intel®
INTEL CORP. 3065 Bowers Avenue, Santa Clara, California 95051 • (408) 246-7501

MCS-8 T.M.

A Guide to PL/M Programming

PL/M is a new high level programming language designed specifically for Intel's 8 bit microcomputers. The new language gives the microcomputer systems programmer the same advantages of high level language programming currently available in the mini and large computer fields. Designed to meet the special needs of systems programming, the new language will drastically cut microcomputer programming time and costs without sacrifice of program efficiency. In addition, training, documentation, program maintenance and the inclusion of library subroutines will all be made correspondingly easier. PL/M is well suited for all microcomputer programming applications, retaining the control and efficiency of assembly language, while greatly reducing programming effort. The PL/M compiler is written in ANSI standard Fortran IV and thus will execute on most machines without alteration.

SEPTEMBER 1973
REV. 1

TABLE OF CONTENTS

A GUIDE TO PL/M PROGRAMMING

I. INTRODUCTION TO PL/M.

PL/M is a programming language designed specifically for the INTEL MCS-8 Microcomputer. The language is structurally similar to PL/I (in particular, PL/M closely resembles XPL), with data types and primitive operations which reflect the architecture of the MCS-8 CPU. Thus, the systems designer can use PL/M to quickly and easily express programs which execute on the MCS-8 CPU, with little or no loss in execution efficiency when compared to assembly language programming. In addition, programs written in PL/M are somewhat self-documenting, are easily altered and maintained, and provide upward software compatibility in the INTEL 8000 CPU series. That is, programs written in PL/M for the 8008 CPU can be recompiled for the 8080 CPU with no alteration of the source program. In each case, the resulting object code takes advantage of the particular target CPU architecture.

The discussion of PL/M given here is in two main sections. Section II provides a tutorial description of PL/M; only a minimal amount of programming experience is assumed, and the discussion is mainly expository. Section III presents a more formal approach to PL/M, providing the exact syntactic structure and corresponding actions of each statement in PL/M. Section III is intended as a reference manual, but may be used as an introduction to PL/M by readers who are familiar with block structured languages similar tc PL/I or XPL.

The remaining sections provide system notes on the use of PL/M, including compiler error messages, control toggles, and execution controls and commands. Appendix A contains sample PL/M programs; it may be useful for the reader to refer occassionally to this appendix to find instances of the various statements as they are discussed in Sections II and III.

II. A TUTORIAL APPROACH TO PL/M.

As mentioned above, this section describes the PL/M programming language from a tutorial viewpoint. The various structures of PL/M are introduced at various levels of complexity. Examples of each of the constructs are also given. The overall structure of a PL/M program is given first.

1. The Organization of a PL/M Program.

A PL/M program is arranged as a sequence of declarations and statements separated by semicolons. The declarations allow the programmer to control allocation of storage, define simple macros, and define procedures. Procedures are subroutines which are invoked through certain statements in PL/M. These procedures may contain further declarations which control storage allocation and define nested procedures. The procedure definition capabilities of PL/M allow modular programming; that is, a particular program can be divided into a number of subtasks, such as processing teletype input, converting from binary to decimal forms, and printing output messages. Each of these subtasks is written as a procedure in PL/M. These procedures are conceptually simple, are easy to formulate and debug, are easily incorporated into a large program, and form a basis for library subroutine facilities when writing a number of similar programs.

In addition to the procedure declaration facilities, PL/M allows a number of data types to be declared and used in a program. The two basic data types are Byte and Address. A Byte variable or constant is one which can be represented in an eight-bit word, while an Address variable or constant requires sixteen bits (double byte). The programmer can declare variable names in a PL/M program to represent Byte and Address values. PL/M also allows the vectors of Byte or Address variables to be declared.

A number of arithmetic, logical, and relational operations are defined in PL/M on Byte and Address variables and constants. These operators and values are combined to form expressions which resemble elementary algebraic expressions. The PL/M expression

X * (Y - 3) / R

represents the calculation of the value of X times the quantity Y-3 divided by the value of R. When values in expressions are both Byte and Address type, PL/M automatically converts the Byte value to an Address value.

Expressions are the major components of most PL/M statements. A simple statement form is the PL/M assignment statement which allows the programmer to compute a result and store it in a location defined by a variable name. Thus, the assignment

Q = X * (Y - 3) / R

first causes the computation of the expression to the right of the equal sign. The result of this computation is then saved in the memory location represented by the variable name Q.

Additional statements are provided in PL/M for conditional tests and branching, iteration control, and procedure invocation with parameter passing.

Input and output statements in PL/M allow the programmer to read the eight-bit value latched into a particular MCS-8 input port, or set the value of an eight-bit output port. Procedures can be defined which use these basic input and output statements to perform more complicated I/O functions.

A compile-time macro processing facility is also provided in PL/M. This facility allows the programmer to define a name in the program to represent an arbitrary sequence of characters. Each time the name is encountered, the corresponding character sequence is substituted into the source program.

The section which follows provides a detailed description of the format of a PL/M program.

2. Basic Constituents of a PL/M Program.

PL/M programs are written in free-form. That is, the input lines are column independent and blanks can be freely inserted between the elements of the program. The only requirement is that the declarations and statements are all terminated with a semicolon. The characters recognized by PL/M are given below. These characters can be combined to form <u>identifiers</u> and <u>reserved words</u>.

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

0 1 2 3 4 5 6 7 8 9

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

along with the special characters

$ = . / ( ) + - ' * , < > : ;

all other characters are ignored by PL/M (a blank is substituted for an unrecognized character).

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

<u>Symbol</u> <u>Name</u> <u>Use</u>

$	dollar sign	compiler controls, number and identifier spacer
=	equal	relational test and assignments
:=	assign	imbedded assignments
.	dot	address indicator
/	slash	division symbol and comment delimiter
( )	parens	list and subscript delimiter
+	plus	addition
-	minus	subtraction
'	apostrophe	string delimiter
*	asterisk	multiplication and comment delimiter
<	less	relational tests
>	greater	"
<=	less or equal	"
>=	greater or equal	"
<>	not equal	"
:	colon	label delimiter
;	semicolon	declaration and statement delimiter

2.2. Identifiers and Reserved Words. A PL/M *identifier* is used to represent names of variables, procedure names, macro names, and statement label names. Identifiers can be up to 31 characters in length; the first character must be alphabetic, and the remaining characters can be alphabetic or numeric. Imbedded dollar signs ($) are ignored by PL/M, and can be used to improve readability of a name. Thus, valid identifiers are

X

GAMMA

LONGIDENTIFIER

INPUT$COUNT

Note, however, that there are a number of *reserved words* in PL/M which cannot be used as names in a PL/M program. These reserved words are shown below

Reserved Word	Use
IF	conditional tests and branching
THEN	
ELSE	
DO	statement grouping
PROCEDURE	and procedure definition
END	
DECLARE	data declarations
BYTE	
ADDRESS	
LABEL	
INITIAL	
DATA	
LITERALLY	
BASED	
GO	unconditional branching
TO	and iteration control
BY	
GOTO	
CASE	
WHILE	
CALL	subroutine call
RETURN	subroutine return
HALT	machine stop
OR	logical or
AND	logical and
XOR	logical xor
NOT	logical not
MOD	remainder after division
PLUS	add with carry
MINUS	subtract with carry

EOF end-of-file

Blanks may be inserted freely around identifiers and special characters. Blanks are not necessary, however, when two identifiers are separated by a special character. Thus, the expression

X * (Y - 3) / R

is equivalent to

X*(Y-3)/R

in PL/M.

2.3. Comments. Explanatory remarks can be used throughout a PL/M program to improve readability and provide a measure of self-documentation. Comments are sequences of symbols from the character set of PL/M bounded by the symbol pairs /* and */. Thus, the sequence

/*THIS IS A COMMENT ABOUT COMMENTS*/

is completely ignored by the PL/M compiler, and has no effect on the program. Comments may be freely interspersed in a PL/M program, and may appear anywhere a blank is valid.

3 PL/M Statement Organization.

The statements found in PL/M programs are one of three basic types: simple statements, conditional statements, and groups.

An example of a simple statement is the PL/M assignment

A = B + C * D;

Note that simple statements are always followed by a semicolon. Other forms of simple statements are defined in later sections.

Conditional statements are preceded by the reserved word IF and contain one or more other statements as a part of the statement body. A conditional statement could be written in PL/M as

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

which assigns the value of B to the variable A only if A's value is greater than B's value.

A more complicated conditional statement involves an alternative, denoted by the reserved word ELSE. The conditional

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

assigns the larger of the two values A and B to the variable C.

Statements can be collected together in groups which are delimited by the reserved words DO and END. These groups of statements are then treated as a single statement in the flow of control. The group could, for example, become a part of a conditional statement:

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

which would perform the two assignments to A and B only if A is greater then B.

Simple statements, conditional statements, and groups can be labelled for control flow purposes. The label may be a PL/M identifier, which precedes the statement, and is separated from the statement by a colon (:). Thus,

```
LAB1: A = B + C * D;
```

is an example of a simple statement labelled by LAB1.

The exact details of the various simple, conditional, and statement groups are discussed in following sections.

4. PL/M Data Elements.

PL/M data elements represent single bytes, double bytes, and strings corresponding to 8-bit values, 16-bit values, and ASCII character strings of length greater than two. Data elements can be either variables or constants. Variables are PL/M identifiers corresponding to values which can change during execution of a PL/M program, while constants have a value which is fixed. The expression

X * (Y - 3) / R

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

Variables must declared in PL/M programs before they are used in expressions. The declaration tells the PL/M compiler how to handle expressions and assignments which involve the variable.

4.1. Variable Declarations. A declaration for a variable or set of variables is headed by the reserved word DECLARE and followed by either a single identifier or a list of identifiers enclosed in parenthesis, and terminated by one of the data types BYTE or ADDRESS. Thus, valid PL/M declarations are:

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

Thus, expressions involving only the variables X, Q, R, and S produce single byte operations, while expressions involving U, V, or W would produce double byte operations and results.

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

4.2. Byte and Double Byte Constants. Constants representing single and double byte values can be expressed in several different ways in PL/M. First, PL/M accepts constants in the binary, octal, decimal, and hexadecimal bases. In addition, ASCII strings of length one or two are translated to single and double byte constants.

In general, the base of a constant is represented by one of the letters

B O Q D H

following a sequence of digits. The letter B represents a binary constant, while the letters O and Q denote octal constants. The letter D optionally follows decimal numbers. Hexadecimal numbers consist of sequences of hexadecimal digits (0,1, ... ,9,A,B,C,D,E,F) followed by the letter H. Note that the leading digit of a hexadecimal number must be a decimal digit to avoid confusion with a PL/M identifier (a leading 0 is always sufficient). Any number not followed by one of the letters B, O, Q, D, or H is assumed to be decimal. The numbers must always be capable of representation as a single or double byte value (a maximum of 16 bits). Thus, the following are valid constants in PL/M

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

The dollar sign symbol may be freely inserted within constants to improve readability. Thus, the binary constant

11110110011B

could be expressed as

111$1011$0011B

ASCII strings are represented by PL/M characters enclosed within apostrophe symbols ('). Strings of length one or two translate to byte and double byte values as mentioned previously. Thus, the string

'A'

is the same as 65 decimal. A pair of apostrophes ('') within a string results in a single apostrophe in the internal representation of the string. Thus, the string '''Q' becomes a single apostrophe followed by the character Q.

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 the usual algebraic notation. Thus, an expression consists of a simple data element, such as a number or variable, or an expression can be two (sub)expressions separated by an operator:

<u>expression1 operator expression2</u>

Examples are

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

Operators in expressions have an assumed <u>precedence</u> which determines the order in which the operations in the expression are evaluated. The valid PL/M operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence and are evaluated from left-to-right when they occur in an expression.

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

The expression

```
A + B * C
```

for example, results first in the computation of B times C since the multiplication (*) has a higher precedence than the addition (+). The result of this computation is then added to the value of A.

Parenthesis can be used to override the assumed precedence by enclosing subexpressions which are to be computed first. The expression

```
( A + B ) * C
```

causes A + B to be evaluated first. The result is then multiplied by C's value. Following are a number of well-formed PL/M expressions

```
A + B - C * D
A - ( B + C ) * D
A / ( B + C ) * D
A / ( B + C )
A OR B AND 0FH
A + B > C - D
```

Each expression results in either a single or double byte value. The number of bytes in the result is determined by the number of bytes required by the subexpressions in the result. Generally, if both operands in an expression are byte values, the result is a byte value. If either operand, however, is a double byte, the result is a double byte value. In this case, the shorter operand is padded with high-order zeroes.

Two exceptions to these rules occur in PL/M. The first is in the case of the *, /, and MOD operations. These operators always result in a double byte value. The second exception is the case of relational operators. A relational test results in either a true or false condition. A true condition is represented in PL/M by a byte value equal to 255 (all bits are 1's), and a false condition is represented by the byte value 0.

Suppose the variables X, Y, and Z have been declared as follows:

DECLARE X BYTE ;

DECLARE (Y,Z) ADDRESS;

given these declarations, the expressions below yield results with the precision shown to the right of the expression:

X + 5 single byte result

X + 300 double byte result

X + Y double byte result

Y + Z double byte result

X / 5 double byte result

X + (Y > Z) single byte result

The NOT operator is a <u>unary</u> operator, and thus PL/m expressions involving NOT take the form

NOT <u>expression</u>

The effect of the NOT operator is that all the bits of the expression are inverted (1,s become 0's, and 0's become 1's). In particular, true conditions change to false conditions, and false conditions revert to true. Examples of the use of the NOT operator are

NOT A

NOT (A > B)

NOT A OR B

For convenience, a unary minus sign is also allowed in PL/M expressions. The form of the unary minus in an expression is

\- <u>expression</u>

The effect is exactly the same as the expression

0 - <u>expression</u>

where the "-" in this last case is the subtract operator. The expression -1, for example, is equivalent to 0-1, resulting in the byte value 255.

Recall that the assignment statement is used to store the result of an expression into a variable. The declared precision of the assigned variable affects the resulting store operation. If the assigned variable is a single byte variable, and the expression is a double byte 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 as type ADDRESS, the high order byte is set to zero.

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 equal sign, separated by commas. The variables A, B, and C could all be set to the expression X + Y with the single assignment

A, B, C = X + Y

A special form of the assignment is allowed _within_ expressions in PL/M. The form of an _imbedded assignment_ is

(_variable_ := _expression_)

and may appear anywhere an expression is allowed in PL/M. The expression to the right of the assign symbol (:=) is evaluated and then stored into the variable on the left. The value of the imbedded assignment is the same as the expression on the right. The expression

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

results in exactly the same value as

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

except that the intermediate results C+D and F/C are stored into B and E, respectively. These intermediate computations can then be used at a later point in the program without recomputation.

Note that the form

A = (B := (C := X + Y))

has exactly the same effect as the multiple assignment to A, B, and C given previously.

It is now possible to construct a simple program based upon these expressions and assignments.

6. A Simple Example.

The following PL/M sample program reads data from input ports 0 and 1, and writes the larger of these two values at output port 0. Note that the two pseudo-variables INPUT(0), and INPUT(1) act like PL/M single byte variables, but have the effect of reading the values latched into input ports 0 and 1, respectively. Similarly, the pseudo-variable OUTPUT(0) can be used in an assignment statement in order to write values to output port 0.

The complete PL/M program for performing this simple function is shown below

```
DECLARE (I,J,MAX) BYTE;
/* READ INPUT PORT 0 AND SAVE IN VARIABLE I */
LOOP:
    I = INPUT(0);
/* NOW READ INPUT PORT 1 AND SAVE IN VARIABLE J */
    J = INPUT(1);
/* SET MAX TO THE LARGER OF THESE TWO VALUES */
    IF I > J THEN MAX = I; ELSE MAX = J;
/* WRITE THE VALUE OF MAX AT OUTPUT PORT 0 */
    OUTPUT(0) = MAX;
/* GO BACK AND READ THE INPUT PORTS AGAIN */
GO TO LOOP;
EOF
```

The symbol EOF (end-of-file) is required in PL/M to indicate the end of the program. Note also that the GO TO statement causes program control to restart at the point labelled 'LOOP:' where input values are read again.

In order to effectively construct more comprehensive PL/M programs, it is necessary to consider the structure of PL/M statement groups, including the loop control groups.

7. DO Groups.

As mentioned previously, statements can be grouped together within the bracketing reserved words DO and END as a DO-group. Recall that the simplest DO-group is of the form

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

Several additional DO-groups are defined in PL/M which control program flow. These groups are shown below.

7.1. The DO-WHILE Group. One form of the DO-group is called a DO-WHILE. The DO-WHILE has the form

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

In this case, the expression following the reserved word WHILE is evaluated before the statements within the group are executed. If the expression evaluates to true (i.e., the rightmost bit of the result is 1), the statements up to the corresponding END are executed. At the end of the group, program control is transferred to the top of the DO-group and the expression is evaluated again. The group is executed over and over until the expression results in a false condition (the rightmost bit is 0). Consider the following example:

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

The statement A = A + 1 will be executed exactly three times. The value of A at the end of execution of the group is four.

7.2. The Iterative DO-group. An Iterative DO-group allows a group of statements to be executed a fixed number of times. The simplest form of the Iterative DO-group is

```
DO variable = expression1 TO expression2;
statement-1;
statement-2;
 . . .
statement-n;
END;
```

The effect of this group is to first store expression1 into the variable following the DO. The group is executed with this initial value once, and control returns to the top of the DO. The value of the variable is incremented by 1 and tested against expression2. If the incremented value exceeds expression2, control transfers to the statement following the END; otherwise, the group is executed once again. An example is

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

Note that this 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;
```

A slightly more complicated form of an Iterative DO-group allows a stepping value other than 1. This second form is

```
DO variable = expr1 TO expr2 BY expr3;
statement-1;
statement-2;
 . . .
statement-n;
END;
```

In this case, the variable following the DO is stepped by the value expr3 instead of by 1.

7.3. The DO-CASE. Another form of the DO-group is the DO-CASE statement. The form of a DO-CASE group is

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

The effect of this group is the following. Upon entry to the DO-CASE, the expression following the CASE is evaluated. The result of this expression is a value k which must be between 0 and n-1. This value k is used to select one of the n statements of the DO-CASE to execute. The first case corresponds to k = 0 (statement-1), the second case corresponds to k = 1 (statement-2), and so-forth. Control transfers to the selected statement, the statement is executed, and control then passes to the statement following the END.

An 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;
        /* CASE 2 */
        DO I = 3 TO 10; A = A + I;
        END;
    END /* OF CASES */ ;
```

Before giving more comprehensive examples, it is useful to define the notion of a subscripted variable and its use in a PL/M program.

8. Subscripted Variables and the INITIAL Attribute.

It is often useful in PL/M to reference memory locations with an "offset" from some base address. This feature is allowed in PL/M through subscripting.

8.1. Subscript Declarations and Value References. A subscripted variable is similar to a simple variable with the addition of an expression enclosed within parentheses following the variable name. The location referenced by the subscripted variable is the sum of the base address of the variable and the subscript expression. Any variable name can be subscripted in PL/M.

Suppose a PL/M programmer declares the variables X, Y, and Z as follows

```
DECLARE (X,Y,Z) BYTE;
```

The first memory location can be referenced simply as X or as the subscripted variable X(0). Similarly, X(1) refers to the location Y, and X(2) references Z's location.

PL/M also allows a fixed number of locations to be set aside in the declaration statement. These fixed locations start at the variable name specified in the declare statement. For example, the statement

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

provides a memory area of 100 bytes starting at X. In this case, X is called a vector. Note that the size of a vector must always be a constant.

Several vectors of the same length can be declared in the same declare statement. The statement

```
DECLARE (U,V,W) (50) ADDRESS;
```

causes three vectors of length 50 (each) to be allocated in contiguous memory locations. Note, however, that these vectors are of type ADDRESS, and thus each element requires two bytes; hence, U takes up the first 50 two-byte locations, requiring 100 bytes altogether. The storage for the second vector starts at V and requires the next 100 bytes. Similarly, W occupies the 100 byte area following V.

As mentioned previously, a subscript can be thought of as a displacement from a base address. This displacement, however, is affected by the declared precision of the variable. That is, if the declared precision is BYTE, then the displacement is measured in single bytes. If, however, the variable is type ADDRESS, the displacement is measured in double bytes. Thus, given the declaration of U, V, and W above, the first element of U is U(0), and the last element is U(49). The first element of V is V(0), or U(50). Storage is always arranged so that double byte variables are at memory addresses which are even numbers; hence, there is sometimes one extra word allocated between contigous byte and double byte variables.

Before continuing, it should be noted that the subscripts can be complicated expressions, and not necessarily just the simple constants shown above. Note also that subscripted variables can occur everywhere a simple variable is allowed, including expressions and assignments. A single exception to this rule is that a subscripted variable cannot be used as the indexing variable in an Iterative DO group.

Two _built in functions_ are provided in PL/M which are based upon the declared size of a vector. These functions take the forms

LENGTH(identifier) and LAST(identifier)

where the identitifers correspond to variables declared previously. These forms can appear anywhere an expression is allowed in PL/M, and result in the declared length and last element number of the specified variable, respectively. The following program, for example, 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
```

8.2. The INITIAL Attribute. The values of variables can be initialized in a declaration statement using the INITIAL attribute. This attribute takes the form

INITIAL (constant-1,constant-2,...,constant-n);

and must directly follow the type (BYTE or ADDRESS) in the declare statement.

The purpose of the INITIAL attribute is to preset the values of memory locations starting at the location named in the declarations. The constants given in the INITIAL attribute are placed into memory before the program starts (these constants become a part of the object code and must be loaded into random-access memory). The following are valid variable declarations which use 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,33);
DECLARE U(100) ADDRESS INITIAL(3,4,333Q);
DECLARE (Q,R,S) BYTE INITIAL(0,1,2);
```

Note that the number of bytes required to hold the constants given in the INITIAL attribute need not correspond to the length declared for the variable. The constants are placed into memory without truncation starting at the first byte allocated in the declare statement.

The use of subscripted variables is shown in the example which follows.

9. A Sorting Program.

It is now possible to construct a more complicated program, given the expressions, DO-groups, and subscripted variables which have been presented. In the program which follows, a vector A is initialized to a set of constants in unsorted order. The program below sorts the values of A into ascending order.

```
/* FIRST DECLARE A VECTOR TO HOLD THE
VALUES TO SORT.
ASSUME THERE ARE NO MORE THAN 10 ELEMENTS TO BE
SORTED.  EACH ELEMENT IS BETWEEN 0 AND 65535 */
DECLARE A(10) ADDRESS INITIAL
(33,10,2000,400,410,3,3,33,500,1999);
/* START THE 'BUBBLE SORT' AT THIS POINT
EXAMINE ADJACENT ELEMENTS OF 'A' AND SWITCH INTO
ASCENDING SEQUENCE.  RECYCLE UNTIL NO MORE
SWITCHING OCCURS */
DECLARE (I,SWITCHED) BYTE,
        TEMP ADDRESS;
SWITCHED = 1;
    DO WHILE SWITCHED; SWITCHED = 0;
```

```
/* GO THROUGH 'A' ONCE AND LOOK FOR A PAIR
WHICH NEEDS TO BE REVERSED */
    DO I = 0 TO 8;
    IF A(I) > A(I+1) THEN
        DO; SWITCHED = 1;
        TEMP = A(I); A(I) = A(I+1);
        A(I+1) = TEMP;
        END;
    END;
END;
/* THE VALUES IN 'A' ARE NOW IN ASCENDING ORDER */
EOF
```

10. Procedure Definitions and Procedure Calls.

The procedure capabilities of PL/M are discussed in this section. A procedure, or subroutine, is a section of PL/M source code which is declared, but not executed immediately. Instead, the procedure is called from various parts of the program. The call amounts to a transfer of program control from the calling point to the procedure. The procedure executes, and, upon completion, returns to the statement following the call.

The use of procedures in PL/M allows construction of modular programs, allows construction and use of subroutine libraries, eases programming and documentation, and reduces generated code when similar program segments are used at several points in the program.

Procedures are described in two parts: how to define them, and how to use them.

10.1. Procedure Declarations. A procedure declaration consists of four main parts: the procedure name, specification of values which are sent to the procedure, the type of the returned value (i.e., BYTE, ADDRESS, or no returned value), and the description of the actions of the procedure, called the procedure body. The procedure may be invoked anywhere in the program after it is declared. The form of a procedure declaration is

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

The procedure-name is any valid PL/M identifier, and is used to name the procedure so that it can be called at a later point in the program.

The argument-list takes the form

(argument-1,argument-2,...,argument-n)

where argument-1 through argument-n are valid PL/M identifiers. These identifiers are called <u>formal parameters</u> and are used to hold particular values which are sent to the procedure from the point of invocation. Each of these parameters must also appear in a declarations statement within the procedure body (before the corresponding END). Note that the argument-list can be omitted altogether if no parameters are passed to the procedure.

The procedure-type is either BYTE, ADDRESS, or can be omitted if the procedure does not return a value to the calling point. The procedure-type defines the precision of the value returned so that proper type conversion takes place when the procedure is invoked as a part of an expression.

The execution of a procedure is terminated with a RETURN statement in the procedure body. The RETURN statement takes the form

RETURN;

or

RETURN expression;

The first form is used if the procedure-type is omitted (no value is returned to the calling point). The second form is used if the procedure-type is BYTE or ADDRESS. The expression following the RETURN is brought back to the calling point in this case.

The statements within the procedure body can be any valid PL/M statements, including nested procedure definitions and invocations. A number of valid PL/M procedure declarations are listed below.

```
NULL: PROCEDURE;
    RETURN;
    END NULL;
SUM: PROCEDURE(X,Y);
    DECLARE (X,Y) ADDRESS:
    /* ASSUME U IS PREVIOUSLY DECLARED */
    U = X + Y;
    RETURN;
    END SUM;
ZERO: PROCEDURE BYTE;
    RETURN 0;
    END ZERO;
IDENTITY: PROCEDURE (X) ADDRESS;
    DECLARE X ADDRESS;
    RETURN X;
    END IDENTITY;
PLUSXY: PROCEDURE (X,Y) BYTE;
    DECLARE (I,X,Y) BYTE;
    I = X - Y;
    RETURN X + Y;
    END PLUSXY;
```

10.2. Procedure Calls. Procedures can be invoked anywhere after their declaration. There are two possible forms of the call, depending upon whether the procedure-type is present or omitted in the procedure declaration.

If the procedure-type is omitted, then the procedure does not return a value to the point of invocation. In this case, the form of the call is

CALL procedure-name argument-list

where the procedure-name and argument-list correspond to those defined above. The effect in PL/M is to assign the actual values in the argument-list at the call to the identifiers given in the argument-list in the procedure declaration. The elements of the argument-list in the call are called <u>actual</u> <u>parameters</u>, and are not restricted to simple PL/m identifiers. In fact, any valid PL/M expression can be placed in the argument-list. These expressicns are all evaluated in the actual parameter list before they are assigned to the corresponding identifiers in the formal parameter list. If the procedure is declared with an empty formal parameter list then the actual parameter list is also omitted. Control is then transferred to the beginning of the procedure named by the procedure-name.

Thus, given the procedure definitions above, the following are all valid procedure calls

```
CALL NULL;
CALL SUM (5,3) ;
CALL SUM(Q,R + Z);
```

In the last case, for example, the value of Q is first placed into X in the procedure SUM. The value of R + Z is then computed and stored into the formal parameter Y. Control then passes to the procedure SUM where the variable U is set to the sum of these two values (it is assumed that U has been declared ahead of the procedure SUM). Note that automatic type conversion occurs between BYTE and ADDRESS values when the actual parameters are assigned to the formal parameters.

The second form of a procedure call occurs when the procedure is declared with a procedure-type of BYTE or ADDRESS. In this case, the procedure call results in a value which can be used in an expression. The form of the call is

procedure-name argument-list;

and may appear anywhere a PL/M expression is allowed. The following calls demonstrate a number of valid PL/M procedure invocations

```
I = IDENTITY(I);
X = PLUSXY(X,Y);
X = Q-PLUSXY(X+Y,Q)/(X-Y);
DO I=PLUSXY(Q,R) TO PLUSXY(Z+R,Q)+10; END;
```

As an example of a procedure declaration and call, consider the sorting program given earlier. The segment of the program which performs the sort can be redefined as a procedure. Assume the procedure has a single formal parameter which gives the upper bound of the sort loop. The value returned by the procedure is the number of switches required to sort the vector.

```
DECLARE A(10) ADDRESS INITIAL
(33,10,2000,400,410,3,3,33,500,1999);
SORT: PROCEDURE(N) ADDRESS;
    /* SORT THE VECTOR AT 'A' OF LENGTH
    N + 2.  RETURN THE NUMBER OF SWITCHES
    REQUIRED TO PERFORM THE SORT */
    DECLARE (N,I,SWITCHED) BYTE,
        (T1,T2,COUNT) ADDRESS;
    SWITCHED = 1; COUNT = 0;
        DO WHILE SWITCHED; SWITCHED=0;
            DO I = 0 TO N;
            T1 = A(I); T2=A(I+1);
```

```
                IF T1 > T2 THEN
                    DO; A(I+1) = T1;
                    A(I) = T2; SWITCHED = 1;
                    COUNT = COUNT + 1;
                    END;
                END;
            END;
        RETURN COUNT;
        END SORT;
    /*  THE SORT PROCEDURE IS DECLARED ABOVE.
    CALL SORT WITH N -2 = 10 - 2 = 8 */
    DECLARE NSWITCHES ADDRESS;
    NSWITCHES = SORT (8);
    EOF
```

The program shown above illustrates a difficulty in parameter passing which has not yet been considered. In particular, the SORT procedure would be much more useful as a library subroutine if several different vectors could be processed by the same subroutine. As shown, the SORT procedure is only capable of sorting the particular vector A.

The next section introduces the notion of based variables which overcome this difficulty.

11. Based Variables.

Based variable features of PL/M allow computation of variable addresses during execution of a program. A based variable is similar to the variables discussed previously, except that no storage is allocated for the variable. Instead, corresponding to each based variable is an address variable, called the *base*, which determines the memory address for the based variable during execution.

Based variables are declared using the BASED attribute which specifies the base. The form of the BASED attribute is

BASED identifier

where the identifier is a previously declared ADDRESS variable name. The BASED attribute must immediately follow the name of the based variable in the declaration statement. The following are examples of PL/M based variable declarations

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

In the first case, a byte variable called X is declared. The declaration implies that X will be found at the location given by the address variable A (which must be declared as an ADDRESS variable elsewhere).

The second declaration above defines two based variables X and Y both of type ADDRESS which are located at XA and YA, respectively.

The third declaration defines a vector based variable called Q based at QA. Note that the vector size need not be stated, however, since no storage is allocated to Q by the PL/M compiler. The only use for the vector size is to provide values for the LENGTH(Q) and LAST(Q) built-in functions described previously.

In order to make effective use of based variables, it is necessary to allow programmatic reference to the assigned address of a non-based variable. The memory location assigned to a variable is designated by preceding the variable name with a dot symbol (.). Thus, the expressions

.A and .A(5)

yield the address of A and the address of A(5), respectively. If A is a BYTE variable, the value of .A+5 is the same as .A(5). Similarly, if A is of type ADDRESS, then .A+10 is the same as .A(5). The address reference to a based variable is allow and results simply in the value of the base.

An address reference using the dot symbol can be used anywhere an expression is valid in PL/M.

As an illustration of the use of based variables, consider the following loop which initializes the elements of a vector to their respective element numbers

```
DECLARE A(100) ADDRESS;
DECLARE I BYTE;
    DO I = 0 TO LAST(A);
    A(I) = I;
    END;
EOF
```

This same function can be performed (rather inefficiently) with the following loop using based variables

```
 DECLARE A(100) ADDRESS,
    QA ADDRESS, Q BASED QA ADDRESS;
/* SET QA TO THE BASE ADDRESS OF A*/
QA = .A;
DECLARE I BYTE;
    DO I = 0 TO 99;
    Q = I; QA = QA + 2;
    END;
EOF
```

Note that QA starts at the base of A and moves up by two bytes on each iteration since each element of A occupies two bytes.

Based variables are most commonly found in procedure parameter passing. It is often necessary to return more than one value from a procedure. In this case, the address of an actual parameter can be passed to the procedure instead of the value of the actual parameter. The corresponding formal parameter is declared within the called procedure as an address variable. This formal parameter is then used as a base for a based variable whithin the procedure. Any changes to the based variable then alter the corresponding actual parameter.

In the case of the SORT procedure, for example, the address of a vector to be sorted can be sent as an actual parameter. The SORT procedure then operates upon a locally defined based variable. The revised SORT procedure is shown below

```
SORT: PROCEDURE(Q,N) ADDRESS;
    DECLARE (N,I,SWITCHED) BYTE,
        (Q,T1,T2,COUNT) ADDRESS;
        /* AND THEN SET UP THE BASED
        VARIABLE TO SORT */
        DECLARE A BASED Q ADDRESS;
    SWITCHED = 1; COUNT = 0;
        DO WHILE SWITCHED; SWITCHED=0;
            DO I = 0 TO N;
            T1 = A(I); T2=A(I+1);
            IF T1 > T2 THEN
                DO; A(I+1) = T1;
                A(I) = T2; SWITCHED = 1;
                COUNT = COUNT + 1;
        END;END;END;
    RETURN COUNT;
    END SORT;
DECLARE B(10) ADDRESS INITIAL
    (33,10,2000,400,410,3,3,33,500,1999);
DECLARE C(5) ADDRESS
    INITIAL('A',32,0FFFH,22Q,2D);
/* NOW SORT THE VECTORS B AND C */
```

```
DECLARE (N1,N2) ADDRESS;
N1 = SORT(.B,LAST(B)-1)·;
N2 = SORT(.C,LENGTH(C)-2);
EOF
```

The SORT procedure has two formal parameters Q and N. Q is an ADDRESS variable which gives the base address of the vector to be sorted. The parameter N gives the upper bound in the sort loop, as before. The variable A is declared inside SORT as an ADDRESS variable based at Q. Thus, references to A inside SORT are actually references to memory locations starting at the value of Q.

The SORT procedure is called twice. First, the vector B is sorted by sending the base address of B. The second call sorts C by passing the base address of C as the first actual parameter.

The section which follows introduces the concept of a long constant. These long constants allow manipulation of data which exceed two bytes in length.

12. Long Constants.

Recall that PL/M allows direct representation of numeric and string constants which require a single or double byte internal representation. It is often useful, however, to manipulate constants of indefinite length. This facility is provided in PL/M through the use of *long constants*.

A PL/M long constant is a set of contiguous memory locations represented by the address of the first byte. The memory locations for long constants are allocated in the same area as the program storage, and are initialized to the string and numeric values specified in the constant (program steps and long constants are normally a part of the Read Only Memory portion of storage, and thus cannot be altered during execution). The first form of a long constant is simply

. constant

where the constant is a string or numeric value. The result of this expression is an address value providing the location of the constant. The second form allows several constants to be gathered together and based at the same address. This form is

. (constant-1,constant-2,...,constant-n)

Again, the result of this expression is an address value giving the starting position of the constants in memory.

Valid PL/M long constants are

```
. 335
. 'THIS IS A LONG CONSTANT STRING'
.('THREE','STRING','CONSTANTS')
.(3,'CONSTANTS',0FFE2H)
```

These long constants can appear anywhere a PL/M expression is allowed.

Another form of a long constant allows the constant to be named and accessed as a subscripted variable. This second form is a particular case of the declare statement called a DATA declaration. The form is

DECLARE identifier DATA (constant-1,...,constant-n);

The following are valid PL/M DATA declarations

```
DECLARE X DATA ('LONG STRING');
DECLARE Y DATA (0,1,2,3,'STRING',4);
```

These two declarations have an effect similar to INITIAL declarations except that new values cannot generally be assigned to the elements of X and Y. In addition, there is an automatic vector size assigned to elements declared in a DATA declaration which is the number of bytes required to hold the constants listed in the DATA attribute. In the above case, both X and Y are treated as BYTE variables with vector size 11. As a result, the LENGTH and LAST built-in procedures can be applied to DATA variables to determine the length of the constant string.

Given the above DATA declaration, the expressions below evaluate to the result shown on the right

```
X(0) = 'L'
X(10) = 'G'
Y(3) = 3
LENGTH(Y) = 11
```

As an example, consider the following PL/M procedure, called EQUAL, which compares two long constants for equality. EQUAL has two formal parameters which give the base addresses of two long constants. The last byte of each constant is 0ffh. EQUAL returns a 1 if the constants match, and 0 if not.

```
EQUAL: PROCEDURE (AS1,AS2) BYTE;
    DECLARE (AS1,AS2,I) ADDRESS,
        (S1 BASED AS1, S2 BASED AS2) BYTE,
        (J1,J2) BYTE;
    /* COMPARE UNTIL A MISMATCH OR 0FFH
    IS FOUND IN BOTH STRINGS */
    J1, J2, I = 0;
        DO WHILE J1 = J2;
        IF J1 = 0FFH THEN RETURN 1;
        J1 = S1(I); J2 = S2(I);
        I = I + 1;
        END;
    RETURN 0;
    END EQUAL;
```

Assume that the following declarations occur in the program

```
DECLARE X DATA ('WALLAWALLAWASH',0FFH);
```

```
DECLARE Y DATA ('WALLAWASH',OFFH);
```

The EQUAL procedure can be called by

```
I = EQUAL(.X,.('WALLAWALLAWASH',OFFH));
```

As a result, I is set to 1. The value of I in the case

```
I = EQUAL(.X,.Y)
```

is zero since the strings X and Y differ.

As a final comment, one should note that the fundamental difference between DATA variables and BYTE variables with the INITIAL attribute is in the allocation of storage. DATA variables are stored in the same area as program code, as mentioned previously, and cannot generally be altered through a PL/M assignment. BYTE variables, on the other hand, are allocated in alterable program storage. The INITIAL attribute provides data which is preloaded into these locations before the program executes (and hence is volatile storage). In this case, these initial values can always be changed with assignment statements during execution.

13. Scope of Variables.

An important concept in any block-structured language, such as PL/M, is the notion of variable scope. The scope of a variable in PL/M is the range of statements where the variable can be used in expressions and assignments. The scope of variables is controlled by the arrangement of DO-groups and DECLARE statements. A variable is available for use only within the DO-END statements in which the DECLARE statement for the variable occurs. This range is called the scope of the declared variable.

Consider the following PL/M program, for example:

```
1  DECLARE (A,B,C,D) BYTE;
2  B,C = 10;
3  A = B + C;
```

```
4        DO;
5        DECLARE (Q,R,S) BYTE;
6        Q, R = 20;
7        S = A + Q + R;
8        END;
9    D = 2 + A;
10   EOF
```

The declaration on line 1 defines four variables A, B, C, and D which can be used throughout the program. The DO-group between lines 4 and 8 contains a declaration of three variables Q, R, and S which are defined only within the group; that is, although A, B, C, and D can be used anywhere in the program, the variables Q, R, and S cannot be referenced outside the range of statements beginning on line 4 and ending on line 8. These lines delimit the scope of Q, R, and S.

A more complicated structure is given by the following skeletal PL/M program

```
    DECLARE (A,B,C,D) BYTE; /* BLOCK 1 */
     . . .
         DO; /* BLOCK 2 */
         DECLARE (A,E,F,G) BYTE;
         . . .
             DO; /* BLOCK 3 */
             DECLARE (B,H,I,J) BYTE;
             . . .
             END;/* OF BLOCK 3 */
         . . .
         END; /* OF BLOCK 2 */
     . . .
         DO; /* BLOCK 4 */
         DECLARE (A,E,K,L) BYTE;
         . . .
         END; /* OF BLOCK 4 */
```

```
. . .
/* BLOCK 1 IS COMPLETED */
EOF
```

The declaration of A, B, C, and D at the top of block 1 makes these variables _global_ to any nested inner blocks in the program. That is, they can be referenced anywhere in the program where there is no conflicting declaration.

The variables A, E, F, and G at the top of block 2 are said to be _local_ to block 2 and global to block 3. These variables cannot be referenced outside block 2. Note that the variable A in block 2 conflicts with the declaration of A in block 1. In this case, any reference to A within block 2 refers to the innermost declaration of A. Similarly, the variables B, H, I, and J declared at the top of block 3 cannot be accessed outside block 3. Again, the declaration of B in block 3 overrides the outer block declaration of this variable name.

Block 4 is _parallel_ to block 2 in this program. The variables A, E, K, and L are local to block 4. Thus, the variables E, K, and L are undefined outside block 4, and references to A outside block 4 affect the variable A declared on the first line.

The notion of scope of variable names extends to procedure names and to formal parameters declared within procedures. A procedure declaration is treated the same as a DO-group in defining scope of variables. As an example, consider the following program

```
/* BLOCK 1 */
DECLARE (I,J,K) BYTE;
P1: PROCEDURE(I,Q) BYTE;
    /* BLOCK 2 */
    DECLARE (I,Q,J,R) ADDRESS;
```

```
        . . .
        END P1 /* AND BLOCK 2 */;
    P2: PROCEDURE (J,Q,R) ADDRESS;
        /* BLOCK 3 */
        DECLARE (J,Q,R,S,T) BYTE;
        . . .
        END P2 /* AND ALSO BLOCK 3 */
    . . .
    /* BLOCK 1 IS FINISHED */
    EOF
```

The variables I, J, and K are global to both the P1 and P2 procedures. The procedures P1 and P2 constitute independent parallel blocks, each with their own local variables. Note that the local variable I declared in procedure P1 is used in all references to I within block 2, instead of the global variable declared in line 1. Note also that the variable Q defined in P1 is completely independent of the Q declared in P2.

The principal advantage to the scope of variable concept in PL/M is that subroutines are independent of the program in which they are imbedded, with no problems arising from conflicting declarations. In particular, library subroutines can be written as completely modular subprograms with no dependence upon the names used outside the procedure.

14. Statement Labels and GO TO's.

PL/M allows program statements to be identified with a statement label, and allows unconditional transfer of program control to these labelled statements.

14.1. Label Names. A PL/M labelled statement takes the form label-1: label-2: ... label-n: statement;

where label-1 through label-n are valid PL/M identifiers or constants. Any number of labels may precede a PL/M statement. Valid labelled statements are

```
L1: X = X + 1;
LOOP: Y = 3;
L1: LOOP: X = Y + 5;
30: Y = X -5;
LOOP: 30: L1: Q = 5 + Y;
```

The function of numeric labels is to specify an _origin_ for code generation. The statement "30: Y = X - 5;" for example, specifies that the object code for this statement is to begin at location 30 in memory. The identifier form of a statement label has no effect on the origin of the code, but does provide a destination for GO TO statements.

14.2. GO TO Statements. PL/M allows three distinct forms of an unconditional transfer. The first is

GO TO label;

In this case, the label is an identifier which appears as a label in a labelled statement. Program control transfers directly to the statement with this label.

The second form of a GO TO is

GO TO constant;

The constant is any valid PL/M single or double byte number. Program control transfers to the absolute location in memory given by this number.

The last form is

GO TO variable;

where the variable contains a computed memory address. Control transfers directly to this computed absolute address.

The following program illustrates the use of labelled statements and GO TO's.

```
DECLARE X ADDRESS;
. . .
10:  GO TO KEYIN;
. . .
LOOP:  Q = R + 3;
. . .
IF Q > Z GO TO LOOP;
. . .
GO TO EXIT;
/* COMPUTE AN ADDRESS AND BRANCH */
X = .MEMORY + 13;
GO TO X;
. . .
GO TO 30;
. . .
EXIT: HALT;
EOF
```

14.3. Scope of Labels. It should be noted that the identifier form of a label has an implied scope, similar to variables and procedures. This implied scope can be made explicit through the PL/M label declaration. The form of the label declaration is

DECLARE identifier LABEL;

or

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

The label declaration informs the compiler that a label or set of labels will occur at the same block level as the declaration. The label declaration is only necessary, however, when the implied declaration does not correspond to the programmer's intention. In particular, any occurrence of an undeclared label in either a GO TO statement, or as a statement label results in an immediate automatic declaration of the label. This implied declaration is most easily seen by example. The programs to the left below contain undeclared labels. The implied declarations resulting from these labels are shown in the corresponding programs to the right.

PROGRAM 1

```
. . .                    | DECLARE LOOP LABEL;
LOOP:  X = X + 1;        | LOOP: X = X + 1;
GO TO LOOP;              | GO TO LOOP;
EOF                      | EOF
```

PROGRAM 2

```
. . .                    | DECLARE LOOP LABEL;
LOOP: X=X+1;             | LOOP: X=X+1;
    DO;                  |     DO;
. . .                    |     DECLARE Q1 LABEL;
    GO TO Q1;            |     GO TO Q1;
    Q1: Y=Y+1;           |     Q1: Y = Y+1;
    GO TO LOOP;          |     GO TO LOOP;
    END;                 |     END;
. . .                    | DECLARE EXIT LABEL;
GO TO EXIT;              | GO TO EXIT;
EXIT: HALT;              | EXIT: HALT;
EOF                      | EOF
```

PROGRAM 3

```
X=X+1;                   | X=X+1;
    DO;                  |     DO;
. . .                    |     DECLARE L1 LABEL;
    GO TO L1;            |     GO TO L1;
    L1: Y=Y+1;           |     L1: Y=Y+1;
    END;                 |     END;
. . .                    | DECLARE L1 LABEL;
L1: Q=Q+3;               | L1: Q=Q+3;
GO TO L1;                | GO TO L1;
EOF                      | EOF
```

The only instance which requires explicit declaration of a label is when a GO TO statement in an inner nested block references a label in an outer block, and the label follows the GO TO statement. Consider the following program, for example.

```
/* BLOCK 1 */
X = X + 1;
. . .
    DO; /* BLOCK 2 */
    . . .
    GO TO EXIT;
    . . .
    END /* OF BLOCK 2 */;
. . .
EXIT: HALT;
EOF
```

The implied label declaration created by the PL/M compiler for the label EXIT results in the program

```
X = X + 1;
. . .
    DO;
    DECLARE EXIT LABEL;
    . . .
    GO TO EXIT;
    . . .
    END;
. . .
DECLARE EXIT LABEL;
EXIT: HALT;
EOF
```

Note that the resulting program is in error since the implied declaration of EXIT in block 2 indicates that the scope of EXIT is only block 2, conflicting with its occurrence in block 1. Thus, the label declaration can be used to remedy the situation. The programmer overrides the implied declaration with

```
DECLARE EXIT LABEL;
```

```
X = X + 1;
. . .
    DO;
    . . .
    GO TO EXIT;
    . . .
    END;
. . .
EXIT: HALT;
EOF
```

As a final note, the PL/M programmer is encouraged to use the IF-THEN-ELSE and DO-group constructs in the place of labelled statements and GO TO's whenever possible. The effect in most cases is better object code and improved readability of the source program.

15. Compile-Time Macro Processing.

PL/M allows declaration and expansion of simple macros at compile time. The LITERALLY declaration in PL/M allows the programmer to define an identifier to represent a sequence of arbitrary characters. The PL/M compiler automatically substitutes the defining string at each occurrence of the defined identifier. The form of the LITERALLY declaration is

```
DECLARE identifier LITERALLY string;
```

where the identifier is any valid PL/M name which does not conflict with previous declarations, and the string is an arbitrary PL/M string, not exceeding 255 characters in length.

The following program illustrates the use of the PL/M macro facility

```
DECLARE TRUE LITERALLY '1',
    FALSE LITERALLY '0';
```

```
DECLARE DCL LITERALLY 'DECLARE',
    LIT LITERALLY 'LITERALLY';
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 declarations on lines 1 and 2 allow the programmer to use the symbols TRUE and FALSE instead of 0 and 1, which often makes the program more readable. The declarations for DCL and LIT define abbreviations for DECLARE and LITERALLY, respectively.

The DC FOREVER statement on line 8 first expands to DO WHILE TRUE. The macro expansion of TRUE then results in a loop headed by DO WHILE 1 (which executes indefinitely, until the HALT statement is executed).

The LITERALLY declaration is also useful for declaring fixed parameters for the particular compilation, but which may change from one compilation to the next. Consider the program below, for example:

```
DECLARE ASIZE LITERALLY '300',
    PBASE LITERALLY '4000',
    SUPERVISOR LITERALLY '200';
DECLARE (A(ASIZE),I) ADDRESS;
. . .
PBASE:  A(ASIZE-10) = 50;
    . . .
    GO TO SUPERVISOR;
. . .
EOF
```

In this case, ASIZE defines the size of the vector A. The value of ASIZE can be altered in the LITERALLY declaration without affecting the remainder of the program. Similarly, the value of PBASE defines the starting location of the program since it expands to a numeric label. The expansion of the PBASE macro results in the statement

```
4000:  A(ASIZE-1) = 50;
```

In the case of the SUPERVISOR macro, the statement "GO TO SUPERVISOR" is replaced by "GO TO 200" resulting in a transfer to absolute address 200 in memory.

16. Predeclared Variables and Procedures.

The LENGTH and LAST forms described previously are called built in procedures. A number of additional predeclared variables and procedures are described in this section, which are intended to ease the programming task.

It should be noted that these variables and procedures are assumed to be declared at an outer encompassing block level which is invisible to the programmer. Thus, declarations of variables and procedures with identical names within the program override the predeclared names.

16.1. Condition Code Variables. There are four variable names in PL/M which can be used to test the condition codes in the MCS-8 CPU. These names are

CARRY ZERO SIGN PARITY

Any occurrence of one of these variables generates an immediate test of the corresponding condition code flip-flop for a true condition (value is 1). The use of these variables is somewhat implementation-dependent, and is described more completely in the section on PL/M system notes. In any case, these variables cannot be used as the destination of an assignment.

16.2. The MEMORY Vector. It is often useful to address the area of memory following the last variable allocated in a particular program. PL/M provides this facility by automatically inserting the declaration

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

as the last declaration in every program.

As an example, consider the following program. This program assumes it will execute on a machine with 10 pages (2560 bytes) of memory. The program initializes all remaining space after the program variable storage to 1's.

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

16.3. The TIME Procedure. A built-in procedure, called TIME, is provided in PL/M for waiting a fixed amount of time at a particular point in the program. The form of the call is

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

where the expression evaluates to a byte quantity n between 1 and 255. The wait time is measured in increments of 100 usec; hence, the total time-out for a value n is

n(100 usec).

Thus, the call to TIME shown below results in a 4500 usec (4.5 msec) time-out

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

Since the maximum time-out is 255*100 usec. = 25500 usec = 25.5 msec, longer wait periods are affected by enclosing the call in a loop. The following loop, for example, takes 1 second to execute

```
DO I = 1 TO 40;
```

```
CALL TIME(250);

END;
```

16.4. Type Transfer Procedures. two built-in procedures are provided in PL/M to convert ADDRESS values to BYTE values. The procedure calls take the forms

LOW(expression) and HIGH(expression)

The LOW procedure returns the low-order byte of a double byte value, while the HIGH procedure returns the high-order byte. Either call can be used wherever a byte expression is valid in PL/M.

The built-in procedure DOUBLE converts a BYTE value to an ADDRESS value. The procedure call takes the form

DOUBLE(expression)

16.5 Bit Manipulation Procedures. Six procedures are provided in PL/M for shifting and rotating expressions. These procedure calls take the forms

```
SHL(expression1,expression2);

SHR(expression1,expression2);

SCL(expression1,expression2);

SCR(expression1,expression2);

ROL(expression3,expression2);

ROR(expression3,expression2);
```

In these cases, expression1 can be either byte or double byte, but expression2 and expression3 must be single byte values.

The SHL and SHR procedures shift expression1 to the left or right by an amount given by expression2, respectively. The precision of the result is the same as that of expression1. Note that the value of expression2 must be greater than zero.

The value of SHL(1000$0011B,2), for example, is the byte value 00001100B. The call SHR(1$0000$1100B,1) results in the double byte value 0$1000$0110B.

The SCL and SCR procedures are identical to the SHL and SHR procedures with the exception that SCL and SCR shift in the previous value of the carry flag, where SHL and SHR shift in zeroes. For example, the statements

```
HIGH$ORDER = SHR(0101$0101B,1);

LOW$ORDER = SCR(0101$0101B,1);
```

assign the value 00101010B to HIGH$ORDER and the value 10101010B to LOW$ORDER.

The ROL and ROR procedures rotate the value of the byte expression3 to the right or left by an amount given by expression2, respectively. Again, expression2 must be greater than zero. Both procedures always return a byte value. The value of ROL(1011$0000,2) is 1100$0010B, and the value of ROR(1111$0000B,8) is 1111$0000B.

The SHL, SHR, SCL, SCR, ROL, and ROR calls can appear anywhere a PL/M expression is allowed.

16.6. I/O processing. The built-in procedure INPUT and built-in variable OUTPUT were introduced earlier. In general, the input call takes the form

```
INPUT(constant)
```

where the constant is in the range 0 to 7. The effect of the call is to read the input port designated by the constant. The result of the call is the byte value latched into the port. The call to INPUT can appear as a part of any valid PL/M expression.

The pseudo-variable OUTPUT can only be used as the destination of an assignment. The form is

```
OUTPUT(constant) = expression;
```

where the constant is in the range 0 to 23. The value of the expression is latched into the output port designated by the constant.

This section completes the tutorial introduction to PL/M. The section which follows provides more detailed discussion of the individual statements and constructs of PL/M.

III. A FORMAL APPROACH TO PL/M.

(Section III is currently incomplete. The BNF description of PL/M is included, however, for reference purposes.)

```
 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                        |  ;
15                        |  <LABEL DEFINITION> <BASIC STATEMENT>

16   <IF STATEMENT>  ::=  <IF CLAUSE> <STATEMENT>
17                     |  <IF CLAUSE> <TRUE PART> <STATEMENT>
18                     |  <LABEL DEFINITION> <IF STATEMENT>

19   <IF CLAUSE>  ::=  IF <EXPRESSION> THEN

20   <TRUE PART>  ::=  <BASIC STATEMENT> ELSE

21   <GROUP>  ::=  <GROUP HEAD> <ENDING>

22   <GROUP HEAD>  ::=  DO ;
23                   |  DO <STEP DEFINITION> ;
24                   |  DO <WHILE CLAUSE> ;
25                   |  DO <CASE SELECTOR> ;
26                   |  <GROUP HEAD> <STATEMENT>

27   <STEP DEFINITION>  ::=  <VARIABLE> <REPLACE> <EXPRESSION> <ITERATION CONTROL>

28   <ITERATION CONTROL>  ::=  <TO> <EXPRESSION>
29                          |  <TO> <EXPRESSION> <BY> <EXPRESSION>

30   <WHILE CLAUSE>  ::=  <WHILE> <EXPRESSION>

31   <CASE SELECTOR>  ::=  CASE <EXPRESSION>

32   <PROCEDURE DEFINITION>  ::=  <PROCEDURE HEAD> <STATEMENT LIST> <ENDING>

33   <PROCEDURE HEAD>  ::=  <PROCEDURE NAME> ;
34                       |  <PROCEDURE NAME> <TYPE> ;
35                       |  <PROCEDURE NAME> <PARAMETER LIST> ;
36                       |  <PROCEDURE NAME> <PARAMETER LIST> <TYPE> ;

37   <PROCEDURE NAME>  ::=  <LABEL DEFINITION> PROCEDURE

38   <PARAMETER LIST>  ::=  <PARAMETER HEAD> <IDENTIFIER> )

39   <PARAMETER HEAD>  ::=  (
40                       |  <PARAMETER HEAD> <IDENTIFIER> ,

41   <ENDING>  ::=  END
42               |  END <IDENTIFIER>
43               |  <LABEL DEFINITION> <ENDING>

44   <LABEL DEFINITION>  ::=  <IDENTIFIER> :
45                         |  <NUMBER> :

46   <RETURN STATEMENT>  ::=  RETURN
47                         |  RETURN <EXPRESSION>

48   <CALL STATEMENT>  ::=  CALL <VARIABLE>

49   <GO TO STATEMENT>  ::=  <GO TO> <IDENTIFIER>
50                        |  <GO TO> <NUMBER>

51   <GO TO>  ::=  GO TO
52              |  GOTO

53   <DECLARATION STATEMENT>  ::=  DECLARE <DECLARATION ELEMENT>
54                              |  <DECLARATION STATEMENT> , <DECLARATION ELEMENT>

55   <DECLARATION ELEMENT>  ::=  <TYPE DECLARATION>
56                            |  <IDENTIFIER> LITERALLY <STRING>
57                            |  <IDENTIFIER> <DATA LIST>

58   <DATA LIST>  ::=  <DATA HEAD> <CONSTANT> )

59   <DATA HEAD>  ::=  DATA (
60                  |  <DATA HEAD> <CONSTANT> ,

61   <TYPE DECLARATION>  ::=  <IDENTIFIER SPECIFICATION> <TYPE>
62                         |  <BOUND HEAD> <NUMBER> ) <TYPE>
63                         |  <TYPE DECLARATION> <INITIAL LIST>
```

```
 64   <TYPE>  ::=  BYTE
                |  ADDRESS
 66             |  LABEL

 67   <BOUND HEAD>  ::=  <IDENTIFIER SPECIFICATION> (

 68   <IDENTIFIER SPECIFICATION>  ::=  <VARIABLE NAME>
 69                                |  <IDENTIFIER LIST> <VARIABLE NAME> )

 70   <IDENTIFIER LIST>  ::=  (
 71                       |  <IDENTIFIER LIST> <VARIABLE NAME> ,

 72   <VARIABLE NAME>  ::=  <IDENTIFIER>
 73                     |  <BASED VARIABLE> <IDENTIFIER>

 74   <BASED VARIABLE>  ::=  <IDENTIFIER> BASED

 75   <INITIAL LIST>  ::=  <INITIAL HEAD> <CONSTANT> )

 76   <INITIAL HEAD>  ::=  INITIAL (
 77                    |  <INITIAL HEAD> <CONSTANT> ,

 78   <ASSIGNMENT>  ::=  <VARIABLE> <REPLACE> <EXPRESSION>
 79                  |  <LEFT PART> <ASSIGNMENT>

 80   <REPLACE>  ::=  =

 81   <LEFT PART>  ::=  <VARIABLE> ,

 82   <EXPRESSION>  ::=  <LOGICAL EXPRESSION>
 83                  |  <VARIABLE> : = <LOGICAL EXPRESSION>

 84   <LOGICAL EXPRESSION>  ::=  <LOGICAL FACTOR>
 85                          |  <LOGICAL EXPRESSION> OR <LOGICAL FACTOR>
 86                          |  <LOGICAL EXPRESSION> XOR <LOGICAL FACTOR>

 87   <LOGICAL FACTOR>  ::=  <LOGICAL SECONDARY>
 88                      |  <LOGICAL FACTOR> AND <LOGICAL SECONDARY>

 89   <LOGICAL SECONDARY>  ::=  <LOGICAL PRIMARY>
 90                         |  NOT <LOGICAL PRIMARY>

 91   <LOGICAL PRIMARY>  ::=  <ARITHMETIC EXPRESSION>
 92                       |  <ARITHMETIC EXPRESSION> <RELATION> <ARITHMETIC EXPRESSION>

 93   <RELATION>  ::=  =
 94               |  <
 95               |  >
 96               |  < >
 97               |  < =
 98               |  > =

 99   <ARITHMETIC EXPRESSION>  ::=  <TERM>
100                             |  <ARITHMETIC EXPRESSION> + <TERM>
101                             |  <ARITHMETIC EXPRESSION> - <TERM>
102                             |  <ARITHMETIC EXPRESSION> PLUS <TERM>
103                             |  <ARITHMETIC EXPRESSION> MINUS <TERM>
104                             |  - <TERM>

105   <TERM>  ::=  <PRIMARY>
106            |  <TERM> * <PRIMARY>
107            |  <TERM> / <PRIMARY>
108            |  <TERM> MOD <PRIMARY>

109   <PRIMARY>  ::=  <CONSTANT>
110               |  . <CONSTANT>
111               |  <CONSTANT HEAD> <CONSTANT> )
112               |  <VARIABLE>
113               |  . <VARIABLE>
114               |  ( <EXPRESSION> )

115   <CONSTANT HEAD>  ::=  . (
116                     |  <CONSTANT HEAD> <CONSTANT> ,

117   <VARIABLE>  ::=  <IDENTIFIER>
118                |  <SUBSCRIPT HEAD> <EXPRESSION> )

119   <SUBSCRIPT HEAD>  ::=  <IDENTIFIER> (
120                      |  <SUBSCRIPT HEAD> <EXPRESSION> ,

121   <CONSTANT>  ::=  <STRING>
122                |  <NUMBER>

123   <TO>  ::=  TO

124   <BY>  ::=  BY

125   <WHILE>  ::=  WHILE
```

IV. COMPIIING AND DEBUGGING PL/M PROGRAMS.

This section discusses procedures for compiling and debugging PL/M programs. A complete compilation of a PL/M program is performed in two distinct parts: the first phase, referred to as PLM1, scans the source program, and produces an intermediate form. The second phase, called PLM2, accepts this intermediate form and produces the machine code for the MCS-8 CPU. All errors in program syntax are detected in PLM1.

The debugging process begins following successful compilation of a PL/M program. This debugging phase consists of an execution of INTERP/8 which accepts the machine code produced by PLM2 and simulates the actions of the MCS-8 CPU. INTERP/8 has a number of facilities which allow monitoring of CPU action, allowing symbolic and absolute reference to machine code and variable storage locations (see Appendix III of the INTEL publication "MCS-8 Micro Computer Set 8008 Users Manual") These three phases are described in detail in the sections which follow.

1. PLM1 Operating Procedures.

The first pass of the PL/M compiler scans the source program, and detects improperly formed declarations and statements. A listing of the source program can be obtained during this pass. Errors are listed by line number whether the source listing is produced or not. An error message produced by PLM1 takes the form:

(nnnnn) ERROR m NEAR s

The number nnnnn corresponds to the line where the error occurred, s is a symbol on the line near the error, and m corresponds to the particular error message as given in

Figure IV-1.

Before discussing the files referenced by PLM1, it is necessary to present the file naming scheme used throughout the three programs PLM1, PLM2, and INTERP/8. These three programs are written in ANSI standard FORTRAN with the intention of being as independent from the host computer as possible. Thus, only a few assumptions can be made about the physical input and output devices or FORTRAN logical unit numbers and corresponding file names used in any particular implementation. Instead, these three programs use an internal file numbering scheme which is consistent between the three programs, but which may differ in terms of FORTRAN logical units from installation to installation. The machine-independent approach here is to give the file numbering in terms of devices types, and allow any particular implementation to assign the most convenient FORTRAN units.

The file numbers used throughout PLM1, PLM2, and INTERP/8, along with the corresponding device types, are shown in Figure IV-2. Two examples of FORTRAN unit number assignments for the PDP-10 and IBM System/360 computers are shown in Figure IV-3.

A number of compiler control switches are used during the execution of PLM1 to control I/O based upon this file numbering scheme. Additional switches are provided to control other compile-time functions during this pass, as given below. Compiler control switches come in two forms: compiler toggles, and compiler parameters. Compiler toggles can take on only the values 0 and 1 (generally specifying an "on" or "off" condition), while compiler parameters can be any non-negative value.

A compiler switch is specified to PLM1 by typing a line

ERROR NUMBER	MESSAGE
1	THE SYMBOLS PRINTED BELOW HAVE BEEN USED IN THE CURRENT BLOCK BUT DO NOT APPEAR IN A DECLARE STATEMENT, OR LABEL APPEARS IN A GO TO STATEMENT BUT DOES NOT APPEAR IN THE BLOCK.
2	PASS-1 COMPILER SYMBOL TABLE OVERFLOW. TOO MANY SYMBOLS IN THE SOURCE PROGRAM. EITHER REDUCE THE NUMBER OF VARIABLES IN THE PROGRAM, OR RE-COMPILE PASS-1 WITH A LARGER SYMBOL TABLE.
3	INVALID PL/M STATEMENT. THE PAIR OF SYMBOLS PRINTED BELOW CANNOT APPEAR TOGETHER IN A VALID PL/M STATEMENT (THIS ERROR MAY HAVE BEEN CAUSED BE A PREVIOUS ERROR IN THE PROGRAM).
4	INVALID PL/M STATEMENT. THE STATEMENT IS IMPROPERLY FORMED-- THE PARSE TO THIS POINT FOLLOWS (THIS MAY HAVE OCCURRED BE- CAUSE OF A PREVIOUS PROGRAM ERROR).
5	PASS-1 PARSE STACK OVERFLOW. THE PROGRAM STATEMENTS ARE RECURSIVELY NESTED TOO DEEPLY. EITHER SIMPLIFY THE PROGRAM STRUCTURE, OR RE-COMPILE PASS-1 WITH A LARGER PARSE STACK.
6	NUMBER CONVERSION ERROR. THE NUMBER EITHER EXCEEDS 65535 OR CONTAINS DIGITS WHICH CONFLICT WITH THE RADIX INDICATOR.
7	PASS-1 TABLE OVERFLOW. PROBABLE CAUSE IS A CONSTANT STRING WHICH IS TOO LONG. IF SO, THE STRING SHOULD BE WRITTEN AS A SEQUENCE OF SHORTER STRINGS, SEPARATED BY COMMAS. OTHERWISE, RE-COMPILE PASS-1 WITH A LARGER VARC TABLE.
8	MACRO TABLE OVERFLOW. TOO MANY LITERALLY DECLARATIONS. EITHER REDUCE THE NUMBER OF LITERALLY DECLARATIONS, OR RE- COMPILE PASS-1 WITH A LARGER 'MACROS' TABLE.
9	INVALID CONSTANT IN INITIAL, DATA, OR IN-LINE CONSTANT. PRECISION OF CONSTANT EXCEEDS TWO BYTES (MAY BE INTERNAL PASS-1 COMPILER ERROR).
10	INVALID PROGRAM. PROGRAM SYNTAX INCORRECT FOR TERMINATION OF PROGRAM. MAY BE DUE TO PREVIOUS ERRORS WHICH OCCURRED WITHIN THE PROGRAM.
11	INVALID PLACEMENT OF A PROCEDURE DECLARATION WITHIN THE PL/M PROGRAM. PROCEDURES MAY ONLY BE DECLARED IN THE OUTER BLOCK (MAIN PART OF THE PROGRAM) OR WITHIN DO-END GROUPS (NOT ITERATIVE DO'S, DO-WHILE'S, OR DO-CASE'S).
12	IMPROPER USE OF IDENTIFIER FOLLOWING AN END STATEMENT. IDENTIFIERS CAN ONLY BE USED IN THIS WAY TO CLOSE A PROCEDURE DEFINITION.
13	IDENTIFIER FOLLOWING AN END STATEMENT DOES NOT MATCH THE NAME OF THE PROCEDURE WHICH IT CLOSES.
14	DUPLICATE FORMAL PARAMETER NAME IN A PROCEDURE HEADING.
15	IDENTIFIER FOLLOWING AN END STATEMENT CANNOT BE FOUND IN THE PROGRAM.
16	DUPLICATE LABEL DEFINITION AT THE SAME BLOCK LEVEL.
17	NUMERIC LABEL EXCEEDS CPU ADDRESSING SPACE.
18	INVALID CALL STATEMENT. THE NAME FOLLOWING THE CALL IS NOT A PROCEDURE.
19	INVALID DESTINATION IN A GO TO. THE VALUE MUST BE A LABEL OR SIMPLE VARIABLE.
20	MACRO TABLE OVERFLOW (SEE ERROR 8 ABOVE).
21	DUPLICATE VARIABLE OR LABEL DEFINITION.
22	VARIABLE WHICH APPEARS IN A DATA DECLARATION HAS BEEN PRE- VIOUSLY DECLARED IN THIS BLOCK

Figure IV-1. PLM1 error messages issued during the first pass.

23 PASS-1 SYMBOL TABLE OVERFLOW (SEE ERROR 2 ABOVE).

24 INVALID USE OF AN IDENTIFIER AS A VARIABLE NAME.

25 PASS-1 SYMBOL TABLE OVERFLOW (SEE ERROR 2 ABOVE).

26 IMPROPERLY FORMED BASED VARIABLE DECLARATION. THE FORM IS I BASED J, WHERE I IS AN IDENTIFIER NOT PREVIOUSLY DECLARED IN THIS BLOCK, AND J IS AN ADDRESS VARIABLE.

27 SYMBOL TABLE OVERFLOW IN PASS-1 (SEE ERROR 2 ABOVE).

28 INVALID ADDRESS REFERENCE. THE DOT OPERATOR MAY ONLY PRECEDE SIMPLE AND SUBSCRIPTED VARIABLES IN THIS CONTEXT.

29 UNDECLARED VARIABLE. THE VARIABLE MUST APPEAR IN A DECLARE STATEMENT BEFORE ITS USE.

30 SUBSCRIPTED VARIABLE OR PROCEDURE CALL REFERENCES AN UN-DECLARED IDENTIFIER. THE VARIABLE OR PROCEDURE MUST BE DECLARED BEFORE IT IS USED.

31 THE IDENTIFIER IS IMPROPERLY USED AS A PROCEDURE OR SUB-SCRIPTED VARIABLE.

32 TOO MANY SUBSCRIPTS IN A SUBSCRIPTED VARIABLE REFERENCE. PL/M ALLOWS ONLY ONE SUBSCRIPT.

33 ITERATIVE DO INDEX IS INVALID. IN THE FORM 'DO I = E1 TO E2' THE VARIABLE I MUST BE SIMPLE (UNSUBSCRIPTED).

34 ATTEMPT TO COMPLEMENT A $ CONTROL TOGGLE WHERE THE TOGGLE CURRENTLY HAS A VALUE OTHER THAN 0 OR 1. USE THE '= N' OPTION FOLLOWING THE TOGGLE TO AVOID THIS ERROR.

35 INPUT FILE NUMBER STACK OVERFLOW. RE-COMPILE PASS-1 WITH A LARGER INSTK TABLE.

36 TOO MANY BLOCK LEVELS IN THE PL/M PROGRAM. EITHER SIMPLIFY YOUR PROGRAM (30 BLOCK LEVELS ARE CURRENTLY ALLOWED) OR RE-COMPILE PASS-1 WITH A LARGER BLOCK TABLE.

37 THE NUMBER OF ACTUAL PARAMETERS IN THE CALLING SEQUENCE IS GREATER THAN THE NUMBER OF FORMAL PARAMETERS DECLARED FOR THIS PROCEDURE.

38 THE NUMBER OF ACTUAL PARAMETERS IN THE CALLING SEQUENCE IS LESS THAN THE NUMBER OF FORMAL PARAMETERS DECLARED FOR THIS PROCEDURE.

39 ATTEMPT TO ASSIGN A VALUE TO AN INTRINSIC OR PROCEDURE NAME

Figure IV-1 (Con't)

Input

Internal File Number	Input Device
1	Interactive Console
2	Card Reader
3	Paper Tape
4	Magnetic Tape A
5	Magnetic Tape B
6	Sequential Disk A
7	Sequential Disk B

Output

Internal File Number	Output Device
1	Interactive Console
2	Line Printer
3	Paper Tape
4	Magnetic Tape C
5	Magnetic Tape D
6	Sequential Disk C
7	Sequential Disk D

Figure IV-2. Symbolic Device Assignments for PLM1, PLM2, and INTERP/8.

PASS-1 FILE DEFINITIONS

PDP-1Ø

NUM	INPUT DEVICE	UNIT	NUM	OUTPUT DEVICE	UNIT
1	TTY	5	1	TTY	5
2	CDR	2	2	PTR	3
3	PAP	6	3	PAP	7
4	MAG	16	4	MAG	17
5	DEC	9	5	DEC	1Ø
6	DISK	2Ø	6	DISK	22
7	DISK	21	7	DISK	23

IBM S/36Ø (CP/CMS)

NUM	INPUT DEVICE	UNIT	NUM	OUTPUT DEVICE	UNIT
1	TTY 8Ø	5	1	TTY 12Ø	6
2	CDR 8Ø	1Ø	2	PTR 133	8
3	TAP 8Ø	11	3	PUN 8Ø	7
4	TAP 14Ø	9	4	TAP 133	12
5	DSK 8Ø-LØ	13	5	DSK 8Ø-LØ	13
6	DSK 8Ø	1	6	DSK 8Ø	3
7	DSK 8Ø	2	7	DSK 8Ø	4

PASS-2 FILE DEFINITIONS

PDP-1Ø

NUM	INPUT DEVICE	UNIT	NUM	OUTPUT DEVICE	UNIT
1	TTY	5	1	TTY	5
2	CDR	2	2	PTR	3
3	PAP	6	3	PAP	7
4	MAG	16	4	MAG	17
5	DEC	9	5	DEC	1Ø
6	DISK	22	6	DISK	2Ø
7	DISK	23	7	DISK	21

IBM S/36Ø (CP/CMS)

NUM	INPUT DEVICE	UNIT	NUM	OUTPUT DEVICE	UNIT
1	TTY 8Ø	5	1	TTY 12Ø	6
2	CDR 8Ø	1Ø	2	PTR 133	8
3	TAP 8Ø	11	3	PUN 8Ø	7
4	TAP 14Ø	9	4	TAP 133	12
5	DSK 8Ø-LØ	13	5	DSK 8Ø-LØ	13
6	DSK 8Ø	3	6	DSK 8Ø	1
7	DSK 8Ø	4	7	DSK 8Ø	2

ALL INPUT RECORDS ARE 8Ø CHARACTERS OR LESS. ALL OUTPUT RECORDS ARE 12Ø CHARACTERS OR LESS.
THE FORTRAN UNIT NUMBERS CAN BE CHANGED IN THE SUBROUTINES GNC AND WRITEL (THESE ARE THE ONLY OC-CURRENCES OF REFERENCES TO THESE UNITS).

Figure IV-3. PDP-10 and IBM System/360 real device assignment.

of input with a "$" in column 1, and a switch name starting in column 2 (only the first character of the switch name is significant, and the remaining characters may be omitted). In the case of compiler parameters (and, optionally compiler toggles), the switch name is followed by an equal sign (=) and an integer value. A compiler toggle with the equal sign and number omitted is complemented (a 0 becomes a 1, and a 1 changes to a 0). Compiler switches are not printed in the source listing.

The most commonly used compiler switches for PLM1 are listed in Figure IV-4, along with their default values. Note that compiler toggles are listed in Figure IV-4 without the "= n" option although it is understood that either "= 1" or "= 0" is acceptable. Compiler parameters are listed in the Figure with the "= n" part following the switch name. The value of n is assumed to be in the proper range. Finally, note that the default values shown here are those provided by INTEL in the distribution version of the system and assume a batch processing environment. Any particular implementation may have differing default values (e.g., values may assume a time-sharing mode of processing), and thus the local installation should be consulted.

The operation of the first pass can now be described. PLM1 begins by reading the input file number which is defaulted by the $INPUT switch. Normally, this switch defaults to the card reader if operating in batch mode, and to the terminal if operating in interactive mode. Subsequent switches in the primary file can be used to change these default values, if necessary (e.g., reset the left or right margin, or change to an alternate input file). The first pass normally creates a listing file on output file number 2, an intermediate symbol table on file 6, and an intermediate code file on file 7.

Switch Name	Use	Default
$ANALYZE	Controls the PL/M syntax analysis trace.	0
$BYPASS	Dump the parse stack for syntax errors.	0
$COUNT = n	Start line numbering at line n.	0
$DELETE = n	Delete all trailing characters in the output after posistion n.	120
$EOF	End-of-file on this unit.	0
$GENERATE	Interlist the intermediate language produced by Pass 1.	0
$INPUT = n	Switch to file n for subsequent input (see PL/M file numbering).	1
$LEFTMARGIN = n	Ignore all characters before column n in the input lines.	1
$MEMORY	Include a symbol table in the object tape produced by Pass 2 showing the memory address assignments for variables, labels, and procedures.	0
$OUTPUT = n	Write subsequent output lines to file n (see PL/M file numbering).	1
$PRINT	Print output lines.	1
$RIGHTMARGIN=n	Ignore all characters in the input lines beyond position n.	72
$SYMBOLS	Print a symbol table dump at the end of Pass 1.	0
$TERMINAL	Interactive processing mode.	0
$WIDTH = n	Set output line width to n characters.	72

NOTE: The input lines are a maximum of 80 characters, and the output lines cannot exceed 120 characters.

Figure IV-4. PLM1 "$" compiler switches.

It should be noted that in an interactive mode, PLM1 starts by reading the progammer's console. At this point, the programmer could type the program directly at the console into PLM1. It is usually the case, however, that the programmer first composes his program using the time-sharing system's text editor. When PLM1 reads the console for the first line of input, the programmer redirects the PLM1 input to the disk file containing the edited program using the $INPUT = n compiler switch, where n is one of the input file numbers correspinding externally to the edited program.

The output from PLM1 can be directed to the programmer's console, or to another device such as a disk file or line printer using the $OUTPUT compiler switch placed in the input stream. If the programmer selects the console as an output device, it is often useful to set $TERMINAL = 1 which automatically lists only the error messages at the terminal. The programmer then uses the line numbers, along with the time-sharing system editor to locate the errors and change the source program in preparation for recompilation. In this way, a source listing of the program need never be generated during the first pass. The program is listed as the compilation proceeds if the $TERMINAL toggle is zero.

A practical approach to development of large PL/M programs is to write the program in terms of a number of independent procedures. Each of these procedures can be compiled and debugged separately, and, after all procedures are checked-out, the entire program can be compiled.

As an example, consider the program shown in Figure IV-5. In this case, a procedure is shown, called INDEX, which performs a comparison of two character strings to determine if the second string occurs as a substring in the

```
$MEMORY = 1
  /* THE INDEX PROCEDURE SEARCHES THE STRING STARTING AT
  'A' FOR AN OCCURRENCE OF THE STRING STARTING AT 'B'.
  INDEX RETURNS A ZERO IF THE SECOND STRING IS NOT A SUB-
  STRING OF THE FIRST; OTHERWISE, THE POSITION OF THE
  SECOND STRING IS RETURNED.  THE CHARACTER POSITIONS ARE
  COUNTED STARTING FROM 1 AND ENDING AT 255.          */
DECLARE EOS LITERALLY '0FFH';
/* THE LABELS L0 ... L5 AND C1 ... C3 ARE PRESENT FOR DEBUGGING
 PURPOSES ONLY, AND CAN BE REMOVED WITHOUT AFFECTING THE PROGRAM
 EXECUTION */
  INDEX: PROCEDURE (A,B) BYTE;
L0:    DECLARE (A,B) ADDRESS,
       (SA BASED A, SB BASED B, J,K,L,M) BYTE;
       J = 0;
L1:        DO WHILE SA(J) <> EOS;
           K = 0;
L2:            DO WHILE (L:=SA(J+K)) = (M:=SB(K));
L3:            IF L = EOS THEN RETURN J+1;
               K = K + 1;
               END;
           J = J + 1;
L4:        IF M = EOS THEN RETURN J;
           END;
L5:    RETURN 0;
       END INDEX;

 /* TEST THE INDEX FUNCTION */
DECLARE Q DATA ('WALLAWALLAWASH',EOS),
     (I,J) BYTE;
     DO WHILE 1;
     C1: I = INDEX(.Q,.('WALLA',EOS));
     C2: I = INDEX(.('WALLA',EOS),.Q);
     C3: I = INDEX(.Q,.('WASH',EOS));
     END;
EOF
```

Figure IV-5. A card-image listing of the INDEX procedure.

first string, as described in the comment preceding the procedure declaration. The last part of the program (following the declaration of Q) is present only to test the INDEX procedure and will be removed when INDEX is imbedded within a larger program. Note that this test section includes three sample calls on INDEX which are repeated indefinitely. The labels L0 through L5 within INDEX are used only during the debugging phase, and have no effect upon program execution. In fact, these labels may be removed after the INDEX procedure is checked-out to avoid later confusion as to the purpose of the labels.

Figure IV-6 shows a sample execution of PLM1 using the above source program as input. The exact manner in which PLM1 is started on any particular computer is, of course, implementation dependent. A number of particular systems are considered, however, in Section IV-4. The particular example shown in Figure IV-6 resulted from execution of PLM1 on an IBM System/360 under the CP/CMS time-sharing system using a 2741 console. Thus, all lines shown in lower case in this example, and examples which follow, are typed by the programmer, while upper case lines are output from the program being executed. The PLM1 output shown in this figure indicates that the program is syntactically correct, the intermediate files have been written, and the second pass can be initiated.

2. PLM2 Operating Procedures.

As mentioned previously, PLM2 performs the second pass of the PL/M compilation by reading the intermediate files produced through execution of PLM1. PLM2 then generates machine code for the MCS-8 CPU.

Error messages produced by PLM2 are of the form

(nnnnn) ERROR m

```
PASS-1

$I=2 (could use $o=2 for printer listing, $t=1 for no listing)
2     /* THE INDEX PROCEDURE SEARCHES THE STRING STARTING AT
2     'A' FOR AN OCCURRENCE OF THE STRING STARTING AT 'B'.
2     INDEX RETURNS A ZERO IF THE SECOND STRING IS NOT A SUB-
2     STRING OF THE FIRST; OTHERWISE, THE POSITION OF THE
2     SECOND STRING IS RETURNED.  THE CHARACTER POSITIONS ARE
2     COUNTED STARTING FROM 1 AND ENDING AT 255.          */
2   DECLARE EOS LITERALLY '0FFH';
2   /* THE LABELS L0 ... L5 AND C1 ... C3 ARE PRESENT FOR DEBUGGI
NG
2    PURPOSES ONLY, AND CAN BE REMOVED WITHOUT AFFECTING THE PROG
RAM
2    EXECUTION */
2     INDEX: PROCEDURE (A,B) BYTE;
3   L0:   DECLARE (A,B) ADDRESS,
3         (SA BASED A, SB BASED B, J,K,L,M) BYTE;
3         J = 0;
3   L1:       DO WHILE SA(J) <> EOS;
3             K = 0;
4   L2:           DO WHILE (L:=SA(J+K)) = (M:=SB(K));
4   L3:           IF L = EOS THEN RETURN J+1;
5                 K = K + 1;
5                 END;
4             J = J + 1;
4   L4:       IF M = EOS THEN RETURN J;
4             END;
3   L5:   RETURN 0;
3         END INDEX;
2
2    /* TEST THE INDEX FUNCTION */
2   DECLARE Q DATA ('WALLAWALLAWASH',EOS),
2       (I,J) BYTE;
2       DO WHILE 1;
2       C1: I = INDEX(.Q,.('WALLA',EOS));
3       C2: I = INDEX(.('WALLA',EOS),.Q);
3       C3: I = INDEX(.Q,.('WASH',EOS));
3       END;
2   EOF
NO PROGRAM ERRORS
```

Figure IV-6. Listing produced by PLM1 for the INDEX procedure.

where nnnnn references the line in the source program where the error occurs, and m is an error message number, corresponding to those given in Figure IV-7.

Operation of the second pass is particularly simple. PLM2 begins by reading the card reader (batch mode) or console (time-sharing mode) and will accept any number of "$" switches as input. These switches set the second pass compiling parameters shown in Figure IV-8. PLM2 continues to read these switches until one blank line is encountered. At this point, PLM2 reads the intermediate files produced by PLM1 and generates the MCS-8 machine code.

As in the case of PLM1, the exact manner in which the PLM2 program is initiated is implementation dependent, and will be discussed for some particular systems in Section IV-4.

Figure IV-9 shows the execution of PLM2 using the intermediate files produced by PLM1 for the INDEX procedure given previously. Figure IV-10 lists the BNPF machine code file which results from this execution of PLM2. Note that the machine code file is headed by a symbol table (caused by the $MEMORY=1 entry during PLM1) which will be used by INTERP/8 during the debugging phase which follows.

3. Program Check-Out.

Program verification is accomplished through the use of the MCS-8 CPU software simulator, called INTERP/8. The various commands available in INTERP/8 are described fully in the MCS-8 Users Manual. The PL/M program being checked-out is first compiled using PLM1 and PLM2, as previously described. In order to quickly locate errors in the source program, it is helpful to include the $MEMORY=1 toggle in PLM1 so that a symbol table is produced for the

ERROR NUMBER	MESSAGE
101	REFERENCE TO STORAGE LOCATIONS OUTSIDE THE VIRTUAL MEMORY OF PASS-2. RE-COMPILE PASS-2 WITH LARGER 'MEMORY' ARRAY.
102	"
103	VIRTUAL MEMORY OVERFLOW. PROGRAM IS TOO LARGE TO COMPILE WITH PRESENT SIZE OF 'MEMORY.' EITHER SHORTEN PROGRAM OR RECOMPILE PASS-2 WITH A LARGER VIRTUAL MEMORY.
104	(SAME AS 103).
105	$TOGGLE USED IMPROPERLY IN PASS-2. ATTEMPT TO COMPLEMENT A TOGGLE WHICH HAS A VALUE OTHER THAN 0 OR 1.
106	REGISTER ALLOCATION TABLE UNDERFLOW. MAY BE DUE TO A PRE-
107	REGISTER ALLOCATION ERROR. NO REGISTERS AVAILABLE. MAY BE CAUSED BY A PREVIOUS ERROR, OR PASS-2 COMPILER ERROR.
108	PASS-2 SYMBOL TABLE OVERFLOW. REDUCE NUMBER OF SYMBOLS, OR RE-COMPILE PASS-2 WITH LARGER SYMBOL TABLE.
109	SYMBOL TABLE OVERFLOW (SEE ERROR 108).
110	MEMORY ALLOCATION ERROR. TOO MUCH STORAGE SPECIFIED IN THE SOURCE PROGRAM (16K MAX ON 8008). REDUCE SOURCE PROGRAM MEMORY REQUIREMENTS.
111	INLINE DATA FORMAT ERROR. MAY BE DUE TO IMPROPER RECORD SIZE IN SYMBOL TABLE FILE PASSED TO PASS-2.
112	(SAME AS ERROR 107).
113	REGISTER ALLOCATION STACK OVERFLOW. EITHER SIMPLIFY THE PROGRAM OR INCREASE THE SIZE OF THE ALLOCATION STACKS.
114	PASS-2 COMPILER ERROR IN 'LITADD' -- MAY BE DUE TO A PREVIOUS ERROR.
115	(SAME AS 114).
116	(SAME AS 114).
117	LINE WIDTH SET TOO NARROW FOR CODE DUMP (USE $WIDTH=N)
118	(SAME AS 107).
119	(SAME AS 110).
120	(SAME AS 110, BUT MAY BE A PASS-2 COMPILER ERROR).
121	(SAME AS 108).
122	PROGRAM REQUIRES TOO MUCH PROGRAM AND VARIABLE STORAGE. (PROGRAM AND VARIABLES EXCEED 16K).
123	INITIALIZED STORAGE OVERLAPS PREVIOUSLY INITIALIZED STORAGE.
124	INITIALIZATION TABLE FORMAT ERROR. (SEE ERROR 111).
125	INLINE DATA ERROR. MAY HAVE BEEN CAUSED BY PREVIOUS ERROR.
126	BUILT-IN FUNCTION IMPROPERLY CALLED.
127	INVALID INTERMEDIATE LANGUAGE FORMAT. (SEE ERROR 111).
128	(SAME AS ERROR 113).

Figure IV-7. PLM2 error messages issued during the second pass.

129 INVALID USE OF BUILT-IN FUNCTION IN AN ASSIGNMENT.

130 PASS-2 COMPILER ERROR. INVALID VARIABLE PRECISION (NOT SINGLE BYTE OR DOUBLE BYTE). MAY BE DUE TO PREVIOUS ERROR.

131 LABEL RESOLUTION ERROR IN PASS-2 (MAY BE COMPILER ERROR).

132 (SAME AS 108).

133 (SAME AS 113).

134 INVALID PROGRAM TRANSFER (ONLY COMPUTED JUMPS ARE ALLOWED WITH A 'GO TO').

135 (SAME AS 134).

136 ERROR IN BUILT-IN FUNCTION CALL.

137 (NOT USED)

138 (SAME AS 107).

139 ERROR IN CHANGING VARIABLE TO ADDRESS REFERENCE. MAY BE A PASS-2 COMPILER ERROR, OR MAY BE CAUSED BY PREVOUS ERROR.

140 (SAME AS 107).

141 INVALID ORIGIN. CODE HAS ALREADY BEEN GENERATED IN THE SPECIFIED LOCATIONS.

142 A SYMBOL TABLE DUMP HAS BEEN SPECIFIED (USING THE $MEMORY TOGGLE IN PASS-1), BUT NO FILE HAS BEEN SPECIFIED TO RECEIVE THE BNPF TAPE (USE THE $BNPF=N CONTROL).

143 INVALID FORMAT FOR THE SIMULATOR SYMBOL TABLE DUMP (SEE ERROR 111).

Figure IV-7. (Con't)

Switch Name	Use	Default
$ANALYZE = n	Print a trace of the register allocation stack if n=1. Include assigned registers if n = 2.	0
$BNPF = n	Do not write a BNPF tape if n=0. Otherwise, write a BNPF tape to file n (see PL/M file numbering).	0
$COUNT = n	(Same as Pass 1)	
$DELETE = n	(Same as Pass 1)	
$EOF	(Same as Pass 1)	
$FINISH	Print a decoded dump of the generated machine code at the finish of Pass 2.	0
$GENERATE = n	Print a cross reference of source line numbers verses machine code locations if n = 1. If n = 2, print a trace of the intermediate language as it is read, as well.	0
$HEADER = n	Start machine code generation at location n when producing a code dump or BNPF tape.	0
$INPUT = n	(same as Pass 1)	
$LEFTMARGIN=n	(same as Pass 1)	
$MAP	Print a memory map showing symbol numbers and address assignments at the end of Pass 2.	0
$OUTPUT = n	(same as Pass 1)	
$PRINT	(same as Pass 1)	
$RIGHTMARGIN=n	(same as Pass 1)	
$TERMINAL	(same as Pass 1, default value suppresses the listing of the intermediate files as they are read)	0
$VARIABLES = n	The first page of Random-access Memory (RAM) is page n (numbering 0, 1,...,63)	0
$WIDTH = n	(same as Pass 1)	

Figure IV-8. PLM2 "$" compiler switches.

PASS-2

```
$generate = 1 (cross reference line numbers and locations in code)
$bnpf = 6      (write bnpf tape to internal file number 6)

   12=0003H   13=000EH   15=0011H   16=001EH   17=0026H   18=0043H
   19=0067H   20=006DH   21=0071H   22=0077H   23=0084H   24=0087H
   25=0089H   26=008AH   29=009CH   32=00A5H   33=00BEH   34=00E1H
   35=00E6H
```

Figure IV-9. Sample output from PLM2 corresponding to the INDEX procedure.

```
 1 CARRY 00362
 2 ZERO 00363
 3 SIGN 00364
 4 PARITY 00365
 5 MEMORY 00400
19 INDEX 00003
20 A 00366
21 B 00370
23 L0 00016
26 J 00372
27 K 00373
28 L 00374
29 M 00375
31 L1 00021
35 L2 00054
38 L3 00132
41 L4 00170
43 L5 00207
44 Q 00215
46 I 00376
47 J 00377
50 C1 00234
52 C2 00265
53 C3 00316
$
```

```
**************************************************************
********
**************************************************************
********
  0 BNPNNNPNNF BPNNNPNPNF BNNNNNNNNF BNNPNPPPNF
    BNNNNNNNNF BNNPPNPPNF BPPPPNPPNF BPPPPPNNPF
  8 BNNPPNNNNF BPPPPPNPNF BNNPPNNNNF BPPPPPNPPF
    BNNPPNNNNF BPPPPPPNNF BNNPPNNNNF BNNPPPPPNF
 16 BNNNNNNNNF BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF
    BPPPPPNPNF BPPNNNPPPF BNNPPNPPNF BPPPPNPPNF
 24 BPNNNNPPPF BNNPPNNNNF BPPNNPNNNF BNNNNNPPNF
    BNNNNNNNNF BPNNNPPPPF BPPPPNNNPF BPPPNPNNNF
 32 BPPNNNPPPF BNNNPNPNNF BPPPPPPPPF BNPPNPNNNF
    BPNNNNPPPF BNNNNNNNNF BNNPNPPPNF BNNNNNNNNF
 40 BNNPPNPPNF BPPPPPNPPF BNNPPPPPNF BNNNNNNNNF
    BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF BPPPPPNPNF
 48 BPPNNNPPPF BNNPPNNNNF BPNNNNPPPF BNNPPNPPNF
    BPPPPNPPNF BPNNNNPPPF BNNPPNNNNF BPPNNPNNNF
 56 BNNNNNPPNF BNNNNNNNNF BPNNNPPPPF BPPPPNNNPF
    BPPPNPNNNF BPPNNNPPPF BNNPNPPPNF BNNNNNNNNF
 64 BNNPPNPPNF BPPPPPPNNF BPPPPPNNNF BNNPPNNNPF
    BPPNNPNNNF BPPNNNPPPF BNNPPNPPNF BPPPPPNNNF
 72 BPNNNNPPPF BNNPPNNNNF BPPNPNNNNF BNNNNNPPNF
    BNNNNNNNNF BPNNNPPPPF BPPPPNNPNF BPPPNPNNNF
 80 BPPNNNPPPF BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF
    BPPPPPPNPF BPPPPPNNNF BPNNPNNNPF BNPNNPNNNF
 88 BNPPPNNNPF BNNNNNNNNF BNNPPNNNPF BPPNNNPPPF
    BNNNPNPNNF BPPPPPPPPF BNPNNPNNNF BNPPNNPPPF
 96 BNNNNNNNNF BNNPPNPPNF BPPPPPNPNF BPPNNPPPPF
    BNNNNPNNNF BPPNNNNNPF BNNNNNPPPF BNNPNPPPNF
104 BNNNNNNNNF BNNPPNPPNF BPPPPPNPPF BPPNNPPPPF
    BNNNNPNNNF BPPPPPNNPF BNPNNNPNNF BNNPNPPNNF
112 BNNNNNNNNF BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF
    BPPPPPNPNF BPPNNPPPPF BNNNNPNNNF BPPPPPNNPF
120 BNNPPNPPNF BPPPPPPNPF BPPNNNPPPF BNNNPNPNNF
    BPPPPPPPPF BNPNNPNNNF BPNNNPNNNF BNNNNNNNNF
128 BNNPPNPPNF BPPPPPNPNF BPPNNNPPPF BNNNNNPPPF
    BNPNNNPNNF BNNNPNNNPF BNNNNNNNNF BPNPNPNNNF
136 BNNNNNPPPF BNNNNNPPPF BNPNNNPNNF BPNNNPPNNF
    BNNNNNNNNF BNPNPNPPPF BNPNNNNNPF BNPNNPPNNF
144 BNPNNPPNNF BNPNNNNNPF BNPNPNPPPF BNPNNNNNPF
    BNPNNPPNNF BNPNNPPNNF BNPNNNNNPF BNPNPNPPPF
152 BNPNNNNNPF BNPNPNNPPF BNPNNPNNNF BPPPPPPPPF
    BNPNNNPNNF BPNPNNPNPF BNNNNNNNNF BNPNPNPPPF
160 BNPNNNNNPF BNPNNPPNNF BNPNNPPNNF BNPNNNNNPF
    BPPPPPPPPF BNNNNPPPNF BPNNNPPNPF BNNNPNPPNF
168 BNNNNNNNNF BNNNPPPPNF BPNNPPPPPF BNNPNNPPNF
    BNNNNNNNNF BNPNNNPPNF BNNNNNNPPF BNNNNNNNNF
176 BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF BPPPPPPPNF
    BPPPPPNNNF BNPNNNPNNF BPNPPPPPNF BNNNNNNNNF
184 BNPNPNPPPF BNPNNNNNPF BNPNNPPNNF BNPNNPPNNF
    BNPNNNNNPF BPPPPPPPPF BNNNNPPPNF BPNPPPNNNF
192 BNNNPNPPNF BNNNNNNNNF BNNNPPPPNF BPNNNPPNPF
    BNNPNNPPNF BNNNNNNNNF BNPNNNPPNF BNNNNNNPPF
200 BNNNNNNNNF BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF
    BPPPPPPPNF BPPPPPNNNF BNPNNNPNNF BPPNPNPPNF
208 BNNNNNNNNF BNPNPNPPPF BNPNNNNNPF BNPNPNNPPF
    BNPNNPNNNF BPPPPPPPPF BNNNNPPPNF BPNNNPPNPF
216 BNNNPNPPNF BNNNNNNNNF BNNNPPPPNF BPPNPNNNPF
    BNNPNNPPNF BNNNNNNNNF BNPNNNPPNF BNNNNNNPPF
224 BNNNNNNNNF BNNPNPPPNF BNNNNNNNNF BNNPPNPPNF
    BPPPPPPPNF BPPPPPNNNF BNPNNNPNNF BPNNPPPNNF
232 BNNNNNNNNF BPPPPPPPPF
```

Figure IV-10. Symbol table and BNPF tape produced by PLM2 for the INDEX procedure.

simulation. In addition, key statements in the source program should be labelled so that important points can be referenced symbolically during program check-out (see the use of the labels L0, ... L5, and C1, C2, and C3 in Figure IV-6, for example).

The generated symbol table and compiled object code is loaded into INTERP/8. Simulated program execution can then be monitored, the values of memory locations can be examined and altered, and program errors are readily detected. Program check-out is usually more effective if debugging is carried-out at the symbolic rather than absolute level. That is, INTERP/8 allows reference to memory through both symbolic locations (using the generated symbol table) and absolute addresses. As a result, it is generally much easier to follow the execution using the symbolic features of INTERP/8 than it is to trace the execution using absolute memory addresses. Thus, it is well worth the effort to become familiar with INTERP/8 symbolic debugging facilities.

A number of features have been added to the INTERP/8 program which enhances its use in debugging PL/M programs. These features augment the commands described in Appendix III of the MCS-8 Users Manual. These additions are given below.

First, note that symbolic names can be duplicated in a PL/M program. That is, a programmer could declare variables with the same name in block levels which do not conflict with one another. Consider the two procedures below, for example

```
P1: PROCEDURE(A) BYTE;
    DECLARE (A,B) ADDRESS;
    . . .
    END P1;
P2: PROCEDURE (Q) ADDRESS;
```

```
DECLARE (Q,A,B) BYTE;
. . .
END P2;
```

Recall that although there are variables in procedures P1 and P2 which have the same names (i.e., A and B), these variables are all given separate storage locations. In order to distinguish these variables, a construct of the form

S1 / S2 / ... Sn

is allowed as a symbolic reference in INTERP/8. The interpretation of this construct is as follows: INTERP/8 first searches for the symbol S1, then looks further to S2, and so-forth until Sn is found. This new construct can appear anywhere a "symbolic name" is allowed in the current INTERP/8 command structure. Note that in particular, the definition of a "range element" is extended to include this new form. Thus, the command

DISPLAY MEMORY A TO B+1.

is the same as

DISP MEM P1/A TO P1/B+1.

The second occurrences of A and B can only be located by first searching for the name P2. Thus, these two variables could be displayed using the command

DI MEM P2/A TO P2/B.

A second change to the INTERP/8 commands allows reference to a symbolic location when setting the value of the program stack (PC, PS 0, ... PS 7) or the value of the memory address register (HL). With this addition, the following are valid commands

```
SET PC = P2, PS 5 = P1.
SET HL = B.
SET HL = P2 / A + 1.
```

Two additional $ switches have been added to INTERP/8. The first is of the form

$MAXCYCLE = n

When this switch has a non-zero value, the CPU simulation is prevented from running more than n cycles before returning to the card reader or console for more input (n is initially zero). The toggle

$GENLABELS

was added to cause INTERP/8 to print the closest symbolic name to the current program counter whenever a break point is encountered. INTERP/8 prints

break AT n = label displacement

where "break" is one of the break point types: CYCLE, ALTER, or REFER, and n is an absolute location. The value of "label" is the closest symbolic name in the program, while the displacement is a positive or negative distance from the name to the location counter.

The last change to INTERP/8 allows imbedded dollar signs within numbers and identifiers, as in PL/M.

These features are demonstrated in the example described below. Figure IV-11 gives a sample run of INTERP/8 using the symbol table and machine code produced by PLM2 corresponding to the program containing the INDEX procedure given previously. Again, the initiation of INTERP/8 is system dependent and thus is not shown here. The symbol table is first loaded from file 6, followed by the machine code, also from file 6. Note that these file numbers must correspond to the BNPF tape file written by PLM2 (see the $BNPF switch in PLM2). The listing produced by PLM1 is used, along with the symbolic reference features of INTERP/8 to follow the program execution.

```
INTERP/8 VERS 1.0

/* first load the symbol table and bnpf tape from internal

   file number 6 (corresponding to the $bnpf=6 in pass2) */

load 6 6.
234 LOAD OK

/* then look at the symbol table */

display symbols.
0003620 00242 00F2H  CARRY
0003630 00243 00F3H  ZERO
0003640 00244 00F4H  SIGN
0003650 00245 00F5H  PARITY
0004000 00256 0100H  MEMORY
0000030 00003 0003H  INDEX
0003660 00246 00F6H  A
0003700 00248 00F8H  B
0000150 00014 000EH  L0
0003720 00250 00FAH  J
0003730 00251 00FBH  K
0003740 00252 00FCH  L
0003750 00253 00FDH  M
0000210 00017 0011H  L1
0000540 00044 002CH  L2
0001320 00090 005AH  L3
0001700 00120 0078H  L4
0002070 00135 0087H  L5
0002150 00141 008DH  Q
0003760 00254 00FEH  I
0003770 00255 00FFH  J
0002340 00156 009CH  C1
0002650 00181 00B5H  C2
0003160 00206 00CEH  C3
/* set break points at places in the index procedures

   labelled by 10, 11,    ... ,15 */

refer 10,11,12,13,14,15.
REFER OK

/* it will probably be useful to examine the program

   at the beginning and end of each call to index, so...*/

ref c1,c2,c3.
REFER OK

/* now run the program to the first reference variable */

go 1000.
GO OK
REFER AT 156=C1

/* we are at location 156 decimal, or equivalently, label c1 */

base hex.
HEX BASE OK

display symb *.
C1
/* look at cpu registers ...*/

di cpu.
CYZSP  A   B   C   D   E   H   L    HL   SP   PSC
*0000*00H*00H*00H*00H*00H*00H*00H*0000H*00H*009CH

di sym 9ch.
C1
```

Figure IV-11. Sample execution of INTERP/8.

```
di memory q to q+10.
008DH 57H 41H 4CH 4CH 41H 57H 41H 4CH 4CH 41H 57H

/* that must be the hex representation of WALLAWALLAW */

di sy q.
0002150 00141 008DH

/* now run the program to entry of the subroutine */

go 1000.
GO OK
REFER AT EH=L0

/* now at label L0, so examine the value of a */

di mem a.
00F6H 8DH

di mem a to a+1.
00F6H 8DH 00H

/* the first string is based    at a, so look at it..*/

di mem 8dh to 90h.
008DH 57H 41H 4CH 4CH

/* looks good, now examine b's value */

di mem b to b+1.
00F8H 9FH 00H

conv 9fh.
  10011111B  237Q  159  9FH

di mem 159 to 165.
009FH 57H 41H 4CH 4CH 41H FFH 0EH

/* looks good too, so run the index procedure down to

label 12 (also, to save typing go 1000, we can set maxcycle

to 1000 so the simulation will never run more than 1000 cycles

before stopping) */

$maxcycle = 1000
go.
REFER AT 11H=L1

go.
REFER AT 2CH=L2

/* examine the values of the local variables */

di mem index/j to index/m dec.
00FAH 000 000 000 000

di mem j to m.
00FAH 00H 00H 00H 00H

di sy 0fah.
J

/* run the procedure to label 13 */

go.
REFER AT 5AH=L3

/* both l and m should contain a 'w' */

di mem l to m.
00FCH 57H 57H
```

```
/* we should get a match on characters W A L L A

   and then return with the matching position 1 */

go. di m 1 to m.
REFER AT 2CH=L2
00FCH 57H 57H

go. di m 1 to m.
REFER AT 5AH=L3
00FCH 41H 41H

go . go. di m 1 to m.
REFER AT 2CH=L2
REFER AT 5AH=L3
00FCH 4CH 4CH

/* so far we have matched W A L */

go. go. di m 1 to m.
REFER AT 2CH=L2
REFER AT 5AH=L3
00FCH 4CH 4CH

/* turn off the break point at L2 since it is getting

   in the way */

noref 12.
REFER OK

go. di m 1 to m.
REFER AT 5AH=L3
00FCH 41H 41H

/* this time we should return */

go.
REFER AT 78H=L4

di mem m.
00FDH FFH

/* m = eos, so we should end up at label c2 */

ref 12. go.
REFER OK
REFER AT B5H=C2

/* the value of i should be 1 */

di m i.
00FEH 01H

di m i dec.
00FEH 001

/* now try the second call */

go.
REFER AT EH=L0

di mem a to b+1.
00F6H B8H 00H 8DH 00H

base dec.
DEC BASE OK

di mem a to b+1.
00246 184 000 141 000
```

```
di mem 184 to 190, mem 141 to 147.
00184 087 065 076 076 065 255 014
00141 087 065 076 076 065 087 065
/* strings are being sent properly, so we can continue.

   we should return a 0 this time since the larger string

   is not a substring of the smaller, so set reference

   breakpoint only at 15 */

noref 10,11,12,13,14. go.
REFER OK
REFER AT 135=L5

/* looks good, so let the subroutine return */

go.
REFER AT 206=C3

di mem i.
00254 000

noref 15. /* let the subroutine run, and see if
REFER OK

              it returns the proper value */

go.
CYCLE AT 50=L2+6

/* we just ran over 1000 cycles, so let it continue */

go 5000.
GO OK
REFER AT 156=C1

/* we are now back around the loop.  i will be    an 11

    if all is well */

di mem i.
00254 011

/* everything looks good, so we can now do a little

fooling around to show some of the other debugging

features -- first we will look at the operand break

point */

noref 0 to 256.
REFER OK

/* all reference break points are reset.  we will now

set a break point so that program execution stops when

the variables local to index are referenced. */

refer j to k.
REFER OK

go.
REFER AT 15=L0+1

/* we stopped at the first instruction in index...

look to see what instructions are there */
```

```
di mem * to *+10 code.
00015 LMI,00H LHI,00H LLI,FAH LAM LLI F6H ADM INL

di hl.
HL = 250

di sy 250.
J
/* thus program execution has stopped because there

was an attempt to store a zero into a variable set

in the refer command   run the program further...*/

go.
REFER AT 21=L1+4

di hl. di mem * code.
HL = 250
00021 LAM

di sy 250.
J
/* breakpoint now occurs because of the reference to

the variable j.        reset the break points, and

break only if the variable is being altered */

noref j to m. alter j to m.
REFER OK
ALTER OK

go.
ALTER AT 42=L2-2

di hl. di m * code.
HL = 251
00042 LMI

di sy 251.
K
/* now stopped because of attempt to alter variable k*/

go.
ALTER AT 66=L2+22

di hl.
HL = 252

di sy 252.
L
di me * to * + 10 code.
00066 LMA DCL LBA LAM LLI,F8H ADM INL LCA LAI,00H

di a.
A = 87

/* we are about to store     the accumulator into the

variable l.  look to see what is currently in l, and

then run one cycle, examine again. */

di mem l.
00252 255

go 1.
GO OK
CYCLE AT 67=L2+23
```

```
di mem l.
00252 087

/*       stored ok  now reset all operand breakpoints,

and go back and try the call over again */

noalter j to m.
ALTER OK

di sy cl.
000234Q 00156 009CH

di cpu.
CYZSP  A   B   C   D   E   H   L    HL   SP   PS0   PS1
*0101*087*141 000*159 000 000*252*00252*001*00176*00067

set pc = cl. di cpu.
SET OK
CYZSP  A   B   C   D   E   H   L    HL   SP   PS0   PS1
 0101 087 141 000 159 000 000 252 00252 001 00176*00156

/* we had better get out of the                 subroutine
call, so .... */

set sp = 0. set pc=cl. di cpu.
SET OK
SET OK
CYZSP  A   B   C   D   E   H   L    HL   SP   PS0
 0101 087 141 000 159 000 000 252 00252*000*00156

/* that looks a lot better.  now try the call again */

go.
CYCLE AT 62=L2+18

go.
CYCLE AT 64=L2+20

ref cl,c2,c3.
REFER OK

go.
REFER AT 181=C2

di mem i.
00254 001

/* same as before.  now       try some selective

program execution and tracing.  we will set the

values of some local variables and execute only

the code between 12 and 13 */

set cpu. di cpu.
SET OK
CYZSP  A   B   C   D   E   H   L    HL   SP   PS0
*0000*000*000 000*000 000 000*000*00000 000*00000

/* display the code between 12 and 13 */

di mem 12 to 13 cod.
00044 LHI,00H LLI,FAH LAM INL ADM LLI,F6H ADM INL LBA LAI,00H ACM LLB
00060 LHA LAM LHI,00H LLI,FCH LMA DCL LBA LAM LLI,F8H ADM INL LCA LAI
00076,00H ACM LLC LHA LAM LHI,00H LLI,FDH LMA SUB JFZ,71H,00H DCL

set mem j to m = 0. di mem j to m.
SET OK
00250 000 000 000 000
```

```
/* set the address pointers for a and b up in memory

   somewhere */

set mem a to b+1 = 0 1h 10h 1h. di m a to b+1.
SET OK
00246 000 001 016 001


/* now place data into these locations */

set mem 100h to 120h = 1 2 3 4 5 6 7.
SET OK

di mem 100h to 120h.
00256 001 002 003 004 005 006 007 001 002 003 004 005 006 007 001 002
00272 003 004 005 006 007 001 002 003 004 005 006 007 001 002 003 004

/* set j to 3 and k to 2 */

set mem j=3, mem k=2. di m j t k.
SET OK
00250 003 002

/* now trace this section of code */

trace 12-3 to 13+5.
TRACE OK

go 5.
GO OK
REFER AT 156=C1

/* move the program counter up to this section */

di pc, sp.
PC = 156
SP = 0

di b.
B = 0

di cpu.
CYZSP  A   B   C   D   E   H   L    HL   SP   PS0
 0000 000 000 000 000 000 000 000 00000 000*00156

set ps 0 = 12. /* same as set pc=12*/
SET OK

go 5.
GO OK
 0000 000 000 000 000 000 000 000 00000 000*00044
LHI 0
 0000 000 000 000 000 000 000 000 00000 000*00046
LLI 250
 0000 000 000 000 000 000 000*250*00250 000*00048
LAM
 0000*003 000 000 000 000 000 250 00250 000*00049
INL
*0010 003 000 000 000 000 000*251*00251 000*00050
ADM
CYCLE AT 51=L2+7

base hex.
HEX BASE OK

go 30
GO OK
```

```
*0001*05H 00H 00H 00H 00H 00H FBH 00FBH 00H*0033H
LLI F6H
 0001 05H 00H 00H 00H 00H 00H*F6H*00F6H 00H*0035H
ADM
 0001 05H 00H 00H 00H 00H 00H F6H 00F6H 00H*0036H
INL
*0010 05H 00H 00H 00H 00H 00H*F7H*00F7H 00H*0037H
LBA
CYZSP  A    B    C    D.   E    H    L    HL   SP   PSO
 0010 05H*05H 00H 00H 00H 00H F7H 00F7H 00H*0038H
LAI 0H
 0010*00H 05H 00H 00H 00H 00H F7H 00F7H 00H*003AH
ACM
*0000*01H 05H 00H 00H 00H 00H F7H 00F7H 00H*003BH
LLB
 0000 01H 05H 00H 00H 00H 00H*05H*0005H 00H*003CH
LHA
 0000 01H 05H 00H 00H 00H*01H 05H*0105H 00H*003DH
LAM
 0000*06H 05H 00H 00H 00H 01H 05H 0105H 00H*003EH
LHI 0H
 0000 06H 05H 00H 00H 00H*00H 05H*0005H 00H*0040H
LLI FCH
 0000 06H 05H 00H 00H 00H 00H*FCH*00FCH 00H*0042H
LMA
 0000 06H 05H 00H 00H 00H 00H FCH 00FCH 00H*0043H
DCL
*0010 06H 05H 00H 00H 00H 00H*FBH*00FBH 00H*0044H
LBA
CYZSP  A    B    C    D    E    H    L    HL   SP   PSO
 0010 06H*06H 00H 00H 00H 00H FBH 00FBH 00H*0045H
LAM
 0010*02H 06H 00H 00H 00H 00H FBH 00FBH 00H*0046H
LLI F8H
 0010 02H 06H 00H 00H 00H 00H*F8H*00F8H 00H*0048H
ADM
*0001*12H 06H 00H 00H 00H 00H F8H 00F8H 00H*0049H
INL
*0011 12H 06H 00H 00H 00H 00H*F9H*00F9H 00H*004AH
LCA
 0011 12H 06H*12H 00H 00H 00H F9H 00F9H 00H*004BH
LAI 0H
 0011*00H 06H 12H 00H 00H 00H F9H 00F9H 00H*004DH
ACM
*0000*01H 06H 12H 00H 00H 00H F9H 00F9H 00H*004EH
LLC
 0000 01H 06H 12H 00H 00H 00H*12H*0012H 00H*004FH
LHA
 0000 01H 06H 12H 00H 00H*01H 12H*0112H 00H*0050H
LAM
CYZSP  A    B    C    D    E    H    L    HL   SP   PSO
 0000*05H 06H 12H 00H 00H 01H 12H 0112H 00H*0051H
LHI 0H
 0000 05H 06H 12H 00H 00H*00H 12H*0012H 00H*0053H
LLI FDH
 0000 05H 06H 12H 00H 00H 00H*FDH*00FDH 00H*0055H
LMA
 0000 05H 06H 12H 00H 00H 00H FDH 00FDH 00H*0056H
SUB
*1011*FFH 06H 12H 00H 00H 00H FDH 00FDH 00H*0057H
JFZ 71H
CYCLE AT 73H=L4-5H

/* that should be enough of a check-out, so retire...*/

$eof
```

4. Implementation-Dependent Operating Procedures.

As mentioned previously, the exact manner in which PLM1 and PLM2 are initiated on any particular computer is implementation-dependent. Several sample implementations are given, however, in Figures IV-12 through IV-15. These figures provide a sample execution of both passes for the INTEL PDP-10, and the commercial time-sharing services Tymshare, Applied Logic, and General Electric, respectively. In each case, the FORTRAN unit names are specified for each of the major files accessed by PLM1 and PLM2.

When using the Tymshare version (Figure IV-13), for example, the programmer places the PL/M source program into a file named FOR20.DAT, which corresponds to the internal file number 6. This file is read when the $I=6 switch is encountered during the PLM1 execution. PLM1 produces the intermediate files FOR22.DAT and FOR23.DAT, along with an optional listing in FOR03.DAT (under control of the $C=2 and $T=0 or $T=1 switches).

PLM2 is then initiated and automatically reads the intermediate files produced by PLM1. Output can be directed to the disk file FOR07.DAT using the $O=3 switch during the PLM2 execution. The $B=7 switch in PLM2 produces a BNPF machine code tape during this second pass.

INTERP/8 can then be intiated for the debugging run, and the "LOAD 7 7." command can be used to read this tape.

```
         SAMPLE RUN ON INTEL PDP-10

.COPY FOR20.DAT=MYPROG.PLM
.SET SPOOL LPT
.R PLM1
$I=6

           PASS 1 OF COMPILER IS INVOKED HERE

.R PLM2
$B=7
(SPACE,CARRIAGE RETURN)

           PASS 2 OF COMPILER IS INVOKED HERE

.PRINT *.LPT
```

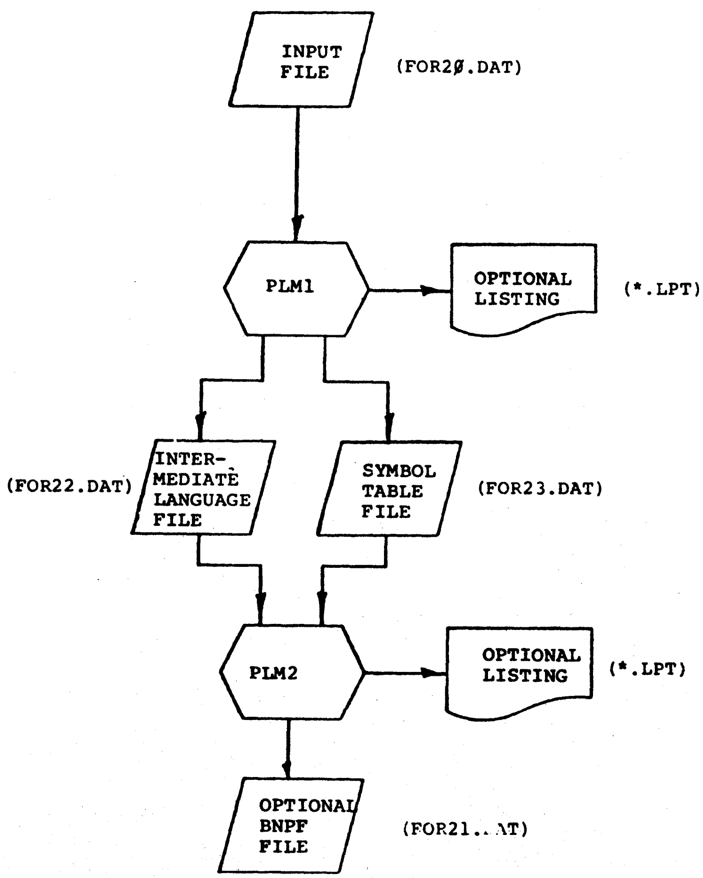

Figure IV-12. The INTEL implementation of PLM1 and PLM2.

```
SAMPLE RUN ON TYMSHARE PDP-10

.COPY MYPROG.PLM,FOR20.DAT
.RUN (UPL) PLM1
$O=2
$M=1
$S=1
$I=6

        PASS 1 OF COMPILER IS INVOKED HERE

.RUN (UPL) PLM2
$F=1
$G=1
$B=7
$M=1
$O=3
(SPACE,CARRIAGE RETURN)

        PASS 2 OF COMPILER IS INVOKED HERE
```

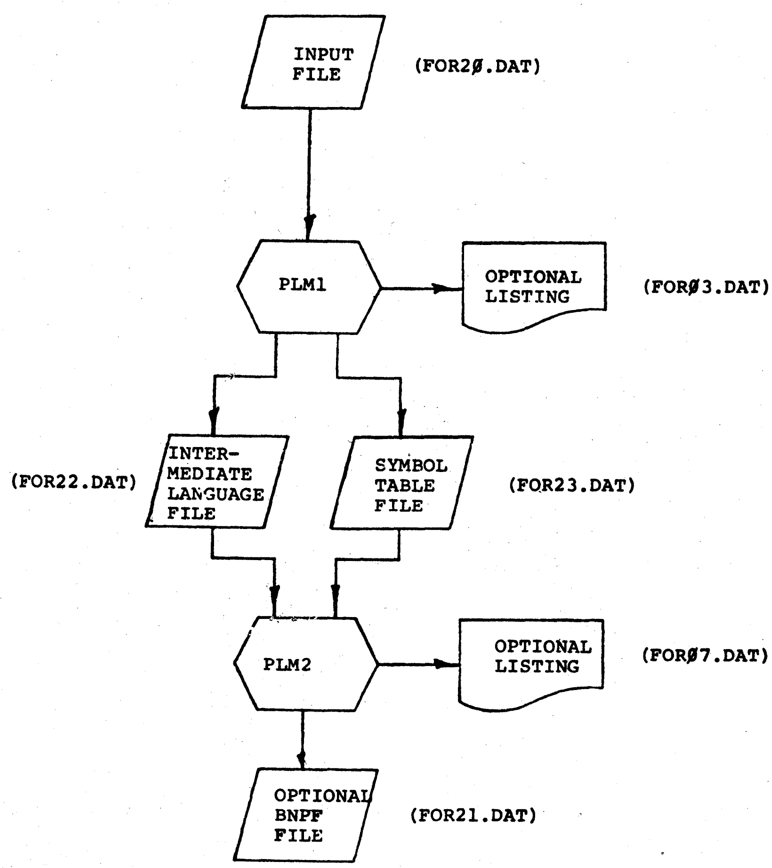

Figure IV-13. The Tymshare implementation of PLM1 and PLM2.

TYMSHARE FILE DEFINITIONS

PASS 1

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FORØ5.DAT	5
2	CDR	FORØ2.DAT	2
3	PTR	FORØ6.DAT	6
4	MTAØ	FOR16.DAT	16
5	DTA1	FORØ9.DAT	9
6	DSKØ	FOR2Ø.DAT	2Ø
7	DSK1	FOR21.DAT	21

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FORØ5.DAT	5
2	LPT	FORØ3.DAT	3
3	PTP	FORØ7.DAT	7
4	MTA1	FOR17.DAT	17
5	DTA2	FOR1Ø.DAT	1Ø
6	DSK2	FOR22.DAT	22
7	DSK3	FOR23.DAT	23

PASS 2

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FORØ5.DAT	5
2	CDR	FORØ2.DAT	2
3	PTR	FORØ6.DAT	6
4	MTAØ	FOR16.DAT	16
5	DTA1	FORØ9.DAT	9
6	DSK2	FOR22.DAT	22
7	DSK3	FOR23.DAT	23

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FORØ5.DAT	5
2	LPT	FORØ3.DAT	3
3	PTP	FORØ7.DAT	7
4	MTA1	FOR17.DAT	17
5	DTA2	FOR1Ø.DAT	1Ø
6	DSKØ	FOR2Ø.DAT	2Ø
7	DSK1	FOR21.DAT	21

```
     SAMPLE RUN ON AL/COM PDP-10

.COPY FILE10.DAT=MYPROG.PLM
.APPLY PLM1
$O=2
$M=1
$S=1
$I=6
          PASS 1 OF COMPILER IS INVOKED HERE

.APPLY PLM2
$F=1
$G=1
$B=7
$M=1
$O=3
(SPACE,CARRIAGE RETURN)
          PASS 2 OF COMPILER IS INVOKED HERE
```

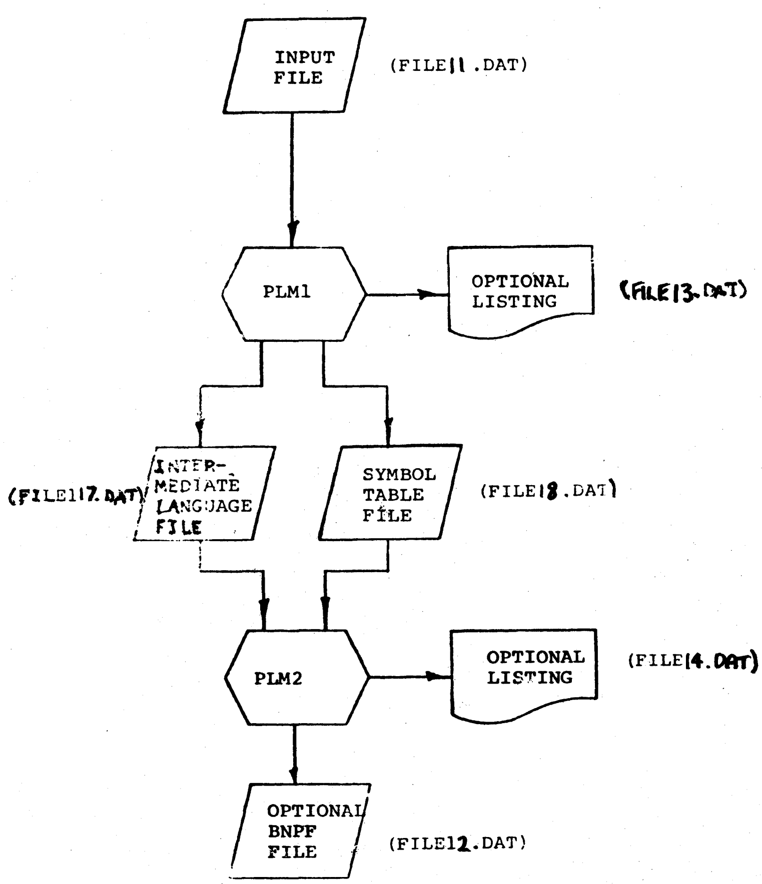

Figure IV-14. The ALCOM implementation of PLM1 and PLM2.

AL/COM FILE DEFINITIONS

PASS 1

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FILE5.DAT	5
2	DSK	FILE7.DAT	7
3	DSK	FILE8.DAT	8
4	DSK	FILE9.DAT	9
5	DSK	FILE1Ø.DAT	1Ø
6	DSK	FILE11.DAT	11
7	DSK	FILE12.DAT	12

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FILE6.DAT	6
2	DSK	FILE13.DAT	13
3	DSK	FILE14.DAT	14
4	DSK	FILE15.DAT	15
5	DSK	FILE16.DAT	16
6	DSK	FILE17.DAT	17
7	DSK	FILE18.DAT	18

PASS 2

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FILE5.DAT	5
2	DSK	FILE7.DAT	7
3	DSK	FILE8.DAT	8
4	DSK	FILE9.DAT	9
5	DSK	FILE1Ø.DAT	1Ø
6	DSK	FILE17.DAT	17
7	DSK	FILE18.DAT	18

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME	FORTRAN UNIT
1	TTY	FILE6.DAT	6
2	DSK	FILE13.DAT	13
3	DSK	FILE14.DAT	14
4	DSK	FILE15.DAT	15
5	DSK	FILE16.DAT	16
6	DSK	FILE11.DAT	11
7	DSK	FILE12.DAT	12

SAMPLE RUN ON GENERAL ELECTRIC TIMESHARE

```
OLD MYPROG
SAVE FILEIN
OLD PLM1
RUN
$O=2
$M
$S
$I=6
```

PASS 1 OF COMPILER IS INVOKED HERE

```
OLD PLM2
RUN
$F
$G
$B=7
$M
$0=2
(SPACE,CARRIAGE RETURN)
```

PASS 2 OF COMPILER IS INVOKED HERE

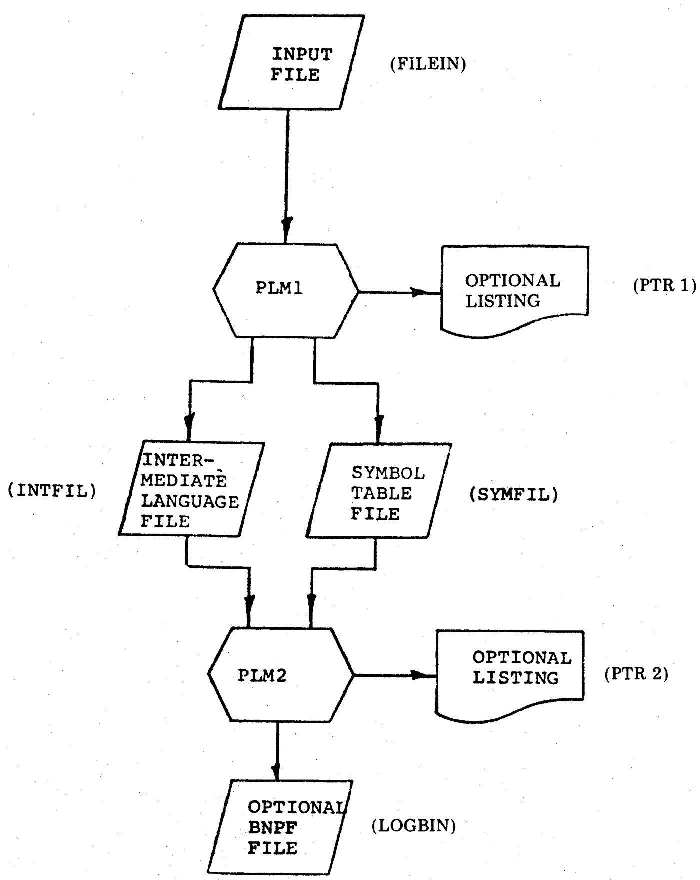

Figure IV-15. The General Electric implementation of PLM1 and PLM2.

ERRATA SHEET
October 24, 1973
GENERAL ELECTRIC FILE DEFINITIONS

PASS 1

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME
1	TERMINAL	---
2	DISK	CDR
3	DISK	PAPI
4	DISK	MAGI1
5	DISK	DECI1
6	DISK	FILEIN
7	DISK	LOGBIN

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME
1	TERMINAL	---
2	DISK	PTR1
3	DISK	PAPO
4	DISK	MAGO
5	DISK	DECO
6	DISK	INTFIL
7	DISK	SYMFIL

PASS 2

INTERNAL FILE NUMBER	INPUT DEVICE	FILENAME
1	TERMINAL	---
2	DISK	CDR
3	DISK	PAPI
4	DISK	MAGI1
5	DISK	DECI1
6	DISK	INTFIL
7	DISK	SYMFIL

INTERNAL FILE NUMBER	OUTPUT DEVICE	FILENAME
1	TERMINAL	---
2	DISK	PTR2
3	DISK	PAPO
4	DISK	MAGO
5	DISK	DECO
6	DISK	LOGOUT
7	DISK	LOGBIN

All "O" in FILENAME are the letter "O", not the character zero ("Ø").

V. PL/M RUN-TIME CONVENTIONS FOR THE 8008 CPU.

This section presents the run-time organization of PL/M programs, including storage allocation and subroutine linkage. The discussion below assumes an 8008 CPU environment, and thus programs which are intended to be independent of CPU architecture should not depend upon the conventions presented here.

1. Storage Allocation.

The overall organization of memory for the INTEL 8008 CPU is shown in Figure V-1. Memory is allocated in three main sections: the Instruction Storage Area (ISA), the Variable Storage Area (VSA), and the Free Storage Area (FSA). The beginning of the ISA is determined by the numeric label of the first statement within the PL/M program. If no numeric label is specified, the origin of the ISA defaults to zero, and the segment marked "unused" in Figure V-1 is empty. The "square root" program given in Appendix A contains a numeric label on the first statement to force the ISA to start at location 2048.

All code generated by the PL/M compiler is "pure." That is, no object code modifications are made at run-time. Thus, the ISA memory portion can be implemented in either RAM (Random-Access Memory) or ROM (Read-Only Memory).

The VSA portion of memory holds values of variables declared within the PL/M program in address-order. The first variable declared in the source program is at the lowest address in the VSA, while the last variable declared is at the highest address. It should be noted that double-byte (ADDRESS) variables are always aligned on an

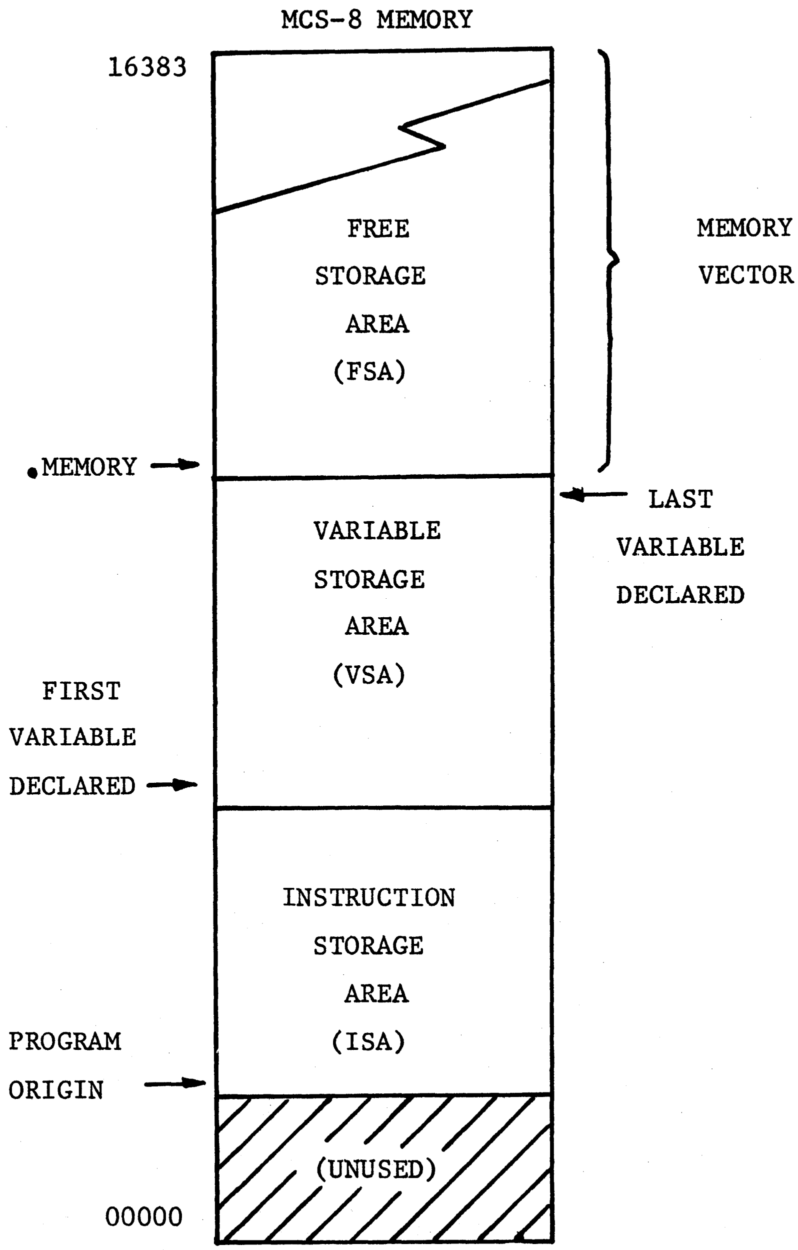

Figure V-1. Run-Time Storage Organization for the 8008 CPU.

even address boundary; thus, contiguous BYTE and ADDRESS declarations in the source program may or may not lead to contiguous allocation of these variables in the VSA. In addition, note that declarations with the DATA attribute cause allocation of the corresponding value in the ISA, not the VSA. Hence, DATA variables cannot be altered if the ISA is implemented in ROM.

The VSA is placed after the ISA, but never begins before the page indicated by the $VARIABLES compiler switch in PLM2 (the default value of this switch is zero). Suppose, for example, that pages 0, 1, and 2 of memory are implemented in unalterable ROM (recall that there are 256 bytes per page). The programmer would then set the switch

$VARIABLES = 3

during PLM2 to indicate that page number 3 is the first page in which variables can be allocated. If the ISA is contained within pages 0, 1, and 2 then the VSA begins in page 3. If the ISA extends past the first three pages into RAM then the length of the ISA determines the beginning of the VSA. The end of the VSA is always at an even page boundary.

Recall that there is one predeclared BYTE vector, called "MEMORY," which is automatically included in every PL/M program. The MEMORY vector is started after the last variable in the VSA, and thus represents the last area of memory, called the FSA, shown in Figure V-1. The length of the MEMORY vector is, of course, dependent upon the amount of memory physically attached to the particular 8008 CPU being used, and the length of the ISA and VSA. The length of MEMORY can be effectively computed at run-time, however, by attempting to read and write the first location in each page of the FSA. A subroutine for this purpose is shown in Figure V-2.

```
2     /*  THE MEM$LENGTH PROCEDURE RETURNS THE NUMBER OF
2     BYTES IN THE FREE STORAGE AREA (FSA) */
2     DECLARE TEST$VALUE LITERALLY '1010$1010B';
2     MEM$LENGTH: PROCEDURE ADDRESS;
3         DECLARE (I,MAX) ADDRESS;
3         I = 0; MAX = 4000H - .MEMORY;
3         /* MAX IS THE LARGEST POSSIBLE SIZE FOR THE FSA
3         IN A FULL 16K 8008 SYSTEM */
3         IF .MEMORY <> 0 THEN /* AT LEAST ONE FREE PAGE */
3   LOOP:     DO WHILE I < MAX;
3             /* WRITE THE TEST VALUE INTO THE FIRST WORD OF
3             THE PAGE */
3             MEMORY(I) = TEST$VALUE;
4             IF MEMORY(I) = TEST$VALUE THEN
4                 I = I + 256; ELSE MAX = 0;
4             END;
3         RETURN I;
3         END MEM$LENGTH;
2
2     /* TEST THE ABOVE PROCEDURE */
2     DECLARE RESULT ADDRESS;
2     START: RESULT = MEM$LENGTH;
2     FINISH: GO TO START;
2   EOF
NO PROGRAM ERRORS
```

Figure V-2. A PL/M Procedure for Determining MEMORY Length.

2. Subroutine Linkage Conventions.

The methods used for activating procedures and binding actual parameters to formal parameters in PL/M is given below. Again, note that the conventions given here are dependent upon the 8008 CPU environment.

Subroutine parameter passing is performed as follows. First, note that formal parameters declared in the procedure definition are treated the same as locally defined variables. That is, each parameter is allocated storage sequentially in memory as if it were a variable local to the procedure. Formal parameters, however, are initialized to their corresponding evaluated actual parameters at the time the procedure is invoked. Thus, all parameters are "call by value" in PL/M. This initialization of formal parameters is performed in two different ways, depending upon the number of arguments declared in the procedure. If there is only one parameter, the low-order byte is passed in CPU register B, while the high-order byte is sent in register C. If there are two parameters, the first is passed as above, and the second is passed in CPU registers D (low-order byte) and E (high-order byte). When there are more than two parameters, the <u>last</u> two are sent as described above, and the others are sent by generating implied assignment statements at the calling point which store the evaluated actual parameters into the variables representing the formal parameters.

The CPU registers are also used to hold values on return from procedures which have the BYTE or ADDRESS attribute. In the case of a BYTE procedure, the value returned is in the A register, while an ADDRESS procedure returns the low-order byte in register A, and the high-order byte in register C.

The eight-level program counter stack mechanism of the 8008 CPU is used to hold return addresses when subroutines are called. Although this stack size is sufficient for most PL/M programming applications, the user should be aware that the 8008 stack size limits nesting of subroutine calls to seven levels at run-time.

3. Use of Assembler Language Subroutines with PL/M.

Assembler language subroutines can be incorporated into PL/M programs if these subroutines account for the PL/M procedure conventions discussed previously.

The assembly language subroutines are first assembled into absolute locations, usually starting at low addresses in memory, as shown in Figure V-3. Each subroutine should end with a RET (return) operation code. The beginning address of each subroutine is obtained after assembly, denoted by S1, S2, ... ,Sn in Figure V-3.

For each subroutine S1, S2, ... ,Sn, write dummy PL/M interface procedures P1, P2, ... ,Pn where each Pi is a procedure containing the single statement

```
GO TO Si;
```

The procedure Pi can have zero, one, or two parameters of type BYTE or ADDRESS, and can return either a BYTE or ADDRESS value, or simply return with no value at all. Note that if more than two parameters are to be sent, or if more than one value is to be returned, ADDRESS variables can be used to "point to" parameters or results.

The subroutine Si then obtains parameters from the CPU registers B, C, D, and E, as given in the conventions above, and returns values through registers A and C.

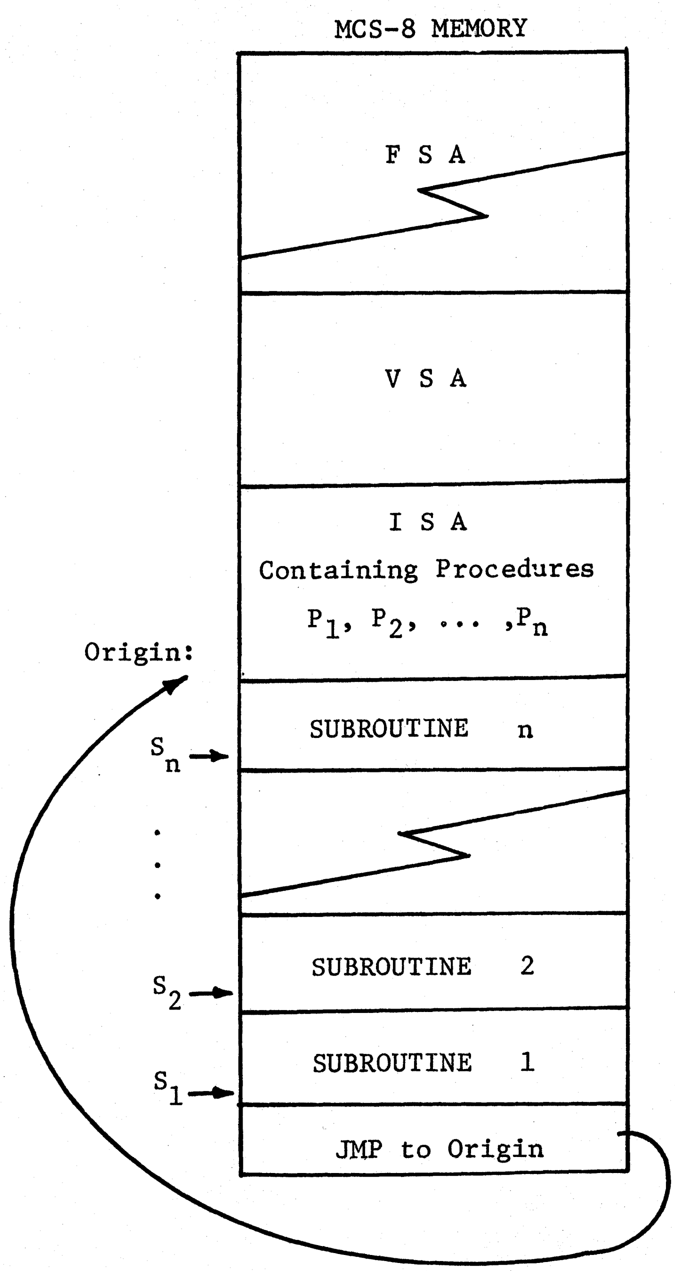

Figure V-3. Including Assembly Language Subroutines in PL/M Programs.

Suppose, for example, a programmer codes three subroutines in assembly language for handling teletype I/O. The subroutine S1 sends a line-feed-carriage-return, and is found at location 50 in memory. The subroutine S2 writes a single character at the teletype and returns. Assume S2 assembles starting at location 75. The subroutine S3 reads one character from the teletype, and is located between addresses 120 and 150 in memory. The following PL/M program then provides interface procedures for these assembly language subroutines.

```
150: DECLARE CRLFS LITERALLY '50',
         TTYOUTS LITERALLY '75',
         TTYINS LITERALLY '120 ';
     CRLF: PROCEDURE;
         GO TO CRLFS;
         END CRLF;
     TTYOUT: PROCEDURE (CHAR);
         DECLARE CHAR BYTE;
         GO TO TTYOUTS;
         END TTYOUT;
     TTYIN: PROCEDURE BYTE;
         GO TO TTYINS;
         END TTYIN;
```

The CRLF, TTYOUT, and TTYIN procedures can then be called in the same manner as any internally-defined procedure.

If the assembly language subroutines are not fully checked-out and thus are undergoing revisions, it may be worthwhile constructing a "jump vector" at the beginning of memory. The jump vector contains jump instructions to addresses of the currently assembled subroutines S1 through Sn in lower memory. The corresponding PL/M interface procedures then branch indirectly through this jump vector. If the subroutines are reassembled at different locations, only the jump vector need be changed, since it is not necessary to recompile the PL/M program.

As a final note, the programmer is reminded that assembly language subroutines should be used only when absolutely necessary. Changes to the PL/M system for future machine architecture will necessitate changes in subroutine conventions, resulting in loss of upward software compatibility in all programs which depend upon these conventions.

Appendix A

A Sample Program in PL/M

```
PASS-1

2      2048: /* IS THE ORIGIN OF THIS PROGRAM */
2  DECLARE TTO LITERALLY '2', CR LITERALLY '15Q', LF LITERALLY '0AH',
2          TRUE LITERALLY '1', FALSE LITERALLY '0';
2
2  SQUARE$ROOT: PROCEDURE(X) BYTE;
3      DECLARE (X,Y,Z) ADDRESS;
3      Y = X; Z = SHR(X+1,1);
3          DO WHILE Y <> Z;
3          Y = Z; Z = SHR(X/Y + Y + 1, 1);
4          END;
3      RETURN Y;
3      END SQUAREROOT;
2
2  PRINT$CHAR: PROCEDURE (CHAR);
3      DECLARE BIT$CELL LITERALLY '91',
3          (CHAR,I) BYTE;
3      OUTPUT (TTO) = 0;
3      CALL TIME (BIT$CELL);
3          DO I = 0 TO 7;
3          OUTPUT(TTO) = CHAR; /* DATA PULSES */

4          CHAR = ROR(CHAR,1);
4          CALL TIME(BIT$CELL);
4          END;
3      OUTPUT (TTO) = 1;
3      CALL TIME (BIT$CELL+BITCELL);
3      /* AUTOMATIC RETURN IS GENERATED */
3      END PRINT$CHAR;
2
2  PRINT$STRING: PROCEDURE(NAME,LENGTH);
3      DECLARE NAME ADDRESS,
3          (LENGTH,I,CHAR BASED NAME) BYTE;
3          DO I = 0 TO LENGTH - 1;
3          CALL PRINT$CHAR(CHAR(I));
4          END;
3      END PRINT$STRING;
2
2  PRINT$NUMBER: PROCEDURE(NUMBER,BASE,CHARS,ZERO$SUPPRESS);
3      DECLARE NUMBER ADDRESS, (BASE,CHARS,ZERO$SUPPRESS,I,J) BYTE;
3      DECLARE TEMP (16) BYTE;
3      IF CHARS > LAST(TEMP) THEN CHARS = LAST(TEMP);
3          DO I = 1 TO CHARS;
3          J = NUMBER MOD BASE + '0';
4          IF J > '9' THEN J = J + 7;
4          IF ZERO$SUPPRESS AND I <> 0 AND NUMBER = 0 THEN
4              J = ' ';
4          TEMP(LENGTH(TEMP)-I) = J;
4          NUMBER = NUMBER / BASE;
4          END;
3      CALL PRINT$STRING(.TEMP + LENGTH(TEMP) - CHARS, CHARS);
3      END PRINT$NUMBER;
2
2  DECLARE I ADDRESS,
2      CRLF LITERALLY 'CR,LF',
2      HEADING DATA (CRLF,LF,LF,
2      '                        TABLE OF SQUARE ROOTS', CRLF,LF,
2      ' VALUE  ROOT VALUE  ROOT VALUE  ROOT VALUE  ROOT',
2      CRLF,LF);
2
2      /* SILENCE TTY AND PRINT COMPUTED VALUES */
2      OUTPUT(TTO) = 1;
2      DO I = 1 TO 1000;
2      IF I MOD 5 = 1 THEN
3          DO; IF I MOD 250 = 1 THEN
4              CALL PRINT$STRING(.HEADING,LENGTH(HEADING));
4          END; ELSE
3      CALL PRINTSTRING(.(CR,LF),2);
3      CALL PRINT$NUMBER(I,10,6,TRUE /* TRUE SUPPRESSES LEADING ZEROES */);
3      CALL PRINT$NUMBER(SQUARE$ROOT(I), 10,6, TRUE);
3      END;
2
2  DECLARE MONITOR$USES (10) BYTE;
2  EOF
NO PROGRAM ERRORS
```

PASS-1 SYMBOL TABLE

```
SYMBOL  ADDR WDS CHRS   LENGTH PR TY
S00078* 0326  3   11 R 000010  1  1 MONITORUSES
S00077  0322  1    1 R 000006  1  6 6
S00076  0319  0    0 R 000000  4  4
S00075  0316  0    0 R 000000  4  4
S00074  0312  1    3 R 000250  1  6 250
S00073  0309  0    0 R 000000  4  4
S00072  0305  1    1 R 000005  1  6 5
S00071  0302  0    0 R 000000  4  4
S00070  0298  1    4 R 001000  2  6 1000
S00069  0295  0    0 R 000000  4  4
S00068  0280 12   60 R 000000  3  5 ' VALUE   ROOT VALUE   ROOT VALUE   ROOT VALUE   R(
S00067  0268  9   45 R 000000  3  5 '                                  TABLE OF SQUARE ROOTS
S00066  0264  1    2 R 000010  1  6 0A
S00065  0260  1    2 R 000013  1  6 15
S00064* 0255  2    7 R 000115  3  1 HEADING
S00063* 0251  1    1 R 000001  2  1 I
S00062  0247  1    1 R 000032  1  5 ' '
S00061  0244  0    0 R 000000  4  4
S00060  0241  0    0 R 000000  4  4
S00059  0237  1    1 R 000057  1  5 '9'
S00058  0233  1    1 R 000048  1  5 '0'
S00057  0230  0    0 R 000000  4  4
S00056  0227  0    0 R 000000  4  4
S00055  0224  0    0 R 000000  4  4
S00054* 0220  1    4 R 000016  1  1 TEMP
S00053* 0216  1    1 R 000001  1  1 J
S00052* 0212  1    1 R 000001  1  1 I
S00051  0209  0    0 R 000000  4  4
S00050* 0203  3   12 R 000001  1  1 ZEROSUPPRESS
S00049* 0199  1    5 R 000001  1  1 CHARS
S00048* 0195  1    4 R 000001  1  1 BASE
S00047* 0190  2    6 R 000001  2  1 NUMBER
S00046* 0184  3   11 R 000004  0  3 PRINTNUMBER


S00045  0181  0    0 R 000000  4  4
S00044  0178  0    0 R 000000  4  4
S00043* 0173  1    4 B 000001  1  1 CHAR  00000027H
S00042* 0169  1    1 R 000001  1  1 I
S00041  0166  0    0 R 000000  4  4
S00040* 0161  2    6 R 000001  1  1 LENGTH
S00039* 0157  1    4 R 000001  2  1 NAME
S00038* 0151  3   11 R 000002  0  3 PRINTSTRING
S00037  0148  0    0 R 000000  4  4
S00036  0144  1    1 R 000007  1  6 7
S00035  0141  0    0 R 000000  4  4
S00034  0137  1    2 R 000091  1  6 91
S00033  0133  1    1 R 000000  1  6 0
S00032  0129  1    1 R 000002  1  6 2
S00031* 0125  1    1 R 000001  1  1 I
S00030  0122  0    0 R 000000  4  4
S00029* 0118  1    4 R 000001  1  1 CHAR
S00028* 0113  2    9 R 000001  0  3 PRINTCHAR
S00027  0110  0    0 R 000000  4  4
S00026  0107  0    0 R 000000  4  4
S00025  0103  1    1 R 000001  1  6 1
S00024* 0099  1    1 R 000001  2  1 Z
S00023* 0095  1    1 R 000001  2  1 Y
S00022  0092  0    0 R 000000  4  4
S00021* 0088  1    1 R 000001  2  1 X
S00020* 0083  2   10 R 000001  1  3 SQUAREROOT
S00019  0079  1    4 R 002048  2  6 2048
S00018  0074  2    6 R 000001  2  2 DOUBLE
S00017  0070  1    4 R 000003  0  2 MOVE
S00016  0066  1    4 R 000001  1  2 LAST
S00015  0061  2    6 R 000001  1  2 LENGTH
S00014  0056  2    6 R 000001  1  2 OUTPUT
S00013  0052  1    5 R 000001  1  2 INPUT
S00012  0048  1    3 R 000001  1  2 LOW
S00011  0044  1    4 R 000001  1  2 HIGH
S00010  0040  1    4 R 000001  0  2 TIME
S00009  0036  1    3 R 000002  1  2 SHR
S00008  0032  1    3 R 000002  1  2 SHL
S00007  0028  1    3 R 000002  1  2 ROR
S00006  0024  1    3 R 000002  1  2 ROL
S00005  0019  2    6 R 000000  1  1 MEMORY
S00004  0014  2    6 R 000001  1  1 PARITY
S00003  0010  1    4 R 000001  1  1 SIGN
S00002  0006  1    4 R 000001  1  1 ZERO
S00001  0002  1    5 R 000001  1  1 CARRY
```

PASS-2

LINE NUMBER - ADDRESS CORRESPONDENCE

```
   2=0800H     6=0803H     7=080AH     8=081DH     9=0838H    10=089DH
  11=08A9H    12=08B1H    13=08B2H    16=08B5H    17=08BAH    18=08BCH
  19=08C5H    20=08C8H    21=08D2H    22=08D7H    23=08D8H    24=08E8H
  25=08EBH    26=08EFH    27=08F7H    28=08F8H    30=08FBH    31=0904H
  32=0907H    33=090DH    34=0923H    36=092DH    37=092EH    38=0931H
  39=0938H    40=093CH    41=093FH    42=0952H    43=096FH    44=0972H
  45=0999H    46=099DH    47=09AFH    48=09CCH    49=09D1H    51=09F5H
  52=09F6H
S00064 02553   115
    0DH 0AH 0AH 0AH 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H
20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 54H 41H 42H 4CH 45H 20H 4FH
46H 20H 53H 51H 55H 41H 52H 45H 20H 52H 4FH 4FH 54H 53H 0DH 0AH 0AH 20H
56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H
20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H
56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H
20H 52H 4FH 4FH 54H 0DH 0AH 0AH
  60=0A6CH    61=0A6FH    62=0A78H    63=0AA4H    64=0AC3H
  65=0AC6H    66=0ACFH
S00081 02773      2
    0DH 0AH  .
  67=0AD7H    68=0AF7H    69=0B09H    70=0B28H
S00001 00BCCH   S00002 00BCDH   S00003 00BCEH   S00004 00BCFH
S00005 00C00H   S00021 00BD0H   S00023 00BD2H   S00024 00BD4H
S00029 00BD6H   S00031 00BD7H   S00039 00BD8H   S00040 00BDAH
S00042 00BDBH   S00047 00BDCH   S00048 00BDFH   S00049 00BE0H
S00050 00BE1H   S00052 00BE2H   S00053 00BE3H   S00054 00BE4H
S00063 00BF4H   S00078 00BF6H   S00079 00BCAH   S00080 00BC8H
0000H HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT HLT
```

GENERATED OBJECT CODE

```
0800H JMP,B2H,08H LHI,0BH LLI,D0H LMB INL LMC DCL LBM INL LCM INL LMB
0810H INL LMC LLI,D0H LAM INL LCM ADI,01H LBA LAC ACI,00H ORA RAR LCA
0820H LAB RAR LLI,D4H LMA INL LMC LHI,0BH LLI,D2H LAM INL LCM INL SUM

0830H INL LBA LAC SBM ORB JTZ,A9H,08H DCL LBM INL LCM LLI,D2H LMB INL
0840H LMC DCL LBM INL LCM LLI,C8H LMB INL LMC LLI,D0H LBM INL LCM LLI
0850H,CAH LMB INL LMC JMP,8AH,08H LEM DCL LDM LMI,11H LBI,00H LCB LAD
0860H RAL LDA LAE RAL LEM DCE LME LEA RTZ LAB RAL LBA LAC RAL LCA DCL
0870H DCL LAB SUM LBA INL LAC SBM LCA JFC,83H,08H DCL LAB ADM LBA INL
0880H LAC ACM LCA INL SBA SBI,80H JMP,5FH,08H CAL,57H,08H LAD LLI,D2H
0890H ADM INL LDA LAE ACM LEA LAD ADI,01H LDA LAE ACI,00H ORA RAR LEA
08A0H LAD RAR INL LMA INL LME JMP,27H,08H LHI,0BH LLI,D2H LAM INL LCM
08B0H RET RET JMP,F8H,08H LHI,0BH LLI,D6H LMB XRA 010 LBI,5BH DCB JTZ
08C0H,C5H,08H JMP,BEH,08H INL LMI,00H LAI,07H LHI,0BH LLI,D7H SUM JTC
08D0H,E8H,08H DCL LAM 010 LAM RRC LMA LBI,5BH DCB JTZ,E1H,08H JMP,DAH
08E0H,08H INL LBM INB LMB JMP,C8H,08H LAI,01H 010 LAI,5BH ADI,5BH LBA
08F0H DCB JTZ,F7H,08H JMP,F0H,08H RET JMP,2EH,09H LHI,0BH LLI,D8H LMB
0900H INL LMC INL LMD INL LMI,00H LHI,0BH LLI,DAH LBM DCB LAB INL SUM
0910H JTC,20H,09H LAM LLI,D8H ADM INL LBA LAI,00H ACM LLB LHA LAM LBA
0920H CAL,B5H,08H LHI,0BH LLI,DBH LBM INB LMB JMP,07H,09H RET JMP,F6H
0930H,09H LHI,0BH LLI,E0H LMB INL LMD LAI,0FH DCL SUM JFC,41H,09H LMI
0940H,0FH LHI,0BH LLI,E2H LMI,01H LHI,0BH LLI,E0H LAM LLI,E2H SUM JTC
0950H,D9H,09H LLI,DFH LBM LLI,C8H LMB INL LMI,00H LLI,DCH LBM INL LCM
0960H LLI,CAH LMB INL LMC CAL,57H,08H LAB ADI,30H LBA LAC ACI,00H LLI
0970H,E3H LMB LAI,39H SUM JFC,7CH,09H LAM ADI,07H LMA LHI,0BH LLI,E2H
0980H LAM SUI,00H ADI,FFH SBA DCL NDM LLI,DCH LBA LAM INL LDM SUI,00H
0990H LCA LAD SBI,00H ORC SUI,01H SBA NDB RRC JFC,A1H,09H LLI,E3H LMI
09A0H,20H LAI,10H LHI,0BH LLI,E2H SUM LLI,E4H ADL LBA LAH ACI,00H DCL
09B0H LDM LLB LHA LMD LHI,0BH LLI,DFH LBM LLI,C8H LMB INL LMI,00H LLI
09C0H,DCH LBM INL LCM LLI,CAH LMB INL LMC CAL,57H,08H LLI,DCH LMD INL
09D0H LME LLI,E2H LBM INB LMB JMP,47H,09H LHI,0BH LLI,E4H LCH LAL ADI
09E0H,10H LBA LAC ACI,00H LCA LAB LLI,E0H SUM LBA LAC SBI,00H LLI,E0H
09F0H LDM LCA CAL,FBH,08H RET JMP,6CH,0AH R01 RRC RRC RRC INE INE INE
0A00H INE INE INE INE INE INE INE INE INE INE INE INE INE INE INE INE
0A10H INE INE INE INE INE JMP,41H,42H JMP,45H,2CH I07 CAL,20H,53H 008
0A20H 010 I00 CFS,45H,20H CFS,4FH,4FH JMP,53H,0DH RRC RRC INE CAL,41H
0A30H,4CH 010 I02 INE INE CFS,4FH,4FH JMP,20H,56H I00 JMP,55H,45H INE
0A40H INE CFS,4FH,4FH JMP,20H,56H I00 JMP,55H,45H INE INE CFS,4FH,4FH
0A50H JMP,20H,56H I00 JMP,55H,45H INE INE CFS,4FH,4FH JMP,20H,56H I00
0A60H JMP,55H,45H INE INE CFS,4FH,4FH JMP,0DH,0AH RRC LAI,01H 010 LHI
0A70H,0BH LLI,F4H LMI,01H INL LMI,00H LAI,E8H LCI,03H LHI,0BH LLI,F4H
0A80H SUM INL LBA LAC SBM JTC,28H,0BH LLI,C8H LMI,05H INL LMI,00H LLI
0A90H,F4H LBM INL LCM LLI,CAH LMB INL LMC CAL,57H,08H LAB SUI,01H LBA
0AA0H LAC SBI,00H ORB JFZ,D2H,0AH LLI,C8H LMI,FAH INL LMI,00H LLI,F4H
0AB0H LBM INL LCM LLI,CAH LMB INL LMC CAL,57H,08H LAB SUI,01H LBA LAC
0AC0H SBI,00H ORB JFZ,CFH,0AH LBI,F9H LCI,09H LDI,73H CAL,FBH,08H JMP
0AD0H,E0H,0AH JMP,D7H,0AH R01 RRC LBI,D5H LCI,0AH LDI,02H CAL,FBH,08H
0AE0H LHI,0BH LLI,F4H LBM INL LCM LLI,DCH LMB INL LMC LLI,DFH LMI,0AH
0AF0H LBI,06H LDI,01H CAL,31H,09H LHI,0BH LLI,F4H LBM INL LCM CAL,03H
0B00H,08H LHI,0BH LLI,DCH LMA INL LMI,00H LLI,DFH LMI,0AH LBI,06H LDI
0B10H,01H CAL,31H,09H LHI,0BH LLI,F4H LAM INL LCM ADI,01H LBA LAC ACI
0B20H,00H DCL LMB INL LMA JMP,78H,0AH HLT
```

```
 1 CARRY 05714
 2 ZERO 05715
 3 SIGN 05716
 4 PARITY 05717
 5 MEMORY 06000
20 SQUAREROOT 20017
21 X 05720
23 Y 05722
24 Z 05724
28 PRINTCHAR 21327
29 CHAR 05726
31 I 05727
38 PRINTSTRING 21757
39 NAME 05730
40 LENGTH 05732
42 I 05733
46 PRINTNUMBER 22307
47 NUMBER 05734
48 BASE 05737
49 CHARS 05740
50 ZEROSUPPRESS 05741
52 I 05742
53 J 05743
54 TEMP 05744
63 I 05764
64 HEADING 04771
78 MONITORUSES 05766
****************************************************
  2048 BNPNNNPNNF BPNPPNNPNF BNNNNPNNNF BNNPNPPPNF
       BNNNNPNPPF BNNPPNPPNF BPPNPNNNNF BPPPPPNNPF
  2056 BNNPPNNNNF BPPPPPNPNF BNNPPNNNPF BPPNNPPPPF
       BNNPPNNNNF BPPNPNPPPF BNNPPNNNNF BPPPPPNNPF
  2064 BNNPPNNNNF BPPPPPNPNF BNNPPNPPNF BPPNPNNNNF
       BPPNNNPPPF BNNPPNNNNF BPPNPNPPPF BNNNNNPNNF
  2072 BNNNNNNNPF BPPNNPNNNF BPPNNNNPNF BNNNNPPNNF
       BNNNNNNNNF BPNPPNNNNF BNNNPPNPNF BPPNPNNNNF
  2080 BPPNNNNNPF BNNNPPNPNF BNNPPNPPNF BPPNPNPNNF
       BPPPPPNNNF BNNPPNNNNF BPPPPPNPNF BNNPNPPPNF
  2088 BNNNNPNPPF BNNPPNPPNF BPPNPNNPNF BPPNNNPPPF
       BNNPPNNNNF BPPNPNPPPF BNNPPNNNNF BPNNPNPPPF
  2096 BNNPPNNNNF BPPNNPNNNF BPPNNNNPNF BPNNPPPPPF
       BPNPPNNNPF BNPPNPNNNF BPNPNPNNPF BNNNNPNNNF
  2104 BNNPPNNNPF BPPNNPPPPF BNNPPNNNNF BPPNPNPPPF
       BNNPPNPPNF BPPNPNNPNF BPPPPPNNPF BNNPPNNNNF
  2112 BPPPPPNPNF BNNPPNNNPF BPPNNPPPPF BNNPPNNNNF
       BPPNPNPPPF BNNPPNPPNF BPPNNPNNNF BPPPPPNNPF
  2120 BNNPPNNNNF BPPPPPNPNF BNNPPNPPNF BPPNPNNNNF
       BPPNNPPPPF BNNPPNNNNF BPPNPNPPPF BNNPPNPPNF
  2128 BPPNNPNPNF BPPPPPNNPF BNNPPNNNNF BPPPPPNPNF
       BNPNNNPNNF BPNNNPNPNF BNNNNPNNNF BPPPNNPPPF
  2136 BNNPPNNNPF BPPNPPPPPF BNNPPPPPNF BNNNPNNNPF
       BNNNNPPPNF BNNNNNNNNF BPPNPNNNPF BPPNNNNPPF
  2144 BNNNPNNPNF BPPNPPNNNF BPPNNNPNNF BNNNPNNPNF
       BPPPNNPPPF BNNPNNNNPF BPPPPPPNNF BPPPNNNNNF
  2152 BNNPNPNPPF BPPNNNNNPF BNNNPNNPNF BPPNNPNNNF
       BPPNNNNPNF BNNNPNNPNF BPPNPNNNNF BNNPPNNNPF
  2160 BNNPPNNNPF BPPNNNNNPF BPNNPNPPPF BPPNNPNNNF
       BNNPPNNNNF BPPNNNNPNF BPNNPPPPPF BPPNPNNNNF
  2168 BNPNNNNNNF BPNNNNNPPF BNNNNPNNNF BNNPPNNNPF
       BPPNNNNNPF BPNNNNPPPF BPPNNPNNNF BNNPPNNNNF

  2664 BNPNPNPNNF BNNNNPPNPF BNNNNPNPNF BNNNNPNPNF
       BNNNNNPPNF BNNNNNNNPF BNPNPNPNPF BNNPNPPPNF
  2672 BNNNNPNPPF BNNPPNPPNF BPPPPNPNNF BNNPPPPPNF
       BNNNNNNNPF BNNPPNNNNF BNNPPPPPNF BNNNNNNNNF
  2680 BNNNNNPPNF BPPPNPNNNF BNNNPNPPNF BNNNNNNPPF
       BNNPNPPPNF BNNNNPNPPF BNNPPNPPNF BPPPPNPNNF
  2688 BPNNPNPPPF BNNPPNNNNF BPPNNPNNNF BPPNNNNPNF
       BPNNPPPPPF BNPPNNNNNF BNNPNPNNNF BNNNNPNPPF
  2696 BNNPPNPPNF BPPNNPNNNF BNNPPPPPNF BNNNNNPNPF
       BNNPPNNNNF BNNPPPPPNF BNNNNNNNNF BNNPPNPPNF
  2704 BPPPPNPNNF BPPNNPPPPF BNNPPNNNNF BPPNPNPPPF
       BNNPPNPPNF BPPNNPNPNF BPPPPPNNPF BNNPPNNNNF
  2712 BPPPPPNPNF BNPNNNPPNF BNPNPNPPPF BNNNNPNNNF
       BPPNNNNNPF BNNNPNPNNF BNNNNNNNPF BPPNNPNNNF
  2720 BPPNNNNPNF BNNNPPPNNF BNNNNNNNNF BPNPPNNNPF
       BNPNNPNNNF BPPNPNNPNF BNNNNPNPNF BNNPPNPPNF
  2728 BPPNNPNNNF BNNPPPPPNF BPPPPPNPNF BNNPPNNNNF
       BNNPPPPPNF BNNNNNNNNF BNNPPNPPNF BPPPPNPNNF
  2736 BPPNNPPPPF BNNPPNNNNF BPPNPNPPPF BNNPPNPPNF
       BPPNNPNPNF BPPPPPNNPF BNNPPNNNNF BPPPPPNPNF
  2744 BNPNNNPPNF BNPNPNPPPF BNNNNPNNNF BPPNNNNNPF
       BNNNPNPNNF BNNNNNNNPF BPPNNPNNNF BPPNNNNPNF
  2752 BNNNPPPNNF BNNNNNNNNF BPNPPNNNPF BNPNNPNNNF
       BPPNNPPPPF BNNNNPNPNF BNNNNPPPNF BPPPPPNNPF
  2760 BNNNPNPPNF BNNNNPNNPF BNNNPPPPNF BNPPPNNPPF
       BNPNNNPPNF BPPPPPNPPF BNNNNPNNNF BNPNNNPNNF
  2768 BPPPNNNNNF BNNNNPNPNF BNPNNNPNNF BPPNPNPPPF
       BNNNNPNPNF BNNNNPPNPF BNNNNPNPNF BNNNNPPPNF
  2776 BPPNPNPNPF BNNNPNPPNF BNNNNPNPNF BNNNPPPPNF
       BNNNNNNPNF BNPNNNPPNF BPPPPPNPPF BNNNNPNNNF
  2784 BNNPNPPPNF BNNNNPNPPF BNNPPNPPNF BPPPPNPNNF
       BPPNNPPPPF BNNPPNNNNF BPPNPNPPPF BNNPPNPPNF
  2792 BPPNPPPNNF BPPPPPNNPF BNNPPNNNNF BPPPPPNPNF
       BNNPPNPPNF BPPNPPPPPF BNNPPPPPNF BNNNNPNPNF
  2800 BNNNNPPPNF BNNNNNPPNF BNNNPPPPNF BNNNNNNNPF
       BNPNNNPPNF BNNPPNNNPF BNNNNPNNPF BNNPNPPPNF
  2808 BNNNNPNPPF BNNPPNPPNF BPPPPNPNNF BPPNNPPPPF
       BNNPPNNNNF BPPNPNPPPF BNPNNNPPNF BNNNNNNPPF
  2816 BNNNNPNNNF BNNPNPPPNF BNNNNPNPPF BNNPPNPPNF
       BPPNPPPNNF BPPPPPNNNF BNNPPNNNNF BNNPPPPPNF
  2824 BNNNNNNNNF BNNPPNPPNF BPPNPPPPPF BNNPPPPPNF
       BNNNNPNPNF BNNNNPPPNF BNNNNNPPNF BNNPPPPPNF
  2832 BNNNNNNNPF BNPNNNPPNF BNNPPNNNPF BNNNNPNNPF
       BNNPNPPPNF BNNNNPNPPF BNNPPNPPNF BPPPPNPNNF
  2840 BPPNNNPPPF BNNPPNNNNF BPPNPNPPPF BNNNNNPNNF
       BNNNNNNNPF BPPNNPNNNF BPPNNNNPNF BNNNNPPNNF
  2848 BNNNNNNNNF BNNPPNNNPF BPPPPPNNPF BNNPPNNNNF
       BPPPPPNNNF BNPNNNPNNF BNPPPPNNNF BNNNNPNPNF
  2856 BNNNNNNNNF
$
NO PROGRAM ERRORS
```

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051
(408) 246-7501

intel

MCS TECHNICAL MEMORANDUM 15 March 1974

A GUIDE TO PL/M PROGRAMMING

This MCS Technical Memorandum provides replacement pages for the following MCS manual: A Guide to PL/M Programming.

The changed pages document the availability of PL/M Version 3.0. Note that prior to Version 3.0 some features of the language and the compiler are either not implemented in full or are not available.

Pages to be replaced or added are:

- 47 - 48
- 54, 58
- 65 - 67
- 95 - 102

File this memo at the back of the manual to provide a record of changes.

MCS108-0774-1K

129 INVALID USE OF BUILT-IN FUNCTION IN AN ASSIGNMENT.

130 PASS-2 COMPILER ERROR. INVALID VARIABLE PRECISION (NOT SINGLE BYTE OR DOUBLE BYTE). MAY BE DUE TO PREVIOUS ERROR.

131 LABEL RESOLUTION ERROR IN PASS-2 (MAY BE COMPILER ERROR).

132 (SAME AS 108).

133 (SAME AS 113).

134 INVALID PROGRAM TRANSFER (ONLY COMPUTED JUMPS ARE ALLOWED WITH A 'GO TO').

135 (SAME AS 134).

136 ERROR IN BUILT-IN FUNCTION CALL.

137 (NOT USED)

138 (SAME AS 107).

139 ERROR IN CHANGING VARIABLE TO ADDRESS REFERENCE. MAY BE A PASS-2 COMPILER ERROR, OR MAY BE CAUSED BY PRE-VOUS ERROR.

140 (SAME AS 107).

141 INVALID ORIGIN. CODE HAS ALREADY BEEN GENERATED IN THE SPECIFIED LOCATIONS.

142 A SYMBOL TABLE DUMP HAS BEEN SPECIFIED (USING THE $MEMORY TOGGLE IN PASS-1), BUT NO FILE HAS BEEN SPECIFIED TO RE-CEIVE THE BNPF TAPE (USE THE $BNPF=N CONTROL).

143 INVALID FORMAT FOR THE SIMULATOR SYMBOL TABLE DUMP (SEE ERROR 111).

144 STACK NOT EMPTY AT END OF COMPILATION. POSSIBLY CAUSED BY PREVIOUS COMPILATION ERROR.

145 PROCEDURES NESTED TOO DEEPLY (HL OPTIMIZATION) SIMPLIFY NESTING, OR RE-COMPILE WITH LARGER PSTACK

146 PROCEDURE OPTIMIZATION STACK UNDERFLOW. MAY BE A RETURN IN OUTER BLOCK.

147 RESTART LOCATIONS FOR SUBSCRIPT AND BASED VARIABLE SUBROUTINES OVERLAP (CHECK $1 THROUGH $7 PARAMETERS)

Figure IV-7. (Con't)

Switch Name	Use	Default
$0 = n	Controls branch to starting location if any restart toggle is specified ($1 thru $7)	0
$1 = n thru $7 = n	If n is greater than 7, inline code is generated for address computation. If n is between 0 and 7, a restart subroutine will be emitted in restart location n, and inline code in the program will be replaced by restart instructions (see section V.4).	8
$ANALYZE = n	Print a trace of the register allocation stack if n = 1. Include assigned registers is n = 2.	0
$BINARY = n	Do not write a object tape if n = 0. Otherwise, write a object tape to file n (see PL/M file numbering). The object tape format is determined by the setting of $QUICKDUMP.	0
$COUNT = n	(Same as Pass 1)	
$DELETE = n	(Same as Pass 1)	
$EOF	(Same as Pass 1)	
$FINISH	Print a decoded dump of the generated machine code at the end of Pass 2.	0
$GENERATE = n	Print a cross reference of source line numbers verses machine code locations if n = 1. If n = 2, print a trace of the intermediate language as it is read, as well.	0
$HEADER = n	Start machine code generation at location n when producing code dump or object tape.	0
$INPUT = n	(Same as Pass 1)	
$LEFTMARGIN = n	(Same as Pass 1)	
$MAP	Print a memory map showing symbol numbers and address assignments at the end of Pass 2.	0
$OUTPUT = n	(Same as Pass 1)	

$PRINT	(Same as Pass 1)
$QUICKDUMP = n	If n = 0, the object tape format will be BNPF. If n = 1, the object tape format will be hexidecimal, with 16 bytes per record. If n is greater than 1, the object tape will hexidecimal with n bytes per record.
$RIGHTMARGIN=n	(Same as Pass 1)
$TERMINAL	(Same as Pass 1)
$VARIABLES = n	The first page of random-access memory (RAM) is page n (numbering 0, 1, ... , 63).
$WIDTH = n	(Same as Pass 1)
$- = n	If n = 0, code is produced for the 8008 (500KHz clock). If n = 1, code is produced for the 8008-1 (800KHz clock).

Figure IV-8. PLM2 "$" compiler switches.

PASS-2

```
$generate = 1 (cross reference line numbers and locations in code)
$bnpf = 6      (write bnpf tape to internal file number 6)

   12=0003H   13=000EH   15=0011H   16=001EH   17=0026H   18=0043H
   19=0067H   20=006DH   21=0071H   22=0077H   23=0084H   24=0087H
   25=0089H   26=008AH   29=009CH   32=00A5H   33=00BEH   34=00E1H
   35=00E6H
```

Figure IV-9. Sample output from PLM2 corresponding to the INDEX procedure.

Suppose, for example, a programmer codes three subroutines in assembly language for handling teletype I/O. The subroutine S1 sends a line-feed-carriage-return, and is found at location 50 in memory. The subroutine S2 writes a single character at the teletype and returns. Assume S2 assembles starting at location 75. The subroutine S3 reads one character from the teletype, and is located between addresses 120 and 150 in memory. The following PL/M program then provides interface procedures for these assembly language subroutines.

```
150;DECLARE CRLFS LITERALLY '50',
          TTYOUTS LITERALLY '75',
          TTYINS LITERALLY '120';
     CRLF: PROCEDURE;
          GO TO CRLFS;
          END CRLF;
     TTYOUT: PROCEDURE (CHAR);
          DECLARE CHAR BYTE;
          GO TO TTYOUTS;
          END TTYOUT;
     TTYIN: PROCEDURE BYTE;
          GO TO TTYINS;
          END TTYIN;
```

The CRLF, TTYOUT, and TTYIN procedures can then be called in the same manner as any internally-defined procedure.

If the assembly language subroutines are not fully checked-out and thus are undergoing revisions, it may be worthwhile constructing a "jump vector" at the beginning of memory. The jump vector contains jump instructions to addresses of the currently assembled subrotines S1 through Sn in lower memory. The corresponding PL/M interface procedures then branch indirectly through this jump vector. If the subroutines are reassembled at different locations, only the jump vector need be changed, since it is not necessary to recompile the PL/M program.

As a final note, the programmer is reminded that assembly language subroutines should be used only when absolutely necessary. Changes to the PL/M system for future machine architecture will necessitate changes in subroutine conventions, resulting in loss of upward software compatibility in all programs which depend upon these conventions.

4. PL/M Restart Functions

The size of PL/M programs which make extensive use of based or subscripted variables may be significantly reduced by permitting the compiler to use the 8008 restarts. The compiler will then emit short 'subroutines' in the selected restart locations and substitute restart instructions for inline code in the body of the PL/M program. Seven restart subroutines are provided to handle various PL/M subscript and based variable constructs. Any combination of these seven available restart subroutines may be specified prior to starting pass 2, by entering the corresponding control toggles and restart numbers to be used. PL/M constructs and the associated control toggles are given in figure V-4. The toggles used should be selected on the basis of occurence of these constructs in the user's PL/M program. Figure V-4 lists typical code reduction, in bytes, for each use of each restart.

In general, all but the most trivial programs will benefit from the use of the restart subroutines. The restarts required for the constructs of figure V-4 are:

1) Based scalar variables require only control toggle 1.

2) Byte vectors with byte subscripts require control toggles 2 and 5.

3) Address vectors require control toggles 2 and 6, and in addition, 3 if byte subscripted and 4 if address subscripted.

4) Subscripted based variables require control toggles 2 and 7.

The default value of all the restart toggles is eight, indicating that neither the restart subroutine nor restart instructions will be produced. Setting a toggle to a value n between 0 and 7 selects the restart option, and forces the restart subroutine to be emitted at locations 8*n through 8*n+7.

The starting location of the user program will be that following the highest restart locations used, for example,

```
$2=4 $4=2 $6=3
```

will result in a starting location of 40 for the user program (subroutine 2 occupies locations 32 (8*4) through 39 (8*4+7)).

A program's starting address may be altered by setting the $HEADER control toggle, or by specifing an origin in the source code. Progam origins are not permitted which would origin the PL/M program at or below the last location used for the restart subroutines.

If any of the restart toggles are selected, the compiler will include a branch to the starting location of the program in location 0 through 2. Thus, a restart 0 may be used to start or restart the user program. Generation of the branch at location 0 is controlled by the control toggle 0. The default value of this toggle is 0, which forces the normal branch to the PL/M program's starting location. If the toggle is set to 1, no branch will be produced. Setting the toggle to a value n greater than 1 will force a branch at location 0 to the absolute address n.

Users of the Intellec 8 should be aware that the monitor uses locations 3 through 15 for all commands other than 'READ'. If a restart toggle is set to '1', the restart subroutine will be occupy locations 8 through 15. The program may be loaded using the monitor, but it may be started only by use of the reset switch to force a restart 0.

Control Toggle	Code Reduction	PL/M Construct
$1	4	Scalar based variables; subscripted based variables when $7 is not selected
$2	1	Complex expressions involving a subscripted variable and either a procedure call of another subscripted variable (may be called prior to calling restart 5, 6, or 7)
$3	3-4	Address vectors with byte subscripts
$4	3-4	Address vectors with address subscripts
	1-2	Address vectors with byte subscripts when $3 is not selected
$5	3-4	Byte vectors with byte subscripts
$6	3-4	All address vectors
	1-2	All byte vectors
$7	7-8	All based subscripted variables

Figure V-4. PL/M restart toggles and associated constructs

Appendix A

A Sample Program in PL/M

Source Listing

```
1       2048; /* IS THE ORIGIN OF THIS PROGRAM */
1   DECLARE TTO LITERALLY '2', CR LITERALLY '15Q', LF LITERALLY '0AH',
1           TRUE LITERALLY '1', FALSE LITERALLY '0';
1
1   SQUARE$ROOT: PROCEDURE(X) BYTE;
2       DECLARE (X,Y,Z) ADDRESS;
2       Y = X; Z = SHR(X+1,1);
2           DO WHILE Y <> Z;
2           Y = Z; Z = SHR(X/Y + Y + 1, 1);
3           END;
2       RETURN Y;
2       END SQUAREROOT;
1
1   PRINT$CHAR: PROCEDURE (CHAR);
2       DECLARE BIT$CELL LITERALLY '91',
2           (CHAR,I) BYTE;
2       OUTPUT (TTO) = 0;
2       CALL TIME (BIT$CELL);
2           DO I = 0 TO 7;
2           OUTPUT(TTO) = CHAR; /* DATA PULSES */
3           CHAR = ROR(CHAR,1);
3           CALL TIME(BIT$CELL);
3           END;
2       OUTPUT (TTO) = 1;
2       CALL TIME (BIT$CELL+BITCELL);
2       /* AUTOMATIC RETURN IS GENERATED */
2       END PRINT$CHAR;
1
1   PRINT$STRING: PROCEDURE(NAME,LENGTH);
2       DECLARE NAME ADDRESS,
2           (LENGTH,I,CHAR BASED NAME) BYTE;
2           DO I = 0 TO LENGTH - 1;
2           CALL PRINT$CHAR(CHAR(I));
3           END;
2       END PRINT$STIPING;
1
1   PRINT$NUMBER: PROCEDURE(NUMBER,BASE,CHARS,ZERO$SUPPRESS);
2       DECLARE NUMBER ADDRESS, (BASE,CHARS,ZEPO$SUPPRESS,I,J) BYTE;
2       DECLARE TEMP (16) BYTE;
2       IF CHARS > LAST(TEMP) THEN CHARS = LAST(TEMP);
2           DO I = 1 TO CHARS;
2           J = NUMBER MOD BASE + '0';
3           IF J > '9' THEN J = J + 7;
3           IF ZERO$SUPPRESS AND I <> 1 AND NUMBER = 0 THEN
3               J = ' ';
3           TEMP(LENGTH(TEMP)-I) = J;
3           NUMBER = NUMBER / BASE;
3           END;
2       CALL PRINT$STRING(.TEMP + LENGTH(TEMP) - CHARS, CHARS);
2       END PRINT$NUMBER;
1
1   DECLARE I ADDRESS,
1       CRLF LITEPALLY 'CR,LF',
1       HEADING DATA (CRLF,LF,LF,
1       '                       TABLE OF SQUAPE ROOTS', CRLF,LF,
1       ' VALUE  ROUT VALUE  ROOT VALUE  ROOT VALUE  ROOT VALUE  ROOT',
1       CRLF,LF);
1
1       /* SILENCE TTY AND PRINT COMPUTED VALUES */
1       OUTPUT(TTO) = 1;
1       DO I = 1 TO 1000;
1       IF I MOD 5 = 1 THEN
2           DO; IF I MOD 250 = 1 THEN
3               CALL PRINT$STRING(.HEADING,LENGTH(HEADING));
3           ELSE
3               CALL PRINISTPING(.(CR,LF),2);
3           END;
2       CALL PRINT$NUMPER(I,10,6,TRUE /* TRUE SUPPPESSES LEADING ZEROES */);
2       CALL PPINT$NUMBER(SQUARE$ROOT(I), 10,6, TRUE);
2       END;
1
1   DECLARE MONITOR$USES (10) BYTE;
1   EOF
NO PROGRAM ERRORS
```

Symbol Table

```
S00083 MONITORUSES
S00082 6
S00078 250
S00076 5
S00074 1000
S00072 ' VALUE   ROOT VALUE   ROOT VALUE   ROOT VALUE   ROOT VALUE   ROOT'
S00071 '                           TABLE OF SQUARE ROOTS'
S00070 0A
S00069 15
S00068 HEADING
S00066 I
S00065 ' '
S00062 '9'
S00061 '0'
S00057 TEMP
S00056 J
S00055 I
S00053 ZEROSUPPRESS
S00052 CHARS
S00051 BASE
S00050 NUMBER
S00049 PRINTNUMBER
S00046 CHAR
S00045 I
S00043 LENGTH
S00042 NAME
S00041 PRINTSTRING
S00039 7
S00037 91
S00036 0
S00035 2
S00034 I
S00032 CHAR
S00031 PRINTCHAR
S00028 1
S00027 Z
S00026 Y
S00024 X
S00023 SQUAREROOT
S00022 2048
S00020 DOUBLE
S00019 MOVE
S00018 LAST
S00017 LENGTH
S00016 OUTPUT
S00015 INPUT
S00014 LOW
S00013 HIGH
S00012 TIME
S00011 SCR
S00010 SCL
S00009 SHR
S00008 SHL
S00007 ROR
S00006 ROL
S00005 MEMORY
S00004 PARITY
S00003 SIGN
S00002 ZERO
S00001 CARRY
```

Source Line Number - Code Location Cross Reference

```
  2=0800H    6=0803H    7=080AH    8=081DH    9=0838H   10=089DH
 11=08A9H   12=08AFH   16=08B4H   18=08B6H   19=08BFH   20=08C2H
 21=08CCH   22=08D1H   23=08D2H   24=08E2H   25=08E5H   26=08E9H
 27=08F1H   28=08F2H   30=08FBH   32=08FEH   33=0904H   34=091AH
 36=0922H   37=0923H   38=092AH   40=092EH   41=0931H   42=0942H
 43=095FH   44=0962H   45=0988H   46=098CH   47=099CH   48=09B9H
 49=09BEH   51=09DEH   52=09DFH
S00068 02527   115
   0DH 0AH 0AH 0AH 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H
20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 54H 41H 42H 4CH 45H 20H 4FH
46H 20H 53H 51H 55H 41H 52H 45H 20H 52H 4FH 4FH 54H 53H 0DH 0AH 0AH 20H
56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H
20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H
56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H
20H 52H 4FH 4FH 54H 0DH 0AH 0AH
 60=0A52H   61=0A55H   62=0A5EH
 63=0A8AH   65=0AA9H   66=0AB5H
S00086 02744     2
   0DH 0AH
 67=0ABAH   68=0AC3H   69=0AD8H
 70=0AF1H   71=0B03H
```

Variable Address Map

```
S00001 00BCCH  S00002 00BCDH  S00003 00BCEH  S00004 00BCFH
S00005 00C00H  S00021 00BD0H  S00024 00BD0H  S00026 00BD2H
S00027 00BD4H  S00032 00BD6H  S00034 00BD7H  S00042 00BD8H
S00043 00BDAH  S00045 00BDBH  S00050 00BDCH  S00051 00BDFH
S00052 00BE0H  S00053 00BE1H  S00055 00BE2H  S00056 00BE3H
S00057 00BE4H  S00066 00BF4H  S00083 00BF8H  S00084 00BCAH
S00085 00BC8H
```

Generated Object Code

```
0800H JMP,52H,0AH LHI,0BH LLI,D0H LMB INL LMC DCL LBM INL LCM INL LMB
0810H INL LMC LLI,D0H LAM INL LCM ADI,01H LBA LAC ACI,00H ORA RAR LCA
0820H LAB RAR LLI,D4H LMA INL LMC LHI,0BH LLI,D2H LAM INL LCM INL SUM
0830H INL LBA LAC SBM ORB JTZ,A9H,08H DCL LBM INL LCM LLI,D2H LMB INL
0840H LMC DCL LBM INL LCM LLI,C8H LMB INL LMC LLI,D0H LBM INL LCM LLI
0850H,CAH LMB INL LMC JMP,8AH,08H LEM DCL LDM LMI,11H LBI,00H LCB LAD
0860H RAL LDA LAE RAL LEM DCE LME LEA RTZ LAB RAL LBA LAC RAL LCA DCL
0870H DCL LAB SUM LBA INL LAC SBM LCA JFC,83H,08H DCL LAB ADM LBA INL
0880H LAC ACM LCA INL SBA SBI,80H JMP,5FH,08H CAL,57H,08H LAD LLI,D2H
0890H ADM INL LDA LAE ACM LEA LAD ADI,01H LDA LAE ACI,00H ORA RAR LEA
08A0H LAD RAR INL LMA INL LME JMP,27H,08H LLI,D2H LAM INL LCM RET LHI
08B0H,0BH LLI,D6H LMB XRA 010 LBI,5BH DCB JTZ,BFH,08H JMP,B8H,08H INL
08C0H LMI,00H LAI,07H LHI,0BH LLI,D7H SUM JTC,E2H,08H DCL LAM 010 LAM
08D0H RRC LMA LBI,5BH DCB JTZ,D8H,08H JMP,D4H,08H INL LBM INB LMB JFZ
08E0H,C2H,08H LAI,01H 010 LAI,5BH ADI,5BH LBA DCB JTZ,F1H,08H JMP,EAH
08F0H,08H RET LHI,0BH LLI,D8H LMB INL LMC INL LMD INL LMI,00H LHI,0BH
0900H LLI,DAH LBM DCB LAB INL SUM JTC,22H,09H LAM LLI,D8H ADM INL LBA
0910H LAI,00H ACM LLB LHA LAM LBA CAL,AFH,08H LLI,DBH LBM INB LMB JFZ
0920H,FEH,08H RET LHI,0BH LLI,E0H LMB INL LMD LAI,0FH DCL SUM JFC,33H
0930H,09H LMI,0FH LLI,E2H LMI,01H LHI,0BH LLI,E0H LAM LLI,E2H SUM JTC
0940H,C6H,09H LLI,DFH LBM LLI,C8H LMB INL LMI,00H LLI,DCH LBM INL LCM
0950H LLI,CAH LMB INL LMC CAL,57H,08H LAB ADI,30H LBA LAC ACI,00H LLI
0960H,E3H LMB LAI,39H SUM JFC,6CH,09H LAM ADI,07H LMA DCL LBM DCB LAI
0970H,FFH JFZ,75H,09H XRA DCL NDM LLI,DCH LBA LAM INL LDM SUI,00H LCA
0980H LAD SBI,00H ORC SUI,01H SBA NDB RRC JFC,90H,09H LLI,E3H LMI,20H
0990H LAI,10H LLI,E2H SUM LLI,E4H ADL LBA LAH ACI,00H DCL LDM LLB LHA
09A0H LMD LHI,0BH LLI,DFH LBM LLI,C8H LMB INL LMI,00H LLI,DCH LBM INL
09B0H LCM LLI,CAH LMB INL LMC CAL,57H,08H LLI,DCH LMD INL LME LLI,E2H
09C0H LBM INB LMB JFZ,37H,09H LLI,E4H LCH LAL ADI,10H LBA LAC ACI,00H
09D0H LCA LAB LLI,E0H SUM LBA LAC SBI,00H LDM LCA CAL,F2H,08H RET
09DFH 0DH 0AH 0AH 0AH 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H
09EFH 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 20H 54H 41H 42H 4CH
09FFH 45H 20H 4FH 46H 20H 53H 51H 55H 41H 52H 45H 20H 52H 4FH 4FH 54H
0A0FH 53H 0DH 0AH 0AH 20H 56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H
0A1FH 20H 56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH
0A2FH 55H 45H 20H 20H 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H 20H
0A3FH 52H 4FH 4FH 54H 20H 56H 41H 4CH 55H 45H 20H 20H 52H 4FH 4FH 54H
0A4FH 0DH 0AH 0AH
```

```
0A52H LAI,01H 010 LHI,0BH LLI,F4H LMI,01H INL LMI,00H LAI,E8H LCI,03H
0A62H LHI,0BH LLI,F4H SUM INL LBA LAC SBH JTC,03H,0BH LLI,C8H LMI,05H
0A72H INL LMI,00H LLI,F4H LBM INL LCM LLI,CAH LMB INL LMC CAL,57H,08H
0A82H LAB SUI,01H LBA LAC SBI,00H ORB JFZ,C3H,0AH LLI,C8H LMI,FAH INL
0A92H LMI,00H LLI,F4H LBM INL LCM LLI,CAH LMB INL LMC CAL,57H,08H LAB
0AA2H SUI,01H LBA LAC SBI,00H ORB JFZ,BAH,0AH LBI,DFH LCI,09H LDI,73H
0AB2H CAL,F2H,08H JMP,C3H,0AH
0AB8H 0DH 0AH
0ABAH LBI,B8H LCI,0AH LDI,02H CAL,F2H,08H LLI,F4H LBM INL LCM LLI,DCH
0ACAH LMB INL LMC LLI,DFH LMI,0AH LBI,06H LDI,01H CAL,23H,09H LLI,F4H
0ADAH LBM INL LCM CAL,03H,08H LLI,DCH LMA INL LMI,00H LLI,DFH LMI,0AH
0AEAH LBI,06H LDI,01H CAL,23H,09H LLI,F4H LAM INL LCM ADI,01H LBA LAC
0AFAH ACI,00H DCL LMB INL LMA JMP,5EH,0AH HLT
NO PROGRAM ERRORS
```

BNPF Object Tape

```
   1 CARRY 05714
   2 ZERO 05715
   3 SIGN 05716
   4 PARITY 05717
   5 MEMORY 06000
  23 SQUAREROOT 04003
  24 X 05720
  26 Y 05722
  27 Z 05724
  31 PRINTCHAR 04257
  32 CHAR 05726
  34 I 05727
  41 PRINTSTRING 04362
  42 NAME 05730
  43 LENGTH 05732
  45 I 05733
  49 PRINTNUMBER 04443
  50 NUMBER 05734
  51 BASE 05737
  52 CHARS 05740
  53 ZEROSUPPRESS 05741
  55 I 05742
  56 J 05743
  57 TEMP 05744
  66 I 05764
  68 HEADING 04737
  83 MONITORUSES 05766
 $
*************************************************************
*************************************************************
   2048 BNPNNNPNNF BNPNPNNPNF BNNNNPNPNF BNNPNPPPNF
        BNNNNPNPPF BNNPPNPPNF BPPNPNNNNF BPPPPPNNPF
   2056 BNNPPNNNNF BPPPPPNPNF BNNPPNNNPF BPPNNPPPPF
        BNNPPNNNNF BPPNPNPPPF BNNPPNNNNF BPPPPPNNPF
   2064 BNNPPNNNNF BPPPPPNPNF BNNPPNPPNF BPPNPNNNNF
        BPPNNNPPPF BNNPPNNNNF BPPNPNPPPF BNNNNNPNNF
   2072 BNNNNNNNPF BPPNNPNNNF BPPNNNNPNF BNNNNPPNNF
        BNNNNNNNNF BPNPPNNNNF BNNNPPNPNF BPPNPNNNNF
   2080 BPPNNNNNPF BNNNPPNPNF BNNPPNPPNF BPPNPNPNNF
        BPPPPPNNNF BNNPPNNNNF BPPPPPNPNF BNNPNPPPNF
   2088 BNNNNPNPPF BNNPPNPPNF BPPNPNNPNF BPPNPNPPPF
        BNNPPNNNNF BPPNPNPPPF BNNPPNNNNF BPNNPNPPPF



   2776 BNNPPNPPNF BPPPPNPNNF BPPNNPPPPF BNNPPNNNNF
        BPPNPNPPPF BNPNNNPPNF BNNNNNNPPF BNNNNPNNNF
   2784 BNNPPNPPNF BPPNPPPNNF BPPPPPNNNF BNNPPNNNNF
        BNNPPPPPNF BNNNNNNNNF BNNPPNPPNF BPPNPPPPPF
   2792 BNNPPPPPNF BNNNNPNPNF BNNNNPPPNF BNNNNNPPNF
        BNNNPPPPNF BNNNNNNNPF BNPNNNPPNF BNNPNNNPPF
   2800 BNNNNPNNPF BNNPPNPPNF BPPPPNPNNF BPPNNNPPPF
        BNNPPNNNNF BPPNPNPPPF BNNNNNPNNF BNNNNNNNPF
   2808 BPPNNPNNNF BPPNNNNPNF BNNNNPPNNF BNNNNNNNNF
        BNNPPNNNPF BPPPPPNNPF BNNPPNNNNF BPPPPPNNNF
*************************************************************
   2816 BNPNNNPNNF BNPNPPPPNF BNNNNPNPNF BPPPPPPPPF
*************************************************************
```

Hexidecimal Object Tape

```
 1 CARRY 05714
 2 ZERO 05715
 3 SIGN 05716
 4 PARITY 05717
 5 MEMORY 06000
23 SQUAREROOT 04003
24 X 05720
26 Y 05722
27 Z 05724
31 PRINTCHAR 04257
32 CHAR 05726
34 I 05727
41 PRINTSTRING 04362
42 NAME 05730
43 LENGTH 05732
45 I 05733
49 PRINTNUMBER 04443
50-NUMBER 05734
51 BASE 05737
52 CHARS 05740
53 ZEROSUPPRESS 05741
55 I 05742
56 J 05743
57 TEMP 05744
66 I 05764
68 HEADING 04737
83 MONITORUSES 05766
$
*****************************************
:1008000044520A2E0B36D0F930FA31CF30D730F9B6
:1008100030FA36D0C730D70401C8C20C00B01AD0A5
:10082000C11A36D4F830FA2E0B36D2C730D73097EB
:1008300030C8C29FB169A90831CF30D736D2F9305D
:10084000FA31CF30D736C8F930FA36D0CF30D73674
:10085000CAF930FA448A08E731DF3E110E00D1C3ED
:1008600012D8C412E721FCE02BC112C8C212D03149
:1008700031C197C830C29FD040830831C187C8306A
:10088000C26FD030981C80445F08465708C336D2C8
:100890008730D8C48FE0C30401D8C40C00B01AE07C
:1008A000C31A30F830FC44270836D2C730D7072E99
:1008B0000B36D6F9A8550E5B0968BF0644B8083056
:1008C0003E0006072E0B36D79760E20831C755C7A2
:1008D0000AF80E5B0968DB0844D40830CF08F948F1
:1008E000C208060155065B045FC80966F10844EAC2
:1008F00008072E0B36D8F930FA30FB303E002E0BAD
:1009000036DACF09C13097602209C736D8F730C898
:1009100006008FF1E8C7C846AF0836DBCF08F948B4
:10092000FE08072E0B36E0F930FB060F319740 33F7
:10093000093E0F36E23E012E0B36E0C736E29760E5
:10094000C60936DFCF36C8F9303E0036DCCF30D7A7
:1009500036CAF930FA465708C10430C8C20C00360E
:10096000E3F9063997406C09C70407F831CF090647
:10097000FF487509A831A736DCC8C730DF1400D09E
:10098000C31C00B2140198A10A40900936E33E202E
:10099000061036E29736E466C8C50C0031CFF1E870
:1009A000FB2E0B36DFCF36C8F9303E0036DCCF3089
:1009B0000736CAF930FA46570836DCFB30FC36E247
:1009C000CF08F948370936E4D5C60410C8C20C0070
:1009D000D0C136E097C8C21C00DFD046F208070D30
:1009E0000A0A0A2020202020202020202020202049
:1009F0002020202020202020202020205441424C452F
:100A0000204F462053515541524520524F4F545389
:100A10000D0A0A2056414C55452020524F4F542074
:100A200056414C55452020524F4F542056414C556D
:100A3000452020524F4F542056414C554520205262E
:100A40004F4F542056414C55452020524F4F540D86
:100A50000A0A0601552E0B36F43E01303E0006E828
:100A600016032E0B36F49730C8C29F60030B36C8AE
:100A70003E05303E0036F4CF30D736CAF930FA465C
:100A80005708C11401C8C21C00B148C30A36C83E99
:100A9000FA303E0036F4CF30D736CAF930FA46572E
:100AA00008C11401C8C21C00B148BA0A0EDF1609F9
:100AB0001E7346F20844C30A0D0A0EB8160A1E0237
:100AC00046F20836F4CF30D736DCF930FA36DF3E5E
:100AD0000A0E061E0146230936F4CF30D746030816
:100AE00036DCF8303E0036DF3E0A0E061E01462395
:100AF0000936F4C730D70401C8C20C0031F930F808
:040B0000445E0AFF46
*****************************************
:0000000000
$
```

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 246-7501

Printed in U.S.A. MCS280-0974-1K