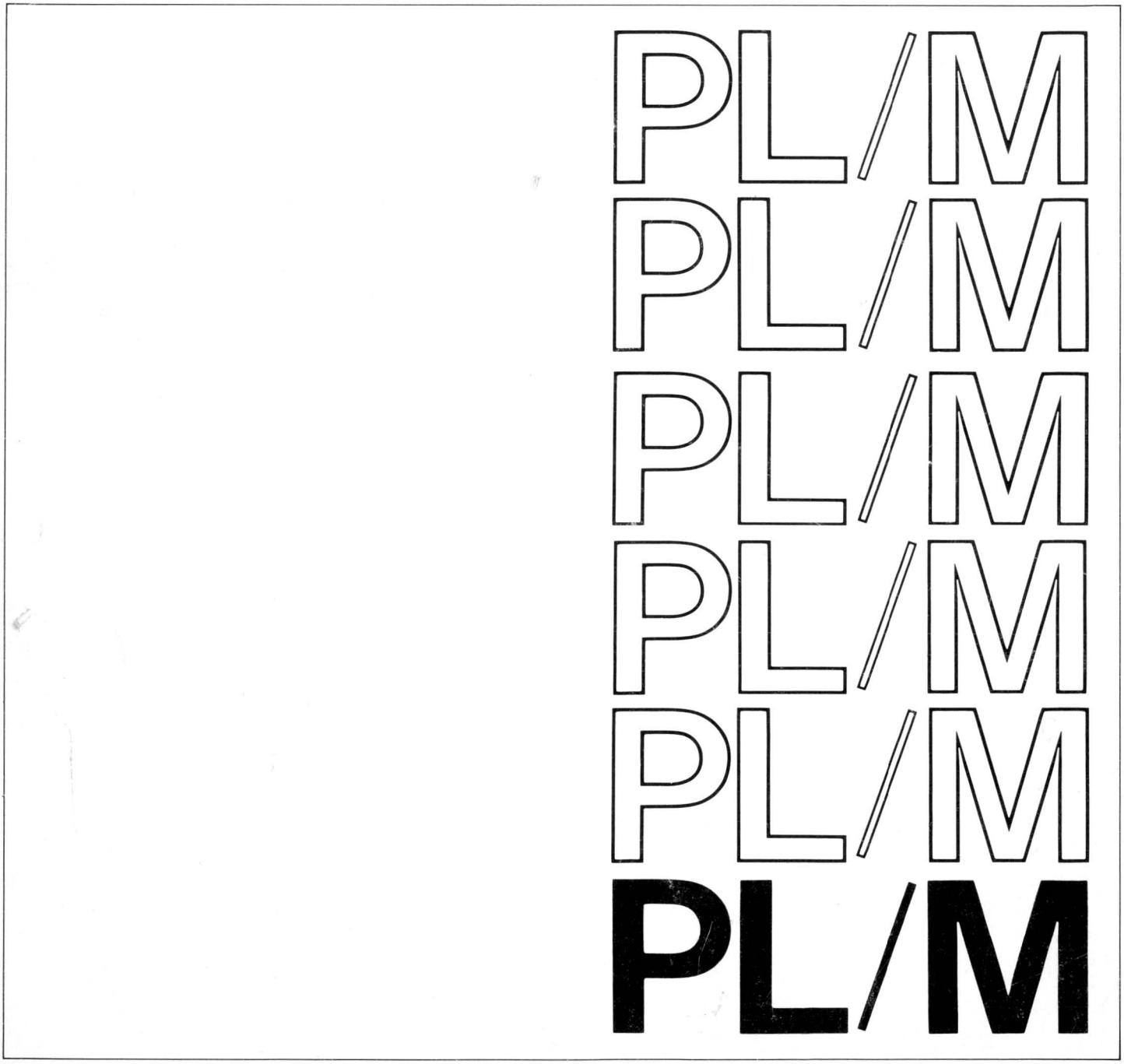
intel®

PL/M-80
PROGRAMMING MANUAL



98-268B

PL/M-80 PROGRAMMING MANUAL

Document Number 98-268B

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

PREFACE

This is a programming manual for the PL/M-80 language, as implemented by the PL/M-80 Compiler. Throughout this manual, the name "PL/M" refers specifically to this implementation.

For information on the use of the PL/M-80 Compiler itself, the reader is referred to the *ISIS-II PL/M-80 Compiler Operator's Manual,* Intel document number 98-300. For information on the ISIS-II operating system facilities related to PL/M-80 programming, see *ISIS-II System User's Guide,* Intel document number 98-306.

Readers of the previous version (Rev. A) of this manual should note the following significant changes in this revision:

- Section 8.1.6 — revised to reflect the support of interrupt numbers higher than 7, for systems with the necessary hardware.
- Section 8.1.7 — revised to state that a procedure called by a reentrant procedure should also be reentrant.

HOW TO USE THIS MANUAL

This manual is intended to be read from front to back by a new PL/M programmer. Chapter 1 contains a synopsis intended to give an intuitive feel for the language, and also introduces certain important concepts in preparation for the complete discussions of language features in subsequent chapters.

An index is included at the end of the manual for reference purposes.

Appendix A contains a complete formal syntax description, written in a modified BNF notation. Most readers will not need this formal syntax, but it is included for completeness. Note that the terminology of Appendix A is not exactly the same as the less formal terminology of the main body of the manual.

The remaining appendices are lists of ASCII codes, PL/M special characters, PL/M reserved words, and PL/M predeclared identifiers, included for convenience in using the manual for reference purposes.

CONTENTS

ILLUSTRATIONS

CHAPTER 1
INTRODUCTION

1.1 WHAT IS PL/M?

PL/M is a high-level language designed for system and applications programming for the Intel® 8080 microprocessor.

A PL/M program is a sequence of *PL/M statements*. The PL/M-80 Compiler accepts the statements as input and produces a machine-code program module as output. As will be seen in the remainder of this chapter, a PL/M statement may be translated by the compiler into a single 8080 instruction, or a sequence of instructions, or none at all — it may cause the compiler to allocate storage, for example, instead of producing any machine instructions.

PL/M statements are divided into two basic categories:

1. DECLARE and PROCEDURE statements. DECLARE statements cause computational "objects" (such as variables) to be defined, associate "identifiers" (that is, names) with objects, and allocate memory storage for objects. PROCEDURE statements are described later in this chapter.

2. *Executable statements*, which are all the PL/M statements other than DECLARE and PROCEDURE statements. Most executable statements cause machine code to be generated.

The following is a simple example of a DECLARE statement:

```
DECLARE WIDTH BYTE;
```

This statement introduces an *identifier*, WIDTH, and associates it with the contents of one byte of memory. The programmer need not know the location of the byte — he will henceforth refer to the contents of this byte by using the identifier, WIDTH.

Notice the semicolon at the end of the statement. Every PL/M statement is terminated with a semicolon.

The following is a simple executable statement:

```
CLEARANCE = WIDTH + 2;
```

Here we have an identifier, CLEARANCE, and another identifier, WIDTH. Both must be declared previous to this executable statement. This executable statement is called an *assignment statement*, and it produces machine code to do the following:

- Retrieve the value associated with the identifier WIDTH from memory.
- Add 2 to this value.
- Store the sum into a memory location associated with the identifier CLEARANCE.

But the PL/M programmer need not think in terms of memory locations. CLEARANCE and WIDTH are *variables,* and the assignment statement assigns the value of the *expression* WIDTH + 2 to the variable CLEARANCE. The compiler automatically generates all the machine code necessary to retrieve data from memory, evaluate the expression, and store the result in the proper location.

PL/M also provides facilities for declaring and calling *procedures.* A *procedure declaration* is a block of PL/M code that is not executed at the point where it is written, but may be "activated" from other points in the program. A reference to the procedure causes the procedure to be activated. The activation may include passing parameters to the procedure and passing a value back from the procedure. When a procedure is finished executing, control is returned to the point from which it was activated.

This procedure capability permits programs to be constructed in a modular fashion, which has numerous advantages including efficiency of coding, readability of programs, ease of debugging, and the possibility of using the same procedure in more than one program.

In the following overview, these and other features of PL/M are examined in greater detail.

1.2 OVERVIEW OF THE LANGUAGE

The following sections are capsule descriptions of some of the important features of PL/M.

1.2.1 IDENTIFIERS

In the examples above, we saw the characters CLEARANCE and WIDTH used as identifiers — that is, names for objects. CLEARANCE and WIDTH were used as identifiers for variables. However, PL/M identifiers can be associated with a wide range of different kinds of objects — variables, collections of variables, procedures, PL/M statements, and macros. PL/M identifiers are chosen by the programmer and are not restricted to alphabetic characters as in the examples above.

The following are examples of PL/M identifiers:

ABC

X2

PRODUCT

K

Further discussion of identifiers will be found in Chapter 2.

1.2.2 DECLARE STATEMENTS

A simple DECLARE statement has already been shown above. A DECLARE statement is a non-executable statement that introduces some object or collection of objects, associates an identifier or identifiers with them, and allocates storage if necessary. The most important use of DECLARE is for declaring variables.

A variable may be a *scalar* — that is, a single quantity — or an *array,* or a *structure.*

An array is a collection of scalars which are all associated with the same identifier and differentiated from each other by the use of *subscripts*.

A structure is a collection of scalars and/or arrays all associated with the same identifier and differentiated from each other by their own *member-identifiers*.

The following statements declare scalars:

```
DECLARE APPROX ADDRESS;
DECLARE (OLD, NEW) BYTE;
DECLARE POINT ADDRESS, VAL BYTE;
```

A scalar always has a *type*, either BYTE or ADDRESS. A BYTE scalar is an 8-bit quantity occupying one byte of memory. An ADDRESS scalar is a 16-bit quantity occupying two contiguous bytes of memory. Values of scalars are always interpreted as unsigned integers.

The first example above declares a single scalar variable of type ADDRESS, with the identifier APPROX.

The second example declares two scalars, OLD and NEW, both of type BYTE.

The third example declares two scalars of different types — POINT is of type ADDRESS, and VAL is of type BYTE.

The following statements declare arrays:

```
DECLARE DOMAIN (128) BYTE;
DECLARE GAMMA (10) ADDRESS;
```

The first statement declares an array called DOMAIN, with 128 scalar elements each of type BYTE. These elements are distinguished by subscripts ranging from 0 to 127 — for example, the third element of the array can be referred to as DOMAIN(2).

The second statement declares an array called GAMMA, with 10 scalar elements of type ADDRESS. The subscripts for this array range from 0 to 9.

The following statement declares a structure with two scalar members:

```
DECLARE RECORD STRUCTURE (KEY BYTE, INFO ADDRESS);
```

The two members are a BYTE scalar that can be referred to as RECORD.KEY and an ADDRESS scalar that can be referred to as RECORD.INFO.

Further discussion of variables and variable declarations will be found in Chapters 3 and 6.

1.2.3 PROCEDURE DECLARATIONS

A procedure declaration begins with a PROCEDURE statement and ends with a matching END statement. It may be thought of as a "sub-program" which will be executed when called from elsewhere in the program.

PROCEDURE Statements

The following is an example of a PROCEDURE statement:

```
SUMSQUARE: PROCEDURE (A, B) ADDRESS;
```

This statement introduces a complete procedure declaration, which will be shown in the next section. The *name* of the procedure is SUMSQUARE. This name is used for calling the procedure.

A and B are identifiers for *formal parameters*. They will appear again as variables in the procedure body. Specifying them in the PROCEDURE statement indicates that we will supply values for them when the procedure is called. Not all procedures have parameters; they are left out of the PROCEDURE statement if not needed.

This is a *typed* procedure, with type ADDRESS. The appearance of a type in the PROCEDURE statement means that the procedure will return a value to the point from which it is called — in this case, a 16-bit (ADDRESS) value. (The meaning of this is explained below.)

Procedure Declaration Blocks

Using the same PROCEDURE statement given above, we can construct the complete declaration of the procedure (known as a procedure declaration block or simply a procedure declaration):

```
SUMSQUARE: PROCEDURE (A, B) ADDRESS;
    DECLARE (A, B) ADDRESS;
    RETURN A*A + B*B;
END SUMSQUARE;
```

A and B are declared to be scalar variables of type ADDRESS. The RETURN statement contains an expression in which A and B are both squared (note the use of the * as a multiplication operator), and the squares are added. The effect of the RETURN statement is to cause this value to be returned to the point of call.

A typed procedure, such as SUMSQUARE, is called by referring to it as an operand in an expression. Suppose that having written the procedure declaration above, we now write an assignment statement like the following:

```
NEWVAL = OLDVAL - SUMSQUARE(PREV, NEXT);
```

where NEWVAL, OLDVAL, PREV, and NEXT are all previously declared variables. The text SUMSQUARE(PREV, NEXT) is a *procedure call* to the procedure SUMSQUARE, with *actual parameters* PREV and NEXT.

The values of PREV and NEXT are passed to the procedure SUMSQUARE as parameters. SUMSQUARE takes the sum of their squares and returns this value. *The returned value replaces the procedure call,* and the expression in the assignment statement can now be evaluated.

For example, suppose that when the above assignment statement is executed, OLDVAL has a value of 100, PREV has a value of 4, and NEXT has a value of 5. Then SUMSQUARE returns a value of 16 + 25, or 41. This is subtracted from the value of OLDVAL and the result, 59, is assigned to the variable NEWVAL.

Not all procedures return values. A procedure that has no type in its PROCEDURE statement does not return a value, and is called an untyped procedure. An untyped procedure is activated by means of a CALL statement.

A complete discussion of procedures will be found in Chapter 8.

1.2.4 EXECUTABLE STATEMENTS

The following is a list of all PL/M executable statements and the numbers of the sections in which they are fully discussed:

Assignment Statement	(Section 4.6)
GOTO Statement	(Section 5.3)
IF Statement	(Section 5.2)
Simple DO Statement	(Section 5.1.1)
Iterative DO Statement	(Section 5.1.4)
DO WHILE Statement	(Section 5.1.3)
DO CASE Statement	(Section 5.1.5)
END Statement	(Section 5.1)
CALL Statement	(Section 8.2)
RETURN Statement	(Section 8.1.3)
HALT Statement	(Section 5.4)
ENABLE Statement	(Section 8.1.6)
DISABLE Statement	(Section 8.1.6)
Null Statement	(Section 5.1.5)

The following sections give simplified descriptions of some of the executable statements, in order to provide a feeling for PL/M before going on to the full descriptions in later chapters.

Assignment Statement

The assignment statement has already been introduced. It is fundamental to PL/M programming, and although its form is quite simple, the expression in an assignment statement may be quite complex and result in a considerable amount of computation, as will be seen in Chapter 4.

IF Statement

The following is an example of an IF statement:

```
IF WEIGHT > MINWT
    THEN COUNT = COUNT + 1;
    ELSE COUNT = 0;
```

Notice how this has been broken into three lines, with indentation, to make it more readable. As explained in Chapter 2, blanks (spaces, tabs, carriage returns, and line feeds) may be freely inserted between the elements of a statement without changing the meaning.

WEIGHT, MINWT, and COUNT are assumed to be previously declared scalar variables. This IF statement has three parts:

- An "IF part" consisting of the reserved word IF and a condition, WEIGHT > MINWT
- A "THEN part" consisting of the reserved word THEN and a statement, COUNT = COUNT + 1;
- An "ELSE part" consisting of the reserved word ELSE and another statement, COUNT = 0;.

The meaning of the IF statement is that if the condition in the IF part is "true," then the statement in the THEN part will be executed. Otherwise, the statement in the ELSE part will be executed.

In this particular case, if the value of WEIGHT is greater than the value of MINWT, then the value of COUNT will be incremented by 1. Otherwise, the value 0 will be assigned to COUNT.

The ELSE part of an IF statement may be omitted. Chapter 5 contains a full description of IF statements.

DO and END Statements

DO and END statements are used to construct "DO blocks." A DO block begins with a DO statement and ends with a matching END statement (just as a procedure declaration block begins with a PROCEDURE statement and ends with a matching END statement).

There are four kinds of DO statements, used to construct four kinds of DO blocks.

A *simple DO block* begins with a *simple DO statement* and has the property (like all blocks) that it may be used wherever a single statement can be used. The following is an example of a simple DO block used in place of a single statement in the THEN part of an IF statement:

```
IF TMP >= 4 THEN
DO;
    INCR = INCR*2;
    COUNT = COUNT + INCR;
END;
ELSE COUNT = 0;
```

This allows two assignment statements to be executed if the condition is "true."

An *iterative DO statement* introduces an *iterative DO block*, and causes the executable statements within the block to be executed repeatedly. The following is an example:

```
DO J = 0 TO 9;
    VECTOR(J) = 0;
END;
```

where J is a previously declared scalar variable and VECTOR is a previously declared array having at least 10 elements. The assignment statement is executed 10 times, with values of J starting at 0 and increasing by 1 each time around until all of the integers from 0 through 9 have been used. Since J is used as a subscript for specifying which element of VECTOR is referenced in the assignment statement, the effect of this iterative DO block is to assign the value 0 to all elements of VECTOR from element 0 through element 9.

The *DO WHILE statement* contains a condition (like the condition in the IF part of an IF statement) and causes the executable statements in a *DO WHILE block* to be executed repeatedly as long as the condition is "true."

In the following example a DO WHILE block is used to step through the elements of an array called TABLE, until an element is found that is not greater than the value of a scalar variable called LEVEL.

```
I = 0;
DO WHILE TABLE(I) > LEVEL;
    I = I + 1;
END;
```

Here TABLE is a previously declared array and LEVEL and I are previously declared scalars. I is first assigned a value of 0, and then used as a subscript for TABLE. It is incremented each time through the DO WHILE block, so each time the DO WHILE statement is executed, a different element of TABLE is compared with LEVEL. When an element is found that is not greater than LEVEL, the condition in the DO WHILE statement is no longer true and the block is not repeated again — control passes to the next statement after the END statement. At this point the value of I is still the subscript of the first element of TABLE that is not greater than LEVEL.

Finally, there is the *DO CASE block*, introduced by a *DO CASE statement*, which allows the value of some expression to be used to select a statement to be executed. In the following example, assume that the expression TST – 1 in the DO CASE statement can have any value from 0 to 3:

```
DO CASE TST - 1;
    RED = 0;
    BLUE = 0;
    GREEN = 0;
    GREY = 0;
END;
```

If the value of the expression is 0, then only the first assignment statement will be executed, and the value 0 will be assigned to RED. If the value of the expression is 1, then only the second assignment statement will be executed, and the value 0 will be assigned to BLUE. Expression values of 2 or 3 will cause GREEN or GREY, respectively, to be assigned the value 0.

DO statements and DO blocks are considered flow control statements and are discussed in Chapter 5.

Statement Labels

An executable statement may be labeled by prefixing it with an identifier and a colon, as in the following example:

```
SET: SUM = 0;
```

The identifier SET is the label of the assignment statement. A statement may also have more than one label. Labels are useful for readability, and in connection with GOTO statements. Labels are discussed in Section 5.3.

1.2.5 BLOCK STRUCTURE AND SCOPE

Block Structure

We have already noted five kinds of blocks: the procedure declaration block, and the four kinds of DO blocks. A PL/M program consists entirely of one or more blocks. (The compiler accepts as its input file one "module," and a module is simply a labeled simple DO block that is not nested inside any other block.)

Blocks may be nested within each other. This leads to the concept of "levels" within a program — the outermost level is that of the module, and the contents of a block nested within the module are at an "inner" level.

This structure makes it possible to have rules of *scope* for declared objects.

Scope

The concept of scope is important in PL/M. The scope of an object (such as a variable or procedure) is simply the part of the program in which its identifier is recognized and the object handled according to its declaration.

In simplified form, the rules of scope are as follows:

- The scope of an object does not extend beyond the block in which it is declared.
- The scope of an object does not include any block that is nested inside the block where the object is declared, if the nested block contains a new declaration using the same identifier.

The complete rules of scope involve some slight modifications of these statements, as explained in Chapter 9. However, the above rules will suffice for understanding Chapters 2 through 8.

The effect of these rules is that when writing a block, and declaring objects solely for use inside the block, one does not need to worry about whether the same identifier has already been used in another block. Even if the same identifier is used elsewhere, it refers to a different object.

1.2.6 EXPRESSIONS

We have already seen simple expressions. A PL/M expression is made up of operands and operators, and resembles a conventional algebraic expression.

Operands include numeric constants (such as 3 or 105) and variables (as well as other types discussed in Chapter 4). The operators include + and – for addition and subtraction, * and / for multiplication and division, and MOD for modulo arithmetic.

As in an algebraic expression, elements of a PL/M expression may be grouped with parentheses.

Expressions are evaluated using unsigned integer arithmetic — that is, the value of an expression is always a positive integer (either type BYTE or type ADDRESS).

1.2.7 MODULAR STRUCTURE OF PL/M PROGRAMS

The definition of a PL/M program is that it consists of one or more modules, one of which must be a "main program module." A main program module is a module that contains executable statements at its outer level. Other modules contain nothing but DECLARE and PROCEDURE statements at their outer levels.

The modules of a program (if there are more than one) are written and compiled separately and combined into a program by use of the linking facility of ISIS-II.

1.2.8 INPUT AND OUTPUT

PL/M does not provide I/O functions in the usual sense of the term. In particular, PL/M I/O does not resemble FORTRAN I/O.

There are four basic methods for moving data to or from memory under PL/M program control.

The INPUT and OUTPUT Facilities

The built-in procedure INPUT and the built-in variable array OUTPUT are described in detail in Chapter 11. INPUT causes the program to read the 8-bit quantity latched in one of the 256 input ports of the 8080. A reference to OUTPUT causes the program to latch an 8-bit quantity into one of the 256 output ports of the 8080.

ISIS-II Facilities

In a system running under ISIS-II, the program can make use of facilities described in the *ISIS-II System User's Guide*, Intel® Document number 98-306.

Direct Memory Access (DMA) Techniques

A peripheral device that has direct access to 8080 memory is called a DMA device. The program can use INPUT and OUPUT facilities to communicate with such a device (if the system is appropriately configured) and cause it to perform data input and output functions.

Memory-Mapped I/O Techniques

In memory-mapped I/O, the program executes ordinary read and store operations using addresses that do not correspond to any actual locations in memory. A peripheral device connected to the CPU recognizes these addresses when they appear on the address bus, and either accepts data for output (if the CPU operation is a store) or supplies data for input (if the CPU operation is a read).

In effect, the peripheral device appears to the CPU as if it were part of memory (although timing is, of course, different from the timing for an actual memory access).

Caution: The PL/M-80 Compiler optimizes the machine code that it produces. This optimization may interfere with the operation of memory-mapped I/O. If a variable is located in memory by means of the AT attribute (see Section 6.2.8), accesses to that variable are not optimized.

1.3 NOTATIONAL CONVENTIONS IN THIS MANUAL

Throughout this manual, certain conventions are used to represent the syntactic form of PL/M statements.

Words in capital letters are reserved words in the PL/M vocabulary, and are to be entered as shown. Semicolons are to be entered as shown.

The items in lower case are to be replaced by actual PL/M code.

Square brackets [ ] around an item indicate that it is optional — that is, the statement is syntactically correct if the item is omitted.

For example, in Section 5.1 the form of the END statement is given as follows:

```
END [label] ;
```

This means that an END statement consists of the following parts:

- The reserved word END
- A label, which may be omitted as shown by the square brackets
- A semicolon.

Note that this is not all the important information about the END statement — it is only the syntax. There is, for example, an important restriction on the label (if any) in an END statement. Such information about PL/M statements is given in the text.

Note also that these notational conventions apply only to the way that the *forms* of PL/M statements are shown. When an example of an *actual* PL/M statement is given, capital letters are used for all of the code and lower-case is used only for comments (see Section 2.4).

In examples it is sometimes necessary to indicate that part of a sequence of statements has been omitted. Three periods (...) are used for this purpose.

CHAPTER 2
BASIC CONSTITUENTS OF A PL/M PROGRAM

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

2.1 PL/M CHARACTER SET

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

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

along with the special characters

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

and the blank characters

space tab carriage-return line-feed

If a PL/M program contains any character that is not in this set, the compiler may treat the character as an error.

Upper- and lower-case letters are not distinguished from each other (except in string constants — see Section 3.2). For example, xyz and XYz are interchangeable. In this manual, all PL/M code is in upper-case letters to help distinguish it visually from explanatory text.

Blanks are not distinguished from each other. Any blank is considered to be the same as any other blank. Moreover, any unbroken sequence of blanks is considered to be the same as a single blank.

Special characters and combinations of special characters have particular meanings in a PL/M program, as described in the remainder of this manual. Appendix C is a concise list of special characters and their meanings.

The above applies to everything in a PL/M program except character string constants (see Section 3.2) and comments (see Section 2.4).

2.2 IDENTIFIERS AND RESERVED WORDS

Identifiers are used to name variables, procedures, macros, and statement labels. An identifier may be up to 31 characters in length. The first character must be alphabetic, and the remainder may be either alphabetic or numeric.

Embedded dollar signs are totally ignored by the compiler, and may be used to improve the readability of an identifier. An identifier containing a dollar sign is exactly equivalent to the same identifier with the dollar sign deleted.

Examples of valid identifiers are

```
X
GAMMA
LONGIDENTIFIERWITHNUMBER3
INPUT$COUNT
INPUTCOUNT
```

The last two examples are interchangeable.

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

2.3 TOKENS, SEPARATORS, AND THE USE OF BLANKS

Just as an English sentence is made up of words — the smallest meaningful units of English — so a PL/M statement is made up of *tokens*. Every token belongs to one of the following classes:

- Identifiers.
- Reserved words.
- Simple delimiters — all of the special characters, except the dollar sign, are simple delimiters.
- Compound delimiters — these are certain combinations of two special characters, namely

  `<>  <=  >=  :=  /*  */`

- Numeric constants (see Chapter 3).
- Character string constants (see Chapter 3).

For the most part, it is obvious where one token ends and the next one begins. For example, in the assignment statement

```
EXACT=APPROX*(OFFSET-3)/SCALE;
```

EXACT, APPROX, OFFSET, and SCALE are identifiers, 3 is a numeric constant, and all the other characters are simple delimiters.

In some cases, identifiers, reserved words, and numeric constants must be separated from each other. If a simple or compound delimiter does not occur between two identifiers, reserved words, or numeric constants, a blank must be placed between them as a separator. (Instead of a single blank, any unbroken sequence of blank characters may be used.)

Also, a comment (see below) may be used as a separator.

Blanks may also be inserted freely around any token, without changing the meaning of the PL/M statement. Thus the assignment statement

```
EXACT = APPROX * ( OFFSET - 3 ) / SCALE ;
```

is equivalent to

```
EXACT=APPROX*(OFFSET-3)/SCALE;
```

2.4 COMMENTS

Explanatory *comments* should be interleaved with PL/M program text, to improve readability and provide program documentation. A PL/M comment is a sequence of characters delimited on the left by the character pair /* and on the right by the character pair */. These delimiters instruct the compiler to ignore any text between them, and not to consider such text part of the program proper.

A comment may contain any printable ASCII character and may also include space, carriage-return, line-feed, and tab characters.

A comment may not be embedded inside a character string constant (see Chapter 3). Apart from this, it may appear anywhere that a blank character may appear — that is, anywhere except embedded within a token. Thus comments may be freely distributed throughout a PL/M program.

Here is a sample PL/M comment:

```
/* This procedure copies one structure to another. */
```

In this manual, comments are presented in upper and lower case letters, to help distinguish them visually from program code, which is always presented in upper case.

CHAPTER 3
PL/M DATA ELEMENTS AND INTRODUCTION TO DECLARATIONS

PL/M data elements can be either variables or constants. Variables are data objects whose values may change during execution of the program and are referred to by identifiers. Constants have fixed values and are referred to directly. The expression

```
APPROX * (OFFSET – 3) / SCALE
```

involves the variables APPROX, OFFSET, and SCALE, and the constant 3.

3.1 NUMERIC CONSTANTS

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

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

B O Q D H

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

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

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

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

Numeric constants may range in value from 0 to 65535, or 0FFFFH, or 177777Q, or 1111$1111$1111$1111B. The upper limit is the largest number that can be represented in two bytes (16 bits).

3.2 CHARACTER STRING CONSTANTS

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

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

(see Appendix B for ASCII character codes). Character strings longer than 2 characters cannot, of course, be used as numeric values, since numeric values are limited to 16 bits. However, longer character strings can be used in certain ways (see Sections 4.1.3 and 6.2.9).

The maximum length of a string constant is limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

3.3 SCALAR VARIABLES AND INTRODUCTION TO DECLARATIONS

A scalar variable is an object whose value is not necessarily known at compile time and may change during the execution of the program. It is therefore referred to by means of an identifier. In this manual, the term "scalar variable" or simply "scalar" always means a variable having a single numeric value.

The term "variable" has a more general meaning: a variable may be a scalar variable, or it may be a set of scalars referred to by a single identifier. Such variables ("arrays" and "structures") are introduced in Sections 3.4 and 3.5.

Each variable used in a PL/M program must be declared in a DECLARE statement before it can be referred to. This declaration defines the variable by introducing the identifier and giving necessary information about it. In the simplest type of DECLARE statement, the only information provided is a "type." (Only simple declarations are described in this chapter. Chapter 6 describes DECLARE statements in complete detail.)

3.3.1 TYPES

A scalar variable takes one of two "types": type BYTE, or type ADDRESS. A BYTE variable is an 8-bit value occupying a single byte of storage. An ADDRESS variable is a 16-bit value occupying two bytes of storage. The type of every variable must be formally declared in its DECLARE statement.

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

```
DECLARE UNKNOWN BYTE;
DECLARE POINTER ADDRESS;
DECLARE (WIDTH, LENGTH, HEIGHT) ADDRESS;
```

The first of these DECLARE statements introduces the identifier UNKNOWN and states that it refers to a BYTE value. The compiler, when it processes this declaration, will allocate one byte of storage for this scalar variable.

The second DECLARE statement introduces the identifier POINTER and states that it refers to an ADDRESS value. The compiler, when it processes this declaration, will allocate two bytes of storage for this scalar variable.

The statement

```
DECLARE (WIDTH, LENGTH, HEIGHT) ADDRESS;
```

is equivalent to the following sequence:

```
DECLARE WIDTH ADDRESS;
DECLARE LENGTH ADDRESS;
DECLARE HEIGHT ADDRESS;
```

(except that contiguous storage is guaranteed for variables declared in a single parenthesized list, while variables declared in consecutive declarations may not be stored contiguously).

The three identifiers WIDTH, LENGTH, and HEIGHT are introduced and stated to refer to three distinct scalars of type ADDRESS — that is, 16-bit, 2-byte values. Two contiguous bytes of storage are allocated for each of these scalars.

3.4 ARRAYS

It is often desirable to use a single identifier to refer to a whole group of scalars, and distinguish the individual scalars from one another by means of a subscript. Such a group is called an array.

3.4.1 ARRAY DECLARATIONS

An array is declared by using a "dimension specifier." The dimension specifier is a numeric constant enclosed in parentheses. The value of the constant specifies the number of array elements (individual scalar variables) making up the array. For example,

```
DECLARE ITEMS (100) BYTE;
```

causes the identifier ITEMS to be associated with 100 array elements, each of type BYTE. One byte of storage is allocated for each of these scalars.

The declaration

```
DECLARE (WIDTH, LENGTH, HEIGHT) (100) ADDRESS;
```

is equivalent to the following sequence:

```
DECLARE WIDTH (100) ADDRESS;
DECLARE LENGTH (100) ADDRESS;
DECLARE HEIGHT (100) ADDRESS;
```

(except that contiguous storage is guaranteed for variables declared in a single parenthesized list, while variables declared in consecutive declarations may not be stored contiguously).

This causes the 3 identifiers WIDTH, LENGTH, and HEIGHT each to be associated with 100 array elements of type ADDRESS, so that 300 elements of type ADDRESS have been declared in all. For each of these scalars, two contiguous bytes of storage are allocated.

3.4.2 SUBSCRIPTED VARIABLES

To refer to a single element of an array (previously declared), one uses the array identifier followed by a subscript enclosed in parentheses. This is called a "subscripted variable."

For example, the DECLARE statement

```
DECLARE ITEMS(100) BYTE;
```

actually declares 100 scalars of type BYTE, which can be referred to as ITEMS(0), ITEMS(1), ITEMS(2), and so on up to ITEMS(99).

Notice that the first element of an array has subscript 0 — *not* 1.

If we want to add the third element of the array ITEMS to the fourth, and store the result in the fifth, we can write the PL/M assignment statement

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

Much of the power of a subscripted variable lies in the fact that the subscript need not be a numeric constant, but can be another variable, or in fact any PL/M expression. Thus the construction

```
VECTOR (ITEMS(3) + 2)
```

refers to some element of the array VECTOR; which element depends on the expression ITEMS(3) + 2, and this in turn depends on the value stored in ITEMS(3), the fourth element of array ITEMS, at the time when the reference is processed by the running program.

If ITEMS(3) contains the value 5, then ITEMS(3) + 2 is equal to 7 and the reference is to VECTOR(7), the eighth element of the array VECTOR.

The following sequence of statements will sum the elements of the 10-element array NUMBERS by using an "index variable," I, which takes on values from 0 to 9:

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

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

Subscripted variables are permitted anywhere PL/M permits an expression.

3.5 STRUCTURES

Just as an array allows one identifier to refer to a collection of elements of the same type, a *structure* allows one identifier to refer to a collection of *structure members* which may have different types. Each member of a structure has a *member-identifier.*

The following is an example of a structure declaration:

```
DECLARE AIRPLANE STRUCTURE (SPEED BYTE, ALTITUDE ADDRESS);
```

This declares two scalars — a BYTE scalar and an ADDRESS scalar — both associated with the identifier AIRPLANE. A byte of storage is allocated for the BYTE scalar, and two bytes for the ADDRESS scalar. Once this declaration has been made, the first scalar can be referred to as AIRPLANE.SPEED and the second can be referred to as AIRPLANE.ALTITUDE. These are the two members of this structure.

A structure may have multiple members (see *ISIS-II PL/M-80 Compiler Operator's Manual* for limits on the number of members allowed).

3.5.1 ARRAYS OF STRUCTURES

We have already seen arrays of BYTE scalars and arrays of ADDRESS scalars. PL/M also allows arrays of structures. The following DECLARE statement declares an array of structures which can be used to store SPEED and ALTITUDE (as in the previous example) for twenty AIRPLANEs instead of one:

```
DECLARE AIRPLANE (20) STRUCTURE (SPEED BYTE, ALTITUDE ADDRESS);
```

This declares twenty structures associated with the array identifier AIRPLANE, distinguished by subscripts from 0 to 19. Each of these structures consists of two members: a BYTE scalar and an ADDRESS scalar. Thus storage is allocated for 20 BYTE scalars and 20 ADDRESS scalars.

To refer to the ALTITUDE of AIRPLANE number 17, one would write AIRPLANE(17).ALTITUDE.

3.5.2 ARRAYS WITHIN STRUCTURES

An array may be used as a member of a structure, as in the following DECLARE statement:

```
DECLARE PAYCHECK STRUCTURE (
    LAST$NAME (15) BYTE,
    FIRST$NAME (15) BYTE,
    MI BYTE,
    DOLLARS ADDRESS,
    CENTS ADDRESS);
```

This structure consists of the following members: two 15-element BYTE arrays, PAYCHECK.LAST$NAME and PAYCHECK.FIRST$NAME; the BYTE scalar PAYCHECK.MI; and the two ADDRESS scalars PAYCHECK.DOLLARS and PAYCHECK.CENTS.

To refer to the fourth element of the array PAYROLL.LASTNAME, we would write PAYROLL.LASTNAME(3).

3.5.3 ARRAYS OF STRUCTURES WITH ARRAYS INSIDE THE STRUCTURES

We have just seen that an array can be made up of structures, and a structure can have arrays as members. By combining these two constructions, we can write a DECLARE statement like the following:

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

The identifier X refers to an array of 100 structures, each of which contains one array of 100 BYTE scalars. This could be thought of as a 100-by-100 matrix of BYTE scalars. To reference a particular scalar value — say element 46 of structure 35 — we would write X(35).Y(46).

We can alter the PAYCHECK structure declaration above to make it an array of structures, as follows:

```
DECLARE PAYROLL (100) STRUCTURE (
    LAST$NAME (15) BYTE,
    FIRST$NAME (15) BYTE,
    MI BYTE,
    DOLLARS ADDRESS,
    CENTS ADDRESS);
```

Now we have an array of 100 structures, each of which can be used during program execution to store the last name, first name, middle initial, dollars, and cents for one employee. LAST$NAME and FIRST$NAME in each structure are 15-BYTE arrays for storing the names as character strings. To refer to the Kth character of the first name of the Nth employee, we would write PAYROLL(N).FIRST$NAME(K), where N and K are previously declared variables to which we have assigned appropriate values. This might be convenient in a routine for printing out payroll information.

With this much complexity — an array of structures with arrays inside the structures — we have reached a limit, since PL/M does not allow structures within structures.

3.6 REFERENCE TO VARIABLES

In the preceding sections, we have seen numerous examples of variable references. A variable reference is simply the use, in program text, of the identifier of a variable that has been declared.

A variable reference may be "fully qualified," "partially qualified," or "unqualified."

3.6.1 FULLY QUALIFIED VARIABLE REFERENCES

A fully qualified variable reference is one that uniquely specifies a single BYTE or ADDRESS scalar. For example, if we have the declarations

```
DECLARE AVERAGE ADDRESS;
DECLARE ITEMS (100) BYTE;
DECLARE RECORD STRUCTURE (KEY BYTE, INFO ADDRESS);
DECLARE NODE (25) STRUCTURE (SUBLIST (100) BYTE, RANK BYTE);
```

then AVERAGE, ITEMS (5), RECORD.INFO, and NODE(21).SUBLIST(32) are all fully qualified variable references: each refers unambiguously to a single scalar.

It should be noted that qualification may only be applied to variables that have been appropriately declared. A subscript may only be applied to an identifier that has been declared with a dimension specifier. A member-identifier may only be applied to an identifier declared as a structure identifier.

3.6.2 UNQUALIFIED AND PARTIALLY QUALIFIED VARIABLE REFERENCES

Unqualified and partially qualified variable references are allowed only in "location references" (see Section 4.1.3) and in the built-in procedures LENGTH, LAST, and SIZE (see Section 11.1.2).

An unqualified variable reference is the identifier of a structure or array, without any member-identifier or subscript. For example, with the above declarations, ITEMS and RECORD are unqualified variable references. An unqualified variable reference is a reference to the *entire* array or structure.

A partially qualified variable reference is the use of an identifier with a subscript and/or member-identifier, if the reference does not uniquely refer to a single BYTE or ADDRESS value. For example, NODE(15) and NODE(12).SUBLIST are partially qualified variable references, given the above declarations. Note that NODE.SUBLIST is not permitted: in referring to an array made up of structures, a subscript must be given before a member-identifier can be added to the reference.

3.6.3 POINTERS AND INDIRECT REFERENCES: BASED VARIABLES

Sometimes a direct reference to a PL/M data element is either impossible or inconvenient. This happens, for example, when the location of a data element must remain unknown until it is computed at run-time. In such cases it may be necessary to write PL/M code to manipulate the addresses of data elements rather than the data elements themselves, considering that the addresses "point to" the data.

To permit this type of manipulation, PL/M uses "based variables." A based variable is a variable which is pointed to by another variable, called its "base." A based variable is not allocated storage by the compiler. At different times during the program run it may actually be in different places in memory, since its base may be changed by the program. A based variable is declared by first declaring its base, which must be of type ADDRESS, and then declaring the based variable itself:

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

Given these declarations, a reference to ITEM is, in effect, a reference to whatever BYTE value is pointed to by the current value of ITEM$POINTER. This means that the sequence

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

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

A variable is made BASED by inserting in its declaration the word BASED and the identifier of the base (which must already have been declared).

The following restrictions apply to bases:

- The base must be of type ADDRESS.
- The base may not be subscripted — that is, it may not be an array element.
- The base may not itself be a based variable.

The word BASED must *immediately* follow the name of the based variable in its declaration, as in the following examples:

```
DECLARE (AGE$PTR, INCOME$PTR, RATING$PTR, CATEGORY$PTR) ADDRESS;
DECLARE AGE BASED AGE$PTR BYTE;
DECLARE (INCOME BASED INCOME$PTR, RATING BASED RATING$PTR)
      ADDRESS;
DECLARE (CATEGORY BASED CATEGORY$PTR) (100) ADDRESS;
```

In the first DECLARE statement, the ADDRESS variables AGE$PTR, INCOME$PTR, RATING$PTR, and CATEGORY$PTR are declared. They are used as base variables in the next three DECLARE statements.

In the second DECLARE statement, a BYTE variable called AGE is declared. The declaration implies that whenever AGE is referenced by the running program, its value will be found at the location given by the value of the ADDRESS variable AGE$PTR at that same time.

The third DECLARE statement declares two based variables, both of type ADDRESS.

The fourth DECLARE statement defines a 100-element ADDRESS array called CATEGORY, based at CATEGORY$PTR. This means that when any element of CATEGORY is referenced at run time, the value of CATEGORY$PTR at that same time is the address of the first element of CATEGORY. The other elements follow contiguously. The parentheses around the tokens CATEGORY BASED CATEGORY$PTR are optional. They help to make the statement more readable, and may be omitted.

For example, CATEGORY(5) is a reference to the sixth element of array CATEGORY. The address of CATEGORY(5) will be the value of CATEGORY$PTR (at the time of the reference) plus 10 (since 10 bytes are required for 5 ADDRESS elements).

So far, the examples have only shown numeric constants being assigned to bases. As will be seen in Section 4.1.3, a PL/M construct called the "dot operator" allows much greater flexibility by

permitting the address of a variable to be assigned to another variable, which is the base for a based variable.

3.7 CONTIGUITY OF STORAGE

PL/M guarantees that variables will be stored in contiguous memory locations in certain situations:

- The elements of an array are stored contiguously, with the 0th element in the lowest address and the last element in the highest address. (No storage is allocated for a based array, but the elements are considered to be contiguous in memory.)

- The members of a structure are stored contiguously, in the order in which they are specified. (No storage is allocated for a based structure, but the members are considered to be contiguous in memory.)

- Non-based variables declared in a "factored" declaration — that is, variables within a parenthesized list — are stored contiguously, in the order specified. (If a based variable occurs in a parenthesized list, it is ignored in allocating storage.)

These are the *only* guarantees.

CHAPTER 4
EXPRESSIONS AND ASSIGNMENTS

A PL/M expression consists of operands (values) combined by means of the various arithmetic, logical, and relational operators. Examples are

```
A + B
A + B - C
A*B + C/D
A*(B + C) - (D - E)/F
```

where +, –, *, and / are operators for addition, subtraction, multiplication, and division, and A, B, C, D, E, and F represent operands. The parentheses serve to group operands and operators, as in ordinary algebra.

4.1 OPERANDS

Operands are the building blocks of expressions. An operand must be something which has a specific value at run time. Thus in the examples above, A, B, C, etc. might be the identifiers of scalar variables which have values at run time.

Numeric constants and fully qualified variable references may appear as operands in expressions. The following sections describe all of the types of operands that are permitted.

4.1.1 CONSTANTS

Any numeric constant may be used as an operand in an expression.

A numeric constant is implicitly of type BYTE if it is not greater than 255. If it is greater than 255, it is implicitly of type ADDRESS.

A string constant containing not more than two characters may also be used as an operand. If it has only one character, it is treated as a BYTE numeric constant whose value is the eight-bit ASCII code for the character. If it is a two-character string, it is treated as an ADDRESS numeric constant whose value is formed by stringing together the ASCII codes for the two characters, with the code for the first character forming the most significant eight bits of the sixteen-bit number.

4.1.2 VARIABLE REFERENCES

As we have seen, a fully qualified variable reference refers unambiguously to a single scalar value. Any fully qualified variable reference may be used as an operand in an expression. When the expression is evaluated, the reference is replaced by the value of the scalar.

In addition to the kinds of variable reference described in Section 3.6, there is another kind called a "function reference."

A function reference is the name of a "typed procedure" that has previously been declared, along with any parameters required by the procedure declaration. The value of a function reference is the value returned by the procedure. For a complete discussion of procedures and function references, see Chapter 8.

4.1.3 LOCATION REFERENCES: THE DOT OPERATOR

A "location reference" is formed by using the dot operator. There are two forms of location reference. The first is

```
.identifier
```

where the identifier is a fully qualified, partially qualified, or unqualified variable reference. The value of this location reference is the location — the actual memory address at run time — of the variable. Thus the value of any location reference is always of type ADDRESS.

If the variable reference is partially qualified or unqualified, it specifies an array or structure. The location is the location of the first element or member of the array or structure. If it is partially qualified, it is the location of the first scalar of the smallest group of scalars that is uniquely identified by the variable reference.

For example, suppose that we have the following declarations:

```
DECLARE RESULT ADDRESS;
DECLARE XNUM (100) BYTE;
DECLARE RECORD STRUCTURE (KEY BYTE, INFO(25) BYTE, HEAD ADDRESS);
DECLARE LIST (128) STRUCTURE (
    KEY BYTE,
    INFO (25) BYTE,
    HEAD ADDRESS);
```

Then .RESULT is the address of the ADDRESS scalar RESULT, while .XNUM(5) is the address of the 6th element of the array XNUM and .XNUM is the address of the beginning of the array — that is, the address of the first element (element 0).

Also, .RECORD.HEAD is the address of the ADDRESS scalar RECORD.HEAD, while .RECORD is the address of the BYTE scalar RECORD.KEY; and .RECORD.INFO is the address of the first element of the 25-BYTE array RECORD.INFO, whereas .RECORD.INFO(7) is the address of the 8th element of the same array.

LIST is an array of structures. The location reference .LIST(5).KEY is the address of the scalar LIST(5).KEY. Note that .LIST.KEY is illegal. The location reference .LIST(0).INFO(6) is the address of the scalar LIST(0).INFO(6). Also, .LIST(0).INFO is the address of the first element of the same array.

A special case exists when the identifier is the name of a procedure. The procedure must be declared at the outer level of the program module (see Chapter 10). No actual parameters may be given (even if the procedure declaration includes formal parameters). The value of the location reference in this case is the address of the entry point of the procedure.

The other form of location reference is

```
.(constant list)
```

where the "constant list" is a sequence of one or more constants separated by commas, and enclosed in parentheses. When this type of location reference is made, space is allocated for the constants, the constants are stored in this space (contiguously, in the order given by the list), and the value of the location reference is the address of the first constant.

Strings may be included in the list. For example, if the operand

```
.('NEXT VALUE')
```

appears in an expression, it causes the string NEXT VALUE to be stored in memory (one character per byte, thus occupying 12 contiguous bytes of storage). The value of the operand is the address of the first of these bytes — in other words, a pointer to the string.

Location References and Based Variables

An important use of location references is to supply values for bases. Thus the dot operator, together with the based variable concept, gives PL/M a complete facility for manipulating pointers.

For example, suppose that we have three different BYTE variables, NORTH$ERROR, EAST$ERROR, and HEIGHT$ERROR. We want to be able to refer to them at different times by means of the single identifier ERROR. This can be done as follows:

```
DECLARE (NORTH$ERROR, EAST$ERROR, HEIGHT$ERROR) BYTE;
DECLARE ERROR$POINTER ADDRESS;
DECLARE ERROR BASED ERROR$POINTER BYTE;

...

ERROR$POINTER = .NORTH$ERROR;
```

At this point, the value of ERROR$POINTER is the address of NORTH$ERROR. A reference to ERROR will be, in effect, a reference to NORTH$ERROR. Later in the program, we can write

```
ERROR$POINTER = .HEIGHT$ERROR;
```

Now a reference to ERROR will be, in effect, a reference to HEIGHT$ERROR. In the same way, we can cause the value of the pointer to be the address of EAST$ERROR, and a reference to ERROR will be a reference to EAST$ERROR.

This kind of technique is useful for manipulating complicated data structures and for passing addresses to procedures as parameters. Examples are given in Chapter 8.

4.1.4 SUBEXPRESSIONS

A subexpression is simply an expression enclosed in parentheses. A subexpression may be used as an operand in an expression. This is the same as saying that parentheses may be used to group portions of an expression together, just as in ordinary algebraic notation.

4.2 ARITHMETIC OPERATORS

There are five principal arithmetic operators in PL/M (two others are described in Chapter 12). The five principal operators are

+ - * / MOD

All of the above operators perform *unsigned integer* (binary) arithmetic, combining two operands. Each operand may have either a BYTE or an ADDRESS value.

4.2.1 THE "+" AND "-" OPERATORS

The operators + and - perform addition and subtraction. If both operands are of the same type, the result is also of that same type. If one operand is of type ADDRESS and the other operand is of type BYTE, the BYTE operand is extended by 8 high-order zero bits, the operation is performed in 16-bit arithmetic, and the result is of type ADDRESS.

See Section 4.5.2 for the case where a larger value is subtracted from a smaller one.

4.2.2 THE UNARY "-" OPERATOR

A *unary* "-" operator is also defined in PL/M. It takes a single operand, to which it is prefixed. In other words, a minus sign that does not have an operand to the left of it is taken to be a unary minus.

Its effect is such that (-A) — where A is any operand — is equivalent to (0-A). Thus -1, for example, is equivalent to 0-1, resulting in the BYTE value 255 or 0FFH as explained below in Section 4.5.2.

The result of a unary "-" is always of the same type as the operand.

4.2.3 THE "*" AND "/" OPERATORS

The operators * and / perform unsigned binary multiplication and division, on operands of type BYTE or ADDRESS. The result is always of type ADDRESS. In the event that arithmetic overflow occurs during multiplication, the result is undefined. The division operator always rounds down to an integer result, and the result of division by zero is undefined.

4.2.4 THE "MOD" OPERATOR

MOD performs similarly to /, except that the result of the operation is not the quotient from the division, but the remainder.

4.3 LOGICAL OPERATORS

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

NOT AND OR XOR

These operators perform logical operations on 8 or 16 bits in parallel.

NOT is a unary operator, taking one operand only. It produces a result in which each bit is the complement of the corresponding bit of its operand.

The remaining operators each take 2 operands, and perform bitwise "and," "or," and "exclusive or" respectively. If both operands are of type BYTE, the operation is an 8-bit operation, and delivers a result of type BYTE. If at least one operand is of type ADDRESS, the operation is a 16-bit operation, and delivers a result of type ADDRESS. In this case, the BYTE operand, if any, is first extended to 16 bits by the addition of 8 high-order zero bits. Examples are

NOT 11001100B	— result is 00110011B
10101010B AND 11001100B	— result is 10001000B
10101010B OR 11001100B	— result is 11101110B
10101010B XOR 11001100B	— result is 01100110B

4.4 RELATIONAL OPERATORS

Relational operators are used to compare operands. They are

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

Relational operators are always binary operators, taking two operands. The operands may be of type BYTE or ADDRESS. The comparison is always performed assuming that the operands are unsigned binary integers. If the specified relation between the operands is "true," a value of 0FFH (or 1111$1111B) results. Otherwise the result is 00H (or 0000$0000B). Thus in all cases the result is of type BYTE, with all 8 bits set to 1 for a "true" condition, or to 0 for a "false" condition. For example:

(6 > 5)	— result is 0FFH ("true")
(6 <= 4)	— result is 00H ("false")

Also, notice that "true" and "false" values can be combined meaningfully by means of logical operators. Thus

NOT(6 > 5)	— result is 00H ("false")
(6 > 5) AND (1 > 2)	— result is 00H ("false")
(6 > 5) OR (1 > 2)	— result is 0FFH ("true")
(LIM = Y) XOR (Z < 2)	— result is 0FFH ("true") if LIM = Y or if Z < 2, but result is 00H ("false") if both relations are "true" or both "false."

Values of "true" and "false" resulting from relational operations are useful in conjunction with DO WHILE statements and IF statements, as will be seen in Chapter 5. (In the context of a DO WHILE statement or IF statement, only the least significant bit of a "true" or "false" value is used.)

4.5 EXPRESSION EVALUATION

4.5.1 PRECEDENCE OF OPERATORS

Operators in PL/M have an implied precedence, which is used to determine the manner in which operators and operands are grouped together. A+B*C causes A to be added to the product of B and C. In this case B is said to be "bound" to the operator * rather than the operator +, as a result of which the multiplication is performed first.

In general, operands are bound to the adjacent operator of highest precedence, or to the left one in the case of a tie.

NOTE

Technically speaking, PL/M does not guarantee the order of evaluation of operands and operations within an expression, but merely defines the association (binding) of operators and operands. This is significant in any case where the order of evaluation of operands can affect the value of the expression, as may occur when embedded assignments (see Section 4.6.3) or function references are used as operands. In such cases, the value of the expression is *undefined*.

Valid PL/M operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence.

```
unary "-"
*  /  MOD
+  -
<  <=  <>  =  >=  >
NOT
AND
OR  XOR
```

The application of the precedence ranking can be seen in the following:

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

Parentheses can be used to override the assumed precedence in the same way as they are used in ordinary algebra. Thus the expression (A + B) * C will cause the sum of A and B to be multiplied by C, instead of adding A to the product of B and C.

4.5.2 UNSIGNED INTEGER ARITHMETIC

When writing a PL/M expression, it is important to bear in mind that all numerical values in PL/M — i.e, numeric constants, values of variables, and values of expressions — are treated as *unsigned binary integers*. Unsigned binary arithmetic is used in evaluating expressions.

This means, in particular, that if a larger value is subtracted from a smaller one, the result is the twos complement of the absolute difference between the two values. For example, 0–1 is equal to 255.

Note that since the relational operators perform unsigned comparisons, if EXP stands for some variable or expression, EXP >= 0 is always "true" for *any* value of EXP.

4.6 ASSIGNMENT STATEMENTS

Results of computations can be stored as values of scalar variables. At any given moment, a scalar variable has only one value — but this value may change with program execution. The PL/M *assignment statement* changes the value of a variable. Its form is

```
variable = expression ;
```

The expression to the right of the equal sign may be any PL/M expression, as described in the preceding sections. This expression is evaluated, and the resulting value is assigned to (that is, stored in) the variable named on the left side of the equal sign. This variable may be any fully qualified variable reference except a function reference. The old value of the variable is lost.

For example, following execution of the statement

```
RESULT = 3;
```

the variable RESULT will have a new value of 3.

4.6.1 TYPE CONVERSIONS

If the type of the value of the expression is not the same as the type of the variable on the left side of an assignment statement, the value of the expression is automatically converted to match the type of the variable.

Conversion from BYTE to ADDRESS is done by appending 8 high-order zero bits to the BYTE value. Conversion from ADDRESS to BYTE is done by dropping the 8 high-order bits of the ADDRESS value.

4.6.2 MULTIPLE ASSIGNMENT

It is often convenient to assign the same value to several variables at the same time. This is accomplished in PL/M by listing all the variables to the left of the equals sign, separated by commas. The variables LEFT, CENTER, and RIGHT can all be set to the value of the expression INIT + CORR with the single assignment statement

```
LEFT, CENTER, RIGHT = INIT + CORR;
```

4.6.3 EMBEDDED ASSIGNMENTS

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

variable := expression

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

ALT + (CORR := TCORR+PCORR) – (ELEV := HT/SCALE)

results in exactly the same value as

ALT + (TCORR+PCORR) – (HT/SCALE)

The only difference is the side-effect of storing the intermediate results TCORR+PCORR and HT/SCALE into CORR and ELEV, respectively. These intermediate results can then be used at a later point in the program without calculating them again.

CHAPTER 5
FLOW CONTROL STATEMENTS

This chapter describes statements that affect the sequence of execution of statements in a PL/M program and the grouping of PL/M statements into blocks.

5.1 DO AND END STATEMENTS: DO BLOCKS

DO and END statements act as brackets to form "DO blocks." There are four different kinds of DO statements, described in the following sections. They are

- The *simple DO statement*
- The *DO WHILE statement*
- The *iterative DO statement*
- The *DO CASE statement*

The END statement has the form

```
END [label] ;
```

where the optional label, if used, must be the label of the DO statement that begins the DO block.

For example, the statement

```
END FIND;
```

could be used to end a block that begins with a DO statement bearing the label FIND.

If the DO statement has more than one label, the label in the END statement must match the last — that is, the rightmost — of these labels. The label in an END statement has no effect on the program. It is allowed as a means of making programs easier to understand and as a debugging aid. The compiler will detect an incorrect label and may thus alert the programmer to a mistake in his program structure.

5.1.1 SIMPLE DO BLOCKS

A simple DO block begins with a simple DO statement and has the form

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

The following is an example:

```
DO;
    NEW$VALUE = OLD$VALUE + TEMP;
    COUNT = COUNT + 1;
END;
```

There are three principal uses of simple DO blocks:

- A simple DO block may be regarded as a single PL/M statement, and may appear anywhere in a program that a single executable statement may appear. This is useful in DO CASE blocks and IF statements, as will be seen in Sections 5.1.5 and 5.2.
- A simple DO block delimits the scope of variables as explained in Chapter 9.
- As explained in Chapter 10, a program module is a simple DO block (with certain other requirements).

Each statement within a simple DO block may be any PL/M statement, including both executable statements and declarations, with the restriction that all declarations within the outer level of the DO block must appear before the first executable statement that occurs at the outer level.

The executable statements (if any) within the DO block are executed in normal sequence just as if they were not enclosed within DO and END statements. (Notice that if any other flow control statements occur within the DO block, they may alter the normal sequence as explained in the following sections.)

DO blocks may be nested within each other as shown in the following:

```
DO;
    statement-1;
    statement-2;
    DO;
        statement-a;
        statement-b;
        statement-c;
    END;
    statement-3;
    statement-4;
END;
```

The first DO statement and the second END statement bracket one simple DO block. The second DO statement and the first END statement bracket a different DO block inside the first one. Notice how indentation (using tabs or spaces) can be used to make the sequence readable, so that it can be seen at a glance that one DO block is nested inside another. It is recommended that this practice be followed in writing PL/M programs.

Nesting is not restricted to simple DO blocks. Any DO block may be nested within any other DO block.

The number of levels to which DO blocks can be nested is limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

5.1.2 "TRUE" AND "FALSE" VALUES

Before describing DO WHILE blocks, it is worth commenting here on the relationship between the logical operators and the DO WHILE statement. These comments also apply to the IF statement (see Section 5.2). We have seen (Section 4.4) that relational operations result in 0FFH for "true" or 00H for "false." Such values may be used to control a DO WHILE statement or IF statement. However, DO WHILE and IF statements examine only the least significant bit of the value of the expression, and the expression need not have a value of 00H or 0FFH. It may have any BYTE or ADDRESS value. If the value is an odd number (least significant bit = 1) it will be considered "true." If it is even (least significant bit = 0) it will be considered "false."

5.1.3 DO WHILE BLOCKS

A DO WHILE block begins with a DO WHILE statement, and has the form

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

The effect of this statement is as follows:

- First the expression following the reserved word WHILE is evaluated. If the result is a quantity whose rightmost bit is 1, then the sequence of statements up to the END is executed.

- When the END is reached, the WHILE expression is evaluated again, and again the sequence of statements is executed only if the value of the expression has a rightmost bit of 1.

- The block is executed over and over until the expression has a value whose rightmost bit is 0, at which time execution of the statements in the block is skipped, and program control passes to the statement following the END statement.

NOTE

The above description assumes that the block does not contain any flow control statements that could cause control to pass out of the block prematurely. For example, a GOTO statement (see Section 5.3.2) could transfer control out of the block without regard to the condition in the DO WHILE statement.

Consider the following example:

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

The statement AMOUNT = AMOUNT+1 is executed exactly 3 times. The value of AMOUNT when program control passes out of the block is 4. (This exemplifies the programming rule that "completion of a DO WHILE causes its condition to become false.")

Like a simple DO block, a DO WHILE block can be considered as a single PL/M statement.

However, unlike a simple DO block, a DO WHILE block may not contain declarations at its outermost level. (It may contain a nested simple DO block which contains declarations.)

5.1.4 ITERATIVE DO BLOCKS

An iterative DO block begins with an iterative DO statement and executes the statements within the block repeatedly as described below.

The *simplest* form of the iterative DO block is

```
DO index = initial-expr TO limit-expr;
    statement-1;
    statement-2;
    ...
    statement-n;
END;
```

where the "index" is a reference to a scalar variable (not subscripted), and "initial-expr" and "limit-expr" are both PL/M expressions.

This block operates as follows:

1. The initial-expr is evaluated and its value is assigned to the index variable. This is done only once, before the first time through the block.

2. The limit-expr is evaluated and its value is compared to that of the index variable. If the value of the index variable is greater than the value of the limit-expr, Steps 3 and 4 below are skipped and control passes to the statement following the END statement.

3. The executable statements in the block are executed.

4. The value of the index variable is incremented by 1. If this causes the new value to be *less* than the old value (because of "wrap-around" due to modulo arithmetic), control passes immediately to the statement following the END statement and the block will not be repeated. If the new value of the index variable is *not* less than the old value, we go back to Step 2 above.

NOTES

The values of both the initial-expr and the limit-expr are converted to the same type as the index variable.

The initial-expr is evaluated only once. The limit-expr is evaluated each time the block is repeated.

Step 4 above provides for stopping the repetition if the value of the index variable is incremented past 255 (for a BYTE index variable) or 65535 (for an ADDRESS index variable).

The above description assumes that the block does not contain any flow control statements that could cause control to pass out of the block prematurely. For example, a GOTO statement (see Section 5.3.2) could transfer control out of the block without regard to the values of the index-variable and the limit-expr.

An example is

```
DO I = 1 TO 10;
    CALL BELL;
END;
```

where BELL is the name of a procedure that causes a bell to be rung. The bell is rung ten times.

Another example shows how the index-variable can be used within the block.

```
AMOUNT = 0;
DO I = 1 TO 10;
    AMOUNT = AMOUNT + I;
END;
```

Both AMOUNT and I are scalar variables. The assignment statement is executed 10 times, each time with a new value for I. The result is to sum the integers from 1 to 10 (inclusive) and leave the sum (namely 55) as the value of AMOUNT.

The more general form of the iterative DO block allows a stepping value other than 1. This more general form is

```
DO index = initial-expr TO limit-expr BY step-expr;
    statement-1;
    statement-2;
    ...
    statement-n;
END;
```

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

```
/*Compute the product of the first N odd integers */

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

NOTES

Since PL/M uses only unsigned arithmetic, there is no such thing as a negative step. For example, if the step-expr is –5, it means the same as a step-expr of 251.

In PL/M, it is not possible to step "downwards" to a limit-expr value that is less than the initial-expr value, because the iterative DO block will *always* terminate if the value of the index-variable is *greater* than the value of the limit-expr.

The value of the step-expr is converted to the same type as the index-variable.

The step-expr is evaluated each time through the block, just after the executable statements in the block are executed.

Like a simple DO block, an iterative DO block can be considered as a single PL/M statement.

However, unlike a simple DO block, an iterative DO block may not contain declarations at its outermost level. (It may contain a nested simple DO block which contains declarations.)

5.1.5 DO CASE BLOCKS

A DO CASE block begins with a DO CASE statement, and selectively executes *one* of the statements in the block. The statement is selected by the value of an expression. The form of the DO CASE block is

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

First, the expression in the DO CASE statement is evaluated. The result of this is a value which must lie between 0 and n–1. (Call this value K.) K is used to select one of the n statements in the DO CASE block, which is then executed. The first case (statement-1) corresponds to K=0, the second case (statement-2) corresponds to K=1, and so forth. *Only one* statement from the block is selected. This statement is then executed *only once*. Control then passes to the statement following the END statement of the DO CASE block.

NOTE

The above description assumes that the block does not contain any flow control statements that could cause control to pass from one case to another. For example, a GOTO statement (see Section 5.3.2) could transfer control from the selected case to another case, or out of the DO CASE block.

If the run-time value of the expression in the DO CASE statement is greater than the number of cases (statements) in the DO CASE block, then the effect of the DO CASE statement is undefined.

An example of a DO CASE block is

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

When execution of this CASE statement begins, the variable SCORE must be in the range 0 - 6. If SCORE is 0, 4, or 5 then a null statement (consisting of only a semicolon, and having no effect) is executed; otherwise the appropriate variable is incremented.

A more complex DO CASE block is the following:

```
DO CASE COUNT-5;

    X = X+1;                              /* Case 0 */

    DO;                                   /* Begin Case 1 */
        X = X+10;
        Y = Y+1;
    END;                                  /* End Case 1 */

    DO I = LAST$HI+1 TO TOP;              /* Begin Case 2 */
        CALL WRITEOUT(.TABLE(I), 1);
    END;                                  /* End Case 2 */

END;                                      /* End DO CASE block */
```

This example illustrates the use of a simple DO block as a single PL/M statement. The DO CASE statement can select Case 1 and cause two statements to be executed. This is only possible because they are grouped as a simple DO block, which acts as a single statement. Also, the iterative DO block of Case 2 appears as a single statement. The CALL statement within the iterative DO block is executed repeatedly.

Like a simple DO block, a DO CASE block can be considered as a single PL/M statement.

However, unlike a simple DO block, a DO CASE block may not contain declarations at its outermost level. (It may contain a nested simple DO block which contains declarations.)

5.2 THE IF STATEMENT

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

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

The reserved word THEN and the statement following it are called the "THEN part," while the reserved word ELSE and the statement following it are the optional "ELSE part."

The IF statement has the following effect: first the expression following the reserved word IF is evaluated. If the result is "true" (see Section 5.1.2) then statement-1 is executed. If the result is "false" then statement-2 is executed. Following execution of the chosen alternative, control passes to the next statement following the IF statement. Thus of the two statements (statement-1 and statement-2) one and only one is executed.

Consider the following program fragment:

```
IF NEW>OLD THEN RESULT=NEW;
ELSE RESULT=OLD;
```

Here RESULT is assigned the value of NEW or the value of OLD, whichever is greater. This code causes exactly one of the two assignment statements to be executed. RESULT always gets assigned some value. However, only one assignment to RESULT is executed.

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

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

Here statement-1 is executed only if the value of the expression has a rightmost bit of 1. Otherwise nothing happens, and control immediately passes on to the next statement following the IF statement.

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

```
INIT = 5;
IF THRESHOLD > INIT THEN INIT = THRESHOLD;
INDEX = INIT;
```

The power of the IF statement is enhanced by using DO blocks in the THEN and ELSE parts. Since a DO block is allowed wherever a single statement is allowed, each of the two statements in an IF statement may be a DO block. For example:

```
IF A=B THEN
    DO;
        EQUAL$EVENTS = EQUAL$EVENTS + 1;
        PAIR$VALUE = A;
        BOTTOM = B;
    END;
ELSE
    DO;
        UNEQUAL$EVENTS = UNEQUAL$EVENTS + 1;
        TOP = A;
        BOTTOM = B;
    END;
```

DO blocks nested within an IF statement can contain further nested DO blocks, IF statements, variable and procedure declarations, and so on.

5.2.1 NESTED IF STATEMENTS

An IF statement (including the ELSE part, if any) may be considered a single PL/M statement (although it is *not* a block). Thus IF statements may be nested within one another.

There is only one restriction when an IF statement is nested in an outer IF statement:

- If an IF statement is nested within the THEN part of an outer IF statement, the outer IF statement may not have an ELSE part.

In other words, the construction

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

would be ambiguous if it were not for this restriction — to which IF statement would the ELSE part belong? Because of the restriction, it is illegal for the ELSE part to belong to the outer IF statement, and so it must belong to the inner one. The construction above is equivalent to the following:

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

and it should be noted that if it is written this way it is much more readable and offers less opportunity for error.

If the intention is for the ELSE part to belong to the outer IF statement, then the nesting *must* be done by means of a DO block:

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

5.3 STATEMENT LABELS AND GOTOs

5.3.1 LABELS AND LABEL DEFINITIONS

PL/M *executable* statements may be labeled for identification and reference (DECLARE and PROCEDURE statements may not be labeled). A labeled statement takes the form

```
label-1: label-2: ... label-n: statement;
```

where each label is a valid PL/M identifier. Multiple labels may precede the statement. See *ISIS-II PL/M-80 Compiler Operator's Manual* for limits on the number of labels allowed.

The appearance of a label in the format shown above — that is, in front of a statement and separated by a colon — is called a "label definition," and it *implicitly* declares the label, exactly as if the label were *explicitly* declared with a label declaration at the beginning of the block.

Here are some examples of labeled statements:

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

The text LOOP: is the definition of the label LOOP, the text L1: is the definition of the label L1, and the text CLEAN$UP: is the definition of the label CLEAN$UP. L1 and CLEAN$UP are labels for the same statement.

5.3.2 GOTO STATEMENTS

A GOTO statement alters the sequential order of program execution by transferring control directly to a labeled statement whose label is referenced in the GOTO statement. Sequential execution then resumes, beginning with the "target" statement. The GOTO statement has the following form:

```
GOTO label ;
```

An example is the following:

```
GOTO ABORT;
```

The appearance of a label in a GOTO statement is *not* a "label definition" — it is a label reference.

The reserved word GOTO can also be written GO TO, with an embedded blank.

Discussion of label scope, which affects the legality of certain GOTOs, is postponed to Section 9.3.

NOTE

In addition, there is an important restriction on the use of a GOTO to exit from a procedure: the label in the GOTO must be the label of a statement in the outermost level of the main program module (see Chapters 9 and 10).

The use of GOTOs is necessary in some situations. However, in most situations where control transfers are desired, the use of iterative DO, DO WHILE, DO CASE, IF, or a procedure call (see Chapter 8) is preferable. Indiscriminate use of GOTOs will result in a program that is difficult to understand, correct, and maintain.

5.4 THE HALT STATEMENT

The HALT statement has the form

```
HALT ;
```

It causes the 8080 to come to a halt with interrupts enabled (see Section 8.1.6).

5.5 THE CALL AND RETURN STATEMENTS

The CALL and RETURN statements are mentioned here only for completeness, since they do control the flow of a program. However, they are not discussed in detail until Chapter 8.

The CALL statement is used to activate an untyped procedure (one that does not return a value).

The RETURN statement is used within a procedure body to cause a return from the procedure to the point from which it was called.

CHAPTER 6
DECLARE STATEMENTS

6.1 GENERAL

As we have seen in Chapter 3, a variable must be declared before it can be referred to by its identifier. This is done by means of a DECLARE statement. Chapter 3 provided some examples of simple DECLARE statements, without describing all the kinds of information that can be included in declarations. This chapter gives complete information on DECLARE statements.

Labels may also be declared in DECLARE statements, although this is usually not necessary, as has been seen in Chapter 5. Label declarations are covered in this chapter, as are macro declarations.

Procedures must also be declared. However, the declaration of procedures is treated as a separate topic in Chapter 8.

6.1.1 PURPOSE OF DECLARATIONS

The purpose of a declaration is to introduce an identifier and define it by giving a list of its properties. Depending on the properties, the identifier then becomes either a label, a macro, or the name of a variable.

The DECLARE statement also causes storage to be allocated for variables, in cases where this is necessary (see Section 3.7 for rules on contiguity of storage).

6.1.2 SCOPE

The *scope* of a declared object is the portion of the program within which it is recognized according to its declaration. The scope depends on the location of the declaration within the program text.

When a DECLARE statement has been made, all occurrences of the declared identifier that are within the scope are recognized and treated according to the information in the DECLARE statement.

As mentioned in Section 1.2.5, PL/M is a block-structured language, and the scope defined by any declaration is limited to the block in which it occurs (unless it has extended scope as described below in Section 6.2.8).

Furthermore, if any sub-block nested within the block contains a declaration which declares the same identifier, then the scope defined by the outer declaration excludes the sub-block.

Scope is discussed in detail in Chapter 9.

6.1.3 WHERE DECLARATIONS MAY OCCUR

Declarations may occur only at the head of a simple DO block (see Chapter 5) or procedure block (see Chapter 8) — that is, between the DO or PROCEDURE statement and the first executable statement in the block.

Once a declaration has been made, it is illegal to make a new declaration using the same identifier at the outer level of the same block.

In addition, declarations containing certain "attributes" and "initializations" may occur only at the outer level of a program module (see Chapter 10).

6.1.4 NOTE ON SYNTAX

The general syntax of DECLARE statements is quite complex. Accordingly, rather than try to start by giving the general syntax, we build up by starting with the simplest cases and proceeding to more complex ones.

Complete formal syntax rules for PL/M can be found in Appendix A.

6.2 VARIABLE DECLARATIONS

6.2.1 BASIC SYNTAX OF A VARIABLE DECLARATION

In simple form, the syntax of a variable declaration is as follows:

```
DECLARE identifier [base specifier] [dimension specifier] type [attributes] [initialization];
```

(Attributes and initializations are discussed below in Sections 6.2.8 and 6.2.9.)

Thus the simplest possible variable declaration consists of the reserved word DECLARE, an identifier, a type, and the terminating semicolon.

An example is

```
DECLARE HIGH$VALUE BYTE;
```

In subsequent sections, we will see more complex forms of the variable declaration.

6.2.2 IDENTIFIERS

Identifiers are discussed in Section 2.2. In the above syntax, the identifier is the name of the variable being declared.

6.2.3 BASE SPECIFIERS

Immediately after the identifier, we can insert a base specifier to declare a based variable. Based variables have already been described in Sections 3.6.3 and 4.1.3.

A base specifier has the form

BASED base-identifier

where the base-identifier is the identifier of an ADDRESS scalar which has already been declared. The base may not be subscripted, and it may not itself be a based variable. It may be a structure member.

No storage is allocated for a based variable. Instead, the value of the based variable is defined to be the contents of the memory location whose address is the value of the base.

A declaration of a single based variable has the form

DECLARE identifier BASED base-identifier [dimension specifier] type ;

Notice that the word BASED must immediately follow the identifier, and the base-identifier immediately follows the word BASED. A based variable may not have any attributes or initializations in its declaration.

The following is an example of a declaration of a based variable:

```
DECLARE LIST$ITEM BASED LIST$ITEM$PTR BYTE;
```

It is assumed that LIST$ITEM$PTR is an ADDRESS variable that has already been declared. Now LIST$ITEM is a BYTE variable whose value is the contents of the byte whose address is found in LIST$ITEM$PTR when a reference is made to LIST$ITEM.

6.2.4 FACTORED VARIABLE DECLARATIONS

Instead of using a DECLARE statement to declare a single variable, we can write a parenthesized list of identifiers (with or without base specifiers) as in the following example:

```
DECLARE (RATE, PRICE, NUM BASED NUM$PTR, K) ADDRESS;
```

This is a "factored" variable declaration, which is equivalent to

```
DECLARE RATE ADDRESS;
DECLARE PRICE ADDRESS;
DECLARE NUM BASED NUM$PTR ADDRESS;
DECLARE K ADDRESS;
```

(except that contiguous storage is guaranteed for non-based variables declared in a single parenthesized list, while variables declared in consecutive declarations may not be stored contiguously).

Type information that follows the parenthesized list (in this case the type ADDRESS) applies to each variable in the list. If a dimension specifier or a PUBLIC or EXTERNAL attribute is present, it also applies to each identifier in the list. The meaning of an AT attribute with a factored list is explained in Section 6.2.8. The meaning of an initialization with a factored list is explained in Section 6.2.9.

The number of variables that can be declared in a single factored declaration is limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

6.2.5 DIMENSION SPECIFIERS

Arrays have already been introduced in Chapter 3.

A dimension specifier is a numeric constant in parentheses, and gives the number of array elements to be associated with the identifier. If no dimension specifier is used in the DECLARE statement, each identifier that is declared in that statement refers to a single variable, which may be either a BYTE or ADDRESS scalar or a structure.

The numeric constant in the dimension specifier may not be zero.

An *implicit* dimension specifier is an asterisk enclosed in parentheses. It can be used only with initializations, and is discussed below in Section 6.2.9.

The elements of an array are located in contiguous memory locations, with the first element (element 0) at the lowest address.

6.2.6 THE BYTE AND ADDRESS TYPES

Every PL/M variable must be declared with a "type." There are three types: BYTE, ADDRESS, and STRUCTURE. The STRUCTURE type is discussed separately in Section 6.2.7 below.

Each BYTE data element is an 8-bit scalar value occupying a single byte of storage. Each ADDRESS data element is a 16-bit scalar value occupying two contiguous bytes of storage. The type of each variable element must be formally declared in its DECLARE statement.

A BYTE variable may assume any value from 0 to 255 (or 0FFH, or 377Q, or 1111$1111B).

An ADDRESS variable may assume any value from 0 to 65535 (or 0FFFFH, or 177777Q, or 1111$1111$1111$1111B).

6.2.7 THE STRUCTURE TYPE

Structures have already been introduced in Chapter 3. Notice that in terms of the syntax of DECLARE statements, STRUCTURE is a "type" like BYTE and ADDRESS, and occupies the same position in the statement — immediately following the dimension specifier (if any), or the identifier or list of identifiers.

However, the STRUCTURE type does not consist of the single word STRUCTURE, but also includes the parenthesized list of structure members. The following are examples of valid DECLARE statements using the STRUCTURE type.

```
DECLARE ENTRY (12) STRUCTURE (
    HEAD ADDRESS,
    NEXT ADDRESS,
    INFO ADDRESS,
    SIZE BYTE);

DECLARE POINT STRUCTURE (
    A$COORD (3) BYTE,
    B$COORD (3) BYTE,
    RANGE ADDRESS);

DECLARE ITEM BASED ITEM$POINTER STRUCTURE (
    HEAD ADDRESS,
    PREDECESSOR ADDRESS,
    SUCCESSOR ADDRESS,
    INFO (200) BYTE);
```

Notice that an array can be made up of structures (first example), that a structure can contain arrays (second and third examples), and that a structure can be based (third example).

The members of a structure are physically located in contiguous memory locations, in the order in which they are specified.

Structure Member Specifications

Structure member specifications are the items inside the parentheses of a STRUCTURE type. A structure member specification has the form

```
member-identifier [dimension specifier] type
```

where a member-identifier is formed in the same way as any PL/M identifier — that is, a string of up to 31 alphanumeric characters, the first of which must be a letter. Dollar signs may be embedded.

The following restrictions apply to structure member specifications:

- Factored structure member specifications are not allowed. Each member-identifier must be written with its own dimension specifier (if any) and type.
- Member-identifiers may not be based, although the structure itself may be based.
- The type must be BYTE or ADDRESS — no structures within structures are permitted.
- No attributes or initializations are allowed in a structure member specification, although the structure itself may have attributes and may be initialized, as will be seen below.

The number of member-identifiers that can be included in the declaration of a structure is limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

6.2.8 ATTRIBUTES

There are three "attributes" that can be used in a variable declaration: PUBLIC, EXTERNAL, and AT. As explained below, PUBLIC and AT may be used together but no other combinations of attributes are permitted. The PUBLIC and EXTERNAL attributes allow the programmer to extend the scope of certain variables beyond the boundaries of a module. The AT attribute allows the programmer to force a variable to be located at a specified address in memory.

The PUBLIC and EXTERNAL Attributes: Extended Scope

The PUBLIC and EXTERNAL attributes permit the programmer to extend the scope of variables (see Chapter 9) so as to allow linkage between separate modules of a program (see Chapter 10). They may only be used in declarations at the outer level of a module, and may not be used with based variables.

For example, the following declaration makes FLAG accessible from other program modules:

```
DECLARE FLAG BYTE PUBLIC;
```

The following declaration would be used in another module to indicate that all references to FLAG within that module are references to the FLAG declared PUBLIC in the declaration above:

```
DECLARE FLAG BYTE EXTERNAL;
```

The PUBLIC and EXTERNAL attributes are mutually exclusive. That is, they may not be used together in the same declaration.

A declaration with the PUBLIC attribute is called the "defining" declaration of each variable declared. It is the declaration that gives all necessary information about each variable and causes storage to be allocated.

A declaration with the EXTERNAL attribute is called a "usage" declaration. It says that each variable declared in the declaration is defined in a defining declaration in another module. The usage declaration does not cause any storage to be allocated.

The effect of declaring a variable PUBLIC in one module is to extend its scope to include every other module in which it is declared EXTERNAL. More specifically, within each module the PUBLIC or EXTERNAL declaration of the variable gives it a certain scope. The extended scope is the combination of these scopes (see Chapter 9 for a discussion of scope).

The following rules apply to declarations with the PUBLIC attribute:

- Within any program, each variable with extended scope must have exactly one defining declaration.
- The PUBLIC attribute may only be used in a declaration at the outer level of a module (see Chapter 10).
- The PUBLIC attribute may not be used with a based variable (however, the base of a based variable may be PUBLIC).

The following rules apply to declarations with the EXTERNAL attribute:

- The EXTERNAL attribute may only be used in a declaration at the outer level of a module (see Chapter 10).

- The EXTERNAL attribute may only be used with a variable that is declared PUBLIC in another module (see Chapter 10) of the same program.

- The EXTERNAL attribute may not be used with a based variable (however, the base of a based variable may be declared EXTERNAL).

- The EXTERNAL attribute may not be used in combination with the AT attribute, or with an initialization. Note, however, that the defining declaration of a variable may have the AT attribute and/or an initialization.

- When a scalar variable is declared EXTERNAL, it must have the same type as in the defining declaration.

- When an array is declared EXTERNAL, it must have the same number of elements and the same type as in the defining declaration.

- When a structure is declared EXTERNAL, it must have the same list of members as in the defining declaration. Strictly speaking, the members do not have to have the same member-identifiers — it is only necessary to have the members correspond as to their dimension specifiers (if any) and their types. However, it is good practice to make the member-identifiers the same also.

It should be noted that the PUBLIC and EXTERNAL attributes may also be applied to procedure declarations. When this is done, there are some additional rules. These rules are given in Chapter 8.

The AT Attribute

The AT attribute has the form

AT (restricted expression)

A *restricted expression* is a sequence of one or more operands separated by operators, with the following restrictions:

- o The first operand may only be a numeric constant or a location reference. If it is a location reference, it must refer to a variable that has already been declared; and if the location reference contains a subscript expression, this expression may not contain any operands except numeric constants or any operators except + and –.

- o If any operands follow the first operand, they may only be numeric constants.

- o Only the + and – operators are allowed.

The following are examples of valid AT attributes:

AT (4096)

AT (.BUFFER)

```
AT (.BUFFER + 128)

AT (.NAMES(INDEX + 1) + OFFSET - 3)
```

In the last example, INDEX and OFFSET represent numeric constants that have been previously declared with "LITERALLY" declarations (see Section 6.4). The compiler replaces these names with the declared numeric constants, thus satisfying the restrictions given above.

The effect of an AT attribute is to cause a variable to be located at the address specified by the restricted expression. The variable located is the first scalar in the declaration. Other scalars in the same declaration will follow in sequence.

For example, the declaration

```
DECLARE (CHAR$A, CHAR$B, CHAR$C) BYTE AT (.BUFFER);
```

causes the BYTE variable CHAR$A to be located at the address of the array BUFFER. The variables CHAR$B and CHAR$C are located in the next two bytes after CHAR$A.

The declaration

```
DECLARE T (10) STRUCTURE(
    X (3) BYTE,
    Y (3) BYTE,
    Z (3) BYTE)
    AT (.DATA$BUFFER);
```

causes the beginning of the structure T — namely the scalar T(0).X(0) — to be located at the same address as a previously declared variable called DATA$BUFFER. The other scalars making up the structure will follow this address in logical order: T(0).X(1), T(0).X(2), and so on up to T(9).Z(2), the last scalar, which is located in the 89th byte after the address of DATA$BUFFER.

However, no memory locations for these 90 scalars are allocated by this declaration. It is up to the programmer to know what else, if anything, will be stored in the memory space starting at .DATA$BUFFER.

The following rules apply to the AT attribute:

- The AT attribute cannot be used with based variables.
- It can be used with the PUBLIC attribute, in which case it immediately follows the word PUBLIC.
- It cannot be used with the EXTERNAL attribute.

The AT attribute can be used to make variables "equivalent," providing more than one way of referring to the same information. For example,

```
DECLARE DATUM ADDRESS;
DECLARE ITEM BYTE AT (.DATUM);
```

causes ITEM to be declared a BYTE variable at the same location that has just been allocated for the ADDRESS variable DATUM. The result is that any reference to ITEM is in effect a reference to the *low-order* byte of DATUM.

The following is another example:

```
DECLARE VECTOR (6) BYTE;
DECLARE SHORT$VECTOR STRUCTURE (
    FIRST (3) BYTE,
    SECOND (3) BYTE)
    AT (.VECTOR);
```

Here we first declare a six-element BYTE array, VECTOR. Then we declare a structure of two three-BYTE arrays, SHORT$VECTOR.FIRST and SHORT$VECTOR.SECOND. The first scalar of this structure — SHORT$VECTOR.FIRST(0) — is located at the same address as the first element of the array VECTOR.

Thus we have two different ways of referring to the same six bytes. For example, the fifth byte in the group can be referenced as either VECTOR(4) or SHORT$VECTOR.SECOND(1).

When a variable is declared with the AT attribute, the PL/M-80 Compiler does not optimize the machine code generated for accesses to that variable. This is useful in connection with memory-mapped I/O.

6.2.9 INITIALIZATIONS

Initializations are used to supply values for variables at compile time. There are two kinds of initialization, INITIAL and DATA.

INITIAL causes initialization during program loading of a variable that has storage allocated for it. Thus a variable that is initialized with INITIAL can subsequently be changed during the program run, like any other variable. It will *not* be reinitialized on a program restart.

The DATA initialization not only initializes a variable, but also causes it to be stored with the program code. In ROM-based systems, this means that a variable initialized in this way can never be changed by the program. It is a "variable" in form only, and should never appear on the left side of an assignment statement.

The following rules apply to INITIAL and DATA initializations:

- INITIAL and DATA may not be used together in the same declaration.
- INITIAL may only be used in a declaration at the outer level of a program module (see Chapter 10). DATA may be used in a DECLARE statement at any level in the program structure.
- No initializations may be used with based variables or with the EXTERNAL attribute.
- An initialization may be used with the AT attribute. However, if this causes multiple initialization, the result is undefined. Also, if the AT attribute causes a variable to be externally located, the variable may not have an initialization. For example, if a variable ABC has been declared EXTERNAL, and another variable XYZ is declared with the attribute AT (.ABC), then XYZ is externally located and may not have an initialization.

The INITIAL Initialization

The INITIAL initialization has the form

```
INITIAL (value list)
```

where the value list is a sequence of one or more values separated by commas. Each value may be either a *restricted expression* as described in Section 6.2.8 in connection with AT, or a string.

Values are taken one at a time from the value list and used to initialize the individual scalars being declared. The declaration

```
DECLARE THRESHOLD BYTE INITIAL (48);
```

declares the BYTE scalar THRESHOLD in the usual way, and also initializes it to a value of 48.

The declaration

```
DECLARE (COUNTER, LIMIT, INCR) ADDRESS INITIAL (0, 1024, 2);
```

declares the ADDRESS scalars COUNTER, LIMIT, and INCR, and initializes COUNTER to a value of 0, LIMIT to a value of 1024, and INCR to a value of 2.

The declaration

```
DECLARE EVEN (5) BYTE INITIAL (2,4,6,8,10);
```

declares the BYTE array EVEN and initializes its five scalar elements to 2, 4, 6, 8, and 10 respectively.

The declaration

```
DECLARE COORD STRUCTURE (
    HIGH$BOUND ADDRESS,
    VALUE (3) BYTE,
    LOW$BOUND BYTE)
        INITIAL (302, 3, 6, 12, 0);
```

declares the structure COORD and initializes it as follows:

```
COORD.HIGH$BOUND to 302
COORD.VALUE(0) to 3
COORD.VALUE(1) to 6
COORD.VALUE(2) to 12
COORD.LOW$BOUND to 0.
```

When a restricted expression is used to initialize a BYTE scalar, its value must not be greater than 255. If it is used to initialize an ADDRESS scalar, its value must not be greater than 65535.

If a string appears in the constant list, it is taken apart from left to right and the pieces are stored in the scalars being initialized. One character is stored in each BYTE scalar, and two in each ADDRESS scalar. For example,

```
DECLARE GREETING (5) BYTE INITIAL ('HELLO');
```

causes GREETING (0) to be initialized with the ASCII code for H, GREETING (1) with the ASCII code for E, and so forth.

So far, all of the examples have shown constant lists that match up one-for-one with the scalars being declared. It is permissible for the constant list to have *fewer* elements than are being declared. Thus

```
DECLARE DATUM (100) BYTE INITIAL (3, 5, 7, 8);
```

is permissible. The first 4 elements of the array DATUM are initialized with the 4 elements in the constant list, and the remainder of the array is left uninitialized. However, the constant list may not have *more* elements than are being declared.

Often, when one initializes an array, one wants the array to have the same number of elements as the constant list. This can be done conveniently by using the *implicit dimension specifier*. This is used in place of an ordinary dimension specifier (that is, a parenthesized constant), and has the form

```
(*)
```

The implicit dimension specifier may not be used in the following cases:

- After the parenthesized list of identifiers in a factored declaration.
- To specify an array whose elements are structures.
- To specify an array which is a member of a structure.

It may only be used with an initialization.

The implicit dimension specifier causes the number of elements in the array to be the same as the number of elements in the constant list of the initialization. Thus the declaration

```
DECLARE FAREWELL (*) BYTE INITIAL ('GOODBYE, NOW');
```

declares a BYTE array, FAREWELL, with enough elements to contain the string 'GOODBYE, NOW' (namely 12), and initializes the array elements with the characters of the string.

The implicit dimension specifier may be used with any constant list — it is not restricted to use with strings.

The DATA Initialization

The DATA initialization has the form

```
DATA (value list)
```

The DATA initialization is identical to the INITIAL initialization, except for three differences which have already been mentioned. They are:

- o The DATA initialization causes storage to be allocated along with the program code. In ROM-based systems, this means that variables declared with a DATA initialization can never be changed by the program.

- An identifier declared with the DATA initialization should never appear on the left side of an assignment statement.

- Unlike the INITIAL initialization, the DATA initialization can be used in a declaration at any block level in the program.

NOTE

This description assumes that the AT attribute is not used with the DATA initialization. Since the AT attribute forces a variable to be located at a specified address, it may defeat the purpose of the DATA initialization.

6.3 LABEL DECLARATIONS

A label is an identifier that is associated with a particular executable statement in a PL/M source program and refers to it. Normally, it is not necessary to declare labels, since a label is *implicitly* declared when it appears in a "label definition" as explained in Section 5.3.1. Under certain circumstances, however, it may be desirable to declare a label explicitly in order to give it extended scope (see Section 6.2.8). The label declaration makes this possible.

6.3.1 IMPLICIT LABEL DECLARATIONS

As noted in Section 5.3.1, the appearance of a label in front of an executable statement is called a "label definition."

If the label is not explicitly declared by a label declaration at the beginning of the smallest block that encloses the label definition, then the label definition not only defines the label but also declares it implicitly. The resulting scope of the label is as if the label had been declared explicitly at the beginning of the smallest enclosing block.

If a label is explicitly declared at the beginning of the smallest enclosing block that encloses the label definition, then the label definition does *not* implicitly declare the label (if it did, it would be illegal, since it is illegal to re-declare something within the outer level of the same block where it was first declared).

Some special consequences of implicit label declaration are described in Section 9.3.

6.3.2 EXPLICIT LABEL DECLARATIONS — BASIC SYNTAX

In simple form, the syntax of a label declaration is as follows:

```
DECLARE identifier LABEL [attribute];
```

The identifier is the PL/M identifier being declared as a label. The attribute may be PUBLIC or EXTERNAL (see Section 6.3.4 below), or may be omitted, as indicated by the brackets.

Note that this syntax is much simpler than the syntax for a variable declaration — there is no type, dimension specifier, or initialization.

6.3.3 FACTORED LABEL DECLARATIONS

Instead of a single identifier, we can write a parenthesized list of identifiers separated by commas, as in the following example:

```
DECLARE (ENTRY, EXIT, MAIN, ERROR1, ERROR2) LABEL PUBLIC;
```

which is exactly equivalent to

```
DECLARE ENTRY LABEL PUBLIC;
DECLARE EXIT LABEL PUBLIC;
DECLARE MAIN LABEL PUBLIC;
DECLARE ERROR1 LABEL PUBLIC;
DECLARE ERROR2 LABEL PUBLIC;
```

When a factored label declaration has an attribute (PUBLIC or EXTERNAL), it applies to each identifier in the list.

The effect of this example is to declare five labels — ENTRY, EXIT, MAIN, ERROR1, AND ERROR2 — and give them all the PUBLIC attribute. These labels can be declared EXTERNAL in other program modules, making it possible to transfer control from other modules to this one by means of GOTO statements (subject to restrictions given in Section 9.3).

The number of labels that can be declared in a single factored label declaration is limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

6.3.4 ATTRIBUTES OF LABELS

The only attributes allowed for labels are PUBLIC and EXTERNAL. They may only be used in label declarations at the outer level of a program module (see Chapter 10). The effect of these attributes is to give labels extended scope, just as with variables. The rules given in Section 6.2.8 above apply to label declarations as well as to variable declarations.

To be meaningful, an explicit label declaration with the PUBLIC attribute must be accompanied by a label definition (since the explicit declaration does not define the location of the label). This label definition must be at the outer level of the same block as the explicit declaration — otherwise, it will be an implicit declaration, that is, it will not be the same label declared in the explicit declaration. In fact, for reasons given in Chapter 9, both the explicit declaration and the label definition must be at the outer level of the "main program module" (see Chapter 10 for discussion of modules).

6.4 MACRO DECLARATIONS ("LITERALLY" DECLARATIONS)

A declaration using the reserved word LITERALLY defines a parameterless *macro* for expansion at compile-time. An identifier is declared to represent a character string, which will then be substituted for each occurrence of the identifier in subsequent text. The form of the declaration is

```
DECLARE identifier LITERALLY 'string';
```

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

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

DECLARE ROUGH BYTE;
DECLARE (X, Y, DELTA) ADDRESS;

...

ROUGH = TRUE;
DO WHILE ROUGH;
    X = SMOOTH(X, Y, DELTA);
    /*SMOOTH is a procedure declared elsewhere.*/
    IF (X–FINAL) < DELTA THEN ROUGH = FALSE;
END;

...
```

The LITERALLY declaration defines the boolean values TRUE and FALSE in a manner consistent with the way PL/M handles relational operators (see Section 4.4). This often makes a program more readable.

Another use of the macro declaration is the declaration of quantities which are fixed for one compilation, but may change from one compilation to the next. Consider the example below:

```
DECLARE BUFFER$SIZE LITERALLY '32';

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

PRINTBUFFER (BUFFERSIZE–10) = 'G';
...
```

A future change to BUFFER$SIZE can be made in one place at the first declaration, and the compiler will propagate it throughout the program during compilation. Thus the programmer is saved the tedious and error-prone process of searching his program for the occurrences of "32" that are buffer-size references, and not some other 32's.

6.5 COMBINING DECLARE STATEMENTS

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

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

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

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

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

Previous to this section, all examples have shown only one declaration element in each DECLARE statement. A declaration element is the text for declaring one identifier (or one factored list of identifiers). In the example above, the text CHR BYTE INITIAL ('A') is one declaration element, and the text COUNT ADDRESS is another.

The declaration elements appearing in a single DECLARE statement are completely independent of each other. It is as if they were declared in separate DECLARE statements.

CHAPTER 7
SAMPLE PROGRAM #1

At this point, we have examined all of the constructions available in PL/M except procedures, and we can now consider a complete PL/M program.

The following sample program implements a straight insertion sort algorithm based on Knuth's "Algorithm S" in *The Art of Computer Programming*, Vol. 3, page 81. Readers who look up Knuth's algorithm should note the following differences:

- The algorithm has been adapted to PL/M usage by using an array of structures to represent the records to be sorted. The sort key for each record is a member of the structure for that record.

- It has been modified by using a DO WHILE block to achieve the same logical effect as the GOTOs implied in steps S3 and S4 of Knuth's algorithm.

- The index I is used in a slightly different manner (it is initialized to J instead of J-1).

The effect of the algorithm is to arrange 128 records in order according to the values of their keys, with the smallest key at the beginning (lowest memory address) and the largest key at the end (highest address).

The sorting method is as follows. Assume that the records are all in memory, stored as an array of structures. The key for each record is a member of the structure.

Now we go through the array from the second record (record number 1) upwards. When we reach any given record (the "current" record), we will already have sorted the preceding records. (The first time through, when we look at record number 1, record number 0 is the only preceding record.)

We take the current record, store it temporarily in a buffer, and look backwards through the preceding records until we find one whose key is not greater than that of the current record. Then we put the current record just after this record.

The program below is followed by a detailed explanation. Please study the program and the explanation until you understand how the program works (especially the DO WHILE block, which is controlled by a more complex condition expression than we have seen up to this point).

```
M: DO;        /*Beginning of module*/

      DECLARE RECORD (128) STRUCTURE (
                  KEY BYTE,
                  INFO ADDRESS);

      DECLARE CURRENT STRUCTURE (
                  KEY BYTE,
                  INFO ADDRESS);

      DECLARE (J, I) BYTE;

      /*Data is read in to initialize the records.*/

SORT: DO J = 1 TO 127;
         CURRENT.KEY = RECORD(J).KEY;
         CURRENT.INFO = RECORD(J).INFO;
         I = J;

FIND:     DO WHILE I > 0 AND RECORD(I-1).KEY > CURRENT.KEY;
                  RECORD(I).KEY = RECORD(I-1).KEY;
                  RECORD(I).INFO = RECORD(I-1).INFO;
                  I = I-1;
         END FIND;

         RECORD(I).KEY = CURRENT.KEY;
         RECORD(I).INFO = CURRENT.INFO;
      END SORT;

      /*Data is written out from the records.*/

   END M;        /*End of module*/
```

Let us now consider the code of this program. First we declare the following variables:

- RECORD, an array of 128 structures to hold the 128 records. Each structure has a BYTE member which is the sort key, and an ADDRESS member which could contain anything (in a working program, this would be the data content of the record).

- CURRENT, a structure used as a buffer to hold the current record while we look back through the records already sorted. Its members are like those of one structure element of RECORD.

- J, which will be used as an index variable in an iterative DO statement. J is always the subscript of the current record. When J becomes greater than 127, the sort is done.

- I, which will be used like an index variable in controlling a DO WHILE block. I–1 is always the subscript of a previously sorted record.

A working program would include code at this point to read data into the array RECORD. At the end of the program, there would be code to write out the data from RECORD. In this example, we omit this code because it would make the example too lengthy and because the method used for I/O would depend on the particular system used to execute the program. Comments have been inserted in place of this code.

The executable part of the program is organized as two DO blocks, one nested within the other. The outer block (labeled SORT) is an iterative DO block which goes through the records one at a time. The record selected by the index variable J each time through this block is the "current record." (Notice that J is never 0 — because of the way the algorithm is defined, we must have a preceding element to look back at, and so we start with the second element of the array and look back at the first.)

The first two assignment statements in the block transfer the current record into CURRENT. The next statement sets the initial value for I, which will be used to control the inner block.

The inner block (labeled FIND) is the one that looks back through previously sorted records to find the right place to put the current record. The way this block is controlled is worth examining. The variable I is used like an index variable in an iterative DO, but it is changed explicitly inside the block, instead of automatically as in an iterative DO statement. The DO WHILE construction is used instead of an iterative DO because it allows two or more tests to be combined — in this case, by means of an AND operator.

I is set to J before the first time through the DO WHILE block, and decremented each time through. As long as I remains greater than 0, the first half of the DO WHILE condition is satisfied.

The value I–1 is the subscript of the record being "looked back at." The second half of the DO WHILE condition is that the key of this record must be greater than the key of the current record.

We are looking for a previously sorted record whose key is not greater than the key of the current record. Thus the condition in the DO WHILE statement will cause the DO WHILE block to be repeatedly executed until such a record is found, or until I reaches 0 (meaning that all previously sorted records have been examined).

Each time the DO WHILE block is executed, it moves the I–1st record "up" into the Ith position, and then decrements I.

When the condition in the DO WHILE statement is *not* met, one of the following is true:

- I = 0, because we have looked through all the previously sorted records without finding one whose key is not greater than that of the current record. All of the previously sorted records have been moved "up" by one.

- I–1 is the subscript of a record whose key is not greater than the key of the current record. All of the previously sorted records whose keys *are* greater than that of the current record have been moved "up" by one.

In either case, the failure of the DO WHILE condition means that the current record (being held in CURRENT) belongs in the Ith position. It is transferred into this position by the two assignment statements that form the remainder of the outer DO block.

Now the outer DO block repeats with an incremented value of J, to consider the next unsorted record.

Notice that the entire program is contained within a simple DO block labeled M. This makes it a "module," as described in Chapter 10.

CHAPTER 8
PROCEDURES

A "procedure" is a section of PL/M code which is declared without being executed, and then "called" from other parts of the program. The call "activates" the procedure, causing the procedure code to be executed out of normal sequence: program control is transferred from the point of call to the beginning of the procedure code, the code is executed, and upon exit from the procedure code, program control is passed back to just beyond the point of the call.

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

8.1 PROCEDURE DECLARATIONS

Procedures, like variables, must be declared. Any reference to a procedure must occur within the scope defined by the procedure declaration. Also, a reference to a procedure may not occur until *after* the END statement of the procedure declaration (except as noted below in Section 8.1.7).

A procedure declaration consists of three parts: a PROCEDURE statement, a sequence of statements forming the "procedure body," and an END statement. These parts take the following form:

```
name: PROCEDURE [(parameter list)] [type] [attributes] ;
    statement-1 ;
    statement-2 ;
    ...
    statement-n ;
END [name] ;
```

The following is a simple example:

```
DOOR$CHECK: PROCEDURE;
    IF FRONT$DOOR$LOCKED AND SIDE$DOOR$LOCKED THEN
        CALL POWER$ON;
    ELSE CALL DOOR$ALARM;
END DOOR$CHECK;
```

where POWER$ON and DOOR$ALARM are procedures declared elsewhere in the same program.

NOTE

The name in a PROCEDURE statement has the same appearance as a label definition — but it is *not* considered a label definition, and a procedure name is not a label. PROCEDURE statements may not be labeled.

The name is a PL/M identifier, which is associated with this procedure. The scope of a procedure is governed by the placement of its declaration in the program text, just as the scope of a variable is governed by the placement of its DECLARE statement (see Chapter 9 for a detailed description). Within this scope, the procedure can be called by the name used in the PROCEDURE statement.

A procedure declaration, like a DO block, is a block. As such, it controls the scope of variables as described in Chapter 9. Also, like a simple DO block, a procedure declaration may contain DECLARE statements, and they must precede the first executable statement in the procedure body.

The procedure body *must* contain at least one executable statement, unless the procedure has the EXTERNAL attribute (see Section 8.1.5). A null statement (consisting of nothing but a semicolon) is sufficient.

As in a DO block, the identifier in the END statement has no effect on the program, but helps legibility and debugging. If used, it must be the same as the procedure name.

The parameter list and the type are discussed in the following two sections.

8.1.1 PARAMETERS

Formal parameters are non-based scalar variables declared within a procedure declaration, whose identifiers appear in the parameter list in the PROCEDURE statement. The identifiers in the list are separated by commas and the list is enclosed in parentheses. No subscripts or member-identifiers are allowed in the parameter list.

If the procedure has no formal parameters, the parameter list (including the parentheses) is omitted from the PROCEDURE statement.

Each formal parameter must be declared as a non-based scalar variable in a DECLARE statement preceding the first executable statement in the procedure body.

When a procedure that has formal parameters is called, the CALL statement or function reference contains a list of *actual parameters*. Each actual parameter is an expression whose value is assigned to the corresponding formal parameter in the procedure, before the procedure begins to execute.

NOTE

> Parameters are not stored according to the same rules as other declared variables. In particular, do not assume that a parameter is stored contiguously with other variables declared in the same factored variable declaration.

For example, the following procedure takes four parameters, called POINTER, N, LOWER, and UPPER. It examines N contiguously stored BYTE variables. The parameter POINTER is the address of the first of these variables. If any of these variables is less than the parameter LOWER or greater than the parameter UPPER, the ERRORSET procedure (declared elsewhere in the program) is called.

```
RANGE$CHECK: PROCEDURE (POINTER, N, LOWER, UPPER);
    DECLARE POINTER ADDRESS;
    DECLARE (N, LOWER, UPPER, ITEM BASED POINTER, I) BYTE;

    DO I = 1 TO N;
        IF (ITEM < LOWER) OR (ITEM > UPPER)
        THEN CALL ERRORSET;

        /*ERRORSET is a procedure declared elsewhere.*/

        POINTER = POINTER + 1;

        /*Next time around, look at the next item.*/

    END;
END RANGE$CHECK;
```

Having made this declaration, suppose that we have 25 variables stored contiguously in an array called VALUES. We want to check that all of these variables have values within the range defined by the values of two other BYTE variables, LOW and HIGH.

We write

```
CALL RANGE$CHECK (.VALUES, 25, LOW, HIGH);
```

When this CALL statement is processed, the following sequence occurs:

- The four expressions in the CALL statement — .VALUES, 25, LOW, and HIGH — are evaluated. These values are the actual parameters.
- The four values are assigned to the formal parameters POINTER, N, LOWER, and UPPER, which are declared within the procedure RANGE$CHECK and are named in the parameter list.
- The executable statements of the procedure RANGE$CHECK are executed, and if any of the elements of VALUES are less than the value of LOW or greater than the value of HIGH, the procedure ERRORSET is called.
- Finally, control returns to the statement following the CALL statement.

Notice how the use of a based variable, with the base passed as a parameter, allows the procedure to have its own unchanging name (ITEM) for a set of variables which may be a different set each time the procedure is called.

8.1.2 TYPED AND UNTYPED PROCEDURES

The procedure shown above is an "untyped" procedure. No type is given in the PROCEDURE statement, and it does not return a value. An untyped procedure is called by using its name in a CALL statement, as shown above and as explained in Section 8.2.

A typed procedure has a type, either BYTE or ADDRESS, in its PROCEDURE statement, and it returns a value of this type. It is called by using its name in an expression as a special kind of variable reference called a "function reference." As we have seen in Section 4.1.2, a function reference may

be an operand in an expression.

When the expression is processed at run time, the appearance of the function reference causes the procedure to be executed. The function reference itself is then replaced by the value returned by the procedure. The expression is then evaluated, and program execution continues in normal sequence.

Like an untyped procedure, a typed procedure may have parameters. They are handled in the same way as described above in Section 8.1.1.

The body of a typed procedure must always contain a RETURN statement with an expression, as explained in the following section.

8.1.3 EXIT FROM A PROCEDURE: THE RETURN STATEMENT

The execution of a procedure is terminated in one of three ways:

- By execution of a RETURN statement within the procedure body. A typed procedure *must* contain a RETURN statement with an expression.
- By reaching the END statement that terminates the procedure declaration.
- By executing a GOTO to a statement outside the procedure body. The target of the GOTO must be at the outer level of the main program module (see Chapter 10). This method should be used only when necessary.

The RETURN statement takes one of two forms:

```
RETURN;
```

or

```
RETURN expression;
```

The first form is used in an untyped procedure. The second form is used in a typed procedure. The value of the expression is the value returned by the procedure.

It is evaluated as a quantity of the type given in the PROCEDURE statement — that is, if it yields a BYTE result and the procedure is type ADDRESS, then 8 high-order 0 bits are appended to form an ADDRESS value. If it yields an ADDRESS result and the procedure is type BYTE, the 8 high-order bits of the result are dropped to form a BYTE value.

8.1.4 THE PROCEDURE BODY

The statements within the procedure body may be any valid PL/M statements, including calls and nested procedure declarations.

Example 1

The following is a typed procedure declaration:

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

This procedure could be used as follows:

```
LOW = 3;
HIGH = 4;
MEAN = AVG (LOW, HIGH);
```

The effect would be to assign the value 3 to MEAN (because the division by 2 in the procedure body yields an integer result).

Example 2

The following is an untyped procedure:

```
AOUT: PROCEDURE (ITEM);
    DECLARE ITEM ADDRESS;
    IF ITEM >= 0FFH THEN COUNTER = COUNTER + 1;
    RETURN;
END AOUT;
```

Here COUNTER is some variable declared outside the procedure — that is, a "global" variable. This procedure could be called as follows:

```
CALL AOUT(UNKNOWN);
```

If the value of the variable UNKNOWN is greater than or equal to 0FFH, the value of COUNTER will be incremented.

Example 3

This example demonstrates an important use of based variables:

```
SUM$ARRAY: PROCEDURE (PTR, N) BYTE;
    DECLARE PTR ADDRESS,
        ARRAY BASED PTR (1) BYTE,
        (N, SUM, I) BYTE;
    SUM=0;
    DO I=0 TO N;
        SUM=SUM+ARRAY(I);
    END;
    RETURN SUM;
END SUM$ARRAY;
```

This procedure returns the sum of the first N+1 elements (from the 0th to the Nth) of a BYTE array pointed to by PTR. Notice that ARRAY is declared to have 1 element. Since it is a based variable, no space is allocated for it. It must be declared as an array, so that it can be subscripted in the iterative DO block. The choice of 1 as the constant in the dimension specifier is arbitrary, and does not restrict the value of N that may be supplied when the procedure is called.

The procedure could be used as follows to sum the elements of a 100-element BYTE array named PRICE, and assign the sum to the variable TOTAL:

```
TOTAL = SUM$ARRAY(.PRICE, 99);
```

8.1.5 THE PUBLIC AND EXTERNAL ATTRIBUTES

The PUBLIC and EXTERNAL attributes can be included in PROCEDURE statements to give procedures extended scope. Extended scope is discussed in Section 6.2.8 and Chapter 10.

A procedure declaration with the PUBLIC attribute is called a "defining declaration." The following rules apply to the use of the PUBLIC attribute in a PROCEDURE statement:

- Within any program, each procedure with extended scope must have exactly one defining declaration — that is, it must be declared once with the PUBLIC attribute.
- The PUBLIC attribute may only be used at the outer level of a module (see Chapter 10).

A procedure declaration with the EXTERNAL attribute is called a "usage declaration." The following rules apply to use of the EXTERNAL attribute in a procedure declaration:

- The EXTERNAL attribute may only be used at the outer level of a module (see Chapter 10).
- The EXTERNAL attribute may only be used if the procedure is declared PUBLIC in another module of the same program.
- The EXTERNAL attribute may not be used in the same PROCEDURE statement as a PUBLIC, INTERRUPT, or REENTRANT attribute (see below). Note, however, that the defining declaration of a procedure may have the INTERRUPT and REENTRANT attributes.
- A usage declaration of a procedure should have the same number of parameters as the defining declaration. Variable types and dimension specifiers should match up in the same sequence in both declarations. The names of the parameters need not be the same. Note that the compiler cannot detect a discrepancy between the parameter lists in the defining declaration and a usage declaration.
- The procedure body of a usage declaration may not contain anything except the declarations of the formal parameters. The formal parameters must be declared with the same types as in the defining declaration.
- The END statement of a usage declaration may not be labeled.

For example, we can alter the procedure AVG (from Section 8.1.4) by giving it the PUBLIC attribute as follows:

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

In another module, we can have a usage declaration:

```
AVG: PROCEDURE (X, Y) ADDRESS EXTERNAL;
    DECLARE (X, Y) ADDRESS;
END AVG;
```

Now, in the module with the usage declaration, we can reference AVG in an executable statement:

```
MIDDLE = AVG(FIRST, LATEST);
```

The effect of this is to activate the procedure AVG as declared in the first module.

8.1.6 INTERRUPTS AND THE INTERRUPT ATTRIBUTE

An untyped procedure with no parameters, declared at the outer level of a program module (see Chapter 10), may have the INTERRUPT attribute. A procedure with this attribute is called an "interrupt procedure."

The INTERRUPT attribute has the form

```
INTERRUPT n
```

where *n* is any numeric constant from 0 to 255 (inclusive). The effect of this attribute is that the procedure can be activated without being called, when the 8080 interrupt corresponding to *n* occurs during program execution.

NOTE

> Interrupt numbers higher than 7 can only be handled in a system that incorporates an 8259 Programmable Interrupt Controller chip. The discussion below applies to a system *without* the 8259, supporting only interrupts 0 through 7.

To explain interrupt processing, we must first consider the 8080 interrupt mechanism and the PL/M statements ENABLE and DISABLE.

The interrupt mechanism has two states, "enabled" and "disabled". The CPU always starts in the disabled state. The ENABLE statement forces it into the enabled state, and has the form

```
ENABLE ;
```

The HALT statement also forces interrupts to be enabled (see Section 5.4).

The DISABLE statement forces the interrupt mechanism to be disabled, and has the form

```
DISABLE ;
```

An interrupt is initiated by two actions:

- The 8080 interrupt line is driven high by some peripheral device.
- The peripheral device "jams" an RST instruction onto the 8080 data bus. This instruction contains a number from 0 to 7 (inclusive).

If interrupts are disabled, the 8080 CPU ignores these actions. If interrupts are enabled, the RST is executed.

If a procedure has been declared with an INTERRUPT attribute with the same number as the RST instruction, the effect of the RST is to disable interrupts and activate this procedure. After the procedure terminates (by executing a RETURN or reaching the END of the procedure), interrupts are enabled and control is returned to the point where the program was interrupted. It is also possible (as with other untyped procedures) for the procedure to terminate by executing a GOTO with a target outside the procedure, in the outer level of the main program module. In this case, control will never be returned to the point where the program was interrupted, and interrupts will not be automatically enabled.

Interrupt processing is discussed in greater detail in the *ISIS-II PL/M-80 Compiler Operator's Manual.*

The following is an example of an interrupt procedure for a hypothetical system where a peripheral device jams an RST 3 instruction onto the data bus whenever the temperature of a device exceeds a certain threshold. The interrupt procedure turns on an annunciator light, updates a status word, and returns control to the program.

```
HITEMP: PROCEDURE INTERRUPT 3;
    CALL ANNUNCIATOR (1);
        /*This call will result in an output
        from the 8080 to turn on annunciator light
        number 1, the high-temperature warning.*/
    ALERT = ALERT OR 00000010B;
        /*This puts a 1 in one of the bit positions
        of ALERT, which contains a bit pattern representing
        current alerts.*/
END HITEMP;
```

The following rules apply to the INTERRUPT attribute:

- The INTERRUPT attribute may not be used in combination with the EXTERNAL attribute.
- It may only be used in a PROCEDURE statement at the outer level of a program module.
- The numeric constant in the INTERRUPT attribute may be any number from 0 to 7 (inclusive). Each number may be used only once within a program.
- The procedure must be untyped and may not have any parameters.

A procedure with the INTERRUPT attribute may also be activated by means of a CALL statement, like any other untyped procedure. However, when this is done, the programmer must bear in mind that interrupts are not automatically disabled, and upon termination of the procedure by means of a RETURN statement or the END statement, interrupts are automatically enabled. In every other respect, an interrupt procedure activated by a CALL statement is like any other procedure so activated.

8.1.7 REENTRANCY AND THE REENTRANT ATTRIBUTE

When a procedure does not have the REENTRANT attribute, storage for its variables is allocated statically in memory. This causes an important limitation which can be understood from the following hypothetical example.

Suppose that we have a procedure PROC$A which is activated both from the main program and from an interrupt procedure. The program runs, and PROC$A is activated. While PROC$A is running, an interrupt occurs and execution of PROC$A is suspended. The interrupt procedure is activated, and while it is running it activates PROC$A. In this "second incarnation" of PROC$A, it runs normally and uses the same storage space for variables as the suspended first incarnation. The second incarnation eventually terminates and returns to the point where it was called within the interrupt procedure.

Finally the interrupt procedure terminates and returns to the point at which the first incarnation of PROC$A was suspended.

But the variable values that were in use by the first incarnation of PROC$A have been changed by the second incarnation, and the first incarnation cannot produce correct results.

A similar problem occurs if a procedure calls itself (this is known as "direct recursion") or if it calls a second procedure with the result that the first procedure is called again before the second procedure returns ("indirect recursion").

A procedure with the REENTRANT attribute is called a "reentrant procedure." When a procedure is reentrant, the problems of multiple incarnations are avoided. Instead of being stored statically, the procedure's variables are stored on the stack, and each incarnation of the procedure uses separate storage.

Also, a procedure with the REENTRANT attribute may be called before it is declared, if it is called from within the body of a reentrant procedure — possibly the same procedure.

This permits "direct recursion," where the procedure calls itself, and "indirect recursion," where the procedure calls another procedure and the second procedure calls it back — or calls a third procedure, which calls a fourth, etc., with the result that the first procedure is called before it terminates.

The following rules summarize the use of the REENTRANT attribute:

- Any procedure that may be interrupted and is also activated from within an interrupt procedure should have the REENTRANT attribute. If this reentrant procedure calls any other procedures, they should also be reentrant.
- Any procedure that is directly recursive (calls itself) should have the REENTRANT attribute.
- Any procedure that is indirectly recursive (calls another procedure and is called back as a result) should have the REENTRANT attribute.

The following rules apply to the REENTRANT attribute.

- The REENTRANT attribute cannot be used in the same declaration as the EXTERNAL attribute.
- The REENTRANT attribute may only be used in a PROCEDURE statement at the outer level of a module (see Chapter 10).

- A procedure declaration with the REENTRANT attribute may not have another procedure declaration nested inside it.

Example of Direct Recursion

The procedure SUM$ARRAY (Example 3 in Section 8.1.4 above) can be rewritten as a recursive procedure which would be called in exactly the same way to do the same thing — namely, sum the elements of a BYTE array from 0 to N:

```
SUM$ARRAY: PROCEDURE (PTR, N) BYTE REENTRANT;
    DECLARE PTR ADDRESS,
        ARRAY BASED PTR (1) BYTE,
        N BYTE;
    IF N=0 THEN RETURN ARRAY(0);
    ELSE RETURN ARRAY(N) + SUM$ARRAY(PTR, N-1);
END SUM$ARRAY;
```

The iterative procedure used earlier may be paraphrased thus: "The sum is found by starting with 0 and then adding each array element from 0 to N."

The recursive procedure can be paraphrased thus: "If N=0, the sum is simply the value of the 0th element. Otherwise, the sum is found by taking the Nth element and adding to it the sum of the elements below it."

8.2 PROCEDURE CALLS

There are two forms of procedure call, depending on whether the procedure is typed or untyped. An untyped procedure is called by means of a CALL statement, which has the form

```
CALL name [(parameter list)] ;
```

An example is the following:

```
CALL REORDER (.RANK$TABLE, 3);
```

(An alternate form of the CALL statement is given in Section 8.2.1 below.)

A typed procedure is called by means of a function reference, which is an operand in an expression and has the form

```
name [(parameter list)]
```

This occurs as an operand in an expression, as in the following example:

```
TOTAL = SUBTOTAL + SUM$ARRAY (.ITEMS, COUNT);
```

where SUM$ARRAY is a previously declared typed procedure.

In both forms of procedure call, the elements of the parameter list are called "actual parameters," to distinguish them from the "formal parameters" of the procedure declaration. At the time of the call, each actual parameter is evaluated, and its resulting value is assigned to the corresponding formal

parameter in the procedure declaration. Then the procedure body is executed. An actual parameter may be any PL/M expression.

If the procedure is declared without a formal parameter list, then no actual parameter list is used in the call. If the procedure declaration does have a formal parameter list, then the actual parameter list must match it — that is, it must contain the same number of parameters.

As in expression evaluation and assignment statements (see Chapter 4), automatic type conversions are performed as necessary in activating and returning from a procedure.

8.2.1 CALLING A PROCEDURE BY ITS ADDRESS

The CALL statement, in the form shown above, calls an untyped procedure by its name. It is also possible to call an untyped procedure by its address. This is done by means of a CALL statement with the form

```
CALL identifier[.member-identifier] [(parameter list)] ;
```

The identifier may not be subscripted. It must be a fully qualified ADDRESS type variable reference, and its value is assumed to be the address of the entry point of the procedure being called.

When a CALL statement that uses the *name* of the procedure is compiled, the compiler checks to make sure that the correct number of parameters is supplied, and performs automatic type conversion on the actual parameters. When a procedure is called by *address*, the compiler does not check the number of parameters or perform type conversion. If the number of parameters is wrong or if an actual parameter is not of the same type as the corresponding formal parameter, the results are unpredictable. (The built-in functions LOW, HIGH, and DOUBLE, described in Section 11.1.3, can be used to force the value of an expression to the desired type.)

8.3 SAMPLE PROGRAM #2

In the sample program of Chapter 7, it is necessary at three different points to move a record — that is, a structure with two members — from one location to another. To do this, we used two assignment statements (one for each member) each time a record was moved. But if each record had contained 30 members, it would have required 30 assignment statements to move one record.

Clearly, a different approach would be needed, and the use of a procedure will solve the problem conveniently. Let us rewrite the sample program of Chapter 7, with an untyped procedure named COPY which will be called each time a record is to be moved.

Such a procedure may be useful in many other situations as well, so we will write a procedure that can move a record of any size up to 128 bytes from one location to another, and declare this procedure PUBLIC so that it can be called from other modules. The record need not be a structure. It may be any collection of up to 128 contiguous bytes.

The procedure takes three parameters:

- SRC$PTR is a pointer to the bytes to be moved (the "source"). Specifically, it is the address of the first byte.

- DST$PTR is a pointer to the "destination." Specifically, it is the address the first byte is to be moved to. The remaining bytes will follow in sequence.

- COUNT is the number of bytes to be moved.

SRC$PTR and DST$PTR are declared as variables, and then the arrays SOURCE and DEST — based on these pointers — are declared. Then COUNT is declared, along with J, which will be used as an index variable, and FINAL, which will be the subscript of the last byte to be moved.

The iterative DO block labeled LOOP moves COUNT number of bytes from the array SOURCE to the array DEST. Because these arrays are based at SRC$PTR and DST$PTR respectively, this is all that is required.

This procedure is useful as an example. However, in an actual program it would be more efficient to use the built-in procedure MOVE (see Section 11.1.5).

The remainder of the sort program is exactly as it appeared in Chapter 7, except that the pairs of assignment statements used to move records are replaced by calls on the procedure COPY. Each call supplies location references for SRC$PTR and DST$PTR, and the constant 3 for COUNT (each record contains 3 bytes).

```
M: DO;      /*Beginning of module*/

      COPY: PROCEDURE (SRC$PTR, DST$PTR, COUNT) PUBLIC;

      /*Moves source record to destination record by treating
      both records as BYTE arrays. COUNT is the number of bytes
      to move (limited to 128), SRC$PTR points to SOURCE, DST$PTR
      points to DEST.*/

         DECLARE (SRC$PTR, DST$PTR) ADDRESS;
         DECLARE (SOURCE BASED SRC$PTR, DEST BASED DST$PTR) (128) BYTE;
         DECLARE (COUNT, J, FINAL) BYTE;

         FINAL = COUNT -1;

LOOP:    DO J = 0 TO FINAL;
            DEST(J) = SOURCE(J);
         END LOOP;

         RETURN;
      END COPY;

      DECLARE RECORD (128) STRUCTURE (
               KEY BYTE,
               INFO ADDRESS);

      DECLARE CURRENT STRUCTURE (
               KEY BYTE,
               INFO ADDRESS);

      DECLARE (J, I) BYTE;

      /*Data is read in to initialize records.*/

SORT: DO J = 1 TO 127;
         CALL COPY (.RECORD(J), .CURRENT, 3);
         I = J;

FIND:    DO WHILE I > 0 AND RECORD(I-1).KEY > CURRENT.KEY;
            CALL COPY (.RECORD(I-1), .RECORD(I), 3);
            I = I-1;
         END FIND;

         CALL COPY (.CURRENT, .RECORD(I), 3);

      END SORT;

      /*Data is written out from records.*/

   END M;   /*End of module*/
```

One further point should be noticed. Within the body of the procedure COPY, a variable named J is declared and referenced. But J is also declared and used as an identifier for a variable outside the procedure, at the outer level of the module. If J refers to the same object in both places, the program will not work correctly: it is easy to see that the call to COPY from within the block labeled SORT would disrupt the use of J as index variable for this block.

In fact, the two uses of J refer to entirely different objects. The rules of scope (given in Chapter 9) guarantee that the scope of the J declared in the body of procedure COPY is restricted to the procedure declaration, while the scope of the J declared at the outer level of the module includes all of the module *except* the procedure declaration. No conflict is possible.

CHAPTER 9
BLOCK STRUCTURE AND SCOPE

PL/M is a "block structured" language. This chapter deals with block structure and scope in programs where the PUBLIC and EXTERNAL attributes are not used — that is, where there is no extended scope. Chapter 10 discusses modules and extended scope.

9.1 BLOCKS

There are two kinds of blocks in PL/M: every DO block is a block, and every procedure declaration is a block. As will be seen in Chapter 10, a PL/M program is made up of modules, and a module is a particular kind of DO block. Thus everything in a PL/M program is part of some block. Any kind of block may be nested within any other kind of block. This nesting creates a multi-level structure of blocks in a typical PL/M program.

As we have seen, each type of block has special properties and uses; but the important *common* property of all blocks is that they control the *scope* of the objects declared in the program.

In order to discuss scope, it will be useful to have the following definitions:

- The *inclusive extent* of a block is everything from the DO or PROCEDURE statement that begins the block to the END statement that terminates it. The DO or PROCEDURE statement and the END statement are included in the inclusive extent of the block. However, any label attached to the DO statement that begins a DO block is *not* in the inclusive extent of the block; it is outside the block. See Figure 1.

- The *exclusive extent* of a block is the inclusive extent of the block *minus* the inclusive extents of all blocks nested inside it. See Figure 2.

In other words, the inclusive extent includes nested blocks, and the exclusive extent excludes them. Notice that the exclusive extent of a block is the same thing as the "outer level" of the block.

9.2 SCOPE

Every object that is declared in a PL/M program has scope. This includes

- Variables
- Procedures
- Labels
- Macros

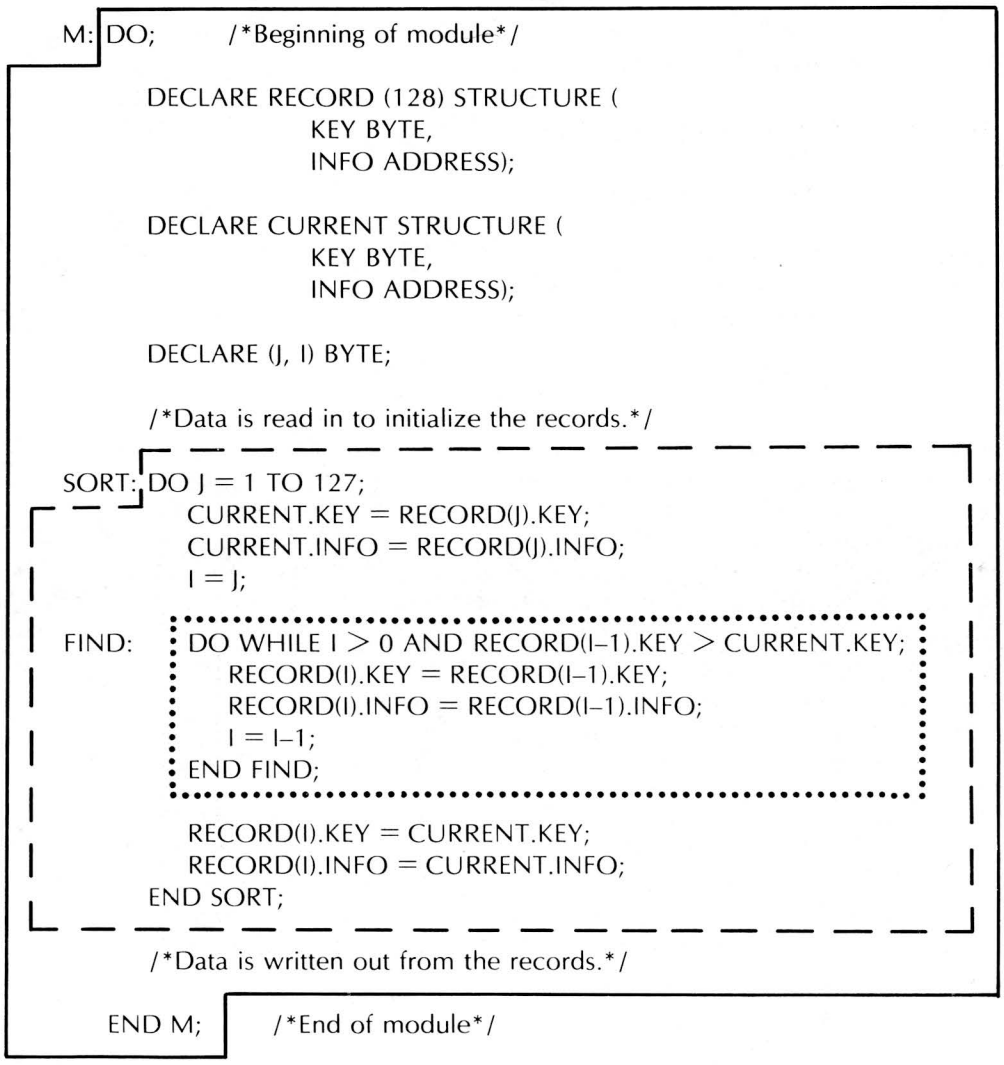


Figure 1: INCLUSIVE EXTENT OF A BLOCK

Everything inside the solid line constitutes the inclusive extent of block M. Everything inside the dashed line constitutes the inclusive extent of block SORT. Everything inside the dotted line constitutes the inclusive extent of block FIND.

```
M: DO;        /*Beginning of module*/

        DECLARE RECORD (128) STRUCTURE (
                        KEY BYTE,
                        INFO ADDRESS);

        DECLARE CURRENT STRUCTURE (
                        KEY BYTE,
                        INFO ADDRESS);

        DECLARE (J, I) BYTE;

        /*Data is read in to initialize the records.*/

SORT:   DO J = 1 TO 127;
            CURRENT.KEY = RECORD(J).KEY;
            CURRENT.INFO = RECORD(J).INFO;
            I = J;

FIND:       DO WHILE I > 0 AND RECORD(I-1).KEY > CURRENT.KEY;
                RECORD(I).KEY = RECORD(I-1).KEY;
                RECORD(I).INFO = RECORD(I-1).INFO;
                I = I-1;
            END FIND;

            RECORD(I).KEY = CURRENT.KEY;
            RECORD(I).INFO = CURRENT.INFO;
        END SORT;

        /*Data is written out from the records.*/

    END M;        /*End of module*/
```

Figure 2: EXCLUSIVE EXTENT OF A BLOCK

The shaded area is the exclusive extent of block SORT.

The scope of an object is defined as follows:

- The scope of an object is the part of the program in which the object's identifier is recognized and handled according to its declaration.

- The declaration of an object is in the exclusive extent of some block. The scope of the object is the inclusive extent of this block, *minus* the inclusive extent of any nested block(s) in which the same identifier is declared.

- The scope of variables, non-reentrant procedures, and macros is restricted: They may not be referred to until after they are declared. The restriction does not apply to reentrant procedures (see Section 8.1.7) or to labels.

The effect of this is that when writing a block, one does not need to worry about inadvertently using an identifier that is already in use elsewhere in the program.

Suppose that we are writing a block called NEWBLOCK, and we declare a variable (or other object) called VBL. Now, if VBL is also declared in some other block called OLDBLOCK, what will happen?

If NEWBLOCK is not nested inside OLDBLOCK, then it is outside the scope of the VBL declared in OLDBLOCK, since the scope of the old VBL cannot go beyond the inclusive extent of OLDBLOCK. Therefore the VBL declared in NEWBLOCK is a *different* VBL from the one declared in OLDBLOCK, just as if a different identifier had been used.

If NEWBLOCK is nested inside OLDBLOCK, then it "interrupts" the scope of the old VBL. The scope of the old VBL will now be the inclusive extent of OLDBLOCK *minus* the inclusive extent of NEWBLOCK. Again, the VBL declared in NEWBLOCK is an entirely different object from the VBL declared in OLDBLOCK.

Now let us consider the reverse situation. Suppose that in NEWBLOCK we want to reference a variable XYZ declared in OLDBLOCK, and have it be the *same* variable.

If NEWBLOCK is nested inside OLDBLOCK, we merely reference XYZ *without* declaring it in NEWBLOCK. The reference will thus be within the scope of the XYZ declared in OLDBLOCK.

If NEWBLOCK is not nested inside OLDBLOCK, there is no way to reference the XYZ declared in OLDBLOCK. The program must be rearranged, either by moving NEWBLOCK so as to nest it inside OLDBLOCK or by nesting both OLDBLOCK and NEWBLOCK inside another block and moving the declaration of XYZ into this outer block.

9.3 SCOPE OF LABELS AND RESTRICTIONS ON GOTOs

Labels are subject to exactly the same rules of scope just given for other objects.

Let us reexamine the definition (Section 9.1) of the inclusive extent of a block. Note that the inclusive extent of a DO block does *not* include any label(s) attached to the DO statement that begins the block. This means that if the same identifier used as a label on the DO statement is used *within* the block to declare a label (implicitly or explicitly), this will interrupt the scope of the label on the DO statement and constitute a different label.

This has an important effect on the label of a DO statement that begins a module (modules are defined in Chapter 10):

- It is not possible to explicitly declare the label of a module. This means that the label of a module may not have the PUBLIC or EXTERNAL attributes.

Moreover, the implicit declaration of labels causes some special effects.

As explained in Section 6.3.1, a label definition implicitly declares the label — unless the label has already been declared explicitly in the exclusive extent of the same block. An implicit label declaration may occur anywhere in the block structure of a program, whereas explicit declarations are limited to simple DO blocks and procedure declarations, and may only appear before the first executable statement in the block.

The rules of scope guarantee that the scope of an implicit declaration is exactly the same as the scope of an explicit declaration at the beginning of the smallest block that encloses the implicit declaration. This means that wherever a labeled statement appears, the scope of the label cannot extend beyond the smallest enclosing block.

This leads to certain important restrictions:

- It is not possible for a GOTO to transfer control from an outer block to a labeled statement in a nested block.

- Moreover, it is not possible for a GOTO to transfer control from one block to *any* block (in the same module) that does not enclose the block containing the GOTO.

In addition, there is the following restriction:

- Any label with the PUBLIC attribute must be attached to an executable statement at the outer level of the main program module.

In fact, the only possible GOTO transfers are the following:

- From one point in the exclusive extent of a block to a statement in the exclusive extent of the same block.

- From an inner block to a statement in the exclusive extent of an enclosing block (not necessarily the smallest enclosing block). However, if the inner block is a procedure block, the transfer may only be to a labeled statement in the outer level of the main program module as stated in Section 8.1.3.

- From any point in one module, in which the label is declared EXTERNAL, to a statement in the outer level of the main program module, in which the label is declared PUBLIC.

CHAPTER 10
PROGRAM MODULES

10.1 DEFINITIONS

In preceding chapters, we have referred to "modules," the " outer level of a module," and the "main program module." The precise definitions of these terms are as follows:

- A *module* is a labeled simple DO block which is not nested in any other block.
- The "outer level of a module" or *module level* is the exclusive extent (see Chapter 9) of a module.
- A *main program module* is a module that contains executable statements at the module level.

10.2 MODULAR STRUCTURE OF A COMPILATION

A "compilation" is one module of PL/M statements. A compilation is not necessarily a complete program. After compilation, the module may be linked with modules from different compilations to build up a program, as described below.

The number of DO blocks and the number of procedure declarations in a module are limited by the PL/M-80 Compiler. See *ISIS-II PL/M-80 Compiler Operator's Manual.*

10.3 MODULAR STRUCTURE OF A PROGRAM

A program is created by means of the linker, using compiled and/or assembled modules as building blocks. A program is built up from one or more modules, including a main program module.

10.4 LINKAGE

The compiled modules produced by the PL/M-80 Compiler are relocatable code. Some of the identifiers have extended scope — those declared with the EXTERNAL and PUBLIC attributes. The references to identifiers declared EXTERNAL need to be associated with the defining declarations. This association of EXTERNAL and PUBLIC identifiers is called "linkage."

To create a program with the linker, one specifies the modules making up the program, in the desired sequence. In choosing this set of modules, the following restrictions must be borne in mind:

- The set of modules must contain a main program module.
- Each identifier with extended scope must have exactly one defining declaration in the total set of modules — that is, exactly one declaration with the PUBLIC attribute.

The linker combines the modules, "satisfying" all references to objects that are declared with the EXTERNAL attribute by associating them with the defining declarations. The effect is to extend the scope of each object declared with the PUBLIC attribute to include the scope of each object (in

another module) that has the EXTERNAL attribute and the same identifier.

This results in a complete relocatable program in which every variable reference, procedure call, and label reference is meaningful.

10.5 EXAMPLE OF MODULAR PROGRAM STRUCTURE

Consider once again the sort program used in Chapter 7 and Section 8.3. This program contains a procedure which might be useful in other programs. Therefore, it might be desirable to compile this procedure as a separate module, so that it can be linked to this sort program and also to any other program that needs to move structures.

Broken into two modules, our example program appears as follows.

```
COPY$MODULE:
  DO;       /*Beginning of module with COPY procedure*/

     COPY: PROCEDURE (SRC$PTR, DST$PTR, COUNT) PUBLIC;

     /*Moves source record to destination record by treating
     both records as BYTE arrays. COUNT is the number of bytes
     to move (limited to 128), SRC$PTR points to SOURCE, DST$PTR
     points to DEST.*/

        DECLARE (SRC$PTR, DST$PTR) ADDRESS;
        DECLARE (SOURCE BASED SRC$PTR, DEST BASED DST$PTR) (128) BYTE;
        DECLARE (COUNT, J, LAST) BYTE;

        LAST = COUNT -1;

LOOP:   DO J = 0 TO LAST;
           DEST(J) = SOURCE(J);
        END LOOP;

        RETURN;
     END COPY;

  END COPY$MODULE; /*End of COPY module*/
```

This module is compiled and can then be kept available for use by any program that is linked to it. The main program module is on the next page. It is the same as the program of Section 8.3, but a usage declaration of the COPY procedure has been substituted for the defining declaration, which is now in the above module.

```
SORT$PROGRAM:
  DO;           /*Beginning of main program module*/

      COPY: PROCEDURE (SRC$PTR, DST$PTR, COUNT) EXTERNAL;

      /*Usage declaration of COPY procedure, which is
      defined in module COPY$MODULE*/

          DECLARE (SRC$PTR, DST$PTR) ADDRESS;
          DECLARE COUNT BYTE;
      END COPY;


      DECLARE RECORD (128) STRUCTURE (
                KEY BYTE,
                INFO ADDRESS);

      DECLARE CURRENT STRUCTURE (
                KEY BYTE,
                INFO ADDRESS);

      DECLARE (J, I) BYTE;

      /*Data is read in to initialize records.*/

SORT: DO J = 1 TO 127;
          CALL COPY (.RECORD(J), .CURRENT, 3);
          I = J;

FIND:     DO WHILE I > 0 AND RECORD(I-1).KEY > CURRENT.KEY;
              CALL COPY (.RECORD(I-1), .RECORD(I), 3);
              I = I-1;
          END FIND;

          CALL COPY (.CURRENT, .RECORD(I), 3);

      END SORT;

      /*Data is written out from records.*/

  END SORT$PROGRAM; /*End of main program module*/
```

CHAPTER 11
BUILT-IN PROCEDURES AND PREDECLARED VARIABLES

Built-in procedures and predeclared variables act as if they were declared in an all-encompassing global block invisible to the programmer.

The identifiers are subject to the rules of scope. This means that if the identifier of a built-in procedure or predeclared variable is used in a declaration within the program, the scope of the predeclared variable or built-in procedure is interrupted by the scope of the declaration in the program. Note that this distinguishes these identifiers from reserved words, which cannot be used as identifiers in declarations.

11.1 BUILT-IN PROCEDURES

11.1.1 INPUT PROCEDURE

INPUT is a BYTE procedure. It is called by a function reference, with the form

```
INPUT (numeric constant)
```

Notice that the actual parameter must be a numeric constant — not an expression. The constant must be in the range from 0 to 255 to specify one of the 256 input ports of the 8080 CPU. The value returned by INPUT is the BYTE quantity latched in the specified input port.

Output is performed by using the predeclared variable OUTPUT; see Section 11.2.1.

11.1.2 LENGTH, LAST, AND SIZE PROCEDURES

PL/M has three built-in procedures that take variable names as actual parameters and return information based on the declarations of the variables.

LENGTH

LENGTH is a typed procedure whose type depends on the value that it calculates. If this value is less than or equal to 255, it is returned as a BYTE value; otherwise it is returned as an ADDRESS value. It is called by a function reference, with the form

```
LENGTH (identifier)
```

where the identifier must be a *non-subscripted* reference to an array. The array may be a member of a structure; it may not be the MEMORY array (see Section 11.2.2).

The BYTE or ADDRESS value returned is the number of elements in the array — that is, it is equal to the dimension specifier in the array declaration.

If the array is not a structure member, then the reference is an unqualified variable reference. If the array is a structure member, then the reference is a partially qualified variable reference (see Section

3.6.2). For example, given the declaration

```
DECLARE RECORD STRUCTURE (
    KEY BYTE,
    INFO (3) ADDRESS);
```

then LENGTH(RECORD.INFO) is a valid function reference and returns the value 3.

If the array is a member of a structure, and the structure is an element of an array, a special case arises. Given the declaration

```
DECLARE LIST (4) STRUCTURE (
    KEY BYTE,
    INFO (3) ADDRESS);
```

then all of the following function references are correct and return the value 3:

```
LENGTH(LIST(0).INFO)
LENGTH(LIST(1).INFO)
LENGTH(LIST(2).INFO)
LENGTH(LIST(3).INFO)
```

In other words, the subscript for the array LIST is irrelevant, since the arrays within the structures are all the same length. Therefore, PL/M allows a "shorthand" form of partially qualified variable reference in the LENGTH function reference:

```
LENGTH(LIST.INFO)
```

This is an exception to the rule given in Section 3.6.2, which says that the subscript is required in a case like this. The same exception applies to the LAST and SIZE built-in procedures described below.

LAST

LAST is a typed procedure whose type depends on the value that it calculates. If this value is less than or equal to 255, it is returned as a BYTE value; otherwise it is returned as an ADDRESS value. It is called by a function reference, with the form

```
LAST (identifier)
```

where the identifier must be a *non-subscripted* reference to an array. The array may be a member of a structure; it may not be the MEMORY array (see Section 11.2.2).

The BYTE or ADDRESS value returned is the subscript of the last element of the array — note that for a given array, LAST will always be one less than LENGTH.

As in the LENGTH procedure, a "shorthand" form of partially qualified variable reference is allowed in the case where the array is a member of a structure and the structure is an array element.

SIZE

SIZE is an ADDRESS procedure. It is called by a function reference, with the form

SIZE (variable-ref)

where the variable-ref is a fully qualified, partially qualified, or unqualified reference to any scalar, array, or structure variable except the MEMORY array (see Section 11.2.2).

The ADDRESS value returned is the number of bytes required by the object referenced.

If the reference is fully qualified, it refers to a scalar and the value is 1 for a BYTE scalar or 2 for an ADDRESS scalar. If the reference is unqualified, it refers to an entire structure or array, and the value is the total number of bytes required for the structure or array.

If the reference is partially qualified, it refers either to a structure member which is an array, or to an array element which is a structure. The value is the number of bytes required for the array or structure.

As in the LENGTH procedure, a "shorthand" form of partially qualified variable reference is allowed in the case where the array or scalar is a member of a structure and the structure is an array element.

11.1.3 LOW, HIGH, AND DOUBLE PROCEDURES

Two built-in BYTE procedures convert ADDRESS values to BYTE values. Calls to these procedures are function references with the following forms:

LOW (expression)
HIGH (expression)

If the expression has an ADDRESS value, LOW returns the low-order (least significant) byte of the value, whereas HIGH returns the high-order (most significant) byte of the value.

If the expression has a BYTE value, then LOW will return this value unchanged. However, HIGH will return zero.

The ADDRESS procedure DOUBLE converts a BYTE value to an ADDRESS value. A call to DOUBLE is a function reference with the form

DOUBLE (expression)

If the expression has a BYTE value, the procedure appends 8 high-order zeros to convert it to an ADDRESS value and returns this ADDRESS value. If the expression has an ADDRESS value, the procedure returns this value unchanged.

11.1.4 SHIFTS AND ROTATIONS

In shift and rotate operations, a value is handled as a pattern of 8 bits (for a BYTE value) or 16 bits (for an ADDRESS value). The pattern is moved to the right or left by a specified number of bits called the "bit count."

In a shift, bits moved off one end of the pattern are lost, and zeros move into the pattern from the other end. In a rotate, bits moved off one end move onto the other end.

Byte Rotation Procedures

ROL and ROR are BYTE procedures. They are called by function references, with the forms

```
ROL (pattern, count)
ROR (pattern, count)
```

where "pattern" and "count" are both expressions. The values of these expressions are converted, if necessary, to BYTE values. The first parameter is handled as an 8-bit pattern which is rotated to the left (by ROL) or to the right (by ROR). The bit count is given by the second parameter. If the value of this expression is 0, the result is undefined.

The following are examples of the action of these procedures:

ROR(10011101B, 1) returns a value of 11001110B.

ROL(10011101B, 2) returns a value of 01110110B.

Logical-Shift Procedures

SHL and SHR are procedures whose type depends on the type of the value of an expression given as an actual parameter. They are called by function references, with the forms

```
SHL (pattern, count)
SHR (pattern, count)
```

where "pattern" and "count" are both expressions.

The value of count will be converted, if necessary, to a BYTE quantity. If the value of count is zero, the result is undefined.

The value of the pattern may be either a BYTE value or an ADDRESS value and will not be converted. If it is a BYTE value, then the procedure will return a BYTE value. If it is an ADDRESS value, then the procedure will return an ADDRESS value.

The value of the first parameter (pattern) is shifted left (by SHL) or right (by SHR), with the bit count given by the second parameter (count).

11.1.5 THE MOVE PROCEDURE

The untyped procedure MOVE is used to transfer a set of contiguous bytes of information from one location in memory to another. The form of the call is:

```
CALL MOVE (count, source, destination) ;
```

where count, source, and destination are all expressions and will be converted, if necessary, to ADDRESS values.

The count parameter is the number of bytes of information to be moved. The source parameter is the memory address of the first byte to be moved, and the destination parameter is the memory address to which this byte is to be moved. Subsequent bytes are taken from subsequent addresses following source and moved to subsequent addresses following destination.

If the source and destination areas of memory overlap, the result is undefined.

11.1.6 THE TIME PROCEDURE

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

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

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

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

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

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

The TIME procedure is based on 8080 CPU cycle times, and assumes that the system is running at 2 MHz without interruptions.

11.2 PREDECLARED VARIABLES

11.2.1 THE OUTPUT ARRAY

The predeclared variable OUTPUT is a BYTE array with 256 elements. Each element corresponds to one of the 256 output ports of the 8080 CPU.

A reference to OUTPUT must always be subscripted with a numeric constant in the range from 0 to 255 and may only appear as the *left* part of an assignment statement or embedded assignment. Anywhere else it is illegal. The effect of such an assignment is to latch the BYTE value of the expression on the right side of the assignment into the specified output port. (Since OUTPUT is a BYTE array, the value of the expression will be automatically converted to type BYTE if necessary.)

11.2.2 THE MEMORY ARRAY

MEMORY is a BYTE array of *unspecified* length which represents the free memory space allocated by the locator. References to MEMORY may be subscripted. The maximum subscript allowed depends on both the system environment and the program. References to MEMORY, either subscripted or unqualified, may be used in location references. Thus, for example, .MEMORY is the address of the beginning of free memory space.

A reference to MEMORY may not be used as an actual parameter for the LENGTH, LAST, and SIZE procedures.

11.2.3 STACKPTR

STACKPTR is a predeclared ADDRESS variable that provides access to the 8080 hardware stack pointer register.

Care must be exercised in setting this register (that is, using STACKPTR on the left side of an assignment). Taking control of the stack away from the compiler frustrates the compile-time checks on stack overflow and invalidates the compiler's assumptions about the run-time status of the stack.

CHAPTER 12
PL/M FEATURES INVOLVING 8080 HARDWARE FLAGS

The PL/M features described in this chapter make use, directly or indirectly, of the 8080 hardware flags or "toggles" — CARRY, ZERO, SIGN, and PARITY. As explained in the following section, these features cannot be guaranteed to produce correct results and the programmer should use them only with caution.

Instead of using these features, it may be more convenient to link the PL/M program to modules containing code to perform the same functions, but written in assembly language.

12.1 OPTIMIZATION AND THE 8080 HARDWARE FLAGS

In order to produce an efficient machine-code program from a PL/M source, the PL/M-80 Compiler performs extensive optimization of the machine code. This means that the exact sequence of machine code produced to implement a given sequence of PL/M source statements cannot be predicted.

Consequently, the state of the 8080 hardware flags cannot be predicted for any given point in the program. For example, suppose that a source program contains the following fragment:

```
...

SUM = SUM + 250;

...
```

where SUM is a BYTE variable. Now, if the value of SUM before this assignment statement is greater than 5, the addition will cause an overflow and the hardware CARRY flag will be set.

If there were no optimization of the machine code, one could follow this assignment statement with one of the PL/M features described in the following sections, and be sure that the feature would operate in a certain fashion depending on whether or not the addition caused the CARRY flag to be set. However, because of optimization, some machine code instructions may occur immediately after the addition, and change the CARRY flag. One cannot safely predict whether this will happen or not.

Accordingly, any PL/M feature that involves the CARRY flag (or any of the other hardware flags) may cause the program to run incorrectly. These features must therefore be used with caution, and any program that uses them must be checked carefully to make sure that it operates correctly.

12.2 THE "PLUS" AND "MINUS" OPERATORS

In addition to the arithmetic operators described in Section 4.2, there are two more: PLUS and MINUS.

PLUS and MINUS perform similarly to + and –, and have the same precedence. However, they take account of the current setting of the 8080 CPU hardware CARRY flag in performing the operation.

12.3 CARRY-ROTATION PROCEDURES

SCL and SCR are built-in procedures whose type depends on the type of the value of an expression given as an actual parameter. They are called by function references, with the forms

```
SCL (pattern, count)
SCR (pattern, count)
```

where "pattern" and "count" are both expressions.

The value of count will be converted, if necessary, to a BYTE quantity. If the value of count is zero, the result is undefined.

The value of the pattern may be either a BYTE value or an ADDRESS value and will not be converted. If it is a BYTE value, then the procedure will return a BYTE value. If it is an ADDRESS value, then the procedure will return an ADDRESS value.

The value of the first parameter (pattern) is rotated left (by SCL) or right (by SCR), with the bit count given by the second parameter (count), just as with the ROL and ROR procedures described in Chapter 11. But with SCL and SCR, the rotation includes the CARRY flag: the bit rotated off one end of the argument is rotated into CARRY, and the old value of CARRY is rotated into the other end of the argument. In effect, SCL and SCR perform 9-bit rotations on 8-bit values, and 17-bit rotations on 16-bit values.

12.4 THE DEC PROCEDURE

DEC is a built-in BYTE procedure which uses the value of the hardware CARRY flag internally. It is called by a function reference, with the form

```
DEC (expression)
```

where the value of the expression will be converted, if necessary, to a BYTE value. The procedure performs a decimal adjust operation on the actual parameter value and returns the result of this operation.

12.5 CARRY, SIGN, ZERO, AND PARITY PROCEDURES

There are four built-in BYTE procedures that return the logical values of the 8080 hardware flags. These procedures take no parameters, and are called by function references with the following forms:

```
CARRY
ZERO
SIGN
PARITY
```

An occurrence of one of these calls (in an expression) generates a test of the corresponding condition flag. If the flag is set (= 1), a value of 0FFH is returned. If the flag is clear (= 0), a value of 0 is returned.

Appendix A
GRAMMAR OF THE PL/M LANGUAGE

This appendix lists the entire BNF syntax of the PL/M language. Since the semantic rules are not included here, this syntax permits certain constructions that are not actually allowed. Also, the terminology used in this BNF syntax has been designed for convenience in constructing concise and rigorous definitions. In some cases, this terminology differs from the terminology used in the main body of manual.

The notation used here is slightly extended from standard BNF. A sequence of three periods (...) is used to indicate that the preceding syntactic element may be repeated any number of times. Curly brackets are used to indicate that exactly one of the items stacked vertically between them is to be used. Square brackets indicate that whatever is between them may be omitted. Also, when items are stacked vertically between square brackets, only one of them may be used, if any.

Following the syntax, the nonterminals in the syntax are listed in alphabetical order. Each nonterminal is tagged with the section number (within this appendix) where its primary definition can be looked up.

SYNTAX

A.1 LEXICAL ELEMENTS

A.1.1 Character Sets

```
<character>::= <apostrophe>
             ¦  <non-quote character>
<non-quote character>::= <letter>
                       ¦  <decimal digit>
                       ¦  $
                       ¦  <special character>
                       ¦  <blank>
<letter>::= <upper case letter>
          ¦  <lower case letter>
<upper case letter>::= 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
<lower case letter>::= 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
<decimal digit>::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9
<special character>::= +¦-¦*¦/¦<¦>¦=¦:¦;¦.¦,¦(¦)
<apostrophe>::= '
```

A.1.2 Tokens

```
<token>::= <delimiter>
         ¦ <identifier>
         ¦ <reserved word>
         ¦ <numeric constant>
         ¦ <string>
```

A.1.3 Delimiters

```
<delimiter>::= <simple delimiter>
             ¦ <compound delimiter>
<simple delimiter>::= +¦-¦*¦/¦<¦>¦=¦:¦;¦.¦,¦(¦)
<compound delimiter>::= <>
                      ¦ <=
                      ¦ >=
                      ¦ :=
```

A.1.4 Identifiers and Reserved Words

```
<identifier>::= <letter> [ <letter>          ] ...
                         [ <decimal digit>   ]
                         [ $                 ]
```

A.1.5 Numeric Constants

```
<numeric constant>::= <binary number>
                    ¦ <octal number>
                    ¦ <decimal number>
                    ¦ <hexadecimal number>

<binary number>::= <binary digit> [ <binary digit> ] ... B
                                  [       $        ]

<octal number>::= <octal digit> [ <octal digit> ] ... { O }
                                [       $       ]     { Q }

<decimal number>::= <decimal digit> [ <decimal digit> ] ... [D]
                                    [       $         ]

<hexadecimal number>::= <decimal digit>
                                  [ <hexadecimal digit> ] ... H
                                  [         $           ]
```

```
<binary digit>::= 0¦1
<octal digit>::= <binary digit> ¦2¦3¦4¦5¦6¦7
<decimal digit>::= <octal digit> ¦8¦9
<hexadecimal digit>::= <decimal digit>¦A¦B¦C¦D¦E¦F
```

A.1.6 Strings

```
<string>::= ' <string body element> ... '
<string body element>::= <printable ASCII character>
                       ¦  ''
```

A.1.7 PL/M Text Structure: Tokens, Blanks, and Comments

```
<pl/m text>::= [<token>    ] ...
               [<separator>]

<separator>::= <blank>
             ¦  <comment>
<comment>::= /* [<character>] ... */
```

A.2 Modules and Compilations

```
<compilation>::= <module> [EOF]
<module>::= <module name> : <simple do block>
<module name>::= <identifier>
```

A.3 DECLARATIONS

```
<declaration>::= <declare statement>
               ¦  <procedure definition>
```

A.3.1 DECLARE Statement

```
<declare statement>::= DECLARE <declare element list>;
<declare element list>::= <declare element>[,<declare element>]...
<declare element>::= <factored element>
                   ¦  <unfactored element>

<unfactored element>::= <variable element>
                      ¦  <literal element>
                      ¦  <label element>

<factored element>::= <factored variable element>
                    ¦  <factored label element>
```

A.3.2 Variable Elements

```
<variable element>::= <variable name specifier>
                      [<dimension specifier>] <variable type>
                      [<variable attributes>][<initialization>]

<variable name specifier>::= <non-based name>
                           ¦  <based name> BASED <base specifier>
<non-based name>::= <variable name>
<based name>::= <variable name>
<variable name>::= <identifier>
<base specifier>::= <identifier> [.<identifier>]

<variable attributes>::= [PUBLIC] [<locator>]
                         ¦  [EXTERNAL]
<locator>::= AT (<restricted expression>)

<dimension specifier>::= <explicit dimension>
                         ¦  <implicit dimension>
<explicit dimension>::= ( <numeric constant> )
<implicit dimension>::= ( * )

<variable type>::= <basic type>
                   ¦  <structure type>
<basic type>::= ADDRESS
                ¦  BYTE
```

A.3.3 Label Element

```
<label element>::= <identifier> LABEL [PUBLIC  ]
                                      [EXTERNAL]
```

A.3.4 Literal Elements

```
<literal element>::= <identifier> LITERALLY <string>
```

A.3.5 Factored Variable Element

```
<factored variable element>::= ( <variable name specifier>
          [,<variable name specifier>]...)
          [<explicit dimension>] <variable type>
          [<variable attributes>][<initialization>]
```

A.3.6 Factored Label Elements

```
<factored label element>::= (<identifier>
          [,<identifier>]...) LABEL [PUBLIC  ]
                                    [EXTERNAL]
```

A.3.7 The Structure Type

```
<structure type>::= STRUCTURE (
         <member element> [,<member element>]...)
<member element>::= <member name>
            [<explicit dimension>] <basic type>
<member name>::= <identifier>
```

A.3.8 Procedure Definition

```
<procedure definition>::= <procedure statement>
         [<declaration>...][<unit>...] <ending>

<procedure statement>::= <procedure name> : PROCEDURE
      [<formal parameter list>][<procedure type>]
      [<procedure attributes>];
<procedure name>::= <identifier>
<procedure type>::= <basic type>
<basic type>::= ADDRESS
              | BYTE
<formal parameter list>::= ( <formal parameter>
      [,<formal parameter>]... )
<formal parameter>::= <identifier>
<procedure attributes>::= {INTERRUPT <numeric constant>} ...
                          {<linkage>                    }
                          {REENTRANT                    }
<linkage>::= PUBLIC
           | EXTERNAL
```

A.3.9 Attributes

A.3.9.1 AT

```
<locator>::= AT ( <restricted expression> )
```

A.3.9.2 INTERRUPT

```
<interrupt>::= INTERRUPT <numeric constant>
```

A.3.9.3 Initialization

```
<initialization>::= {INITIAL} (<initial value> [,<initial value>] ...)
                    {DATA   }
<initial value>::= <restricted expression>
                 ¦  <string>
```

A.4 UNITS

```
<unit>::= <conditional clause>
        ¦  <do block>
        ¦  <basic statement>
        ¦  <label definition> <unit>
<basic statement>::= <assignment statement>
                   ¦  <call statement>
                   ¦  <goto statement>
                   ¦  <null statement>
                   ¦  <return statement>
                   ¦  <8080 dependent statement>
<scoping statement>::= <simple do statement>
                     ¦  <do-case statement>
                     ¦  <do-while statement>
                     ¦  <iterative do statement>
                     ¦  <end statement>
                     ¦  <procedure statement>
<label definition> ::= <identifier> :
```

A.4.1 Basic Statements

A.4.1.1 Assignment Statement

```
<assignment statement>::= <left part> = <expression> ;
<left part>::= <variable reference> [, <variable reference>] ...
```

A.4.1.2 CALL Statement

```
<call statement>::= CALL <simple variable> [<parameter list>];
<parameter list>::= (<expression> [, <expression>] ... )
<simple variable>::= <identifier>
                   ¦  <identifier> . <identifier>
```

A.4.1.3 GOTO Statement

```
<goto statement>::= { GOTO  } <identifier> ;
                    { GO TO }
```

A.4.1.4 Null Statement

```
<null statement>::= ;
```

A.4.1.5 RETURN Statement

```
<return statement>::= <typed return>
                    ¦  <untyped return>
<typed return>::= RETURN <expression> ;
<untyped return>::= RETURN ;
```

A.4.1.6 8080 Dependent Statements

```
<8080 dependent statement>::= <disable statement>
                            ¦  <enable statement>
                            ¦  <halt statement>
<disable statement>::= DISABLE ;
<enable statement>::= ENABLE ;
<halt statement>::= HALT ;
```

A.4.2 Scoping Statements

A.4.2.1 Simple DO Statement

```
<simple do statement>::= DO ;
```

A.4.2.2 DO-CASE Statement

```
<do-case statement>::= DO CASE <expression> ;
```

A.4.2.3 DO-WHILE Statement

```
<do-while statement>::= DO WHILE <expression> ;
```

A.4.2.4 Iterative DO Statement

```
<iterative do statement>::= DO <index part><to part>[<by part>] ;
<index part>::= <index variable> = <start expression>
<to part>::= TO <bound expression>
<by part>::= BY <step expression>
<index variable>::= <simple variable>
<start expression>::= <expression>
<bound expression>::= <expression>
<step expression>::= <expression>
```

A.4.2.5 END Statement

```
<end statement>::= END [<identifier>] ;
```

A.4.2.6 Procedure Statement

```
<procedure statement>::= <procedure name> : PROCEDURE
      [<formal parameter list>][<procedure type>]
      [<procedure attributes>];
<procedure name>::=<identifier>
<procedure type>::= <basic type>
<basic type>::= ADDRESS
              | BYTE
<formal parameter list>::= ( <formal parameter>
     [,<formal parameter>]... )
<formal parameter>::= <identifier>
<procedure attributes>::= { INTERRUPT <numeric constant> } ...
                          { <linkage>                     }
                          { REENTRANT                     }
<linkage>::= PUBLIC
           | EXTERNAL
```

A.4.3 Conditional Clause

```
<conditional clause>::= <if condition> <true unit>
                      | <if condition> <true element> ELSE
                            <false element>
<if condition>::= IF <expression> THEN
<true element>::= [<label definition> ...] <do block>
                | [<label definition> ...] <basic statement>
<false element>::= <unit>
<true unit>::= <unit>
```

A.4.4 DO Blocks

```
<do block>::= <simple do block>
            ¦ <do-case block>
            ¦ <do-while block>
            ¦ <iterative do block>
```

A.4.4.1 Simple DO Blocks

```
<simple do block>::= <simple do statement>
                     [<declaration>...][<unit>...]<ending>
<ending>::=[<label definition>...]<end statement>
```

A.4.4.2 DO-CASE Blocks

```
<do-case block>::= <do-case statement>{<unit>...}<ending>
```

A.4.4.3 DO-WHILE Blocks

```
<do-while block>::= <do-while statement>[<unit>...]<ending>
```

A.4.4.4 Iterative DO Blocks

```
<iterative do block>::=<iterative do statement>[<unit>...]<ending>
```

A.5 EXPRESSIONS

A.5.1 Primaries

```
<primary>::= <constant>
           ¦ <variable reference>
           ¦ <location reference>
           ¦ <subexpression>
<subexpression>::= ( <expression> )
```

A.5.1.1 Constants

```
<constant>::= <numeric constant>
            ¦ <string>
```

A.5.1.2 Variable References

```
<variable reference>::= <data reference>
                      ¦  <function reference>
<data reference>::= <name>[<subscript>][<member specifier>]
<subscript>::= ( <expression> )
<member specifier>::= .<member name>[<subscript>]
<function reference>::= <name>[<actual parameters>]
<actual parameters>::= ( <expression>[,<expression>]...)
<member name>::= <identifier>
<name>::= <identifier>
```

A.5.1.3 Location References

```
<location reference>::= .<constant list>
                      ¦  .<variable reference>
<constant list>::= (<constant>[,<constant>]...)
```

A.5.2 Operators

```
<operator>::= <logical operator>
            ¦  <relational operator>
            ¦  <arithmetic operator>
<logical operator>::= AND
                    ¦  OR
                    ¦  NOT
                    ¦  XOR
<relational operator>::= < ¦ > ¦ <= ¦ >= ¦ <> ¦ =
<arithmetic operator>::= + ¦ - ¦ PLUS ¦ MINUS¦ * ¦ / ¦ MOD
```

A.5.3 Structure of Expressions

```
<expression>::= <logical expression>
              ¦  <embedded assignment>
<embedded assignment>::= <variable reference> :=
                                        <logical expression>
<logical expression>::= <logical factor>
    ¦ <logical expression> <or operator> <logical factor>
<or operator>::= OR
               ¦  XOR
<logical factor>::= <logical secondary>
    ¦ <logical factor> <and operator> <logical secondary>
<and operator>::= AND
<logical secondary>::= [<not operator>] <logical primary>
<not operator>::= NOT
<logical primary>::= <arithmetic expression>
      [<relational operator> <arithmetic expression>]
<relational operator>::= < ¦ > ¦ <= ¦ >= ¦ <> ¦ =
```

```
<arithmetic expression>::= <term>
      ¦ <arithmetic expression> <adding operator> <term>
<adding operator>::= + ¦ - ¦ PLUS ¦ MINUS
<term>::= <secondary>
       ¦ <term> <multiplying operator> <secondary>
<multiplying operator>::= * ¦ / ¦ MOD
<secondary>::= [<unary minus>] <primary>
<unary minus>::= -
```

A.5.4 Restricted Expressions

```
<restricted expression>::= [<restricted reference>
         <restricted adding operator>] <restricted sum>
<restricted reference>::= .<identifier> [<restricted subscript>]
         [.<identifier> [<restricted subscript>]]
<restricted subscript>::= ( <restricted sum> )
<restricted sum>::= <restricted secondary>
         [<restricted adding operator>
         <restricted secondary>] ...
<restricted adding operator>::= + ¦ -
<restricted secondary>::= [<restricted negation operator>]
         <restricted secondary> ¦ <restricted primary>
<restricted negation operator>::= -
<restricted primary>::= <numeric constant>
```

NONTERMINALS	SECTION #
<actual parameters>	A.5.1.2
<adding operator>	A.5.3
<and operator>	A.5.3
<apostrophe>	A.1.1
<arithmetic expression>	A.5.3
<arithmetic operator>	A.5.2
<assignment statement>	A.4.1.1
<base specifier>	A.3.2
<based name>	A.3.2
<basic statement>	A.4
<basic type>	A.3.2
<binary digit>	A.1.5
<binary number>	A.1.5
<bound expression>	A.4.2.4
<by part>	A.4.2.4
<call statement>	A.4.1.2
<character>	A.1.1
<comment>	A.1.7
<compilation>	A.2
<compound delimiter>	A.1.3
<conditional clause>	A.4.3
<constant list>	A.5.1.3
<constant>	A.5.1.1
<data reference>	A.5.1.2
<decimal digit>	A.1.5
<decimal number>	A.1.5
<declaration>	A.3
<declare element list>	A.3.1
<declare element>	A.3.1
<declare statement>	A.3.1
<delimiter>	A.1.3
<dimension specifier>	A.3.2
<disable statement>	A.4.1.6
<do block>	A.4.4
<do-case block>	A.4.4.2
<do-case statement>	A.4.2.2
<do-while block>	A.4.4.3
<do-while statement>	A.4.2.3
<embedded assignment>	A.5.3
<enable statement>	A.4.1.6
<end statement>	A.4.2.5
<ending>	A.4.4.1
<explicit dimension>	A.3.2
<expression>	A.5.3
<factored element>	A.3.1
<factored label element>	A.3.6
<factored variable element>	A.3.5
<false element>	A.4.3
<formal parameter>	A.3.8

Appendix B
ASCII CODES

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

00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Appendix C
PL/ M SPECIAL CHARACTERS

SYMBOL	*NAME*	*USE*
$	dollar sign	number and identifier spacer
=	equal sign	Two distinct uses: (1) relational test operator (2) assignment operator
: =	assign	embedded assignment operator
.	dot	Two distinct uses: (1) address operator (2) structure member qualification
/	slash	division operator
/*		beginning-of-comment delimiter
*/		end-of-comment delimiter
(	left paren	left delimiter of lists, subscripts, and expressions
)	right paren	right delimiter of lists, subscripts, and expressions
+	plus	addition operator
–	minus	subtraction or unary minus operator
'	apostrophe	string delimiter
*	asterisk	multiplication operator
<	less than	relational test operator
>	greater than	relational test operator
<=	less or equal	relational test operator
>=	greater or equal	relational test operator
<>	not equal	relational test operator
:	colon	label delimiter
;	semicolon	statement delimiter
,	comma	list element delimiter

Appendix D
PL/ M RESERVED WORDS

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

Appendix E
PL/M PREDECLARED IDENTIFIERS

These are the identifiers for the built-in procedures and predeclared variables.

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

INDEX

This index does not list every occurrence of each term. Boldface page references indicate the location of primary information on a term. Italicized page references indicate the location of primary information on a term. Other page references indicate the locations of significant references to the term in connection with some other topic.

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

PRINTED IN U.S.A./S82/0477/15K