACE
Moves the carriage back one position.

During editing the following keys can be used to manipulate the lines of the file:

RETURN
Advance to the next line. Will go from last line of the file to the first line of the file.

F1
Delete line.

F2
Insert a group of lines. Press return after each line to be inserted. Press F1 when insertion is completed.

F3
If you press the F3 (or SEARCH) key, the word SEARCH: will appear on the visual display followed by the cursor. Type the text of the line to be located and press the RETURN key. The editor will search each line for the text entered without regard for the position in the line. F8, F3, F4 and F6 will terminate the search.

F4
The system will respond: KEY SEARCH: and you may enter the text to be located. The text must have the same position in the line as the text entered for the search. Leading spaces affect only the position of the data and need not be matched by spaces in the searched-for line. This search is as much as one hundred times faster than the one initiated by F3.

F6
Displays the previous line in the file. Will go from the first line of the file to the last line of the file.

F7
Editing finished. Always press this key before removing the disk, pressing restart, or turning the machine off.

F8
Go to the first line of the file.

The Q1 Print Routine

In order to print an ASCII file, enter the command PRINT FILE1 to the operating system. It is also possible to print the index file with the command PRINT INDEX.

The Q1 Sort Routine

The Q1 Sort Routine sorts record files. The sort key has to be the first element of the structure. A valid sort key is a character string or positive fixed point decimal variable. The order for sorting is defined by the ASCII representations. In the case that the keys of two records are equal, the second element of the structure will be used for comparison. If the second keys are equal, the third element of the structure will be used as key and so on.

In order to call the Sort Routine, enter to the Q1 operating system the command SORT NAME1 NAME2, where NAME1 is the name of the file that is to be sorted and NAME2 the scratch file needed by the sort routine. The scratch file must be able to hold all the records in the sort file. The record length of NAME2 should be at least as long as NAME1.

If the sort key desired does not begin at the first byte, enter SORT,O FILE1 FILE2.

The system will ask:

```
SORT KEY OFFSET IN BYTES:
```

Enter the number of bytes in the record before the start of the sort key.

The SELECT Utility
To copy a portion of FILE1 to FILE2:

```
SELECT FILE1 FILE2
```

The program will ask:

```
FIRST RECORD TO BE COPIED:
```

Key in a character string from the start of the first record to be copied which is long enough to distinguish it from prior records.

The last question should be answered similarly:

```
UP TO BUT NOT INCLUDING:
```

If the answer specifies a non-existent record the remainder of the file will be copied.

If, in the answers to the above questions, the records are to be specified with keys that do not begin at the first of the record, enter:

```
SELECT,O FILE1 FILE2
```

The key will be treated analagous to the same option for SORT.

The JOIN Utility
To attach FILE2 to the end of FILE1, forming a file with the records from both files:

```
JOIN FILE1 FILE2
```

The ALIST Routine

```
ALIST FILE1
```

Prints a file in hexadecimal, with the record numbers at the left margin.

The COMPARE Utility
To check if two files are identical:

```
COMPARE FILE1 FILE2
```

The byte and record numbers of non-identical bytes will be printed. FILE2 must be in drive 2. This allows files with identical names to be compared.

FORM

The FORM program allows data entry and file manipulation for binary, decimal, and character data. Also, reverse and forward scrolling allow the use of records which cannot be displayed on the screen entirely at one time. This program has utility in several areas:

* FORM can be used for customer conversions. This will permit the system to be installed immediately without writing conversion programs.

* Programmers can use FORM to create test files, to analyze file data to isolate program bugs, and to repair customer files damaged by program bugs.

* The system can now be more effectively used as a data entry device.

To call the FORM program, three file names are required:

```
FORM LAYOUT DATA
```

The last two files can be any names. LAYOUT and DATA are examples. LAYOUT is a file which specifies the organization of the information on the display. The control file section describes the creation of this file, which is usually done by programmers rather than operators.

To discard all existing records in DATA and treat it as a new file, enter:

```
FORM,N LAYOUT DATA
```

Once the FORM program has been loaded, the first record of DATA will be displayed. The following control keys can now be used:

TAB
Advances cursor to next field.

REV TAB
Moves cursor back to the start of the previous field.

CHAR ADV, COR
Work like they do with the operating system.

CLEAR ENTRY, DEL, INSRT MODE
Work like they do with the operating system except that they only operate on a field rather than an entire record.

TAB SET/CLR
Duplicates the field from the previous record.

F1
Delete record.

F2
Insert a sequence of records. Push F1 after the last record is inserted.

F4
Key search. After pushing F4, enter the field to be searched for. Leading or trailing blanks or zeros will be ignored. Push return immediately after F4 to repeat the prior search.

F6
Previous record.

F7
Return to "Q1/LMC AT YOUR SERVICE".

F8
First record.

The second file in the calling of the FORM program describes the organization of the data on the display. It should be allocated as a single track with 37 bytes/record (47 for the six or twelve line display. Do not exceed forty records.

Suppose the DATA file were written from the following structure:

```
1 ORG,
  2 RATE FLOAT (7),
  2 NAME CHAR (20),
  2 PAY FIXED (9,2),
  2 NUM BINARY;
```

The LAYOUT file might appear as follows:

```
RATE OF PAY:  fffffffff
NAME:  cccccccccccccccccccc
GROSS PAY:  xxxxxxxxx  EMPLOYEE NO.:  bbbbb
```

The lower case letters do not appear on the display, but represent position where the cursor may appear. A contiguous sequence of lower case letters represents one variable. The letter should correspond to the data type of the variable, and the number of letters is dependent on the length of the variable:

c - character. One 'c' for each character in the variable.

b - binary. 2 or more 'b's.

f - floating point. One 'f' for each digit plus two for sign and decimal point.

x - fixed point. One 'x' for each digit. The operator should not key in the decimal point.

- character with lower case shift mode.

Do not exceed 84 variables.

The Q1 Trace Routine

A trace routine is a usefule debugging aid. By means of various trace procedures, it is possible to see what is in the registers at any given time and to display such information as addressed memory locations, the contents of the program counter, and the instructions about to be executed at any given time. A trace can be applied to all memory if so desired, or it can be limited in its scope by the various commands available under the Q1 trace routine. The particular trace options chosen will depend upon the requirements of the user program. In order to use the Q1 Trace Routine program, enter the following command to the operating system:

```
    TRACE FILE1
```

where FILE1 is the name of a binary file, and then use the following instructions. Optional parameters in hexadecimal are enclosed in brackets for descriptive purposes. If the second parameter is omitted, it will be assumed to be the same as the first.

```
    ALL         [First]          [,Last]
```

The command will cause the trace routine to display the contents of the registers, the addressed memory locations, the program counter, and the instruction about to be executed, if the instruction has an address not less than "First" and not greater than "Last." "First" and "Last" are two hexadecimal addresses. If they are not given, the trace will be applied to all of the memory. The trace will continue when the space bar is depressed.

```
    MEMORY      [First]          [,Last]
```

This comman is similar to the ALL command except that all store instructions that address memory locations in the range "First" to "Last" will be traced, regardless of the instruction address.

```
    GO          [Address]
```

The trace routine will begin tracing a program at "Address" when this command is given. If "Address" is not given, the trace will begin at the instruction it was about to trace when interrupted, or at the starting address of the program on loading if no tracing has taken place.