MCS-8

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION TO PL/M</td>
<td>1</td>
</tr>
<tr>
<td>II. A TUTORIAL APPROACH TO PL/M</td>
<td>2</td>
</tr>
<tr>
<td>1. The Organization of a PL/M Program</td>
<td>2</td>
</tr>
<tr>
<td>2. Basic Constituents of a PL/M Program</td>
<td>4</td>
</tr>
<tr>
<td>2.1. PL/M Character Set</td>
<td>4</td>
</tr>
<tr>
<td>2.2. Identifiers and Reserved Words</td>
<td>5</td>
</tr>
<tr>
<td>2.3. Comments</td>
<td>7</td>
</tr>
<tr>
<td>3. PL/M Statement Organization</td>
<td>7</td>
</tr>
<tr>
<td>4. PL/M Data Elements</td>
<td>9</td>
</tr>
<tr>
<td>4.1. Variable Declarations</td>
<td>9</td>
</tr>
<tr>
<td>4.2. Byte and Double Byte Constants</td>
<td>10</td>
</tr>
<tr>
<td>5. Well-Formed Expressions and Assignments</td>
<td>11</td>
</tr>
<tr>
<td>6. A Simple Example</td>
<td>15</td>
</tr>
<tr>
<td>7. DO-Groups</td>
<td>16</td>
</tr>
<tr>
<td>7.1. The DO-WHILE Group</td>
<td>16</td>
</tr>
<tr>
<td>7.2. The Iterative DO-Group</td>
<td>17</td>
</tr>
<tr>
<td>7.3. The DO-CASE</td>
<td>18</td>
</tr>
<tr>
<td>8. Subscripted Variables and the INITIAL Attribute</td>
<td>19</td>
</tr>
<tr>
<td>8.1. Subscript Declarations and Value References</td>
<td>19</td>
</tr>
<tr>
<td>8.2. The INITIAL Attribute</td>
<td>21</td>
</tr>
<tr>
<td>9. A Sorting Program</td>
<td>22</td>
</tr>
<tr>
<td>10. Procedure Definitions and Procedure Calls</td>
<td>23</td>
</tr>
<tr>
<td>10.1. Procedure Declarations</td>
<td>23</td>
</tr>
<tr>
<td>10.2. Procedure Calls</td>
<td>26</td>
</tr>
<tr>
<td>11. Based Variables</td>
<td>28</td>
</tr>
<tr>
<td>12. Long Constants</td>
<td>32</td>
</tr>
<tr>
<td>13. Scope of Variables</td>
<td>35</td>
</tr>
<tr>
<td>14. Statement Labels and GO TO's</td>
<td>38</td>
</tr>
<tr>
<td>14.1. Label Names</td>
<td>38</td>
</tr>
<tr>
<td>14.2. GO TO Statements</td>
<td>39</td>
</tr>
<tr>
<td>14.3. Scope of Labels</td>
<td>40</td>
</tr>
<tr>
<td>15. Compile-Time Macro Processing</td>
<td>43</td>
</tr>
<tr>
<td>16. Predeclared Variables and Procedures</td>
<td>45</td>
</tr>
<tr>
<td>16.1. Condition Code Variables</td>
<td>45</td>
</tr>
<tr>
<td>16.2. The MEMORY Vector</td>
<td>46</td>
</tr>
<tr>
<td>16.3. The TIME Procedure</td>
<td>46</td>
</tr>
<tr>
<td>16.4. Type Transfer Functions</td>
<td>47</td>
</tr>
<tr>
<td>16.5. Bit Manipulation Procedures</td>
<td>47</td>
</tr>
<tr>
<td>16.6. I/O Processing</td>
<td>48</td>
</tr>
<tr>
<td>III. THE FORMAL DEFINITION OF PL/M</td>
<td>49</td>
</tr>
<tr>
<td>IV. COMPILING AND DEBUGGING PL/M PROGRAMS</td>
<td>51</td>
</tr>
<tr>
<td>1. PLM1 Operating Procedures</td>
<td>51</td>
</tr>
<tr>
<td>2. PLM2 Operating Procedures</td>
<td>51</td>
</tr>
<tr>
<td>3. Program Check-Out</td>
<td>61</td>
</tr>
<tr>
<td>4. Implementation-Dependent Operating Procedures</td>
<td>80</td>
</tr>
<tr>
<td>V. PL/M RUN-TIME CONVENTIONS FOR THE 8008 CPU</td>
<td>88</td>
</tr>
<tr>
<td>1. Storage Allocation</td>
<td>88</td>
</tr>
<tr>
<td>2. Subroutine Linkage Conventions</td>
<td>92</td>
</tr>
<tr>
<td>3. Use of Assembler Language Subroutines with PL/M</td>
<td>93</td>
</tr>
</tbody>
</table>
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 to 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 occasionally 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

\[ X \times (Y - 3) / R \]

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

\[ Q = X \times (Y - 3) / R \]

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 identifiers and reserved words.

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.

Symbol  Name  Use
$  dollar  compiler controls, number
sign  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

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

6
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

\[ X \ast ( Y - 3 ) \div R \]

is equivalent to

\[ X*(Y-3)\div R \]

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

\[ A = B + C \ast D; \]

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

\[
\text{IF } A > B \text{ THEN } A = B;
\]

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

\[
\text{IF } A > B \text{ THEN } C = A; \text{ ELSE } C = B;
\]

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:

\[
\text{IF } A > B \text{ THEN }
\begin{align*}
\text{DO; } & A = B; \text{ B = C;} \\
\text{END;}
\end{align*}
\]

which would perform the two assignments to A and B only if A is greater than 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,

\[
\text{LAB1: } A = B + C * D;
\]

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

\[ X \times (Y - 3) / R \]

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

\[ \text{B O Q D H} \]

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

\[ 2 \text{33Q} \text{110B} \text{33FH} \text{55D} \text{55} \text{0BF3H} \text{65535} \]

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:

expression1 operator expression2

Examples are

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

Operators in expressions have an assumed precedence 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

\[(A + B) \times C\]

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

\[A + B - C \times D\]
\[A - (B + C) \times D\]
\[A / (B + C) \times D\]
\[A / (B + C)\]
\[A \text{ OR B AND OPH}\]
\[A + B \geq C - D\]

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:

\[
\text{DECLARE } X \text{ BYTE ;}
\]
\[
\text{DECLARE } (Y,Z) \text{ ADDRESS;}
\]
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 \textbf{NOT} operator is a \textit{unary} operator, and thus PL/\(m\) expressions involving NOT take the form

\[\text{NOT } \text{expression}\]

The effect of the \textbf{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 \textbf{NOT} operator are

\[
\text{NOT } A
\]
\[
\text{NOT } (A > B)
\]
\[
\text{NOT } A \text{ OR } B
\]

For convenience, a \textit{unary} minus sign is also allowed in PL/\(m\) expressions. The form of the \textit{unary} minus in an expression is

\[\text{- expression}\]

The effect is exactly the same as the expression

\[0 - \text{expression}\]

where the "-" in this last case is the \textit{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,
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:

```plm
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:

\[
A = 1;
DO \text{ WHILE } A \leq 3;
A = A + 1;
END;
\]

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

\[
\begin{align*}
&\text{DO variable = expression1 TO expression2;}
&\text{statement-1;}
&\text{statement-2;}
&\ldots
&\text{statement-n;}
&\text{END;}
\end{align*}
\]

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

\[
\begin{align*}
&\text{DO } I = 1 \text{ TO 10;}
&A = A + I;
&\text{END;}
\end{align*}
\]

Note that this DO-group has exactly the same effect as the following DO-WHILE:

\[
\begin{align*}
&I = 1;
&\text{DO WHILE } I \leq 10;
&A = A + I;
&I = I + 1;
\end{align*}
\]
END;

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

\begin{verbatim}
DO variable = expr1 TO expr2 BY expr3;
statement-1;
statement-2;
...
statement-n;
END;
\end{verbatim}

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

\begin{verbatim}
DO CASE expression;
statement-1;
statement-2;
...
statement-n;
END;
\end{verbatim}

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:

\begin{verbatim}
DO CASE X - 5;
X = X + 5; /* CASE 0 */
\end{verbatim}
DC; /* 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 contiguous 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 identifiers 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.

\[
\text{DECLARE } v(100) \text{ BYTE;}
\]
\[
\text{DECLARE } i \text{ BYTE;}
\]
\[
\text{DO } i = 0 \text{ TO LAST}(v);
\]
\[
v(i) = 5;
\]
\[
\text{END;}
\]
\[
\text{EOF}
\]

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.

\[
\text{DECLARE } x \text{ BYTE INITIAL}(10);
\]
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


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 formal parameters 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

\[ \text{RETURN;} \]

or

\[ \text{RETURN expression;} \]

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 \text{BYTE} or \text{ADDRESS}. The expression following the \text{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.

\begin{verbatim}
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;
\end{verbatim}
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 actual parameters, and are not restricted to simple PL/m identifiers. In fact, any valid PL/M expression can be placed in the argument-list. These expressions 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 = IDENTIFY(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

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

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

```plaintext
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;
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 within 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,OFFFH,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

\[
\begin{align*}
X(0) &= 'L' \\
X(10) &= 'G' \\
Y(3) &= 3 \\
\text{LENGTH}(Y) &= 11
\end{align*}
\]

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  F,C = 10;
3  A = B + C;
DO;
DECLARE (Q,R,S) BYTE;
Q, R = 20;
S = A + Q + R;
END;
D = 2 + A;
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 */
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

```plaintext
/* 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

\[ L1: \quad X = X + 1; \]
\[ LOOP: \quad Y = 3; \]
\[ L1: \quad LOOP: \quad X = Y + 5; \]
\[ 30: \quad Y = X - 5; \]
\[ LOOP: \quad 30: \quad L1: \quad Q = 5 + Y; \]

The function of numeric labels is to specify an \textit{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

\[ \text{GO TO label;} \]
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

\[ \text{GO TO constant;} \]
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

\[ \text{GO TO variable;} \]
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;
\[ X = X + 1; \]
\[ \ldots \]
\[ DO; \]
\[ \ldots \]
\[ GO \; TO \; EXIT; \]
\[ \ldots \]
\[ END; \]
\[ \ldots \]
\[ EXIT: \; HALT; \]
\[ EOF \]

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.


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

\[ n \times (100 \text{ usec}) \]

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.

16.5. Bit Manipulation Procedures. Four procedures are provided in PL/M for shifting and rotating expressions. These procedure calls take the forms
SHL(expression1,expression2);
SHR(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 0000$1100B. The call SHR(1$0000$1100B,1) results in the double byte value 0$1000$0110B.

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$000B,2) is 1100$00010B, and the value of ROR(1111$0000B,8) is 1111$0000B.

The SHL, SHR, 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

\[ \text{INPUT}(\text{constant}) \]

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

\[ \text{OUTPUT}(\text{constant}) = \text{expression}; \]

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 | <LABEL DEFINITION> <BASIC STATEMENT>

15 <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT>

16 | <IF CLAUSE> <TRUE PART> <STATEMENT>

17 | <LABEL DEFINITION> <IF STATEMENT>

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

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

20 <GROUP> ::= <GROUP HEAD> <ENDING>

21 <GROUP HEAD> ::= DO :

22 | DO <STEP DEFINITION> ;

23 | DO <WHILE CLAUSE> ;

24 | DO <CASE SELECTOR> ;

25 | <GROUP HEAD> <STATEMENT>

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

27 <ITERATION CONTROL> ::= <TO> <EXPRESSION>

28 | <TO> <EXPRESSION> <BY> <EXPRESSION>

29 <WHILE CLAUSE> ::= <WHILE> <EXPRESSION>

30 <CASE SELECTOR> ::= CASE <EXPRESSION>

31 <PROCEDURE DEFINITION> ::= <PROCEDURE HEAC> <STATEMENT LIST> <ENDING>

32 <PROCEDURE HEAC> ::= <PROCEDURE NAME> :

33 | <PROCEDURE NAME> <TYPE> :

34 | <PROCEDURE NAME> <PARAMETER LIST> :

35 | <PROCEDURE NAME> <PARAMETER LIST> <TYPE> ;

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

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

38 <PARAMETER HEAD> ::= {

39 | <PARAMETER HEAD> <IDENTIFIER> ,

40 <ENDING> ::= END <IDENTIFIER>

41 | <LABEL DEFINITION> <ENDING>

42 <LABEL DEFINITION> ::= <IDENTIFIER> :

43 | <NUMBER> :

44 <RETURN STATEMENT> ::= RETURN

45 | RETURN <EXPRESSION>

46 <CALL STATEMENT> ::= CALL <VARIABLE>

47 <GO TO STATEMENT> ::= <GO TO> <IDENTIFIER>

48 | <GO TO> <NUMBER>

49 <GO TO> ::= GO TO

50 | GOTO

51 <DECLARATION STATEMENT> ::= DECLARE <DECLARATION ELEMENT>

52 | <DECLARATION STATEMENT> , <DECLARATION ELEMENT>

53 <DECLARATION ELEMENT> ::= <TYPE DECLARATION>

54 | <IDENTIFIER LITERALLY <STRING>

55 | <IDENTIFIER <DATA LIST>.

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

57 <DATA HEAD> ::= DATA {

58 | <DATA HEAD> <CONSTANT> ,

59 <TYPE DECLARATION> ::= <IDENTIFIER SPECIFICATION> <TYPE>

60 | <BOUND HEAD> <NUMBER> ) <TYPE>

61 | <TYPE DECLARATION> <INITIAL LIST>

62 | <TYPE DECLARATION>
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:

\[(\text{n}}\text{nnnn}) \text{ ERROR } \mathbf{m} \text{ NEAR } \text{s}\]

The number \text{n}nnn\text{n} 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
Figure IV-1. PLM1 error messages issued during the first pass.
PASS-1 SYMBOL TABLE OVERFLOW (SEE ERROR 2 ABOVE).

INVALID USE OF AN IDENTIFIER AS A VARIABLE NAME.

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

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.

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

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

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

SUBSCRIPTED VARIABLE OR PROCEDURE CALL REFERENCES AN UNDECLARED IDENTIFIER, THE VARIABLE OR PROCEDURE MUST BE DECLARED BEFORE IT IS USED.

THE IDENTIFIER IS IMPROPERLY USED AS A PROCEDURE OR SUBSCRIPTED VARIABLE.

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

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

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

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

TOO MANY BLOCK LEVELS IN THE PL/M PROGRAM. EITHER SIMPLIFY YOUR PROGRAM (33 BLOCK LEVELS ARE CURRENTLY ALLOWED) OR RE-COMPILE PASS-1 WITH A LARGER BLOCK TABLE.

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

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

Figure IV-1 (Con't)
### Input

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Input Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interactive Console</td>
</tr>
<tr>
<td>2</td>
<td>Card Reader</td>
</tr>
<tr>
<td>3</td>
<td>Paper Tape</td>
</tr>
<tr>
<td>4</td>
<td>Magnetic Tape A</td>
</tr>
<tr>
<td>5</td>
<td>Magnetic Tape B</td>
</tr>
<tr>
<td>6</td>
<td>Sequential Disk A</td>
</tr>
<tr>
<td>7</td>
<td>Sequential Disk B</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Output Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interactive Console</td>
</tr>
<tr>
<td>2</td>
<td>Line Printer</td>
</tr>
<tr>
<td>3</td>
<td>Paper Tape</td>
</tr>
<tr>
<td>4</td>
<td>Magnetic Tape C</td>
</tr>
<tr>
<td>5</td>
<td>Magnetic Tape D</td>
</tr>
<tr>
<td>6</td>
<td>Sequential Disk C</td>
</tr>
<tr>
<td>7</td>
<td>Sequential Disk D</td>
</tr>
</tbody>
</table>

Figure IV-2. Symbolic Device Assignments for PLM1, PLM2, and INTERP/8.
### PASS-1 FILE DEFINITIONS

<table>
<thead>
<tr>
<th>NUM</th>
<th>INPUT DEVICE</th>
<th>UNIT</th>
<th>OUTPUT DEVICE</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>5</td>
<td>TTY</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>2</td>
<td>PTR</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>PAP</td>
<td>6</td>
<td>PAP</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>MAG</td>
<td>16</td>
<td>MAG</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>DEC</td>
<td>9</td>
<td>DEC</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>20</td>
<td>DISK</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>21</td>
<td>DISK</td>
<td>23</td>
</tr>
</tbody>
</table>

### IBM S/360 (CP/CMS)

<table>
<thead>
<tr>
<th>NUM</th>
<th>INPUT DEVICE</th>
<th>UNIT</th>
<th>OUTPUT DEVICE</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>5</td>
<td>TTY</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>10</td>
<td>PTR</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>TAP</td>
<td>11</td>
<td>PUN</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>TAP 140</td>
<td>9</td>
<td>TAP</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>DSK 80-L0</td>
<td>13</td>
<td>DSK 80-L0</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>DSK 80</td>
<td>1</td>
<td>DSK 80</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>DSK 80</td>
<td>2</td>
<td>DSK 80</td>
<td>4</td>
</tr>
</tbody>
</table>

### PASS-2 FILE DEFINITIONS

<table>
<thead>
<tr>
<th>NUM</th>
<th>INPUT DEVICE</th>
<th>UNIT</th>
<th>OUTPUT DEVICE</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>5</td>
<td>TTY</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>2</td>
<td>PTR</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>PAP</td>
<td>6</td>
<td>PAP</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>MAG</td>
<td>16</td>
<td>MAG</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>DEC</td>
<td>9</td>
<td>DEC</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>22</td>
<td>DISK</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>23</td>
<td>DISK</td>
<td>21</td>
</tr>
</tbody>
</table>

### IBM S/360 (CP/CMS)

<table>
<thead>
<tr>
<th>NUM</th>
<th>INPUT DEVICE</th>
<th>UNIT</th>
<th>OUTPUT DEVICE</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>5</td>
<td>TTY</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>10</td>
<td>PTR</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>TAP</td>
<td>11</td>
<td>PUN</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>TAP 140</td>
<td>9</td>
<td>TAP 133</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>DSK 80-L0</td>
<td>13</td>
<td>DSK 80-L0</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>DSK 80</td>
<td>1</td>
<td>DSK 80</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>DSK 80</td>
<td>4</td>
<td>DSK 80</td>
<td>7</td>
</tr>
</tbody>
</table>

All input records are 80 characters or less. All output records are 120 characters or less. The FORTRAN unit numbers can be changed in the subroutines GND and WRTTEL (these are the only occurrences 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 $INPUT switch. Normally, this switch defaults to the card reader if operating in batch mode, and to the terminal if operating in interactive mode. Subsequent switches in the primary file can be used to change these default values, if necessary (e.g., reset the left or right margin, or change to an alternate input file). The first pass normally creates a listing file on output file number 2, an intermediate symbol table on file 6, and an intermediate code file on file 7.
<table>
<thead>
<tr>
<th>Switch Name</th>
<th>Use</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ANALYZE</td>
<td>Controls the PL/M syntax analysis trace.</td>
<td>0</td>
</tr>
<tr>
<td>$COUNT = n</td>
<td>Start the line numbering at line n.</td>
<td>0</td>
</tr>
<tr>
<td>$DELETE = n</td>
<td>Delete all trailing characters in the output after position n.</td>
<td>120</td>
</tr>
<tr>
<td>$EOF</td>
<td>End-of-file on this unit.</td>
<td>0</td>
</tr>
<tr>
<td>$GENERATE</td>
<td>Interlist the intermediate language generated by Pass 1.</td>
<td>0</td>
</tr>
<tr>
<td>$INPUT = n</td>
<td>Switch to file n for subsequent input (see PL/M file numbering).</td>
<td>1</td>
</tr>
<tr>
<td>$LEFTMARGIN=n</td>
<td>Ignore all characters before column n in the input lines.</td>
<td>1</td>
</tr>
<tr>
<td>$MEMORY</td>
<td>Include a symbol table in the BNF tape produced by Pass 2 showing the memory address assignments for variables, labels, and procedures.</td>
<td>0</td>
</tr>
<tr>
<td>$OUTPUT = n</td>
<td>Write subsequent output lines to file n (see PL/M file numbering).</td>
<td>1</td>
</tr>
<tr>
<td>$PRINT</td>
<td>Print output lines (batch mode).</td>
<td>1</td>
</tr>
<tr>
<td>$RIGHTMARGIN=n</td>
<td>Ignore all characters in the input lines beyond position n.</td>
<td>72</td>
</tr>
<tr>
<td>$SYMBOLS</td>
<td>Print a symbol table dump at the end of Pass 1.</td>
<td>0</td>
</tr>
<tr>
<td>$TERMINAL</td>
<td>Interactive processing mode.</td>
<td>0</td>
</tr>
<tr>
<td>$WIDTH = n</td>
<td>Set output line width to n characters.</td>
<td>72</td>
</tr>
</tbody>
</table>

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 programmer'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 corresponding 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 substring 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 'OFFH';
/* 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 ('WALLA', 'WALLAWASH', EOS),
 (1, J) BYTE;
DO WHILE 1;
C1: I = INDEX(.Q, .'WALLA', EOS);
C2: I = INDEX(.Q, .'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

\[(nnnnn) \text{ERROR m}\]
PASS-1
S1=2 (could use $2=2 for printer listing, $1=1 for no listing)
00001 2 /* THE INDEX PROCEDURE SEARCHES THE STRING STARTING AT
00002 2 'A' FOR AN OCCURRENCE OF THE STRING STARTING AT 'R'.
00003 2 INDEX RETURNS A ZERO IF THE SECOND STRING IS NOT A SUB-
00004 2 STRING OF THE FIRST; OTHERWISE, THE POSITION OF THE
00005 2 SECOND STRING IS RETURNED. THE CHARACTER POSITIONS ARE
00006 2 COUNTED STARTING FROM 1 AND ENDING AT 255. */
00007 2 DECLARE EOS LITERALLY 'OFFH';
00008 2 /* THE LABELS L0 ... L5 AND C1 ... C3 ARE PRESENT FOR DEBUGGI-
00009 2 NG PURPOSES ONLY, AND CAN BE REMOVED WITHOUT AFFECTING THE PROG
00010 2 RAM EXECUTION */
00011 2 INDEX: PROCEDURE (A,B) BYTE;
00012 3 L0: DECLARE (A,B) ADDRESS,
00013 3 (SA BASED A, SB BASED B, J,K,L,M) BYTE;
00014 3 J = 0;
00015 3 L1: DO WHILE SA(J) <> EOS;
00016 3 K = 0;
00017 4 L2: DO WHILE (L:=SA(J+K)) = (M:=SR(K));
00018 4 L3: IF L = EOS THEN RETURN J+1;
00019 5 K = K + 1;
00020 5 END;
00021 4 J = J + 1;
00022 4 L4: IF M = EOS THEN RETURN J;
00023 4 END;
00024 3 L5: RETURN 0;
00025 3 END INDEX;
00026 2
00027 2 /* TEST THE INDEX FUNCTION */
00028 2 DECLARE Q DATA ('WALLAWALLAWASH',EOS),
00029 2 (I,J) BYTE;
00030 2 DO WHILE 1;
00031 2 C1: I = INDEX(Q,'WALLA',EOS));
00032 3 C2: I = INDEX(Q,'WALLA',EOS),Q);
00033 3 C3: I = INDEX(Q,'WASH',EOS));
00034 3 END;
00035 2 EOF
NO PROGRAM ERRORS

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

62
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 INDEXT 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

63
<table>
<thead>
<tr>
<th>ERROR NUMBER</th>
<th>MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Reference to storage locations outside the virtual memory of PASS-2. Re-compile PASS-2 with larger 'MEMORY' array.</td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>Virtual memory overflow. Program is too large to compile with present size of 'MEMORY.' Either shorten program or re-compile PASS-2 with a larger virtual memory.</td>
</tr>
<tr>
<td>104</td>
<td>(Same as 103).</td>
</tr>
<tr>
<td>105</td>
<td>$toggle used improperly in PASS-2. Attempt to complement a toggle which has a value other than 0 or 1.</td>
</tr>
<tr>
<td>106</td>
<td>Register allocation table underflow. May be due to a pre-</td>
</tr>
<tr>
<td>107</td>
<td>Register allocation error. No registers available. May be caused by a previous error, or PASS-2 compiler error.</td>
</tr>
<tr>
<td>108</td>
<td>PASS-2 symbol table overflow. Reduce number of symbols, or re-compile PASS-2 with larger symbol table.</td>
</tr>
<tr>
<td>109</td>
<td>Symbol table overflow (see error 108).</td>
</tr>
<tr>
<td>110</td>
<td>Memory allocation error. Too much storage specified in the source program (16K max on 8008). Reduce source program memory requirements.</td>
</tr>
<tr>
<td>111</td>
<td>Inline data format error. May be due to improper record size in symbol table file passed to PASS-2.</td>
</tr>
<tr>
<td>112</td>
<td>(Same as error 107).</td>
</tr>
<tr>
<td>113</td>
<td>Register allocation stack overflow. Either simplify the program or increase the size of the allocation stacks.</td>
</tr>
<tr>
<td>114</td>
<td>PASS-2 compiler error in 'LITADD' -- May be due to a previous error.</td>
</tr>
<tr>
<td>115</td>
<td>(Same as 114).</td>
</tr>
<tr>
<td>116</td>
<td>(Same as 114).</td>
</tr>
<tr>
<td>117</td>
<td>Line width set too narrow for code dump (use $WIDETHAN)</td>
</tr>
<tr>
<td>118</td>
<td>(Same as 107).</td>
</tr>
<tr>
<td>119</td>
<td>(Same as 110).</td>
</tr>
<tr>
<td>120</td>
<td>(Same as 110, but may be a PASS-2 compiler error).</td>
</tr>
<tr>
<td>121</td>
<td>(Same as 108).</td>
</tr>
<tr>
<td>122</td>
<td>Program requires too much program and variable storage. (Program and variables exceed 16K).</td>
</tr>
<tr>
<td>123</td>
<td>Initialized storage overlaps previously initialized storage.</td>
</tr>
<tr>
<td>124</td>
<td>Initialization table format error. (See error 111).</td>
</tr>
<tr>
<td>125</td>
<td>Inline data error. May have been caused by previous error.</td>
</tr>
<tr>
<td>126</td>
<td>Built-in function improperly called.</td>
</tr>
<tr>
<td>127</td>
<td>Invalid intermediate language format. (See error 111).</td>
</tr>
<tr>
<td>128</td>
<td>(Same as error 113).</td>
</tr>
</tbody>
</table>

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 PREVIOUS 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=0 CONTROL).
143 INVALID FORMAT FOR THE SIMULATOR SYMBOL TABLE DUMP (SEE ERROR 111).

Figure IV-7. (Cont')
<table>
<thead>
<tr>
<th>Switch Name</th>
<th>Use</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ANALYZE} = n$</td>
<td>Print a trace of the register allocation stack if $n=1$. Include assigned registers if $n=2$.</td>
<td>0</td>
</tr>
<tr>
<td>$\text{BNPF} = n$</td>
<td>Do not write a BNPF tape if $n=0$. Otherwise, write a BNPF tape to file $n$ (see PL/M file numbering).</td>
<td>0</td>
</tr>
<tr>
<td>$\text{COUNT} = n$</td>
<td>(Same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{DELETE} = n$</td>
<td>(Same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{EOF}$</td>
<td>(Same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{FINISH}$</td>
<td>Print a decoded dump of the generated machine code at the finish of Pass 2.</td>
<td>0</td>
</tr>
<tr>
<td>$\text{GENERATE} = n$</td>
<td>Print a cross reference of source line numbers versus machine code locations if $n=1$. If $n=2$, print a trace of the intermediate language as it is read, as well.</td>
<td>0</td>
</tr>
<tr>
<td>$\text{HEADER} = n$</td>
<td>Start machine code generation at location $n$ when producing a code dump or BNPF tape.</td>
<td>0</td>
</tr>
<tr>
<td>$\text{INPUT} = n$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{LEFTMARGIN}=n$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{MAP}$</td>
<td>Print a memory map showing symbol numbers and address assignments at the end of Pass 2.</td>
<td>0</td>
</tr>
<tr>
<td>$\text{OUTPUT} = n$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{PRINT}$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{RIGHTMARGIN}=n$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
<tr>
<td>$\text{TERMINAL}$</td>
<td>(same as Pass 1, default value suppresses the listing of the intermediate files as they are read)</td>
<td>0</td>
</tr>
<tr>
<td>$\text{VARIABLES} = n$</td>
<td>The first page of Random-access Memory (RAM) is page $n$ (numbering 0, 1,...,63)</td>
<td>0</td>
</tr>
<tr>
<td>$\text{WIDTH} = n$</td>
<td>(same as Pass 1)</td>
<td></td>
</tr>
</tbody>
</table>

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)

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

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

\begin{verbatim}
P1: PROCEDURE (A) BYTE;
    DECLARE (A,B) ADDRESS;
    . . .
    END P1;

P2: PROCEDURE (Q) ADDRESS;
\end{verbatim}
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
$\text{MAXCYCLE} = n$

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

$\text{GENLABELS}$

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

\[ \text{break AT } n = \text{ label displacement} \]

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 $\text{BNPF}$ 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
0000050 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
00000210 00017 0011H L1
00000540 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
0002540 00156 008CH C1
0002650 00181 0055H C2
0003160 00206 00C6H 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.
CY2SP A R C D E' H L HL SP P5C
*0000*00H*00H*00H*00H*00H*00H*00H*00H*00H*00H*00H*00H*00H*00H

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 WALLAHALLAH */
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 2370 159 9FH
di mem 159 to 165.
009FH 57H 41H 4CH 41H FFH 0EH
/* looks good too, so run the index procedure down to
label l2 (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/1 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 l3 */
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 I */

.go. di m 1 to m.
REFER AT 2CH=L1
00FCH 57H 57H

.go. di m 1 to m.
REFER AT 5AH=L2
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.
00FENH 01H

di m i dec.
00FENH 001

/* now try the second call */

.go.
REFER AT E1H=L0
di mem a to b+1.
00F6H B8H C0H 80H 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.
00124 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 1.
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 CK
REFER AT 156=C1
/* we are now back around the loop. i will be an 11
if all is well */
di mem 1.
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 h1.
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 h1, 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 h1, 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 h1.
HL = 252

di sy 252.
l
di me * to * + 10 code.
00066 LMA DCL LBA LAN 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 l.
GO OK
CYCLE AT 67=L2+23

76
di mem 1.
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.
0002340 00156 009CH

di cpu.
CYZSP A B C D E H L HL SP PS0 PS1
*0101*087*141 000*159 000*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 000156

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

set sp = 0. set pc=cl. di cpu.
SET OK
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 000156

/* 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*00000 000*00000

/* display the code between 12 and 13 */

di mem 12 to 13 cod.
00000 LHI,00H LLI,FAH LAM INL ADM LLI,F6H ADM INL LPA LAI,00H ACM LLR
00000 LHA LAM LHI,00H LLI,F6H LHY DCL LBA LAM LLI,F8H ADM INL LCA LAI
00005600H ACM LHY LHA LAM LHI,00H LLI,FDH LHY SUB JFZ,71H,00H DCL

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

77
/* 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
100h 001 002 003 004 005 006 007 001 002 003 004 005 006 007 001 002 003 004
10272 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=01

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

di pc, sp.
PC = 156
SP = 0

di b.
B = 0

di cpu.

C Y Z S P A B C D E H L HL SP PS0
0000 000 000 000 000 000 000 000 000 000 000*00156

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

go 5.
GO OK
0000 000 000 000 000 000 000 000 000 000*00044
LHI 0
0000 000 000 000 000 000 000 000 000*00046
LLI 250
0000 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*251*00251 000*00050
ADM
CYCLE AT 51=L2+7

base hex.
HEX BASE OK

go 30.
GO OK

78
*0001 05H 00H 00H 00H 00H 00H F7H 00F7H 00H 0033H
L1 L6H
0001 05H 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
C0Y05SP A B C D E H L HL SP PS0
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
L1R
0000 01H 05H 00H 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
L1I 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 F6H 00F6H 00H 0044H
LBA
C0Y05SP A B C D E H L HL SP PS0
0010 06H 06H 00H 00H 00H 00H F6H 00F6H 00H 0045H
LAM
0010 02H 06H 00H 00H 00H 00H F6H 00F6H 00H 0046H
L1I F8H
0010 02H 06H 00H 00H 00H 00H F6H 00F6H 00H 0048H
ADM
*0001 12H 06H 00H 00H 00H 00H F6H 00F6H 00H 0049H
INL
*0011 12H 06H 00H 00H 00H 00H F6H 00F6H 00H 004AH
LCA
0011 12H 06H 12H 06H 00H 00H 00H F6H 00F6H 00H 004BH
LAI 0H
0011 06H 12H 06H 00H 00H 00H F6H 00F6H 00H 004CH
ACM
*0000 01H 06H 12H 00H 00H 00H F6H 00F6H 00H 004DH
L1C
0000 01H 06H 12H 00H 00H 00H 00H 12H 0012H 00H 004EH
LHA
0000 01H 06H 12H 00H 00H 01H 12H 0112H 00H 0050H
LAM
C0Y05SP A B C D E H L HL SP PS0
0000 05H 06H 12H 00H 00H 01H 12H 0112H 00H 0051H
LHI 0H
0000 05H 06H 12H 00H 00H 01H 12H 0012H 00H 0053H
L1I F6H
0000 05H 06H 12H 00H 00H 00H FCH 00FCH 00H 0055H
LHA
0000 05H 06H 12H 00H 00H 00H F6H 00F6H 00H 0056H
SUB
*1011 F8H 06H 12H 00H 00H 00H F6H 00F6H 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 PDF-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 $I=1$ 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 initiated for the debugging run, and the "IOAL 7 7." command can be used to read this tape.
SAMPLE RUN ON INTEL FDP-18

COPY FOR22.DAT MYPROG.PLH
.SFT SPOOL LPT
.R PLM1
$1=6

PASS 1 OF COMPILER IS INVOKED HERE

.R PLM2
$8=7
(SPACE,CARRIAGE RETURN)

PASS 2 OF COMPILER IS INVOKED HERE

.PRINT *.LPT

Figure IV-]2. The INTEL implementation of PLM1 and PLM2.

81
SAMPLE RUN ON TYMSHARE PDP-10

COPY WPROG.PLM, FOR22.DAT
.RUN (UPL) PLM1
$0=2
$1=m1
$2=m1
$3=e

PASS 1 OF COMPILER IS INVOKED HERE

.RUN (UPL) PLM2
$2=m1
$3=m1
$4=7
$5=7

(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**

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>INPUT DEVICE</th>
<th>FILENAME</th>
<th>FORTRAN UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FORØ5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>FORØ2.DAT</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>PTR</td>
<td>FORØ6.DAT</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>MTAØ</td>
<td>FOR16.DAT</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>DTA1</td>
<td>FORØ9.DAT</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>DSKØ</td>
<td>FOR2Ø.DAT</td>
<td>2Ø</td>
</tr>
<tr>
<td>7</td>
<td>DSK1</td>
<td>FOR21.DAT</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>OUTPUT DEVICE</th>
<th>FILENAME</th>
<th>FORTRAN UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FORØ5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>LPT</td>
<td>FORØ3.DAT</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>PTP</td>
<td>FORØ7.DAT</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>MTA1</td>
<td>FOR17.DAT</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>DTA2</td>
<td>FOR1Ø.DAT</td>
<td>1Ø</td>
</tr>
<tr>
<td>6</td>
<td>DSK2</td>
<td>FOR22.DAT</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>DSK3</td>
<td>FOR23.DAT</td>
<td>23</td>
</tr>
</tbody>
</table>

**PASS 2**

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>INPUT DEVICE</th>
<th>FILENAME</th>
<th>FORTRAN UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FORØ5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>FORØ2.DAT</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>PTR</td>
<td>FORØ6.DAT</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>MTAØ</td>
<td>FOR16.DAT</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>DTA1</td>
<td>FORØ9.DAT</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>DSK2</td>
<td>FOR22.DAT</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>DSK3</td>
<td>FOR23.DAT</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>OUTPUT DEVICE</th>
<th>FILENAME</th>
<th>FORTRAN UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FORØ5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>LPT</td>
<td>FORØ3.DAT</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>PTP</td>
<td>FORØ7.DAT</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>MTA1</td>
<td>FOR17.DAT</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>DTA2</td>
<td>FOR1Ø.DAT</td>
<td>1Ø</td>
</tr>
<tr>
<td>6</td>
<td>DSKØ</td>
<td>FOR2Ø.DAT</td>
<td>2Ø</td>
</tr>
<tr>
<td>7</td>
<td>DSK1</td>
<td>FOR21.DAT</td>
<td>21</td>
</tr>
</tbody>
</table>

83
SAMPLE RUN ON AL/COM PDP-18

. COPY FILE18.DAT MYPROG.PL1
. APPLY PLM1
  $0=2
  $M=1
  $S=1
  $I=6

  PASS 1 OF COMPILER IS INVOKED HERE

  . APPLY PLM2
  $F=1
  $G=1
  $R=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

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Input Device</th>
<th>Filename</th>
<th>Fortran Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FILE5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>DSK</td>
<td>FILE7.DAT</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>DSK</td>
<td>FILE8.DAT</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>DSK</td>
<td>FILE9.DAT</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>DSK</td>
<td>FILE10.DAT</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>DSK</td>
<td>FILE11.DAT</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>DSK</td>
<td>FILE12.DAT</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Output Device</th>
<th>Filename</th>
<th>Fortran Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FILE6.DAT</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>DSK</td>
<td>FILE13.DAT</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>DSK</td>
<td>FILE14.DAT</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>DSK</td>
<td>FILE15.DAT</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>DSK</td>
<td>FILE16.DAT</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>DSK</td>
<td>FILE17.DAT</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>DSK</td>
<td>FILE18.DAT</td>
<td>18</td>
</tr>
</tbody>
</table>

#### Pass 2

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Input Device</th>
<th>Filename</th>
<th>Fortran Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FILE5.DAT</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>DSK</td>
<td>FILE7.DAT</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>DSK</td>
<td>FILE8.DAT</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>DSK</td>
<td>FILE9.DAT</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>DSK</td>
<td>FILE10.DAT</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>DSK</td>
<td>FILE11.DAT</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>DSK</td>
<td>FILE12.DAT</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Internal File Number</th>
<th>Output Device</th>
<th>Filename</th>
<th>Fortran Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTY</td>
<td>FILE6.DAT</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>DSK</td>
<td>FILE13.DAT</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>DSK</td>
<td>FILE14.DAT</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>DSK</td>
<td>FILE15.DAT</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>DSK</td>
<td>FILE16.DAT</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>DSK</td>
<td>FILE11.DAT</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>DSK</td>
<td>FILE12.DAT</td>
<td>12</td>
</tr>
</tbody>
</table>
SAMPLE RUN ON GENERAL ELECTRIC TIMESHARE

OLD HYPROG
SAVE FILEIN
OLD PLM1
RUN
SO=2
$M
$S
$I=6

PASS 1 OF COMPILER IS INVOKED HERE

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

PASS 2 OF COMPILER IS INVOKED HERE

Figure IV-15. The General Electric implementation of PLM1 and PLM2.
### GENERAL ELECTRIC FILE DEFINITIONS

#### PASS 1

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>INPUT DEVICE</th>
<th>FILENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TERMINAL</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>DISK</td>
<td>CDR</td>
</tr>
<tr>
<td>3</td>
<td>DISK</td>
<td>PAPI</td>
</tr>
<tr>
<td>4</td>
<td>DISK</td>
<td>MAGII</td>
</tr>
<tr>
<td>5</td>
<td>DISK</td>
<td>DECI1</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>FILEIN</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>LOGBIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>OUTPUT DEVICE</th>
<th>FILENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TERMINAL</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>DISK</td>
<td>PTR1</td>
</tr>
<tr>
<td>3</td>
<td>DISK</td>
<td>PAP0</td>
</tr>
<tr>
<td>4</td>
<td>DISK</td>
<td>MAG0</td>
</tr>
<tr>
<td>5</td>
<td>DISK</td>
<td>DECO</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>DISK01</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>DISK02</td>
</tr>
</tbody>
</table>

#### PASS 2

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>INPUT DEVICE</th>
<th>FILENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TERMINAL</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>DISK</td>
<td>CDR</td>
</tr>
<tr>
<td>3</td>
<td>DISK</td>
<td>PAPI</td>
</tr>
<tr>
<td>4</td>
<td>DISK</td>
<td>MAGII</td>
</tr>
<tr>
<td>5</td>
<td>DISK</td>
<td>DECI1</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>DISK01</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>DISK02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERNAL FILE NUMBER</th>
<th>OUTPUT DEVICE</th>
<th>FILENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TERMINAL</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>DISK</td>
<td>PTR2</td>
</tr>
<tr>
<td>3</td>
<td>DISK</td>
<td>PAP0</td>
</tr>
<tr>
<td>4</td>
<td>DISK</td>
<td>MAG0</td>
</tr>
<tr>
<td>5</td>
<td>DISK</td>
<td>DECO</td>
</tr>
<tr>
<td>6</td>
<td>DISK</td>
<td>LOGOUT</td>
</tr>
<tr>
<td>7</td>
<td>DISK</td>
<td>LOGBIN</td>
</tr>
</tbody>
</table>
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.
00001 2 /* THE MEM$LENGTH PROCEDURE RETURNS THE NUMBER OF
00002 2 BYTES IN THE FREE STORAGE AREA (FSA) */
00003 2 DECLARE TEST$VALUE LITERALLY '1010$1010B';
00004 2 MEM$LENGTH: PROCEDURE ADDRESS;
00005 3 DECLARE (I,MAX) ADDRESS;
00006 3 I = 0; MAX = 4000H - .MEMORY;
00007 3 /* MAX IS THE LARGEST POSSIBLE SIZE FOR THE FSA
00008 3 IN A FULL 16K 8008 SYSTEM */
00009 3 IF .MEMORY <> 0 THEN /* AT LEAST ONE FREE PAGE */
00010 3 LOOP: DO WHILE I < MAX;
00011 3 /* WRITE THE TEST VALUE INTO THE FIRST WORD OF
00012 3 THE PAGE */
00013 3 MEMORY(I) = TEST$VALUE;
00014 4 IF MEMORY(I) = TEST$VALUE THEN
00015 4 I = I + 256; ELSE MAX = 0;
00016 4 END;
00017 3 RETURN I;
00018 3 END MEM$LENGTH;
00019 2
00020 2 /* TEST THE ABOVE PROCEDURE */
00021 2 DECLARE RESULT ADDRESS;
00022 2 START: RESULT = MEM$LENGTH;
00023 2 FINISH: GO TO START;
00024 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 last 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
0001 2 2848: */ IS THE ORIGIN OF THIS PROGRAM */
0002 2 DECLARE TTO LITERALLY '2', OR LITERALLY '150', LF LITERALLY '0AH',
0003 2 TRUE LITERALLY '1', FALSE LITERALLY '0';
0004 2
0005 2 SQUARE ROOT: PROCEDURE (X) BYTE;
0006 3 DECLARE (X,Y,Z) ADDRESS;
0007 3 Y = X; Z = SHR(X+1,1);
0008 3 DO WHILE Y <> Z;
0009 3 Y = Z; Z = SHR(X/Y + Y + 1, 1);
0010 4 END;
0011 4 RETURN Y;
0012 4 END SQUARE ROOT;
0013 2
0014 2 PRINT STRING: PRINTCHAR: PROCEDURE (CHAR);
0015 3 DECLARE BIT CELL LITERALLY '91',
0016 3 (CHAR,.) BYTE;
0017 3 OUTPUT (TTO) = 81;
0018 3 CALL TIME (BITCELL);
0019 3 DO I = 0 TO 7;
0020 3 OUTPUT (TTO) = CHAR; /* DATA PULSES */
0021 4 CHAR = ROR (CHAR, 1);
0022 4 CALL TIME (BITCELL);
0023 4 END;
0024 3 OUTPUT (TTO) = 11;
0025 3 CALL TIME (BITCELL); /* AUTOMATIC RETURN IS GENERATED */
0026 3 END PRINT STRING;
0027 3
0028 2 PRINT STRING: PRINT CHAR: PROCEDURE (NAME, LENGTH);
0029 3 DECLARE NAME ADDRESS,
0030 3 (LENGTH,1,CHAR BASED NAME) BYTE;
0031 3 DO I = 0 TO LENGTH - 11;
0032 3 CALL PRINT STRING (CHAR(I));
0033 4 END;
0034 3 END PRINT STRING;
0035 3
0036 2 PRINT NUMBER: PROCEDURE (NUMBER, BASE, CHARs, ZEROSUPPRESS);
0037 3 DECLARE NUMBER ADDRESS, (BASE, CHARs, ZEROSUPPRESS,1,J) BYTE;
0038 3 DECLARE TEMP (16) BYTE;
0039 3 IF CHARs > LAST(TEMP) THEN CHARs = LAST(TEMP);
0040 3 DO I = 1 TO CHARs;
0041 3 J = NUMBER MOD BASE + '0';
0042 3 IF J > '9' THEN J = J - 7;
0043 4 IF ZEROSUPPRESS I <> 0 AND NUMBER = 0 THEN
0044 4 J = 11;
0045 4 TEMP(LENGTH(TEMP)-1) = J;
0046 4 NUMBER = NUMBER / BASE;
0047 4 END;
0048 4 CALL PRINT STRING (TEMP, LENGTH (TEMP) - CHARs, CRF, CHARs);
0049 3 END PRINT NUMBER;
0050 3
0051 2 DECLARE I ADDRESS,
0052 2 CRF LITERALLY 'CRF.LF',
0053 2 HEADING DATA (CRF.LF.LF,
0054 2 ' TABLE OF SQUARE ROOTS', CRF.LF,
0055 2 ' VALUE ROOT VALUE ROOT VALUE ROOT VALUE ROOT ',
0056 2 CRF.LF);
0057 2
0058 2 /* SILENC TTY AND PRINT COMPUTER VALUES */
0059 2 OUTPUT (TTO) = 11;
0060 2 DO I = 1 TO 10000;
0061 2 IF I MOD 5 = 1 THEN
0062 3 DO: IF I MOD 250 = 1 THEN
0063 4 CALL PRINT STRING (.HEADING, LENGTH (HEADING));
0064 4 END: ELSE
0065 3 CALL PRINT STRING (.CRF.LF, 2);
0066 3 CALL PRINT NUMBER (I, 10, 6, TRUE /* TRUE SUPPRESSES LEADING DEGRES */);
0067 3 CALL PRINT NUMBER(SQUARE ROOT(I), 10, 6, TRUE);
0068 3 END;
0069 3
0070 2 DECLARE MONITOR US (10) BYTE;
0071 2 EDF
0072 2 NO PROGRAM ERRORS

97
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ADDR</th>
<th>WDS</th>
<th>CHRS</th>
<th>LENGTH</th>
<th>PR</th>
<th>TY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S00079</td>
<td>0326</td>
<td>3</td>
<td>1</td>
<td>R</td>
<td>00010</td>
<td>1</td>
</tr>
<tr>
<td>S00077</td>
<td>0322</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>00006</td>
<td>1</td>
</tr>
<tr>
<td>S00076</td>
<td>0319</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>00000</td>
<td>4</td>
</tr>
<tr>
<td>S00075</td>
<td>0316</td>
<td>3</td>
<td>0</td>
<td>R</td>
<td>00000</td>
<td>4</td>
</tr>
<tr>
<td>S00074</td>
<td>0312</td>
<td>1</td>
<td>3</td>
<td>R</td>
<td>000250</td>
<td>1</td>
</tr>
<tr>
<td>S00073</td>
<td>0309</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>00000</td>
<td>4</td>
</tr>
<tr>
<td>S00072</td>
<td>0305</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>00005</td>
<td>1</td>
</tr>
<tr>
<td>S00071</td>
<td>0301</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>00000</td>
<td>4</td>
</tr>
<tr>
<td>S00070</td>
<td>0298</td>
<td>1</td>
<td>4</td>
<td>R</td>
<td>000100</td>
<td>2</td>
</tr>
<tr>
<td>S00069</td>
<td>0295</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>00000</td>
<td>4</td>
</tr>
<tr>
<td>S00068</td>
<td>0292</td>
<td>12</td>
<td>6</td>
<td>R</td>
<td>00000</td>
<td>3</td>
</tr>
<tr>
<td>S00067</td>
<td>0289</td>
<td>9</td>
<td>45</td>
<td>R</td>
<td>00000</td>
<td>3</td>
</tr>
<tr>
<td>S00066</td>
<td>0286</td>
<td>2</td>
<td>2</td>
<td>R</td>
<td>000010</td>
<td>1</td>
</tr>
<tr>
<td>S00065</td>
<td>0283</td>
<td>1</td>
<td>2</td>
<td>R</td>
<td>000013</td>
<td>1</td>
</tr>
<tr>
<td>S00064</td>
<td>0255</td>
<td>2</td>
<td>7</td>
<td>R</td>
<td>000015</td>
<td>3</td>
</tr>
<tr>
<td>S00063</td>
<td>0251</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000001</td>
<td>2</td>
</tr>
<tr>
<td>S00062</td>
<td>0247</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000032</td>
<td>1</td>
</tr>
<tr>
<td>S00061</td>
<td>0244</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00060</td>
<td>0241</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00059</td>
<td>0237</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000057</td>
<td>1</td>
</tr>
<tr>
<td>S00058</td>
<td>0233</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000046</td>
<td>1</td>
</tr>
<tr>
<td>S00057</td>
<td>0230</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00056</td>
<td>0227</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00055</td>
<td>0224</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00054</td>
<td>0221</td>
<td>1</td>
<td>4</td>
<td>R</td>
<td>000016</td>
<td>1</td>
</tr>
<tr>
<td>S00053</td>
<td>0216</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>S00052</td>
<td>0212</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>S00051</td>
<td>0209</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>000000</td>
<td>4</td>
</tr>
<tr>
<td>S00050</td>
<td>0206</td>
<td>3</td>
<td>12</td>
<td>R</td>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>S00049</td>
<td>0199</td>
<td>1</td>
<td>5</td>
<td>R</td>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>S00048</td>
<td>0195</td>
<td>1</td>
<td>4</td>
<td>R</td>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>S00047</td>
<td>0190</td>
<td>2</td>
<td>6</td>
<td>R</td>
<td>000001</td>
<td>2</td>
</tr>
<tr>
<td>S00046</td>
<td>0184</td>
<td>3</td>
<td>11</td>
<td>R</td>
<td>000004</td>
<td>0</td>
</tr>
</tbody>
</table>

S00045 0181 0 3 R 000000 4 4
S00044 0178 0 0 R 000000 4 4
S00043 0173 1 4 R 000001 1 1 Char R0000027h
S00042 0169 1 1 R 000001 1 1
S00041 0166 0 0 R 000020 4 4
S00040 0161 2 6 R 000021 1 1 Length
S00039 0157 1 4 R 000021 2 1 Name
S00038 0151 3 11 R 00002 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 000001 1 6 91
S00033 0133 1 1 R 000020 1 6 0
S00032 0129 1 1 R 000002 1 6 2
S00031 0126 1 1 R 000001 1 1 1
S00030 0122 0 0 R 000000 4 4
S00029 0118 1 4 R 000001 1 1 Char
S00028 0115 2 9 R 000001 0 3 Printchar
S00027 0112 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 2
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 0084 2 10 R 000011 1 3 Square Root
S00019 0079 1 4 R 000248 2 6 248
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 2 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