intel®

8008 and 8080 PL/M™ Programming Manual

REVISION A

Intel Corporation, 1975

PREFACE

This manual is a tutorial introduction to the PL/M langauge, as it applies to the INTEL 8008 and 8080 processors. To facilitate a first reading, it has a spiral organization: in the course of a front-to-back reading, the same topics will arise more than once, to be explained more fully on the later occasions.

It is also expected that the PL/M programmer will keep this manual at hand for reference purposes. The table of contents has been elaborated to encourage such use. For information on the use of the PL/M compilers themselves, the reader is referred to the appropriate Compiler User's Manual.

* PL/M is a registered trademark of INTEL Corporation

1. INTRODUCTION

1.1 What is PL/M?

PL/M is a high-level programming language especially designed to simplify the task of system programming for the INTEL 8-bit family of microcomputers -- the 8008 and the 8080.

PL/M provides an effective software tool suited to the requirements of the microcomputer system designer and implementor. It gives the programmer control of the processor sufficient for the needs of system programming, but provides automatic control of many specific processor resources -- e.g., registers, memory, and stack. In consequence, PL/M programs can enjoy a high degree of portabliity between systems.

PL/M has been designed to facilitate the use of modern techniques in structured programming. These techniques can lead to rapid system development and checkout, straightforward maintenance and modification, and a product of high reliability.

1.2 Overview of the Language

A PL/M program is a sequence of "declarations" and "executable statements".

The declarations allow the programmer to control allocation of storage, define simple textual substitutions (macros), and define procedures. PL/M is a "block structured" language: procedures may contain further declarations which control storage allocation and define other procedures.

The procedure definition facility of PL/M allows modular programming: a program can be divided into sections (e.g. teletype input, conversion from binary to decimal forms, and printing output messages). Each of these sections is written as a PL/M procedure. Such procedures are conceptually simple, easy to formulate and debug, and easily incorporated into a large program. They may form a basis for a procedure library, if a family of similar programs is being developed.

PL/M handles two kinds of data, its two basic "data types": BYTE and ADDRESS. A BYTE variable or constant is one that can be represented as an 8-bit quantity; an ADDRESS variable or constant is a 16-bit or double-byte quantity. The programmer can DECLARE variable names to represent BYTE or ADDRESS values. One can also declare vectors (or arrays) of type BYTE or ADDRESS.

In general, executable statements specify the computational processes that are to take place. To achieve this, arithmetic, logical (boolean), and comparison (relational) operators are defined for variables and constants of both types (BYTE and ADDRESS). These operators and operands are combined to form EXPRESSIONS, which resemble those of elementary algebra. For example, the PL/M expression

```
X * (Y - 3) / R
```

represents this calculation: the value of X multiplied by the quantity Y-3, divided by the value of R. Expressions are a major component of PL/M statements. A simple statement form is the PL/M ASSIGNMENT statement, which computes a result and stores it in a memory location defined by a variable name. The assignment

```
Q = X * (Y - 3) / R
```

first causes the computation to the right of the equals sign, as described above. The result of this computation is then saved in a memory location labeled by the variable name ´Q´.

Other statements in PL/M perform conditional tests and branching, loop control, and procedure invocation with parameter passing. The flow of program execution is specified by means of powerful control structures that take advantage of the block-structured nature of the language. Input and output statements read and write 8-bit values from and to 8008 and 8080 input and output ports. Procedures can be defined which use these basic input and output statements to perform more complicated I/O operations.

A method of automatic text-substitution (more specifically, a "compile-time macro facility") is also provided in PL/M. A programmer can declare a symbolic name to be completely equivalent to an arbitrary sequence of characters. As each occurence of the name is encountered by the compiler, the declared character sequence is substituted, so the compiler actually processes the substituted character string instead of the symbolic name.

2. BASIC CONSTITUENTS OF A PL/M PROGRAM

PL/M programs are written free-form. That is, the input lines are column-independent and spaces may be freely inserted between the elements of the program.

2.1 PL/M Character Set

The character set recognized by PL/M is a subset of both ASCII and EBCDIC character sets. The valid PL/M characters consist of the alphanumerics

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9

along with the special characters

$ = . / ( ) + - ´ * , < > : ;

All other characters are unrecognized by PL/M, in the sense that a blank is substituted for each such character.

Special characters and combinations of special characters have particular meanings in a PL/M program, as shown in Appendix C.

2.2 Identifiers and Reserved Words

A PL/M identifier is used to name variables, procedures, macros, and statement labels. An identifier may be up to 31 characters in length, the first of which must be alphabetic, and the remainder either alphabetic or numeric. Imbedded dollar signs are ignored by the PL/M compiler, and are used to improve the readability of an identifier. An identifier containing a dollar sign is exactly equivalent to the same identifier with the dollar sign deleted. Examples of valid identifiers are

X
GAMMA
LONGIDENTIFIERWITHNUMBER3
INPUT$COUNT
INPUTCOUNT

where the PL/M compiler will regard the final 2 examples as instances of the same identifier.

There are a number of otherwise valid identifiers, whose meanings are fixed in advance. Because they are actually part of the PL/M language, they may not be used as programmer-defined identifiers. A list of such RESERVED WORDS is given in Appendix D.

Blanks may be inserted freely around identifiers, reserved words, and special characters. Blanks are not necessary, however, when identifiers or reserved words are separated by special characters or delimiters. Thus the expression

```
X * ( Y - 3 ) / R
```

is equivalent to

```
X*(Y-3)/R
```

2.3 Comments

Explanatory remarks should be interleaved with PL/M program text, to improve readability and provide program documentation. This is the purpose of the COMMENT construction. A PL/M comment is a sequence of characters (from the PL/M character set) delimited on the left by the character pair /* and on the right by the character pair */. These delimiters instruct the compiler to ignore any text bewteen them, and not to consider such text part of the program proper. A comment may appear anywhere a space character may; thus comments may be freely distributed throughout a PL/M program. There is only one restriction on the placement of a comment: it may not begin or end inside a character string. Here is a sample (if atypical) PL/M comment:

```
/* THIS IS A COMMENT ABOUT COMMENTS */
```

3. PL/M PROGRAM ORGANIZATION

STATEMENTS are the building blocks of a PL/M program. A PL/M statement either defines a computational entity, or specifies a computation to be performed. For example, the PL/M statement

```
DECLARE X BYTE;
```

defines a variable named X that has a single-byte (8-bit) value. The PL/M statement

```
X = 3*(Y + Z);
```

causes the computation of the arithmetic quantity, 3 times the sum of Y and Z, and the assignment of that quantity as the new value of the variable X. PL/M statements are frequently (but not necessarily) written one to a line, and invariably terminate with semicolons.

A PL/M program comprises a sequence of PL/M statements, followed by the special identifier EOF. In the absence of statements specifying otherwise, the statements of a PL/M program are executed sequentially, in the order of their appearance. For example, the following program fragment is a sequence of two statements:

```
X = 3;
Y = 4+X;
```

Two successive actions are specified: first, 3 becomes the current value of the variable X; second, a new value for the variable Y is calculated by adding 4 to the current value of X (in this case 3, for a result of 7). It is obvious that in a different sequence, these two statements could have a very different effect.

The strictly sequential execution of statements is interrupted by, for example, an IF-statement:

```
IF A>63 THEN X=3;
ELSE X=9;
Y = 4+X;
```

Here the statement ′X=3′ is executed only if the current value of A is greater than 63; the statement ′X=9′ is executed only if the current value of A is less than or equal to 63; and the statement ′Y = 4+X′ is executed always.

Statements may be collected together in groups, delimited by the reserved words DO and END, to form compound statements, or blocks. These blocks are then treated as single statements with respect to the flow of program control. Such a group could, for example, be a part of a conditional statement:

```
IF A>B THEN
  DO;
  A = B;
  B = C;
  END;
```

This statement performs the two assignments to A and B only if A is greater than B to start with.

Statements may also be grouped to form a ´procedure´, whose execution may then be called for from elsewhere in the program. The following procedure, for example, calculates the sum of the squares of its two arguments:

```
SUM$SQUARE: PROCEDURE (A, B) ADDRESS;
  DECLARE (A, B) ADDRESS;
  RETURN A*A + B*B;
END SUM$SQUARE;
```

After this procedure has been defined, it is available for use -- e.g., for calculating new values for variables. For example, the sequence of statements

```
X = 3;
Y = 5 + SUM$SQUARE (X, 4);
```

results in Y having the new value 30.

The exact details of various kinds of statements and other PL/M language constructs -- assignments, conditional statements, groups, declarations, procedures, and so forth -- are given in the following sections.

4. PL/M DATA ELEMENTS

PL/M data elements can be either variables or constants. Variables are PL/M identifiers whose values may change during execution of the program, while constants have fixed values. The expression

X * (Y - 3) / R

involves the variables X, Y, and R, and the constant 3.

4.1 Numeric Constants

A constant is a value known at compile-time, which does not change during execution of the program. A constant is either a number or a character string. Numeric constants may be expressed as binary, octal, decimal, and hexadecimal numbers.

In general, the base (or radix) of a number is represented by one of the letters

B O Q D H

following the number. The letter B denotes a binary constant; the letters O and Q signal octal constants. The letter D may optionally follow decimal numbers. Hexadecimal numbers consist of sequences of hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) terminated by the letter H. The leading character of a hexadecimal number must be a numeric digit, to avoid confusion with a PL/M identifier; a leading zero is always sufficient. Any number not followed by one of the letters B, O, Q, D, or H is assumed to be decimal. Numbers must be representable in 16 bits. The following are valid constants in PL/M:

```
2  33Q  110B  33FH  55D  55  0BF3H  65535
```

The dollar sign may be freely inserted between the characters of a constant to improve readability. The two following binary constants are exactly equivalent:

```
11110110011B
111$1011$0011B
```

4.2 Character String Constants

Character strings are denoted by PL/M characters enclosed within apostrophes. (To include an apostrophe in a string, write it as a double apostrophe: e.g. the string '''Q' comprises 2 characters, an apostrophe followed by a Q.) The PL/M compiler represents character strings in memory as ASCII codes, one 7-bit character code to each 8-bit byte, with a high-order zero bit. Strings of length 1 translate to single-byte values; strings of length 2 translate to double-byte values. For example,

'A'	is equivalent to	41H
'AG'	is equivalent to	4147H

(see appendix for ASCII character codes). Character strings longer than 2 characters cannot, of course, be used as BYTE or ADDRESS values. But they will turn out to be useful in conjunction with the dot operator, with the INITIAL attribute, and with the DATA declaration.

4.3 Variables and Type Declarations

Each variable used in a PL/M program must be declared in a 'declaration statement' before (earlier in the program text than) its use in expressions. This declaration defines the variable and gives necessary information about it.

A PL/M variable takes one of two 'types': type BYTE, or type ADDRESS. Each BYTE data element is an 8-bit, single-byte object; each ADDRESS data element is a 16-bit, double-byte object. The type of each variable must be formally declared in its declaration statement.

A declaration of a variable (or a list of variables) begins with the reserved word DECLARE. Each single identifier, or list of identifiers enclosed in parentheses, is followed by one of the two reserved words BYTE or ADDRESS. Sample PL/M declarations are

```
DECLARE X BYTE;
DECLARE (Q, R, S) BYTE;
DECLARE (U, V, W) ADDRESS;
```

Additional facilities are present in PL/M for declaring vectors, macros, labels, and data lists. These facilities are discussed in later sections.

5. WELL-FORMED EXPRESSIONS AND ASSIGNMENTS

PL/M expressions can now be more completely defined. A well-formed expression consists of basic data elements combined through the various arithmetic, logical, and relational operators, in accordance with simple algebraic notation. Examples are

```
A + B
A + B - C
A*B + C/D
```

5.1 Arithmetic Operators

There are 7 arithmetic operators in PL/M. These are

+ - PLUS MINUS * / MOD

All of the above operators perform unsigned binary arithmetic on either byte or address values.

The operators + and - perform addition and subtraction. If both operands are of type BYTE, the operation is done in 8-bit arithmetic and the result is of type BYTE. If either operand is of type ADDRESS, the other operand, if it is of type BYTE, will be extended by 8 high-order zero bits, and the operation is then performed in 16-bit arithmetic, returning a value of type ADDRESS. A unary '-' operator is also defined in PL/M. Its effect is such that (-A) is equivalent to (0-A). Thus -1, for example, is equivalent to 0-1, resulting in the BYTE value 255 or 0FFH. PLUS and MINUS perform similarly to + and -, but take account of the current setting of the CPU hardware carry flag in performing the operation.

The operators * and / perform unsigned binary multiplication and division, on operands of type BYTE or ADDRESS. The result is always of type ADDRESS. In the event that arithmetic overflow occurs during multiplication, the result is undefined. The division operator always rounds down to an integer result, and the result of division by zero is undefined. (The setting of the 8080 hardware carry flag by these operations is undefined.) MOD performs similarly to /, except that the result of the operation is not the quotient from the division, but the remainder.

5.2 Logical Operators

There are 4 logical (boolean) operators in PL/M. These are

NOT AND OR XOR

These operators perform logical operations on 8 or 16 bits in parallel. NOT is a unary operator, taking one operand only. It produces a result in which each bit is the complement of the corresponding bit of its operand. The remaining operators each take 2 operands, and perform bitwise AND, OR, and EXCLUSIVE OR respectively. If both operands are of type BYTE, the operation is an 8-bit operation, and delivers a result of type BYTE. If at least one operand is of type ADDRESS, the operation is a 16-bit operation, and delivers a result of type ADDRESS. In this case, the BYTE operand, if any, is first extended to 16 bits by the addition of 8 high-order zero bits. Examples are

```
          NOT 11001100B returns 00110011B
10101010B AND 11001100B returns 10001000B
10101010B OR  11001100B returns 11101110B
10101010B XOR 11001100B returns 01100110B
```

5.3 Relational Operators

Relational operators are used to compare PL/M values. They are

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to
=	equals

Relational operators are always binary operators, taking two operands. The operands may be of type BYTE or ADDRESS. The comparison is always performed assuming that the operands are unsigned binary integers. If the specified relation between the operands holds, a value of 0FFH is returned, otherwise the result is 00H. Thus in all cases the result is of type BYTE, with all 8 bits set to 1 for a true condition, and to 0 for a false condition. For example:

```
(6 > 5)                   returns 11111111B
(6 <= 4)                  returns 00000000B
(6 > 5) OR (1 > 2)        returns 0FFH
(6 > (4+5)) OR (1 > 2)    returns 00H
```

5.4 Expression Evaluation

Operators in PL/M have an implied precedence, which is used to determine the manner in which operators and operands are grouped together. A+B*C causes A to be added to the product of B and C. In this case B is said to be ´bound´ to the operator * rather than the operator +, as a result of which the multiplication will be performed first. In general, operands are bound to the adjacent operator of highest precedence, or to the left one in the case of a tie. Technically speaking, PL/M does not guarantee the order of evaluation of operands and operations within an expression, but merely defines the association (binding) of operators and operands. Valid PL/M operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence.

```
           *  /  MOD
       +  -  PLUS  MINUS
     <  <=  <>  =  >=  >
               NOT
               AND
             OR  XOR
```

Parentheses should be used to override the assumed precedence in the usual way. Thus the expression (A + B) * C will cause the sum of A and B to be multiplied by C. For example,

A + B + C + D	is equivalent to	((A + B) + C) + D
A + B * C	is equivalent to	A + (B * C)
A + B - C * D	is equivalent to	(A + B) - (C * D)

5.5 Assignment Statements

Results of computations are stored as values of variables. At any given moment, a variable has only one value -- but this value may change with program execution. The PL/M ASSIGNMENT STATEMENT re-specifies the value of a variable. Its form is

```
variable = expression ;
```

The expression to the right of the equal sign is evaluated, and the resulting value is assigned to the variable named on the left. The old value of the variable is lost.

For example, following execution of the statement

```
A = 3;
```

the variable A will have a new current value of 3.

The declared precision (BYTE or ADDRESS) of the assigned variable affects the store operation: if the receiving variable is a BYTE variable, and the expression is a double-byte (ADDRESS) result, the high-order byte is omitted in the store. Similarly, if the expression yields a single-byte result, and the receiving variable is declared type ADDRESS, the high-order byte is filled with zeros.

It is often convenient to assign the same expression to several variables. This is accomplished in PL/M by listing all the variables to the left of the equals sign, separated by commas. The variables A, B, and C could all be set to the value of the expression X + Y with the single assignment statement

```
A, B, C = X + Y;
```

A special form of the assignment is used within PL/M expressions. The form of this 'embedded assignment' is

```
(variable := expression)
```

and may appear anywhere an expression is allowed. The expression to the right of the := assignment symbol is evaluated and then stored into the variable on the left. The value of the embedded assignment is the same as that of its right half. For example, the expression

```
A + (B := C+D) - (E := F/G)
```

results in exactly the same value as

```
A + (C+D) - (F/G)
```

The only difference is the side-effect of storing the intermedia results C+D and F/G into B and E, respectively. These intermedi results can then be used at a later point in the program witho calculating them again.

6. DO GROUPS

Statements may be grouped together within the bracketing words DO and END, to form a do-group. (DO and END are reserved words.) The simplest do-group is of the form

```
DO;
  statement-1;
  statement-2;
  ...
  statement-n;
END;
```

A group of statements so bracketed may be regarded as a single PL/M statement, and may appear anywhere in a program that a single statement may. The flow of program control is explicitly controlled by other forms of the do-group; these are shown below.

6.1 The DO-WHILE Group

The DO-WHILE is a do-group of the form

```
DO WHILE expression;
  statement-1;
  statement-2;
  ...
  statement-n;
END;
```

The effect of this statement is: first the expression following the reserved word WHILE is evaluated. If the result is a quantity whose rightmost bit is 1, then the sequence of statements up to the END is executed. When the END is reached, the expression is evaluated again, and again the sequence of statements is executed only if the value of the expression has a rightmost bit of 1. The group is executed over and over until the expression results in a value whose rightmost bit is 0, at which time execution of the statement group is skipped, and program control passes out of the group.

Consider the following example:

```
A = 1;
DO WHILE A <= 3;
  A = A+1;
END;
```

The statement A = A+1 will be executed exactly 3 times. The value of A when program control exits the group will be 4.

It is worth commenting here on the relationship between the logical operators, and the WHILE and IF statements. Recall that the relational operators return a BYTE value of all ones, or all zeros.

It may be helpful to consider any BYTE whose least significant (rightmost) bit is 1, as representing a TRUE condition, and any whose least significant bit is 0, as representing a FALSE condition. With this interpretation, we may consider (1 < 2) as returning a value of TRUE. We may also consider that the do-while statement merely executes the statements of its group as long as the while-expression is TRUE. Note that the logical operators AND, OR, and NOT operate bitwise on all the bits of their operands, and in particular perform the standard actions of boolean algebra on the least significant bit, provided a 1 stands for TRUE and a 0 for FALSE. For example, with the above definitions,

```
NOT(TRUE) is FALSE
NOT(FALSE) is TRUE
```

Finally, observe that these conventions cause a complicated expression to take on its most obvious meaning. For example:

```
DO WHILE (A < 10) AND (A > 4);
...
END;
```

6.2 The Iterative Do-Group

An iterative do-group executes a group of statements a fixed number of times. The simplest form of the iterative do-group is

```
DO var = expr-1 TO expr-2;
  statement-1;
  statement-2;
  ...
  statement-n;
END;
```

where ´var´ is a variable-name, and ´expr-1´ and ´expr-2´ are both PL/M expressions. The effect of this statement is first to store the value of expr-1 into the variable var. Second, the value of the variable var is tested, and if it is less than or equal to expr-2, the grouped statements are executed. When the END is reached, the variable is incremented by 1, and the test is repeated. The group is repeatedly executed until the value of the variable is greater than expr-2, when the test fails, execution of the group is skipped, and control immediately passes out of the range of the do-group. An example is

```
DO I = 1 TO 10;
  A = A+I;
END;
```

This iterative do-group has exactly the same effect as the following DO-WHILE:

```
I = 1;
DO WHILE I <= 10;
  A = A+I;
  I = I+1;
END;
```

The more general form of the iterative do-group allows a stepping value other than 1. This more general form is

```
DO var = expr-1 TO expr-2 BY expr-3;
  statement-1;
  statement-2;
  ...
  statement-n;
END;
```

In this case, the variable 'var' following the DO is stepped by the value of expr-3, instead of 1, each time the END is reached. An example of this form follows:

```
/* COMPUTE THE PRODUCT OF THE
   FIRST N ODD INTEGERS */

PROD = 1;
DO I = 1 TO (2*N-1) BY 2;
   PROD = PROD*I;
END;
```

6.3 The DO-CASE Statement

The final form of the do-group is the DO-CASE statement. Its form is

```
DO CASE expression;
  statement-1;
  statement-2;
  ...
  statement-n;
END;
```

The effect of this statement is first the evaluation of the expression following the CASE. The result of this is a value K which must lie between 0 and n-1. K is used to select one of the n statements of the do-case, which is then executed. The first case (statement-1) corresponds to K=0, the second case (statement-2) corresponds to K=1, and so forth. After only one statement from the group has been selected and then executed only once, control passes beyond the END of the do-case group. If the run-time value of K is greater than the number of cases, then the effect of the CASE statement is undefined.

An example of the DO-CASE is

```
DO CASE SCORE;
   ;
   CONVERSIONS = CONVERSIONS+1;
   SAFETIES = SAFETIES+1;
   FIELDGOALS = FIELDGOALS+1;
   ;
   ;
   TOUCHDOWNS = TOUCHDOWNS+1;
END;
```

When execution of this CASE statement begins, the variable SCORE must be in the range 0 - 7. If SCORE is 0, 4, or 5 then a null statement (consisting of only a semicolon, and having no effect) is executed; otherwise the appropriate variable is incremented.

A more complex example of the DO-CASE is

```
DO CASE X-5;

  X = X+5;            /* CASE 0 */

  DO;                 /* CASE 1 */
    X = X+10;
    Y = X-3;
  END;

  DO I = 3 TO 10;     /* CASE 2 */
    A = A+I;
  END;

END;                  /* END OF CASES */
```

This example illustrates the use of DO-END blocks to group several statements as a single (although compound) PL/M statement.

7. THE IF-STATEMENT

The IF-statement provides alternative execution of statements. It takes the form

```
IF expression THEN statement-1;
ELSE statement-2;
```

and has the following effect: first the expression following the reserved word IF is evaluated. If the result has a low-order (rightmost) bit of 1, then statement-1 is executed; if the result has a rightmost bit of 0 then statement-2 is executed. Following execution of the chosen alternative, control passes to the next statement following the if-construct. Thus of the two subordinate statements (statement-1 and statement-2) one and only one is executed.

The IF-statement tests the rightmost bit of an expression in the same way as the DO-WHILE statement (see section 6.1). The most intuitive interpretation associates TRUE with a rightmost bit of 1, and FALSE with a rightmost bit of 0.

Consider the following program fragment:

```
IF A>B THEN RESULT=A;
ELSE RESULT=B;
```

Here RESULT is assigned the value of A or the value of B, whichever is greater. As program control falls through this fragment, there will be exactly one assignment statement executed. RESULT always gets assigned some value; but only one assignment to RESULT will be executed.

Let us return to the most general form of the IF-statement:

```
IF expression THEN statement-1;
ELSE statement-2;
```

In the event that statement-2 is not needed, the else-clause may be omitted entirely. Such an IF-statement takes the form

```
IF expression THEN statement-1;
```

Here the subordinate statement is executed only if the value of the if-condition has a rightmost bit of 1; otherwise nothing happens, and control falls right through the if-construct.

For example, the following sequence of PL/M statements will assign to INDEX either the number 5, or the value of Y, whichever the larger. The value of X will change during execution of the IF-statement only if Y is greater than 5. The final value of X is always copied to INDEX in any case.

```
X = 5;
IF Y > X THEN X = Y;
INDEX = X;
```

The power of the IF construct in enhanced by compound statements. Since a do-group is itself syntactically equivalent to a single statement, each of the two subordinate statements in an IF-construct may be a do-group. For example:

```
IF A=B THEN
    DO;
    ...
    END;
ELSE
    DO;
    ...
    END;
```

These do-groups can contain further nested if-statements, variable and procedure declarations, and so on.

There is only one restriction on subordinate statements of if-statements: statement-1 (that is, the subordinate statement just following the if-clause) may not itself be an if-statement, unless no ELSE is attached to either of these IF´s. In other words, the construction

```
IF condition-1 THEN
   IF condition-2 THEN statement-3;
ELSE statement-2;
```

is ambiguous and illegal (to which IF does the ELSE belong?), and must be replaced by one of the two following constructions, depending on the actual intention:

```
(1)     IF condition-1 THEN
           DO;
           IF condition-2 THEN statement-3;
           END;
        ELSE statement-2;

(2)     IF condition-1 THEN
           DO;
           IF condition-2 THEN statement-3;
           ELSE statement-2;
           END;
```

8. ARRAYS

8.1 Array Declarations

It is frequently convenient to let one PL/M identifier represent more than one BYTE or ADDRESS value. When this is desired, the identifier must be suitably declared in a DECLARE statement. For example,

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

causes the identifier X to be associated with 100 data elements, each of type BYTE. Furthermore,

```
DECLARE (A, B, C) (100) ADDRESS;
```

causes the 3 identifiers A, B, and C each to be associated with 100 data elements of type ADDRESS, so that 300 elements of type ADDRESS have been declared in all. Variables that have been declared in this manner to name more than a single data element are called arrays, vectors, or subscripted variables.

(In the special case that an array is declared to have a length of zero, no space will be allocated for it in memory. As a result, the variable will be a ghost, which refers to memory not specifically reserved for it.)

8.2 Subscripted Variables

It is sometimes necessary to refer to each element of an array by name. For example,

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

actually declares 100 data elements of type BYTE, with names X(0), X(1), X(2), and so on up to X(99). If we wish to add the third data element to the fourth, and store the result in the fifth, we can write the PL/M assignment statement

```
X(4) = X(2) + X(3);
```

The index in parentheses, which selects the specific data element of the array, is called an array index, or subscript.

Much of the power of a subscripted variable lies in the fact that its subscript need not be a numeric constant, but can be another variable, or in fact any valid PL/M expression. Thus the following program will sum the elements of the array NUMBERS:

```
DECLARE SUM BYTE;
DECLARE NUMBERS (10) BYTE;
DECLARE I BYTE;

SUM = 0;
DO I = 0 TO 9;
  SUM = SUM + NUMBERS(I);
END;

EOF
```

Subscripted variables are permitted anywhere PL/M permits a simple variable, with the one exception that it is not legal to use one as the control variable of an iterative do-group.

9. DECLARATION STATEMENTS

9.1 Objects and Attributes

The purpose of a declaration is to introduce some computational entity (e.g. a procedure, label, or data element), give it a name, and describe some of its attributes. Leaving aside the declaration of procedures, which will be discussed in section 11, declarations are done by means of the declaration statement. The simplest form of a declaration statement is

```
DECLARE object-name attribute-1 attribute-2 ... ;
```

where the attributes are things like (in the case of variables) type, size, addressing method, and initial value. Let us look at the declaration of a typical array:

```
DECLARE FWD (100) BYTE;
```

Here is a name (FWD), a size attribute (100), and a type attribute (BYTE). Certain syntactic rules govern the ordering of attributes; in the example above, the size attribute must precede the type attribute. (All such rules are explicitly gathered in one place at appendix A.)

9.2 The INITIAL Attribute

The values of variables may be initialized in their declaration statements using the INITIAL attribute. This attribute takes the form

```
INITIAL (constant-list)
```

where the ´constant-list´ is a sequence of constants, separated by commas. This attribute must immediately follow the type attribute (BYTE or ADDRESS) in the declaration statement.

The purpose of the INITIAL attribute is to pre-set the values of the variables named. The variable or array is allocated storage as if the INITIAL attribute were not present in the declaration. Then the values given in the INITIAL attribute are placed in memory at program load-time, before the program starts execution.

The user should exercise caution in use of the INITIAL attribute. He should be aware, for example, that neither procedure entry nor program restart will cause any variable initialization -- a complete program re-load is required. In fact, use of the INITIAL attribute is hardly ever recommended; for ROM-based systems at least, the DATA declaration will be far more useful.

The following are valid declarations using the INITIAL attribute:

```
DECLARE X BYTE INITIAL (10);
DECLARE Y(10) BYTE INITIAL (1,2,3,4,5,6,7,8,9,10);
DECLARE Z(100) BYTE INITIAL('SHORT', 'STRING', 0FH);
DECLARE U(100) ADDRESS INITIAL (3, 4, 535Q);
DECLARE (Q, R, S) BYTE INITIAL (0, 1, 2);
```

The number of bytes required to hold the list of constants need not correspond to the length declared for the variable. The constants are placed in memory without truncation, starting at the first byte allocated by the DECLARE statement. It is illegal, however, to specify INITIAL attributes which overlay each other.

9.3 The DATA Declaration

Suppose you want to declare an array and give it initial values, and you want those values never to change with program execution. If your system provides different memories for program code storage and data storage, the answer might be to store this particular array with the read-only program code rather than with the read-write variables. PL/M gives you this kind of control over storage allocation with the DATA declaration. The form is

```
DECLARE identifier DATA (constant-list);
```

As an example of this construction, consider

```
DECLARE MESSAGE DATA ('8080 PL/M');
```

The effect of a DATA declaration is similar to that of an array declaration with an INITIAL attribute, but there are differences in form. No data-type specification appears in the declaration; type BYTE is forced. No explicit array size appears in the declaration; the size is implicitly specified by the length of the constant-list. DATA identifiers are used just like subscripted BYTE array identifiers, with one exception: they should never appear on the left-hand side of an assignment operator.

9.4 Declaration Elements

A separate declaration statement is not required for each and every declaration. Instead of writing the two declaration statements

```
DECLARE CHR BYTE INITIAL ('A');
DECLARE FAB ADDRESS;
```

we may write both declarations in a single declaration statement, like this:

```
DECLARE CHR BYTE INITIAL ('A'), FAB ADDRESS;
```

This declaration statement contains two "declaration elements", separated by the comma. Every declaration statement contains at least one declaration element; if it contains more than one, they are separated by commas.

A declaration element is a textual unit defining one name, or one list of names, as in

```
DECLARE HARE BYTE, (HOUND, HORN) ADDRESS;
```

The declaration elements appearing in a single statement are completely independent of each other; it is as if they had been declared in different statements. The only question is whether the reserved word DECLARE shall be repeated. This is a question of style, not substance.

10. A SORTING PROGRAM

Now we construct an example program using expressions, do-groups, and subscripted variables. Suppose a vector A contains a set of numbers in an arbitrary order, and we wish to sort them into ascending order.

```
/* INITIAL ORDERING OF 'A' IS ARBITRARY */

DECLARE A(10) ADDRESS INITIAL
  (33, 10, 2000, 400, 410, 3, 3, 33, 500, 1999);

                  /* BUBBLE SORT */

/* SWITCHED = (BOOLEAN) HAVE WE DONE ANY
              SWITCHING YET THIS SCAN? */
DECLARE (I, SWITCHED) BYTE, TEMP ADDRESS;

SWITCHED = 1;        /* SWITCHED=TRUE MEANS NOT DONE YET */
DO WHILE SWITCHED;

  SWITCHED = 0;          /* BEGIN NEXT SCAN OF A */
  DO I = 0 TO 8;
    IF A(I) > A(I+1) THEN
      DO;                /* FOUND A PAIR OUT OF ORDER */
      SWITCHED = 1;      /* SET SWITCHED = TRUE */
      TEMP = A(I);       /* SWITCH THEM INTO ORDER */
      A(I) = A(I+1);
      A(I+1) = TEMP;
      END;
  END;
  /* HAVE NOW COMPLETED A SCAN */

END /*WHILE*/;
/* HAVE NOW COMPLETED A SCAN WITH NO SWITCHING */

EOF
```

This program scans the vector A, comparing each adjacent pair of elements. When it finds a pair out of order, it swaps them. It does this repeatedly, until it completes an entire scan of A without having swapped any pair. Then it is done.

The variable SWITCHED keeps track of whether we have done a swap yet, this time through the array. So we zero it each time we start a new scan, and set it each time we do a swap.

Study this program until you understand it. It is the basis of later examples.

11. PROCEDURES

A ´procedure´ is a section of PL/M code which is declared without being executed, and then ´called´ from other parts of the program. The call is in fact a remote execution of the procedure out of normal sequence: program control is transferred from the point of call to the procedure code, the procedure is executed, and when the procedure exits, program control is passed back to the point of the call.

The use of procedures forms the basis of modular programming, facilitates making and using program libraries, eases programming and documentation, and reduces the amount of object code generated by a program. The following 2 sections tell how to define (declare) procedures, and how to invoke (call) procedures.

11.1 Procedure Declarations

A procedure must be defined before it is used. That is, a ´procedure declaration´ for a procedure must occur earlier in the program text than any reference to that procedure. A procedure declaration consists of 4 parts: the procedure name, the specification of any formal parameters, the type of the returned value (if any), and the procedure body (the code itself). These elements take the following form:

```
name: PROCEDURE (argument-list) type;
  statement-1;
  statement-2;
  ...
  statement-n;
END name;
```

The name is a PL/M identifier, which is hereby associated with this procedure. From this point in the program forward, the procedure can be invoked by simply mentioning its name.

The argument-list takes the form

```
(arg-1, arg-2, ..., arg-n)
```

where arg-1 through arg-n are PL/M identifiers. Such identifiers are called ´formal parameters´; they hold values passed to the procedure from the point of its invocation. (PL/M procedures are thus of the "call by value" variety.) Each of these formal parameters must appear in a declaration statement within the procedure body, so its type and size are defined. The argument-list may be omitted altogether if no parameters are passed to the procedure.

The type of the procedure is either BYTE or ADDRESS, if the procedure returns a value to the point of call. If no value is returned, the type is omitted from the procedure declaration. The procedure type defines the precision of the value returned so that proper type conversion takes place when the procedure is invoked as part of an expression.

The execution of a procedure is terminated by execution of a RETURN statement within the procedure body. The RETURN statement takes one of the two forms

```
RETURN;
RETURN expression;
```

The first form is used if no value is returned by the procedure (and hence no procedure type is declared). The second form is used if the procedure type is BYTE or ADDRESS, in which case the value of the expression in the RETURN statement is brought back to the calling point.

The statements within the procedure body may be any valid PL/M statements, including nested procedure declarations and invocations. Here are some sample procedure declarations:

```
AVG: PROCEDURE (X, Y) ADDRESS;
  DECLARE (X, Y) ADDRESS;
  RETURN (X + Y)/2;
END AVG;


AOUT: PROCEDURE (ITEM);
  DECLARE ITEM ADDRESS;
  IF ITEM >= 0FFH THEN OUTPUT(3) = 0FFH;
  ELSE OUTPUT(3) = ITEM;
  RETURN;
END AOUT;


DECLARE GLOBAL$SUM ADDRESS;
SUMSQUARE: PROCEDURE (ARG);
  GLOBAL$SUM = GLOBAL$SUM + ARG*ARG;
END;
```

You may have noticed that there is no RETURN statement in the last example. This is a legal construction; there is an implied RETURN at the END of any procedure body. Of course, if the procedure returns a value, there must be an explicit RETURN to specify it.

A final note: procedures are not allowed to be recursive. This means that a procedure may not call itself, and further that procedures may not call each other circularly.

11.2 Procedure Calls

Procedures can be invoked (i.e. executed, or activated) only following their declaration in the program text. There are two forms of procedure call, depending on whether the procedure returns a value. If a procedure does not return a value, then the procedure-type will be absent from its declaration, and the form of its call is

```
CALL procedure-name (argument-list);
```

which is a self-contained PL/M statement. If a procedure returns a value, then its declaration contains a procedure-type, and the form of its call is

```
procedure-name (argument-list)
```

which is an operand or term to be used in an expression, just as a variable-name would be used.

In both forms of procedure-invocation, the elements of the argument list are called ´actual parameters´, to distinguish them from the ´formal parameters´ of the procedure declaration. At the time of the call, each actual parameter is evaluated, and its resulting value is assigned to the corresponding formal parameter in the procedure declaration. Then the procedure body is executed. Actual parameters can be variable-names, constants, or in fact any PL/M expression. If the procedure is declared without a formal parameter list, then the actual parameter list is absent in the call.

Given the procedure declarations in section 11.1 for AVG, AOUT, and SUMSQUARE, the following are valid procedure calls:

```
X = AVG (X, Y);

CALL AOUT (X);

CALL SUMSQUARE (4);

CALL SUMSQUARE (Y + 3);

CALL AOUT (1 + AVG (X, 4));

DO WHILE AVG(X, Y) < MAX;
  X = X + XDEL;
  Y = Y + YDEL;
END;
```

Whenever there is a disagreement in type between an actual parameter and a formal parameter, automatic type-conversion takes place at the point of call. That is, an actual parameter value of type BYTE will be extended with high-order zeros when it is assigned to a formal parameter of type ADDRESS, and an actual parameter value of type ADDRESS will have its high-order 8 bits truncated when it is assigned to a formal parameter of type BYTE. The same kind of automatic type-conversion happens in two other cases of type disagreement: (1) when there is a disagreement between the value returned by a BYTE or ADDRESS procedure, and its use at the point of call; (2) when there is a disagreement between the value of a RETURN expression, and the type of the procedure.

11.3 Example

As an example of procedure declaration and call, let us consider the sorting program given earlier in section 10. We will take out the segment of the program which actually does the sorting, and declare it as a procedure. We will give this procedure a single formal parameter: the length of the array to be sorted. The procedure will return a value: the number of switches required to sort the array.

```
/* INITIAL ORDER OF A IS ARBITRARY */

DECLARE A(10) ADDRESS INITIAL
  (33, 10, 2000, 400, 410, 3, 3, 33, 500, 1999);

      .         /* BUBBLE-SORT DECLARATION */

SORT: PROCEDURE (N) ADDRESS;

  /* N = LENGTH OF A
     COUNT = NR. OF SWITCHES PERFORMED TO-DATE
     SWITCHED = (BOOLEAN) HAVE WE DONE ANY SWITCHING
                YET ON THIS SCAN?    */
  DECLARE (N, I, SWITCHED) BYTE,
     (TEMP, COUNT) ADDRESS;

  SWITCHED = 1;      /* SWITCHED=TRUE MEANS NOT DONE YET */
  COUNT = 0;
  DO WHILE SWITCHED;

    SWITCHED = 0;        /* BEGIN NEXT SCAN OF A */
    DO I = 0 TO N-2;
      IF A(I) > A(I+1) THEN
        DO;              /* FOUND A PAIR OUT OF ORDER */
        COUNT = COUNT + 1;
        SWITCHED = 1;    /* SET SWITCHED = TRUE */
        TEMP = A(I);     /* SWITCH THEM INTO ORDER */
        A(I) = A(I+1);
        A(I+1) = TEMP;
        END;
    END;
    /* HAVE NOW COMPLETED A SCAN */

  END /*WHILE*/;
  /* HAVE NOW COMPLETED A SCAN WITH NO SWITCHING */
  RETURN COUNT;

END SORT;

                /* BUBBLE-SORT INVOCATION */

DECLARE NSWITCH ADDRESS;
NSWITCH = SORT (10);

EOF
```

Compare this procedure with the program of section 10, which was a one-shot program. If we wanted to write a program in which the array got mixed up, sorted, mixed up, and sorted cyclicly, the program of section 10 would be no help. It would sort the array once and quit. But here, declared as a procedure, we can invoke it as many times as we want the array sorted.

12. POINTERS AND INDIRECT REFERENCES

Sometimes a direct reference to a PL/M data element is either impossible or inconvenient. This happens, for example, when the memory address of a data element must remain unknown until it is computed at run-time. In such cases it may be necessary to write PL/M code to manipulate the addresses of data rather than the data themselves, considering that the addresses "point to" the data. Such pointers have been called "indirect addresses", "references", and "pointers". In PL/M, the double-byte data type is called ADDRESS, to suggest this use. A PL/M programmer handles pointer computations using the language facilities described in this section.

12.1 Based Variables

A ´based variable´ is a variable which is pointed to by another variable, called its ´base´. A based variable is not allocated storage by the compiler; its value is calculated at run-time by an indirect access through its base. A based variable is declared by first declaring its base, which must be of type ADDRESS, and then declaring the based variable itself:

```
        DECLARE ITEM$POINTER ADDRESS;
        DECLARE ITEM BASED ITEM$POINTER BYTE;
```

From this point in your program forward, whenever you write ´ITEM´ you are really saying ´the BYTE value pointed to by the current value of ITEM$POINTER´. This means that the sequence

```
        ITEM$POINTER = 34AH;
        ITEM = 77H;
```

will load the BYTE value 77 (hex) into the memory location 34A (hex).

A variable is made BASED by the occurence of a base-attribute in its declaration. A base-attribute takes the form

```
                    BASED identifier
```

where the identifier names the base, or pointer variable. Unlike other declaration attributes, this base-attribute must immediately follow the name of the based variable in its declaration, as in the following examples:

```
        DECLARE X BASED A BYTE;
        DECLARE (Z BASED ZA, Y BASED YA) ADDRESS;
        DECLARE (Q BASED QA) (100) BYTE;
```

In the first example, a byte variable called X is declared. The declaration implies that X will be found at the location given by the run-time values of the ADDRESS variable A (declared elsewhere).

The second example declares 2 based variables, both of type ADDRESS. The third example defines an array called Q based at QA. The compiler will not allocate any storage to Q at compile time; the size attribute (100) merely provides values for the built-in functions LENGTH and LAST, and documents the intended use of Q.

Based variables may be subscripted like any other variables.

12.2 The Dot Operator

Based variables give us a way of talking about a referent, given its pointer; now we need a way of constructing a pointer, given the referent. This is the dot operator: the memory address of a variable is designated by preceding the variable-name with a dot character. Thus, the expressions

```
.A and .B(5)
```

yield the address of A, and the address of B(5), respectively. If A is a BYTE array, the value of .A(0)+5 is the same as .A(5); if A is an ADDRESS array, the value of .A(0)+10 is the same as .A(5). You can use the dot operator on a based variable; the result is simply the value of the base.

In general, the dot operator takes the forms

```
.variable
.constant
.(constant)
.(constant-list)
```

This means the dot operator can take a constant for an argument, as well as a variable. In this case memory storage is allocated for the constant itself, and the dot operator returns a pointer to it. For example, the construction

```
.37
```

evaluates to an address which points to a memory location containing the number 37. Likewise,

```
.'MESSAGE'
```

returns a pointer to the first character, M, of the ASCII string M-E-S-S-A-G-E. A list of constants separated by commas and enclosed by parentheses may be dotted like this:

```
.(02H, 'MIXED', 0DH, 0AH, 'CONSTANTS', 03H)
```

by the run-time values of the ADDRESS variable A (declared elsewhere).

The second example declares 2 based variables, both of type ADDRESS. The third example defines an array called Q based at QA. The compiler will not allocate any storage to Q at compile time; the size attribute (100) merely provides values for the built-in functions LENGTH and LAST, and documents the intended use of Q.

Based variables may be subscripted like any other variables.

12.2 The Dot Operator

Based variables give us a way of talking about a referent, given its pointer; now we need a way of constructing a pointer, given the referent. This is the dot operator: the memory address of a variable is designated by preceding the variable-name with a dot character. Thus, the expressions

```
.A and .B(5)
```

yield the address of A, and the address of B(5), respectively. If A is a BYTE array, the value of .A(0)+5 is the same as .A(5); if A is an ADDRESS array, the value of .A(0)+10 is the same as .A(5). You can use the dot operator on a based variable; the result is simply the value of the base.

In general, the dot operator takes the forms

```
.variable
.constant
.(constant)
.(constant-list)
```

This means the dot operator can take a constant for an argument, as well as a variable. In this case memory storage is allocated for the constant itself, and the dot operator returns a pointer to it. For example, the construction

```
.37
```

evaluates to an address which points to a memory location containing the number 37. Likewise,

```
.'MESSAGE'
```

returns a pointer to the first character, M, of the ASCII string M-E-S-S-A-G-E. A list of constants separated by commas and enclosed by parentheses may be dotted like this:

```
.(02H, 'MIXED', 0DH, 0AH, 'CONSTANTS', 03H)
```

These last two constructions are useful for passing parameters to procedures. A PRINT procedure, for instance, might take 2 formal parameters, a pointer to a message and a character count of its length. It could then be called this way:

```
CALL PRINT (14, .´STACK OVERFLOW´);
```

An address reference made with the dot operator is valid anywhere a PL/M expression is valid.

12.3 Example: Bubble-Sort

Let us return to the bubble-sort procedure that has been a part of our theme, last seen in section 11.3. There it would sort only the array A, which was declared globally. Would it not be more useful, if it would sort any array we cared to hand it? Passing an entire array is clearly awkward; getting it back is even more so, since a procedure can return at most one value. So pass the procedure a pointer to the array; then the procedure can sort the array where it already sits in memory, in place, and no motion of data is required in the procedure call and return.

More abstractly, what we are doing is passing the procedure a pointer to its parameter, rather than the value of its parameter. Inside the procedure, the formal parameter corresponding to this pointer must be declared type ADDRESS, and then it can be used as a base for a based variable. In the case of our bubble-sort example this strategy results in a program like this:

```
                /* BUBBLE-SORT DECLARATION */

SORT: PROCEDURE (PTR, N) ADDRESS;

  /* N = LENGTH OF ARRAY TO BE SORTED
     PTR = MEMORY ADDRESS OF ARRAY TO BE SORTED
     COUNT = NR. OF SWITCHES PERFORMED TO-DATE
     SWITCHED = (BOOLEAN) HAVE WE DONE ANY SWITCHING
                YET ON THIS SCAN?  */
  DECLARE PTR ADDRESS, A BASED PTR ADDRESS;
  DECLARE (N, I, SWITCHED) BYTE,
    (TEMP, COUNT) ADDRESS;

  SWITCHED = 1;     /* SWITCHED=TRUE MEANS NOT DONE YET */
  COUNT = 0;
  DO WHILE SWITCHED;

    SWITCHED = 0;       /* BEGIN NEXT SCAN OF ARRAY */
    DO I = 0 TO N-2;
      IF A(I) > A(I+1) THEN
        DO;             /* FOUND A PAIR OUT OF ORDER */
        COUNT = COUNT + 1;
        SWITCHED = 1;   /* SET SWITCHED = TRUE */
        TEMP = A(I);    /* SWITCH THEM INTO ORDER */
        A(I) = A(I+1);
        A(I+1) = TEMP;
        END;
    END;
    /* HAVE NOW COMPLETED A SCAN */

  END /*WHILE*/;
  /* HAVE NOW COMPLETED A SCAN WITH NO SWITCHING */
  RETURN COUNT;

END SORT;

                /* BUBBLE-SORT INVOCATION */

DECLARE B(10) ADDRESS INITIAL
  (33, 10, 2000, 400, 410, 3, 3, 33, 500, 1999);
DECLARE C(5) ADDRESS INITIAL
  ('A', 32, 0FFFFH, 22Q, 'EW');
DECLARE (N1, N2) ADDRESS;

N1 = SORT (.B, LENGTH(B));
N2 = SORT (.C, LENGTH(C));

EOF
```

Conceptually, the SORT procedure has a single argument: the array to be sorted. We have implemented this idea by giving the procedure 2 formal parameters: a pointer to tell where to find the array, and a count to tell its size. Compare this formulation with the bubble-sort procedure of section 11.3, which only sorts one array, the one it already knows about, array A. Use of the old procedure to sort a different array B means copying B to A, calling the SORT procedure, then copying A back to B again. Our new SORT procedure can sort any array of any length anywhere in memory: we just tell where and how big.

12.4 Example: String Comparison

This is an example of character-string handling. We declare a procedure EQUAL, which compares two character strings for equality. It is a typed procedure that returns a value TRUE (= 0FFH) if the strings match, FALSE (= 0) if they don't. EQUAL takes two parameters: pointers to the two strings to be compared. Each of the strings must be terminated by a final byte of 0FFH.

```
EQUAL: PROCEDURE (PTR1, PTR2) BYTE;

   DECLARE (PTR1, PTR2) ADDRESS;
   DECLARE (STRING1 BASED PTR1,
            STRING2 BASED PTR2) BYTE;
   DECLARE I ADDRESS, (J1, J2) BYTE;

   J1, J2, I = 0;
   DO WHILE J1=J2;
      IF J1=0FFH THEN RETURN 0FFH;
      J1 = STRING1(I);
      J2 = STRING2(I);
      I = I+1;
   END;
   RETURN 0;

END EQUAL;
```

The idea of this program is to use a do-while loop to keep searching down the strings until either a mismatch or the end of a string is encountered. A mismatch will terminate the do-while, and execution will fall through to the RETURN 0 statement; but the end of a string will provoke a return out of the middle of the do-while.

13. STATEMENT LABELS AND GO-TO´S

13.1 Label Names

Statements (or groups) may be labeled for identification and reference. A labeled statement takes the form

```
LABEL-1: LABEL-2: ... LABEL-N: STATEMENT;
```

where the label-i are vaild PL/M identifiers. Any number of labels may precede the PL/M statement. Here are some examples of labeled statements:

```
LOOP: X = X+1;
L1: CLEAN$UP: I = 0;
```

A label may also be a number. Such a label is like the ´org´ statement of many assemblers. The statement

```
30: Y = X-5;
```

specifies that the object code for this statement is to begin at memory location 30. No more than one numeric label should precede a statement; and when symbolic labels are used in conjunction with a numeric label on the same statement, the numeric label should appear first. Example:

```
128: FISH: X = (X + 2)*3;
```

The symbolic form of a label has no effect on the origin of code. Its purpose is to be a documentation and debugging aid, and to provide a target for GO TO statements.

Labels may be declared, like variables, in declaration statements. Such explicit label declaration is not usually required; normally one simply uses labels as described in this section, and no problems arise. Label declaration is discussed at some length in section 15.3.

13.2 GO TO Statements

A GO-TO statement stops the normally sequential order of program execution by transferring control directly to its ´target´. Sequential execution then resumes, beginning with the target statement. There are three distinct forms for the PL/M GO TO statement:

```
GO TO label-name;
GO TO number;
GO TO variable-name;
```

In the first form, the label-name is an identifier which appears as a label in a labeled statement. The effect of the GO TO is a transfer of program control directly to the labeled statement. In the second form, the number is an absolute memory address, and program control is transferred directly to that address. In the third form; the variable-name is that of a variable containing a pre-computed memory address; control passes directly to this absolute memory address.

These last two forms of the GO-TO are extremely dangerous, as they fail to guarentee the existence of executable code at the GO-TO target. In general, one should never use a numeric GO-TO if a symbolic GO-TO will work.

The reserved word GO TO can also be written GOTO, without the embedded blank.

Discussion of label scope, which affects the legality of certain GO-TO's, and questions of up-level transfers, are postponed to section 15 (Block Structure and Scope).

As a final note on labels: you are encouraged to use IF-THEN-ELSE and DO-group constructs instead of labels and GO TO's wherever possible. The effect in general will be better object code and more readable programs.

14. COMPILE-TIME MACRO PROCESSING

The LITERALLY declaration defines a macro for expansion at compile-time. An identifier is declared to represent a character string, which is then substituted for each occurence of the identifier in subsequent text. The form of the declaration is

```
DECLARE identifier LITERALLY ´string´;
```

where the identifier is any valid PL/M identifier, and the string is a sequence of arbitrary characters from the PL/M set, not exceeding 255 in length, enclosed in apostrophes. The following program illustrates the use of this macro facility.

```
DECLARE LIT LITERALLY ´LITERALLY´,
   DCL LIT ´DECLARE´;
DCL TRUE LIT ´0FFH´, FALSE LIT ´0´;
DCL FOREVER LIT ´WHILE TRUE´;

DCL (X, Y, Z) BYTE;

X = TRUE;
...
DO FOREVER;
   Y = Y+1;
   IF Y > 10 THEN HALT;
END;
...
EOF
```

The first declaration of this program defines abbreviations for the reserved words LITERALLY and DECLARE, which are then used throughout the program. The second declaration defines the boolean values TRUE and FALSE in a manner consistent with the way PL/M handles relational operators (see section 5.3). This often makes a program more readable.

The DO FOREVER statement in the program body first expands to DO WHILE TRUE. The macro expansion of TRUE then yields DO WHILE 0FFH; and since 0FFH has a rightmost bit of 1 (see section 7.1), the effect is an endless loop, terminated only by execution of the HALT statement within the loop.

Another use of the LITERALLY declaration is the declaration of parameters which may be fixed for one compilation, but may change from one compilation to the next. Consider the program below:

```
DECLARE BUFFER$SIZE LITERALLY '300',
   PBASE LITERALLY '4000Q',
   SUPERVISOR LITERALLY '40H';

DECLARE PRINT$BUFFER (BUFFER$SIZE) ADDRESS;
...

PBASE:
PRINTBUFFER (BUFFERSIZE-10) = 'G';
   ...
   IF ERROR THEN GO TO SUPERVISOR;
   ...
EOF
```

A future change to BUFFER$SIZE can be made in one place at the first declaration, and the compiler will propagate it throughout the program during compilation. Thus the programmer is saved the tedious and error-prone process of searching his program for the occurences of "300" that are buffer-size references, and not some other 300's.

Likewise, the starting location of the program (and any references to it from elsewhere) can be changed with a modification in the PBASE declaration. The expansion of this macro in line 5 of our program will create a numeric label; other references (not shown above) might expand into absolute GO TO's, like the statement 'GO TO SUPERVISOR'.

15. BLOCK STRUCTURE AND SCOPE

PL/M is a "block structured" language. This means that certain portions of programs, namely "blocks", can be written so there is no unwanted interaction between the block and its environment. This desirable situation stems mainly from the concept of "scope": entities which are declared within a block are inaccessible to statements or declarations outside the block; and a block may shield itself from the influence of enitities declared outside the block by suitable declarations inside the block. The use of the same identifier for different objects, one inside a block, one outside the block, creates no difficulty.

For example, there are two blocks in the following program:

```
DECLARE (A, B) BYTE;
A = 3;
DO;
   DECLARE C BYTE;
   C = A-17;
END;
B = A+200;
EOF
```

The DO-END group constitutes a block, as does the entire program. The "scope" of the variables A and B comprises the entire program, because they were declared in the outermost block. The scope of variable C is the DO-END group only, because C was declared within that block. This means that the variables A and B may be used anywhere in the program, while use of the variable C is restricted to the DO-END block. A reference to C located outside the DO-END group will be flagged by the compiler as an undefined identifier; outside its scope, the variable C simply does not exist.

15.1 How Scope is Defined

A "block" is any do-group, any procedure body, or the entire program. Each block limits the scope of those identifiers declared within it; they will be unknown outside the block. Given an identifier, its scope is determined by finding the point of its declaration, and looking forward and backward in the program text ("outward" from the declaration), to find the innermost block containing the declaration. The exact scope of the identifier then begins with its declaration, and ends with the end of the block.

The scope of an identifier, so defined, can have "holes" in it. If the scope contains an inner block, and the inner block contains a declaration that redefines the same identifier, then the scope of that inner declaration creates an area in which the outer declaration is temporarily inoperative -- masked by the inner declaration.

Study of the following example will be instructive:

```
0001      DECLARE (A, B) ADDRESS INITIAL (101, 102);
0002
0003      P: PROCEDURE (A) ADDRESS;
0004         DECLARE A BYTE;
0005         RETURN (A*A + B);
0006      END P;
0007
0008      A = P(2);
0009
0010      DO;
0011         DECLARE P(10) ADDRESS, I BYTE;
0012         DO I = 0 TO 9;
0013            P(I) = 500+I;
0014         END;
0015         A = P(2);
0016      END;
0017
0018      EOF
```

First let us consider the scope of the variable I. I is declared on line 11; the innermost block encompassing this declaration is the DO-END group comprising lines 10 to 16. Thus the scope of the variable I begins with its declaration on line 11, and ends with the end of the block on line 16.

The scope of the variable B begins with its declaration on line 1, and ends with the end of the program on line 18 -- that is to say, the scope of B is the entire program. The case of the variable A is similar, since it is declared simultaneously with B, but there is an important difference. The procedure P, whose declaration begins on line 3, contains the declaration of another variable A, whose scope is the body of the procedure P: line 3 to line 6. So there are two distinct variables named A in this program, declared at two different block levels. The outer A´s scope fails to be continuous; it extends from line 1 to line 2, and from line 7 to line 18. It is interrupted by the scope of the inner A, which occupies lines 3 to 6. Thus the multiplication on line 5 uses the inner A, the formal parameter of the procedure P; and the assignment statement on line 8 assigns a new value to the outer A, the A declared on line 1.

The scope of B is not interrupted by any inner declaration in the procedure P. That is why the reference to B on line 5, although within the procedure, is nontheless a reference to the global B declared in line 1.

Let us now consider the scope of the procedure P. Its declaration begins on line 3, and the innermost block encompassing this declaration is the entire program. The scope of the procedure is thus the entire program -- with one exception. Notice that the identifier P is declared again at line 11, this time not as a procedure, but as a 10-element array of addresses. As in the case of the identifier A, this double declaration presents no difficulty because the declaration on line 11 is contained within an inner block, in this case the DO-END group encompassing lines 10 to 16. The scope of the array P is thus from line 11 to line 16. Without this inner declaration of the identifier P, the scope of the procedure P would be the entire program; with it, the scope of the procedure is only from line 3 to line 10, and from line 17 to line 18.

The double declaration of P -- once as a procedure, once as an array -- has a curious consequence. The two statements at lines 8 and 15, although lexically identical, have different meanings. Line 15 falls within the scope of the array declaration on line 11, and thus sets the variable A equal to the third element of the array P (which the iteration of lines 12 to 14 has left equal to 502). On the other hand, line 8 falls outside the scope of the array P, and within the scope of the procedure P. Thus the assignment of line 8 invokes procedure P with an actual parameter of 2; within the procedure body the inner variable A becomes equal to 2; the value 2*2 + B, or 106, is returned as the value of the procedure call; and the outer A gets assigned the new value 106.

15.2 What is Subject to Scope

Variable names, array names, and data names have scope, as explained in the preceding section. The rules explained there apply, with one anomaly: the innermost block encompassing a variable, array, or data declaration must not be a DO-WHILE, a DO-CASE, or an iterative DO. Declarations so placed are illegal.

Procedure names have scope, following the rules explained in the preceding section. The anomaly just described holds for procedure names also: the innermost block encompassing a procedure declaration must not be a DO-WHILE, a DO-CASE, or an iterative DO.

Macro names defined in LITERALLY declarations also have scope, according to the rules of the preceding section. Here again, the innermost block encompassing a LITERALLY declaration must not be a DO-WHILE, a DO-CASE, or an iterative DO.

Labels are also identifiers, and as such have scope. But unlike variables, procedures, and macros, it is not usually required to explicitly declare label names. The first use of an undeclared label is itself an implicit declaration of the label; and this implicit declaration governs the scope of the label according to the rules of the preceding section. But there are times when a programmer must override these implicit declarations with his own explicit declarations. These issues are discussed more completely in the following section.

15.3 Scope of Labels

Just as variables and procedures have an explicit scope, The symbolic form of a statement label has an implied scope. This scope can be made explicit by a label-declaration. The form of the label-declaration element conforms to one or the other of

```
DECLARE identifier LABEL;
DECLARE (identifier-1, ... identifier-n) LABEL;
```

Such a declaration says that the label or set of labels will be defined at the block level of the declaration. This explicit label declaration is necessary only if the implied declaration does not satisfy the programmer´s intention.

Suppose we have a program containing the following statement:

```
LOOP: X = X+1;
```

This program will be compiled as if we had written

```
DECLARE LOOP LABEL;
LOOP: X = X+1;
```

where the implicit label declaration immediately precedes the occurence of the label. Mostly this turns out to be exactly what one would wish for; but here is an example which shows why the explicit declaration is sometimes required.

```
X = X+1;                  /* START OUTER BLOCK */
...
DO;                          /* START INNER BLOCK */
   ...
   GO TO EXIT;
   ...
END;                         /*  END INNER BLOCK  */
...
EXIT: HALT;
EOF                       /*  END OUTER BLOCK  */
```

Our obvious intention is to branch from the inner block to the statement labeled EXIT at the end of the program. But according to the implicit declaration rule for labels, we could have written equivalently

```
        X = X+1;                /* START OUTER BLOCK */
        ...
        DO;                        /* START INNER BLOCK */
           ...
           DECLARE EXIT LABEL;
           GO TO EXIT;
           ...
        END;                       /*  END INNER BLOCK  */
        ...
        DECLARE EXIT LABEL;
        EXIT: HALT;
        EOF                     /* END OUTER BLOCK */
```

At the first use of EXIT, the implicit declaration limits the scope of the label to the do-group. So at the second occurence of EXIT, we are outside that scope, EXIT is again undefined, and a new implicit declaration will occur. Now there are two labels due to implicit declarations, an inner and an outer. Amusingly enough, the inner label is undefined (although declared), and the GO-TO statement has nowhere to go to! To accomplish the original purpose, we should write

```
        DECLARE EXIT LABEL;     /* START OUTER BLOCK */
        X = X+1;
        ...
        DO;                        /* START INNER BLOCK */
           ...
           GO TO EXIT;
           ...
        END;                       /*  END INNER BLOCK  */
        ...
        EXIT: HALT;
        EOF                     /*  END OUTER BLOCK  */
```

Now everything will work out properly: at the first use of the label (in the GO-TO statement) it has already been declared, and this use lies within the scope of that declaration. The implict declarations are suppressed, as they are not required; there is but one label EXIT, and its scope is now the entire program, without restriction.

15.4 Use of Block Structure

Transfer of control from one block nesting level to another should always be done by entering the block at its beginning and leaving it at its end, or (for a procedure body) leaving by means of a RETURN statement.

For example, a GO-TO statement which contrives to jump into the middle of a procedure body will leave the run-time pushdown stack in an undefined state, and continued execution of the program will produce unpredictable results. A procedure body should be entered only by means of a call on the procedure.

A GO-TO leaving a procedure body has similar trouble with the run-time stack, since it by-passes the orderly RETURN mechanism. Because of this, it is illegal to write a GO-TO inside a procedure that transfers control outside the procedure, unless its target is at the outermost block level of the program. Such unconditional up-level transfers are sometimes justified by the convenience of global error exits, or by abort-and-restart conditions.

Need for inter-block GO-TO's is quite rare, and programs may often be rewritten to remove them, using alternative PL/M control structures. Excessive use of GO-TO's will make programs hard to debug and modify.

It is recommended that, within any given block, all declarations be put at the beginning of the block, preceeding executable statements. The scope of identifiers so declared may then be visualized as the extent of the entire block. This simplification also prevents an important class of programming errors: mistaken identification of the "innermost encompassing block".

Programmers find their work greatly facilitated by proper layout of a program on the pages of its program listing. Blocks (procedures, do-groups) are frequently set off by blank lines. The body of each block is indented by a fixed number of spaces from the code in which it occurs; thus the opening and closing lines of the block are vertically aligned. When you look at a program listing it should be easy to see its block nesting structure at a glance, without reading the code in detail.

Block structure in a programming language provides the opportunity to define truly independent program modules, letting the compiler do the work of keeping them independent. Procedures can be made independent of their environment (except for number and types of parameters). Procedures can be moved from one program to another, with no surprises resulting from new declaration conflicts. Complete self-contained modules, together with conventional macro definitions, can form a project or department library -- greatly reducing program development time.

16. PRE-DECLARED VARIABLES AND BUILT-IN PROCEDURES

Pre-declared variables and built-in procedures are assumed to be declared in an all-encompassing global block invisible to the programmer. Such invisible declarations can be overridden by inner declarations -- which distinguishes these special identifiers from reserved words. A list of these pre-declared identifiers is given in appendix E.

16.1 INPUT and OUTPUT

The form of an input call is

INPUT (number)

It is used in expressions exactly as a BYTE procedure call would be, and its value is the 8-bit quantity latched in the specified input port of the CPU. The numeric constant argument must be in the range 0 - 255 for the 8080, and in the range 0 - 7 for the 8008.

The pseudo-variable OUTPUT always appears as the left part of an assignment statement; elsewhere it is illegal. (In particular, it never appears as the destination of an imbedded assignment.) Its form is

OUTPUT(number) = expression;

where the numeric constant argument must be in the range 0 - 255 for the 8080, and in the range 0 - 23 for the 8008. Its effect is to latch the 8-bit value of the expression into the specified output port.

In the 8080 CPU, there are 512 I/O ports: 256 input ports and 256 ouput ports, each group being numbered 0 through 255. These physical (hardware) port designations are identical with the PL/M constants that appear as arguments for INPUT and OUTPUT.

In the 8008 CPU, there are 32 I/O ports, numbered 0 through 31. The first 8 of these are reserved for input, the remaining 24 for ouput. The correspondence between these physical (hardware) port designations, and the PL/M designations, is given by the table below:

8008 physical port number	PL/M
0	INPUT(0)
1	INPUT(1)
2	INPUT(2)
...	...
7	INPUT(7)
8	OUTPUT(0)
9	OUTPUT(1)
...	...
30	OUTPUT(22)
31	OUTPUT(23)

16.2 LENGTH and LAST

PL/M has 2 built-in functions based on the declared sizes of arrays. These functions take the forms

```
LENGTH (identifier)
LAST (identifier)
```

where "identifier" is any previously declared variable name, array name, or data identifier. These forms may appear anywhere an expression is allowed in a PL/M program. They evaluate to the declared length of the variable, and the index of the final element of the variable, respectively. The following program uses the LAST function to set all the elements of a vector V to the constant 5:

```
DECLARE V(100) BYTE;
DECLARE I BYTE;

DO I = 0 TO LAST(V);
  V(I) = 5;
END;

EOF
```

For any identifier VAR, LENGTH(VAR) = 1 + LAST(VAR), on the condition that LAST is defined. LENGTH is defined for all variables, no matter how declared, but LAST is not defined for variables declared to have length zero.

16.3 The Functions LOW, HIGH, and DOUBLE

Two built-in type-transfer procedures convert ADDRESS values to BYTE values. They both return BYTE values, and take ADDRESS arguments, as follows:

```
        LOW (expression)
        HIGH (expression)
```

LOW returns the low-order byte of its argument; HIGH returns the high-order byte of its argument.

A third type-conversion procedure, DOUBLE, converts a byte value to an ADDRESS value by padding it on the left with a high-order byte of zeros.

Calls to these three type-conversion procedures are valid anywhere an expression is valid. They may never appear as the destination of an assignment statement.

16.4 Shift and Rotate Functions

16.4.1 BYTE Rotation Functions

Calls to the two functions ROL and ROR take the forms

```
                    ROL (expr-1, expr-2)
                    ROR (expr-1, expr-2)
```

where both expr-1 and expr-2 must evaluate to BYTE quantities; a single BYTE value is returned in both cases. ROL rotates expr-1 to the left, the bit count of the rotation being given by expr-2. ROR returns the corresponding right rotation. By ´rotate´ we mean that any bits falling off the end in the direction of the rotation, come back in the other end. For example,

```
        ROR(10011101B, 1) returns a value of 11001110B;
        ROL(10011101B, 2) returns a value of 01110110B.
```

ROL and ROR have the side-effect of setting CARRY according to the last bit rotated off the end and around. In the first example above, CARRY will be set; in the second example, CARRY will be cleared.

Important restriction: expr-2 must be non-zero.

16.4.2 CARRY-Rotation Functions

Calls to the two functions SCL and SCR take the forms

```
                    SCL (expr-1, expr-2)
                    SCR (expr-1, expr-2)
```

where expr-2 must evaluate to a BYTE quantity, but expr-1 may be either a BYTE value or an ADDRESS value. If it's of type BYTE, then the function will return a BYTE value; if it's of type ADDRESS, then the functon will return an ADDRESS value.

The first parameter (expr-1) is rotated left (SCL) or right (SCR) according to a count given by the second argument (expr-2), just as with ROL and ROR. But with SCL and SCR, the rotation includes the CARRY bit: the bit rotated off one end of the argument is rotated into CARRY; the old value of CARRY is rotated into the other end of the argument. In effect, SCL and SCR perform 9-bit rotations on 8-bit arguments, and 17-bit rotations on 16-bit arguments.

Suppose that CARRY is clear. Then SCL(10011101B, 1) returns the value 00111010B, and sets CARRY as a side-effect. Similarly, if CARRY starts out clear, then SCR(10011101B, 2) returns the value 10100111B, and clears CARRY as a side-effect. The same principles hold for 16-bit arguments.

Important restriction: expr-2 must be non-zero.

16.4.3 Logical-Shift Functions

Calls to the two functions SHL and SHR take the forms

```
SHL (expr-1, expr-2)
SHR (expr-1, expr-2)
```

where expr-2 must evaluate to a BYTE quantity, but expr-1 may be either a BYTE value or an ADDRESS value. If it's of type BYTE, then the function will return a BYTE value; if it's of type ADDRESS, then the function will return an ADDRESS value.

The first parameter (expr-1) is shifted left (SHL) or right (SHR) according to a bit count given by the second argument (expr-2). Bits shifted off the left end (SHL) or the right end (SHR) are shifted into the CARRY; zeros are shifted in the other end. The previous value of CARRY is always lost. For example, SHL (10011101B, 1) returns the value 00111010B and sets CARRY as a side-effect; SHR (10011101B, 2) returns the value 00100111B, and clears CARRY as a side-effect.

Important restriction: expr-2 must be non-zero.

16.5 Interrupt Control Statements

Two special statements are provided for control of the 8080 interrupt facility: ENABLE and DISABLE. Their functions and usage are explained in some detail in section 18 of this manual, and so will not be repeated here. PL/M for the 8008 does not support these statements.

16.6 Carry, Zero, Sign, Parity

There are four identifiers used to test the 8008 and 8080 CPU condition codes:

CARRY ZERO SIGN PARITY

An occurence of one of these identifiers (in an expression) generates a test of the corresponding condition flip-flop. If the flip-flop is set (= 1), a value of 0FFH is returned; if the flip-flop is clear (= 0), a value of 0 is returned.

16.7 The Decimal Arithmetic Function

A pre-declared function called DEC facilitates computations in BCD (binary-coded-decimal) numbers. This pre-declared DEC function is described in the section of this manual covering decimal arithmetic, section 17. PL/M for the 8008 does not support this facility.

16.8 The MEMORY Vector

Often it is useful to reference the area of free memory that follows the space allocated to variables. This facility is provided by an implicit declaration

```
DECLARE MEMORY (0) BYTE;
```

as the last declaration of every program.

As an example, consider the following program: assuming 10 memory pages of 256 bytes each, we want to leave all unallocated memory set to ones.

```
DECLARE SIZE LITERALLY ´2559´;
DECLARE I ADDRESS;
DO I = .MEMORY TO SIZE;
   MEMORY (I - .MEMORY) = 1;
END;
EOF
```

16.9 The TIME Procedure

The built-in procedure TIME causes a time delay specified by its actual parameter. The form of the call is

```
CALL TIME (expression);
```

where the expression evaluates to a BYTE quantity. The length of time measured by the procedure is a multiple of 100 microseconds: if the actual parameter evaluates to n, then the delay caused by the procedure is 100n usec. For example, the procedure call

```
CALL TIME (45);
```

returns after 4.5 milliseconds. Since the maximum delay offered by the procedure is 25.5 milliseconds, longer delays must be obtained by repeated calls. The following loop takes one second to execute:

```
DO I = 1 TO 40;
   CALL TIME (250);
END;
```

The TIME function is based on the 8008 and 8080 CPU cycle times, and assumes that in your system the memory cycle time is fast enough to permit the CPU to run full speed. If this condition fails, or if the CPU goes into a ´hold´ state during execution of the TIME function, then delay times become unpredictable.

16.10 STACKPTR

The Intel 8080 (unlike the 8008) has a run-time pushdown stack in memory, rather than in the CPU itself. The 8080 references this memory stack by means of a stackpointer register in the CPU, which always contains the memory address of the (current) top item on the stack. 8080 PL/M gives the programmer direct access to this register by means of the pseudovariable STACKPTR, which may be used in the two following constructions only:

```
variable = STACKPTR;
STACKPTR = expression;
```

(´variable´ means non-subscripted PL/M variable; ´expression´ means any PL/M expression.)

A PL/M programmer should require access to the stack pointer only under extreme and unusual circumstances. Taking control of the stack away from the compiler frustrates the compile-time checks on stack overflow, invalidates the compiler´s assumptions about the run-time states of the stack, and results in unreliable programs. If such action seems necessary nontheless, programmers are advised to study the 8080 PL/M run-time environment, before making use of STACKPTR. The necessary documentation will be found in the appropriate compiler manual.

17. DECIMAL ARITHMETIC FACILITIES

8080 PL/M has operations which greatly simplify decimal arithmetic, providing some straightforward conventions are followed. Let all operands (variables and constants) be BYTE values, each containing two 4-bit fields. Each field will represent one decimal digit by a binary number in the range 0 - 9. Such a BYTE value will be called a BCD-pair. (The representation of a number as a string of decimal digits, where each digit is represented by a 4-bit binary number, is called BINARY-CODED-DECIMAL representation.) In an 8080 BCD-pair, the least significant (rightmost) 4-bit field represents the least significant decimal digit of the pair. We will write a BCD-pair as a PL/M hexadecimal number, because each hexadecimal digit will conveniently represent a 4-bit BCD digit. Here are some valid BCD-pairs:

```
23H   96H   10H
```

and here are some invalid BCD-pairs:

```
2FH   0A0H   77Q   31
```

BCD-pairs are added using the + or PLUS operators, and then the result is made into a BCD-pair again by means of the DEC function. The form of a call to DEC is one of the following:

```
DEC (el + e2)
DEC (el PLUS e2)
```

where el and e2 are BCD-pair values: non-subscripted BYTE variables, BCD-pair constants, or expressions resulting in BCD-pair values, such as properly nested calls to DEC. The effect of a call on DEC in one of the above forms, is to return the BCD-pair representing the BCD sum of el and e2, including the possible carry from the low-order to the high-order digit of the pair. If the sum exceeds 99H, the 8080 carry flip-flop is set. Some examples will clarify these definitions. In the table below, the calls to DEC on the left produce the corresponding results on the right:

DEC (22H + 22H)	44H, CARRY=clear
DEC (36H + 36H)	72H, CARRY=clear
DEC (73H + 81H)	54H, CARRY=set
DEC (DEC(22H+22H) + 33H)	77H, CARRY=clear
DEC (DEC(22H+22H)+DEC(22H+22H))	88H, CARRY=clear

The operator PLUS can be used in place of the operator + if decimal numbers with more than two digits must be added. For example, to add 1234 and 4678, we first add the low-order BCD-pairs 34H and 78H, then the high-order BCD-pairs 12H and 46H, taking account of the carry from the low-order pair to the high-order pair. In PL/M, we have

```
DECLARE (UPPER, LOWER) BYTE;
LOWER = DEC (34H + 78H);
UPPER = DEC (12H PLUS 46H);
```

The low-order result 12H is now in LOWER, the high-order 59H in UPPER. It was important not to disturb the 8080 carry flip-flop between the two calls to DEC, so that the PLUS operator was not misled. This can be assured in general by permitting only scalar assignments to separate the calls.

Here is another example: suppose we wish to obtain the sum of two decimal numbers, each 6 digits in length. One number is stored in the three BYTE variables X1, X2, X3; the other is stored in the three BYTE variables Y1, Y2, Y3. We want the result to appear in Z1, Z2, Z3. We write this program:

```
DECLARE (X1, X2, X3) BYTE,
   (Y1, Y2, Y3) BYTE,
   (Z1, Z2, Z3) BYTE;
Z3 = DEC (X3 + Y3);      /* LOW-ORDER RESULT    */
Z2 = DEC (X2 PLUS Y2);   /* MIDDLE-ORDER RESULT */
Z1 = DEC (X1 PLUS Y1);   /* HIGH-ORDER RESULT   */
```

Now we generalize to a procedure which sums two numbers, each represented by a vector of BCD-pairs (X and Y), and leaves the result in the vector Z. The digits of each number are assumed to be stored such that the least significant BCD-pair is at subscript position zero.

```
DECLARE (I, CY) BYTE,
   (X, Y, Z) (10) BYTE, (U, V, W) BYTE;
...
CY = 0;
DO I= 0 TO LAST(X);
   U = X(I);
   V = Y(I);
   W = DEC (U + CY);
   CY = CARRY;
   W = DEC (W + V);
   CY = (CY OR CARRY) AND 1;
   Z(I) = W;
END;
```

No direct facility is provided by 8080 PL/M for decimal arithmetic other than addition. Subtraction is easily accomplished by complement arithmetic: given a BCD-pair X, the value

99H - X

is the nines-complement of X. Subtraction of a number is accomplished by adding its nines-complement. Decimal multiplication and division can be done by repeated addition and subtraction, using shift-and-add or shift-and-subtract algorithms if the application warrants. Unpacked decimal (one digit per byte) is always an option if BCD-pair operations become too involved.

18. INTERRUPT PROCESSING FACILITIES

8080 PL/M includes language facilities for use with the 8080 interrupt mechanism to process interrupts generated externally. Fundamentally, an interrupt is an external asynchronous call on a PL/M procedure. When the interrupt is accepted, the executing process is stopped, the machine state is saved, and a specific interrupt-handling procedure is invoked. When the interrupt procedure does a return, the previous machine state is restored and control returns to the interrupted process.

Up to 8 different interrupt procedures can be included in a PL/M program, corresponding to the 8 restart instructions RST 0 through RST 7.

The 8080 interrupt mechanism is controlled by the PL/M statements

```
DISABLE;
ENABLE;
```

The DISABLE statement causes the 8080 CPU to enter a state wherein interrupts are masked. The ENABLE statement causes the 8080 to leave that state, so that incoming interrupts are processed as they occur. The 8080 CPU starts from power-up with interrupts disabled; interrupts must be explicitly enabled before any interrupt procedures can be invoked.

An interrupt procedure in 8080 PL/M is a parameterless and typeless procedure, with the INTERRRUPT attribute in its declaration. The form of this attribute is

```
INTERRUPT n
```

where n is a number in the range 0 to 7, corresponding to one of the eight possible interrupts. Interrupt procedures must be declared only in the outermost block of a program.

For example, the following interrupt procedure is invoked whenever a RST 3 instruction is jammed into the 8080 interrupt port with interrupts enabled.

```
        DECLARE KEYMAX LITERALLY ´72´;
        DECLARE KEYBUFF (KEYMAX) BYTE, KEYPTR BYTE;
        DECLARE OVERFLOW LABEL;

        KEYBOARD$PROCESS: PROCEDURE INTERRUPT 3;
           DECLARE CHAR BYTE;
           KEYPTR = KEYPTR+1;
           IF KEYPTR > KEYMAX THEN GO TO OVERFLOW;
           IF (CHAR := INPUT(5)) = ´$´ THEN RETURN;
           KEYBUFF(KEYPTR) = CHAR;
        END KEYBOARD$PROCESS;

        KEYPTR = .(KEYBUFF);
        ENABLE;
        /* MAIN PROGRAM */
        ...

        OVERFLOW:
        /* KEYBOARD BUFFER OVERFLOW */
        ...

        EOF
```

In this example, KEYBOARDPROCESS operates on the global variables KEYPTR and KEYBUFF each time RST 3 is executed. If KEYPTR exceeds KEYMAX then control is transferred to the outer block label OVERFLOW and the saved machine state is discarded -- control never returns to the interrupted process. If KEYPTR does not exceed KEYMAX then the value of input port 5 is read and stored into CHAR. If the value of CHAR is ASCII dollar sign, then the interrupt procedure returns immediately to the interrupted process. Otherwise the value of CHAR is placed in the vector KEYBUFF and control returns to the interrupted process.

The 8080 interrupt mechanism is disabled by the occurence of an interrupt, and may be explicitly enabled with an ENABLE statement inside the interrupt procedure. Interrupts are enabled by a return from an interrupt procedure. Caution should be exercised when enabling interrupts inside an interrupt procedure: two activations of the same interrupt procedure must never be in process simultaneously, since there is only one data area for both activations. This exclusion can be accomplished by specifically disabling the interrupt source, or by establishing a priority of interrupts with external circuitry. The safest method is to leave interrupts disabled during all interrupt processing.

Interrupt procedures may contain nested non-interrupt procedures. On completion of a call, these nested procedures return to their point of call inside the interrupt procedure in which they are defined; it is only the RETURN´s at the outermost interrupt procedure level which cause the machine state of the interrupted process to be restored.

Similarly, procedures at the same level, or global to a particular interrupt procedure, can be invoked from inside the interrupt procedure. The programmer must ensure, however, that any data areas referenced by such a global procedure are not sensitive to actions of the interrupt procedure. For example, it would be dangerous to do floating point multiplications or divisions inside an interrupt procedure, because such multiply and divide procedures would almost certainly have local variables. If an interrupt comes during a multiply, and the interrupt procedure re-enters the multiply code, these local data areas will be corrupted. The interrupt procedure will complete its execution correctly, but the return from the interrupt will not be able to restore the original machine state.

Interrupt procedures can be called directly from the outer block of the program, or from another procedure, if desired. The programmer must be aware, of course, that interrupts are always enabled on exit from an interrupt procedure, even though the procedure may have been entered via a call rather than an external interrupt.

Note also that an 8080 in the halt state with interrupts disabled cannot be restarted except by applying the appropriate reset to the 8080 chip. This is why the HALT statement in PL/M enables the interrupt mechanism immediately before stopping the CPU.

THE VOCABULARY

terminal symbols	
1	!
2	;
3	HALT
4	ENABLE
5	DISABLE
6	IF
7	THEN
8	ELSE
9	DO
10	CASE
11	INTERRUPT
12	<number>
13	PROCEDURE
14	<identifier>
15	)
16	(
17	,
18	END
19	:
20	RETURN
21	CALL
22	GO
23	TO
24	GOTO
25	DECLARE
26	LITERALLY
27	<string>
28	DATA
29	BYTE
30	ADDRESS
31	LABEL
32	BASED
33	INITIAL
34	=
35	:=
36	OR
37	XOR
38	AND
39	NOT
40	<
41	>
42	+
43	-
44	PLUS
45	MINUS
46	*
47	/
48	MOD

nonterminals

<program>
<statement list>
<statement>
<basic statement>
<if statement>
<assignment>
<group>
<procedure definition>
<return statement>
<call statement>
<go to statement>
<declaration statement>
<label definition>
<if clause>
<true part>
<expression>
<group head>
<ending>
<step definition>
<while clause>
<case selector>
<variable>
<replace>
<iteration control>
<to>
<by>
<while>
<procedure head>
<procedure name>
<type>
<parameter list>
<parameter head>
<go to>
<declaration element>
<type declaration>
<data list>
<data head>
<constant>
<identifier specification>
<bound head>
<initial list>
<variable name>
<identifier list>
<based variable>
<initial head>
<left part>
<logical expression>
<logical factor>

49	.	<logical secondary>
50	BY	<logical primary>
51	WHILE	<arithmetic expression>
52		<relation>
53		<comp>
54		<term>
55		<primary>
56		<constant head>
57		<subscript head>

<program> is the goal symbol.

THE PRODUCTIONS

```
 1   <program>  ::=   <statement list>

 2   <statement list>  ::=   <statement>
 3                       !   <statement list> <statement>

 4   <statement>  ::=   <basic statement>
 5                  !   <if statement>

 6   <basic statement>  ::=   <assignment> ;
 7                        !   <group> ;
 8                        !   <procedure definition> ;
 9                        !   <return statement> ;
10                        !   <call statement> ;
11                        !   <go to statement> ;
12                        !   <declaration statement> ;
13                        !   HALT ;
14                        !   ENABLE ;
15                        !   DISABLE ;
16                        !   ;
17                        !   <label definition> <basic statement>

18   <if statement>  ::=   <if clause> <statement>
19                     !   <if clause> <true part> <statement>
20                     !   <label definition> <if statement>

21   <if clause>  ::=   IF <expression> THEN

22   <true part>  ::=   <basic statement> ELSE

23   <group>  ::=   <group head> <ending>

24   <group head>  ::=   DO ;
25                   !   DO <step definition> ;
26                   !   DO <while clause> ;
27                   !   DO <case selector> ;
28                   !   <group head> <statement>

29   <step definition>  ::=   <variable> <replace> <expression> <iteration control>

30   <iteration control>  ::=   <to> <expression>
31                          !   <to> <expression> <by> <expression>

32   <while clause>  ::=   <while> <expression>

33   <case selector>  ::=   CASE <expression>

34   <procedure definition>  ::=   <procedure head> <statement list> <ending>
```

```
 35   <procedure head>  ::=   <procedure name> ;
 36                       !   <procedure name> <type> ;
 37                       !   <procedure name> <parameter list> ;
 38                       !   <procedure name> <parameter list> <type> ;
 39                       !   <procedure name> INTERRUPT <number> ;

 40   <procedure name>  ::=   <label definition> PROCEDURE

 41   <parameter list>  ::=   <parameter head> <identifier> )

 42   <parameter head>  ::=   (
 43                       !   <parameter head> <identifier> ,

 44   <ending>  ::=   END
 45               !   END <identifier>
 46               !   <label definition> <ending>

 47   <label definition>  ::=   <identifier> :
 48                         !   <number> :

 49   <return statement>  ::=   RETURN
 50                         !   RETURN <expression>

 51   <call statement>  ::=   CALL <variable>

 52   <go to statement>  ::=   <go to> <identifier>
 53                        !   <go to> <number>

 54   <go to>  ::=   GO TO
 55              !   GOTO

 56   <declaration statement>  ::=   DECLARE <declaration element>
 57                              !   <declaration statement> , <declaration element>

 58   <declaration element>  ::=   <type declaration>
 59                            !   <identifier> LITERALLY <string>
 60                            !   <identifier> <data list>

 61   <data list>  ::=   <data head> <constant> )

 62   <data head>  ::=   DATA (
 63                  !   <data head> <constant> ,

 64   <type declaration>  ::=   <identifier specification> <type>
 65                         !   <bound head> <number> ) <type>
 66                         !   <type declaration> <initial list>

 67   <type>  ::=   BYTE
 68             !   ADDRESS
 69             !   LABEL

 70   <bound head>  ::=   <identifier specification> (
```

```
 71   <identifier specification>  ::=   <variable name>
 72                                 !   <identifier list> <variable name> )

 73   <identifier list>  ::=   (
 74                        !   <identifier list> <variable name> ,

 75   <variable name>  ::=   <identifier>
 76                      !   <based variable> <identifier>

 77   <based variable>  ::=   <identifier> BASED

 78   <initial list>  ::=   <initial head> <constant> )

 79   <initial head>  ::=   INITIAL (
 80                     !   <initial head> <constant> ,

 81   <assignment>  ::=   <variable> <replace> <expression>
 82                   !   <left part> <assignment>

 83   <replace>  ::=   =

 84   <left part>  ::=   <variable> ,

 85   <expression>  ::=   <logical expression>
 86                   !   <variable> := <logical expression>

 87   <logical expression>  ::=   <logical factor>
 88                           !   <logical expression> OR <logical factor>
 89                           !   <logical expression> XOR <logical factor>

 90   <logical factor>  ::=   <logical secondary>
 91                       !   <logical factor> AND <logical secondary>

 92   <logical secondary>  ::=   <logical primary>
 93                          !   NOT <logical primary>

 94   <logical primary>  ::=   <arithmetic expression>
 95                        !   <arithmetic expression> <relation> <arithmetic expression>

 96   <relation>  ::=   =
 97                 !   <
 98                 !   >
 99                 !   <comp>

100   <comp>  ::=   < >
101             !   < =
102             !   > =

103   <arithmetic expression>  ::=   <term>
104                              !   <arithmetic expression> + <term>
105                              !   <arithmetic expression> - <term>
106                              !   <arithmetic expression> PLUS <term>
107                              !   <arithmetic expression> MINUS <term>
```

```
108                               !   - <term>

109   <term>  ::=   <primary>
110             !   <term> * <primary>
111             !   <term> / <primary>
112             !   <term> MOD <primary>

113   <primary>  ::=   <constant>
114                !   . <constant>
115                !   <constant head> <constant> )
116                !   <variable>
117                !   . <variable>
118                !   ( <expression> )

119   <constant head>  ::=   . (
120                      !   <constant head> <constant> ,

121   <variable>  ::=   <identifier>
122                 !   <subscript head> <expression> )

123   <subscript head>  ::=   <identifier> (
124                       !   <subscript head> <expression> ,

125   <constant>  ::=   <string>
126                 !   <number>

127   <to>  ::=   TO

128   <by>  ::=   BY

129   <while>  ::=   WHILE
```

APPENDIX B, ASCII codes

The ASCII (American Standard Code for Information Interchange) was adopted by the American National Standards Institute, Inc. (ANSI) in 1968. The standard itself, as distinct from the summary here presented, is available from ANSI, 1430 Broadway, New York, NY 10018, as USAS X3.4-1968. A previous version of this standard was adopted by the, National Bureau of Standards as a Federal Information Processing Standard (FIPS 1). ASCII is a seven-bit code, which we are representing here by a pair of hexadecimal digits.

00	NUL	20	SP	40	@	60	´
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	´	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	[(braces)
1C	FS	3C	<	5C	\	7C	! (bar)
1D	GS	3D	=	5D	]	7D	] (braces)
1E	RS	3E	>	5E	^	7E	(tilde)
1F	US	3F	_	5F	_	7F	DEL

SYMBOL	NAME	USE
$	dollar sign	compiler toggles, number and identifier spacer
=	equal sign	relational test operator, assignment operator
:=	assign	imbedded assignment operator
.	dot	address operator
/	slash	division operator
/*		left comment delimiter
*/		right comment delimiter
(	left paren	left delimiter of lists, subscripts, and expressions
)	right paren	right delimiter of lists, subscripts, and expressions
+	plus	addition operator
-	minus	subtraction operator
'	apostrophe	string delimiter
*	asterisk	multiplication operator
<	less than	relational test operator
>	greater than	relational test operator
<=	less or equal	relational test operator
>=	greater or equal	relational test operator
<>	not equal	relational test operator
:	colon	label delimiter
;	semicolon	statement delimiter
,	comma	list element delimiter

RESERVED WORD	USE
IF	conditional tests and alternative execution
THEN	
ELSE	
DO	statement grouping and procedure definition
PROCEDURE	
INTERRUPT	
END	
DECLARE	data declarations
BYTE	
ADDRESS	
LABEL	
INITIAL	
DATA	
LITERALLY	
BASED	
GO	unconditional branching and loop control
TO	
BY	
GOTO	
CASE	
WHILE	
CALL	procedure call
RETURN	procedure return
HALT	machine stop
ENABLE	interrupt enable
DISABLE	interrupt disable
OR	boolean operators
AND	
XOR	
NOT	
MOD	remainder after division
PLUS	add with carry
MINUS	subtract with borrow
EOF	end of input file (compiler control)

CARRY
DEC
DOUBLE
HIGH
INPUT
LAST
LENGTH
LOW
MEMORY
OUTPUT
PARITY
ROL
ROR
SCL
SCR
SHL
SHR
SIGN
STACKPTR
TIME
ZERO