PL/M-86
PROGRAMMING MANUAL FOR
8080/8085-BASED
DEVELOPMENT SYSTEMS

Manual Order Number: 9800466-03 Rev. C

Additional copies of this manual or other Intel literature may be obtained from:

A541/282/5K SVP

This is a programming manual for the PL/M-86 language, as implemented by the PL/M-86 Compiler. Throughout this manual, the name "PL/M-86" refers specifically to this implementation.

For information on the use of the PL/M-86 Compiler itself, the reader is referred to the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems*, Intel document number 9800478. For information on the ISIS-II operating system facilities related to PL/M-86 programming, see *ISIS-II System User's Guide*, Intel document number 9800306.

How to Use This Manual

This manual is intended to be read from front to back by a new PL/M-86 programmer. It is assumed that the reader has at least some acquaintance with a higher-level programming language. Readers who are already familiar with PL/M-80 may find it helpful to start by reading Appendix E, which describes the differences between PL/M-80 and PL/M-86.

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-86 reserved words, and PL/M-86 predeclared identifiers, included for convenience in using the manual for reference purposes.

CONTENTS

CONTENTS (Cont'd.)

TABLES

ILLUSTRATIONS

1.1 What Is PL/M-86?

PL/M-86 is a high-level language designed for system and applications programming for the Intel 8086 microprocessor.

A PL/M-86 program is a sequence of *PL/M-86 statements*. The PL/M-86 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-86 statement may be translated by the compiler into a single 8086 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-86 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-86 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-86 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-86 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-86 also provides facilities for declaring and activating *procedures*. A *procedure declaration* is a block of PL/M-86 statements 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 programming, 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-86 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-86.

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-86 identifiers can be associated with a wide range of different kinds of objects — variables, collections of variables, procedures, and PL/M-86 statements. PL/M-86 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-86 identifiers:

    ABC
    X2
    PRODUCT
    K

Further discussion of identifiers will be found in Chapter 2.

1.2.2 DECLARE Statements and Data Types

A simple DECLARE statement has already been shown above (Section 1.1). 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 REAL;
    DECLARE (OLD, NEW) BYTE;
    DECLARE POINT WORD, VAL BYTE;

A scalar always has a *type*, which may be BYTE, WORD, INTEGER, REAL, or POINTER.

* A BYTE scalar is an 8-bit quantity occupying one byte of memory. Values of BYTE scalars are always interpreted as unsigned whole numbers and may range from 0 to 255.

* A WORD scalar is a 16-bit quantity occupying two contiguous bytes of memory. Values of WORD scalars are always interpreted as unsigned whole numbers and may range from 0 to 65535.

* The value of an INTEGER scalar is a *signed* whole number and may range from −32768 to 32767.

* The value of a REAL scalar is a signed floating-point number whose size limits depend on the implementation (see *ISIS-II PL/M-86 Compiler Operator's Manual*, 98-478).

* The value of a POINTER scalar is a location in 8086 storage.

The different types are discussed in detail in Chapter 3.

The first example above declares a single scalar variable of type REAL, 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 WORD, and VAL is of type BYTE.

The following statements declare arrays:

    DECLARE DOMAIN (128) BYTE;
    DECLARE GAMMA (10) WORD;

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 WORD. 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 WORD);

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

Further discussion of variables and variable declarations will be found in Chapters 3, 5, and 8.

1.2.3 Executable Statements

The following is a list of all PL/M-86 executable statements and the numbers of the sections in which they are fully discussed:

Assignment Statement	(Section 4.6)
GOTO Statement	(Section 6.3)
IF Statement	(Section 6.2)
Simple DO Statement	(Section 6.1.1)
Iterative DO Statement	(Section 6.1.4)
DO WHILE Statement	(Section 6.1.3)
DO CASE Statement	(Section 6.1.5)
END Statement	(Section 6.1)
CALL Statement	(Section 9.3)
RETURN Statement	(Section 9.2.3)
HALT Statement	(Section 6.4)
ENABLE Statement	(Section 9.2.6)
DISABLE Statement	(Section 9.2.6)
Null Statement	(Section 6.1.5)

The following sections give simplified descriptions of some of the executable statements, in order to provide a feeling for PL/M-86 before going on to the full descriptions in later chapters.

Assignment Statement

The assignment statement has already been introduced (Section 1.1). It is fundamental to PL/M-86 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.

The simplest form of the assignment statement is

    identifier = expression ;

where the identifier is the identifier of a variable. The expression is evaluated, and the resulting value becomes the value of the variable. Variations of this form are given 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 6 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.

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 or more executable 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 BYTE, WORD, or INTEGER 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 6.

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. Labels are useful for readability, and in connection with GOTO statements. Labels are discussed in Section 6.3.

1.2.4 Procedures

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

PROCEDURE Statements

The following is an example of a PROCEDURE statement:

    SUMSQUARE: PROCEDURE (A, B) REAL;

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 activating 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 activated. Not all procedures have parameters; they are left out of the PROCEDURE statement if not needed.

This is a *typed* procedure, with type REAL. The appearance of a type in the PROCEDURE statement means that the procedure will return a value to the point from which it is activated − in this case, a floating-point (REAL) 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) REAL;
         DECLARE (A, B) REAL;
         RETURN A*A + B*B;
    END SUMSQUARE;

A and B are declared to be scalar variables of type REAL. 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 activation.

A typed procedure, such as SUMSQUARE, is activated 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 REAL variables. The text SUMSQUARE(PREV, NEXT) is a *function reference* 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 function reference*, and the expression in the assignment statement can now be evaluated.

1-7

For example, suppose that when the above assignment statement is executed, OLDVAL has a value of 100.0, PREV has a value of 4.0, and NEXT has a value of 5.0. Then SUMSQUARE returns a value of 16.0 + 25.0, or 41.0. This is subtracted from the value of OLDVAL and the result, 59.0, 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 9.

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-86 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-86. 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 10. However, the above rules will suffice for understanding Chapters 2 through 9.

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 Builtin Procedures and Variables

PL/M-86 provides a large repertoire of builtin procedures and variables. These provide such functions as shifts and rotations, data type conversions, and string manipulation. The builtin procedures and variables are described in Chapter 12.

1.2.7 Expressions

We have already seen simple expressions. A PL/M-86 expression is made up of operands and operators, and resembles a conventional algebraic expression.

Operands include numeric constants (such as 3.78 or 105) and variables (as well as other types discussed in Chapters 3 and 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-86 expression may be grouped with parentheses.

An expression is evaluated using unsigned integer arithmetic, signed integer arithmetic, and/or floating-point arithmetic, depending on the types of operands in the expression (see Chapters 3 and 4 for details).

1.2.8 Modular Structure of PL/M-86 Programs

The definition of a PL/M-86 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 (possibly only null 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.

1.2.9 Input And Output

PL/M-86 does not provide I/O functions in the usual sense of the term. In particular, PL/M-86 I/O does not resemble FORTRAN I/O.

There are three basic methods for moving data to or from memory under PL/M-86 program control.

The INPUT, INWORD, OUTPUT, and OUTWORD Facilities

The builtin procedures INPUT and INWORD and the builtin variable arrays OUTPUT and OUTWORD are described in detail in Chapter 12.

INPUT causes the program to read the 8-bit quantity found in one of the 65K input ports of the 8086. A reference to OUTPUT causes the program to place an 8-bit quantity into one of the 65K output ports of the 8086.

INWORD and OUTWORD have the same effects as INPUT and OUTPUT, except that they handle 16-bit (WORD) quantities instead of 8-bit (BYTE) quantities.

Direct Memory Access (DMA) Techniques

A peripheral device that has direct access to 8086 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 runctions.

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-86 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 8.3), 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-86 statements.

Words in capital letters are reserved words in the PL/M-86 vocabulary, and are to be entered as shown (lower-case letters may be used). Semicolons are to be entered as shown.

The items in lower case are to be replaced by actual PL/M-86 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 6.1 the form of the END statement is given as follows:

        END [label];

This means that an END statement consists of the following parts, in order:
- 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-86 statements is given in the text.

Note also that these notational conventions apply only to the way that the *forms* of PL/M-86 statements are shown. When an example of an *actual* PL/M-86 statement is given, capital letters are used for all of the code and lower-case is used only for comments (see Section 2.6).

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.

PL/M-86 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-86 Character Set

The character set used in PL/M-86 is a subset of both ASCII and EBCDIC character sets. The valid PL/M-86 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-86 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 2.5). For example, xyz and XYz are interchangeable. In this manual, all PL/M-86 code is in upper-case letters to help distinguish it visually from explanatory text.

Blanks are not distinguished from each other (except in string constants). 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-86 program, as described in the remainder of this manual. Appendix B is a concise list of special characters and their meanings.

The above applies to everything in a PL/M-86 program except character string constants (see Section 2.5) and comments (see Section 2.6).

2.2 Identifiers and Reserved Words

Identifiers are used to name variables, procedures, and statement labels (they are also used in "LITERALLY" declarations, described in Chapter 8). An identifier may be up to 31 characters in length. The first character must be alphabetic, and the remainder may be either alphabetic, numeric, or the underscore (__).

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. (Note that a dollar sign is not allowed as the first character of an identifier.)

Examples of valid identifiers are

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

The last two examples are interchangeable, but are different from the first.

There are a number of otherwise valid identifiers whose meanings are fixed in advance. Because they are actually part of the PL/M-86 language, they may not be used as identifiers. A list of such *reserved words* is given in Appendix C.

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-86 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 Section 2.4).

- Character string constants (see Section 2.5).

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-86 statement. Thus the assignment statement

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

is equivalent to

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

2.4 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 whole-number constant, a floating-point constant, or a character-string constant. A whole-number constant may be expressed as a binary, octal, decimal, or hexadecimal number. Floating-point constants are always decimal.

In general, the base (or radix) of a *whole-number constant* 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 are interchangeable and denote an octal constant. The letter D may optionally follow a decimal number for readability.

A hexadecimal number consists of a sequence 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-86 identifier. A leading zero is always sufficient. For example, instead of A3H (which could be an identifier) you must write 0A3H.

Any number not followed by one of the letters B, O, Q, D, or H is assumed to be decimal. Whole-number constants must be representable in 16 bits except when treated as POINTER values (see note below). Thus they may range in value from 0 to 65535, or 0FFFFH, or 177777Q, or 1111$1111$1111$1111B.

The following are examples of valid whole-number constants:

    2 33Q 0110B 33FH 55D 55 0BF3H 65535

A whole-number constant may be treated as a BYTE, WORD, INTEGER, or POINTER value, as explained in Chapter 4.

NOTE

There are only three cases where a whole-number constant can be treated as a POINTER value. These are described in Sections 4.6.1, 8.3, and 8.4.

The size limits for a whole-number constant depend on which of these types it will have, as follows:

BYTE or WORD	0 to 65535
INTEGER	0 to 32767
POINTER	depends on implementation and compiler controls.

Note that a minus sign in front of a constant is not part of the constant. An INTEGER *value* may be negative, but a constant is never negative. This is why the range shown opposite "INTEGER" is a range of positive numbers — it is the range of *constants* that can be treated as INTEGER values, not the range of INTEGER values (which is −32768 to 32767). See Chapter 4.

A *floating-point constant* is indicated by the presence of a decimal point. At least one decimal digit must occur on the left side of the decimal point. The following are valid PL/M-86 floating-point constants:

    5.3 176.0 1.88 3.14159 0.15 0.00032 16.

Also, a floating-point constant may have an "exponent" to indicate multiplication by a power of 10. To indicate an exponent, end the number with the letter E followed by the power of ten (which must be an integer), as in the following examples. The values of these examples are exactly the same as the previous examples:

> 53.0E−1 1.760E2 0.188E1 314159.0E−5 1.5E−1 3.2E−4 1.6E + 1

Note the "unary" plus sign in the exponent of the last example. It has no effect but may be used if desired.

A floating-point constant is always treated as a REAL value. Its size limits depend on the implementation (see Chapter 14).

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

> 11110110011B
> 111$1011$0011B

It is **strongly** advised that you read Chapter 14 before using floating point arithmetic in programs. See also Chapter 6 of the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.

2.5 Character Strings

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 but line-feeds are not. 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 ASCII code table at end of this manual).

Character strings can only be used as BYTE or WORD values. Therefore, character strings longer than 2 characters cannot be used as values, since this would exceed the 16-bit capacity of a WORD value. However, longer character strings can be used in a PL/M-86 program (see Sections 3.6, 8.4, 8.5, and Chapter 12).

The maximum length of a string constant is limited by the PL/M-86 Compiler. See *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.

2.6 Comments

Explanatory *comments* may be interleaved with PL/M-86 program text, to improve readability and provide program documentation. A PL/M-86 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. 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-86 program.

Here is a sample PL/M-86 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.

3.1 General

PL/M-86 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 whole- number constant 3.

As previously mentioned, the values of PL/M-86 variables and constants are classified as BYTE, WORD, INTEGER, REAL, and POINTER values. These "data types" are fundamental to PL/M-86.

PL/M-86 performs calculations using three different types of arithmetic: unsigned, signed, and floating-point. As explained in this chapter and the next, the type of arithmetic used for any calculation depends on the data type(s) involved.

3.1.1 Scalar Variables

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 5.1 and 5.2.

Each variable used in a PL/M-86 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 8 gives additional information about DECLARE statements.)

3.1.2 Types

A scalar variable has one of five "types": BYTE, WORD, INTEGER, REAL, or POINTER. A BYTE variable is an 8-bit value occupying a single byte of storage. A WORD variable is a 16-bit value occupying two bytes of storage. The internal representations of INTEGER, REAL, and POINTER variables depend on the implementation.

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 five reserved words BYTE, WORD, INTEGER, REAL, or POINTER. Sample PL/M-86 declarations are

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

The first of these DECLARE statements introduces the identifier UNKNOWN and states that it refers to a BYTE value.

The second DECLARE statement introduces the identifier PTR and states that it refers to a POINTER value.

The statement

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

is called a "factored declaration." It is equivalent to the following sequence:

```
DECLARE WIDTH REAL;
DECLARE LENGTH REAL;
DECLARE HEIGHT REAL;
```

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

The three identifiers WIDTH, LENGTH, and HEIGHT are introduced and stated to refer to three distinct scalars of type REAL — that is, floating-point values.

The concept of data types applies not only to variables, but to every value that is processed by a PL/M-86 program. This includes values returned by procedures, and values calculated by processing expressions.

3.2 WORD and BYTE Variables: Unsigned Arithmetic

The value of a BYTE variable is an 8-bit binary number ranging from 0 to 255 and occupying one byte of 8086 memory. The value of a WORD variable is a 16-bit binary number ranging from 0 to 65535 and occupying two contiguous bytes of 8086 memory. Values of WORD and BYTE variables are treated as *unsigned binary integers.*

Unsigned integer arithmetic is used in performing any arithmetic operation upon WORD and BYTE variables. All of the PL/M-86 operators may be used with BYTE and WORD variables (see Chapter 4). Arithmetic and logical operations yield a result of type BYTE or WORD, depending on the operation and the operands (see Section 4.2 and Table 4-1). Relational operations always yield a "true" or "false" result of type BYTE. (See Section 4.2 and Table 4-1.)

With unsigned arithmetic, 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, if a BYTE value of 1 (00000001 in binary) is subtracted from a BYTE value of 0 (00000000 binary), the result is a BYTE value of 255 (11111111 binary).

Also, the result of a division operation is always truncated (rounded down) to a whole number. For example, if a WORD value of 7 (0000000000000111 binary) is divided by a BYTE value of 2 (00000010 binary), the result is a WORD value of 3 (0000000000000011 binary).

When declaring a variable that may be used to hold or produce a negative result, it is advisable to make it either INTEGER or REAL. If it may be used to hold or produce a non-integer value, it must be REAL. This will avoid unexpected incorrect results from arithmetic operations. See Chapter 4.

3.3 INTEGER Variables: Signed Arithmetic

INTEGER variables represent signed integers ranging from −32768 to 32767. Internally, an INTEGER value is represented in twos-complement notation.

Arithmetic operations on INTEGER variables use signed integer arithmetic to yield an INTEGER result. Thus addition and subtraction always produce mathematically correct results if overflow does not occur. Relational operations use signed arithmetic comparisons to yield a "true" or "false" result of type BYTE.

However, just as with BYTE and WORD operands, division produces a result which is truncated to an integer. (The result is rounded down if it is positive, or up if it is negative.)

Only the arithmetic and relational operators may be used with INTEGER operands. Logical operators are not allowed. See Chapter 4.

3.4 REAL Variables: Floating-Point Arithmetic

The value of a REAL variable is a signed floating-point number whose size limits depend on the implementation (see Chapter 14). A REAL value may be any floating-point number (within the limits of precision allowed by the implementation).

Operations on REAL operands use signed floating-point arithmetic to yield a result of type REAL. The implementation guarantees that the result of each operation is the closest possible floating-point number to the exact mathematical real-number result (if overflow or underflow does not occur). The relational operators and the arithmetic operators + . −, *, and / may be used with REAL operands—the MOD operator and the logical operators are not allowed. Arithmetic operations yield a result of type REAL, and relational operations yield a "true" or "false" result of type BYTE. (See Section 4.2 and Table 4-1.)

For a description of binary representations of REAL values and a discussion of error handling in floating-point arithmetic, see Chapter 14.

3.5 POINTER Variables and Location References

The value of a POINTER variable is an 8086 storage location. An important use of POINTER variables is as bases for based variables (see Section 5.4).

Only the relational operators may be used with POINTER operands, to yield a "true" or "false" result of type BYTE. No arithmetic or logical operations are allowed. See Chapter 4.

3.5.1 The @ Operator

A "location reference" is formed by using the @ operator. A location reference has a value of type POINTER — that is, a location. An important use of location references is to supply values for POINTER variables.

The basic form of a location reference is

  @variable-ref

where the "variable-ref" is a reference to some variable. The value of this location reference is the actual location at run time of the variable. Thus the value of any location reference is always of type POINTER.

The "variable-ref" may refer to an array or structure. The location is the location of the first element or member of the array or structure.

For example, suppose that we have the following declarations:

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

Then @RESULT is the location of the REAL scalar RESULT, while @XNUM(5) is the location of the 6th element of the array XNUM and @XNUM is the location of the beginning of the array, that is, the location of the first element (element 0).

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

LIST is an array of structures. The location reference @LIST(5).KEY is the location of the scalar LIST(5).KEY. Note that @LIST.KEY is illegal, since it does not identify a unique location.

The location reference @LIST(0).INFO(6) is the location of the scalar LIST(0).INFO(6). Also, @LIST(0).INFO is the location 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 11). 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 location of the entry point of the procedure.

3.5.2 The "DOT" Operator

For compatibility with PL/M-80 programs, a "dot" operator is provided. The dot operator (.) is similar to the @ operator, but produces an address of type WORD. This will not always produce correct results in a PL/M-86 program. See Appendix E.

3.6 Storing Strings and Constants via Location References

Another 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 location 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 10 contiguous bytes of storage). The value of the operand is the location of the first of these bytes—in other words, a pointer to the string.

A PL/M-86 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.

This chapter presents a complete discussion of the rules governing PL/M-86 expressions. Although these rules may appear complex, bear in mind that most of the expressions used in actual programs are simple and easy to understand. In particular, when the operands of arithmetic and relational operators are all of the *same type*, the resulting expression is easy to understand.

4.1 Operands

Operands are the building blocks of expressions. An operand is something that has a value at run time, which can be operated upon by an operator. 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 that contains a decimal point is of type REAL.

A numeric constant that does *not* contain a decimal point is called a whole-number constant. As will be seen below (Section 4.5.2), a whole-number constant may be found in either *signed context* or *unsigned context*. In signed context a whole-number constant is treated as an INTEGER value. In unsigned context a whole-number constant is treated as a BYTE value if it is equal to or less than 255; if it is greater than 255 it is treated as a WORD value. (In three special cases, a single whole-number constant may be treated as a POINTER value. See Sections 4.6.1, 8.3, and 8.4.)

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 constant whose value is the eight-bit ASCII code for the character. If it is a two-character string, it is treated as a WORD 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. Strings of more than two characters are illegal as operands in expressions.

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 previously, 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 9.

4.1.3 Location References

Location references have already been described in Section 3.5.

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.

4.1.5 Compound Operands

All the operand types described above are *primary* operands. An operand may also be a value calculated by evaluating some portion of the total expression. For example, in the expression

    A + B * C

(where A, B, and C are variable references), the operands of the * operator are B and C. The operands of the + operator are A and the *compound operand* B * C — or more precisely, the value obtained by evaluating B * C. Notice that this expression is evaluated as if it had been written

    A + (B * C)

This analysis of an expression to determine which operands belong to which operators, and which groups of operators and operands form compound operands, is discussed in Section 4.5.1.

4.2 Arithmetic Operators

There are five principal arithmetic operators in PL/M-86 (two others are described in Chapter 13). The five principal operators are

    + − * / MOD

All of these operators are used as in ordinary algebra to combine two operands. Each operand may have a BYTE, WORD, INTEGER, or REAL value (except that REAL operands are not allowed with the MOD operator). Arithmetic operations on POINTER variables are not allowed.

4.2.1 The +, −, *, and / Operators

The operators +, −, *, and / perform addition, subtraction, multiplication, and division on operands of any type except POINTER. The following rules govern these operations (see also Table 4-1).

1. If both operands are of the same type, the result is of the same type as the operands, with only one exception: if both operands are of type BYTE, the * and / operations produce results of type WORD. The type of arithmetic depends on the type of the operands (see Sections 3.2 through 3.4).

2. Only one combination of operand types is allowed: a BYTE operand can be combined with a WORD operand. When this is the case, the BYTE operand is extended by 8 high-order "zero" bits to produce a WORD value. Then the operation is performed as if both operands were WORD operands.

If one operand is a whole-number constant, it will be treated as a BYTE, WORD, or INTEGER value according to the following rules (see also Table 4-1 below):

3. If the other operand is a WORD or BYTE operand, the whole-number constant is treated as a BYTE value if it is equal to or less than 255, or as a WORD value if it is greater than 255. Then the operation is performed under Rule 1 or Rule 2 above. If the whole-number constant exceeds 65535, the operation is illegal.

4. If the other operand is an INTEGER operand, the whole-number constant is treated as a positive INTEGER value. Then the operation is performed as if both operands were INTEGER operands. If the whole-number constant exceeds 32767, the operation is illegal.

5. If the other operand is of type REAL or POINTER, the operation is illegal.

6. If both operands are whole-number constants, the operation depends on the context in which it occurs, as explained in Section 4.5.2.

The result of division by 0 is undefined.

A *unary* "−" operator is also defined in PL/M-86. It takes a single operand, to which it is prefixed. In other words, a minus sign that has no operand to the left of it is taken to be a unary minus.

Its effect is that (−A) is equivalent to (0−A), where A is any operand. The 0 is a BYTE value if A is of type BYTE or WORD, an INTEGER value if A is of type INTEGER, or a REAL value if A is of type REAL. If A is a whole-number constant, its type and the unary "−" operation depend on the context as explained in Section 4.5.2.

Finally, a unary " + " operator is defined for the sake of completeness. As in ordinary algebra, a unary " + " has no effect, and (+ A) is exactly equivalent to (A).

4.2.2 The MOD Operator

MOD performs exactly the same as /, except as follows:

- REAL operands are not allowed — only BYTE, WORD, and INTEGER operands can be used.

- The result is not the quotient, but the remainder left after integer division. The result has the same sign as the operand on the left side of the MOD operator.

For example, if A and B are INTEGER variables with values of 35 and 16 respectively, then A MOD B yields an INTEGER result of 3.

Unlike the / operator, the MOD operator must be separated from surrounding letters and digits by blanks or other separators.

4.3 Relational Operators

Relational operators are used to compare operands of all types. 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 to yield a BYTE result (see below).

If both operands are of the same type, unsigned arithmetic is used to compare two BYTE values or two WORD values, signed arithmetic is used to compare two INTEGER values or two REAL values, and POINTER values are compared according to the ordering of 8086 locations. (See the discussion of the Optimize control in the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual.*

As with the arithmetic operators, the only legal combination of operand types is a BYTE operand with a WORD operand. Whole-number operands are treated as BYTE, WORD, or INTEGER values as explained in the rules of Section 4.2.1.

If the specified relation between the operands is "true," a BYTE value of 0FFH (or 1111$1111B) results. Otherwise the result is a BYTE value of 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")

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 6. (In the context of a DO WHILE statement or IF statement, only the least significant bit of a "true" or "false" value is used.)

4.4 Logical Operators

There are 4 logical (boolean) operators in PL/M-86. These are

    NOT AND OR XOR

These operators are used with BYTE, WORD, or whole-number constant operands to perform logical operations on 8 or 16 bits in parallel.

A whole-number constant is treated as a BYTE value if it is equal to or less than 255. It is treated as a WORD value if it is greater than 255. If it exceeds 65535, the operation is illegal.

NOT is a unary operator, taking one operand only. It produces a result of the same type as its operand, in which each bit is the ones 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 BYTE values, the operation is an 8-bit operation, and delivers a result of type BYTE. If both operands are WORD

values, the operation is a 16-bit operation, and delivers a result of type WORD. If one operand is a BYTE value and the other is a WORD value, the BYTE operand is first extended by 8 high-order zero bits to yield a WORD value. The operation is a 16-bit operation and the result is of type WORD.

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

Also, notice that "true" and "false" values resulting from relational operations 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."

4.5 Expression Evaluation

4.5.1 Precedence of Operators: Analyzing an Expression

Operators in PL/M-86 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. B is said to be "bound" to the operator * rather than the operator +, which means that A + B*C is equivalent to A + (B*C) rather than (A + B)*C.

The Rules of Precedence

- If an operand has only one operator immediately adjacent to it, it is bound to that operator.

- If an operand has operators immediately adjacent to it on both sides, it is bound to the operator with the highest precedence (see below).

- *Left-to-right rule:* If the two operators immediately adjacent to an operand have the same precedence, the operand is bound to the operator on the left.

The PL/M-86 operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence.

    unary − unary +

    * / MOD

    + −

    < <= <> = >= >

    NOT

    AND

    OR XOR

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

A + B * C	is equivalent to	A + (B * C)
A + B − C * D	is equivalent to	(A + B) − (C * D)
A + B + C + D	is equivalent to	((A + B) + C) + D
A / B * C / D	is equivalent to	((A / B) * C) / D
A>B AND NOT B<C−1	is equivalent to	(A>B) AND (NOT(B<(C−1)))

In the last four examples, we see the application of the "left-to-right" rule for operators with the same precedence. In the second, third, and fifth examples the left-to-right rule makes no difference in the value of the expression. But in the fourth example, the left-to-right rule is critical.

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.

As mentioned above in Section 4.1.4, paired parentheses and everything between them form a *subexpression*, whose value is considered to be an operand. What this means in practice is that *everything between paired parentheses is evaluated before the surrounding part of the expression*, and this value is used as an operand.

Parentheses are also used around subscripts and around the parameters of function references. These are not subexpressions since they do not become operands. But like subexpressions, they must be evaluated before the parts of the expression outside the parentheses can be evaluated.

To see these rules in action, consider the expression

    (−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)

and assume that A, B, and C are variables of type REAL and SQRT is a procedure of type REAL which returns the square root of the value passed to it as a parameter — in this case, SQRT returns the square root of the value of B*B − 4.0*A*C. Notice that it is necessary to use floating-point constants (4.0 and 2.0) rather than whole-number constants (4 and 2) since it is illegal to combine whole-number constants with REAL variables.

We will indicate the association of operands with operators by drawing brackets over each operator and its operand(s).

Since the expression contains parentheses, we first consider the portions of the expression that are within the innermost parentheses, namely the procedure parameter B*B − 4.0*A*C and the subexpression 2.0*A. Drawing brackets over the highest-precedence operators (which all happen to be * operators), we get

(−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)
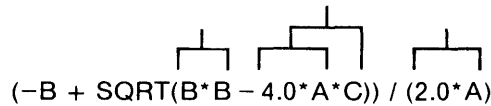
The two operands of the first * operator are both equal to the value of B. The operands of the second * operator are 4.0 and the value of A. The operands of the third * operator are the value of 4.0*A (a compound operand) and the value of C. The operands of the fourth * operator are 2.0 and the value of A.

The subexpression 2.0*A is now completely analyzed, but the parameter expression B*B − 4.0*A*C still contains a − operator that has not been analyzed. Therefore, we draw another bracket:

(−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)
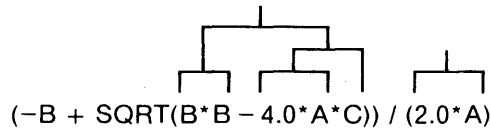
The operands of the − operator are the value of B*B and the value of 4.0*A*C. The parameter expression is now completely analyzed and its value can be calculated, but this value does not become an operand in the overall expression. Instead it is passed to the procedure SQRT which returns the square root of the parameter. It is this returned value that becomes an operand in the overall expression.
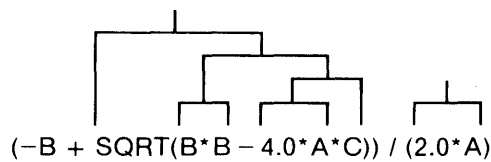
(−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)

Having dealt with the innermost parenthesized items, we next analyze the outer subexpression, −B + SQRT(B*B −4.0*A*C). The − operator has no operand to its left, so it is a unary minus and has the highest precedence. Its operand is the value of B. Next comes the + operator. Its operands are the value of −B and the value of SQRT(B*B − 4.0*A*C).
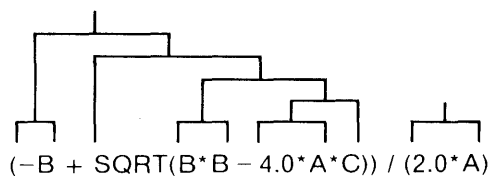
(−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)

We are left with only the / operator, whose operands are the value of (−B + SQRT(B*B − 4.0*A*C)) and the value of (2.0*A).

(−B + SQRT(B*B − 4.0*A*C)) / (2.0*A)
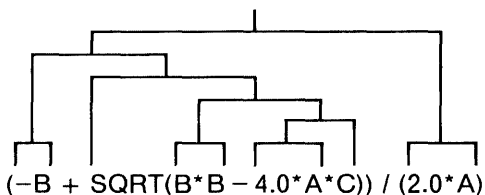
This detailed example has been included not only to illustrate the analytical process but to make three important points.

Types of Compound Operands

The first point is that the compound operands delineated by the brackets have types, just as primary operands do. In the above example, all of the primary operands are of type REAL and so the compound operands are also of type REAL. But in an expression where data types are mixed, it is important to understand how the analysis determines the type of each compound operand. Consider the expression

F > G AND H < K

where F and G are INTEGER variables and H and K are REAL variables. If we mistakenly considered G and H to be the operands of the AND operator, we would have a problem because only BYTE and WORD operands are allowed with logical operators. In fact, however, the expression is analyzed as

F > G AND H < K
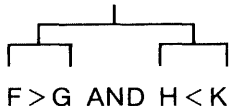
and the operands of the AND operator are the value of F > G and the value of H < K. Both of these values are BYTE values, and the expression is legal. It yields a BYTE result.

Restriction on Relational Operators

The second point is that understanding precedence rules and order of evaluation is critical to successful use of PL/M-86. In ordinary algebra, it is common to write an expression like

a <= x <= b

meaning that the value of x lies between the values of a and b. From the preceding discussion, it is clear that if we used this construction in a PL/M-86 expression, it would not have the desired meaning. First the values of a and x would be compared, yielding a BYTE result of either 0 or FFH. Then b would be compared to this result, instead of being compared to x.

But this construction would not only be mistaken, it would be *illegal*. The PL/M-86 compiler would reject it with an error message, because of the following restriction: *If a compound operand is an operand of a relational operator, the compound operand must not be the result of a relational operation.*

Thus if we write

A <= X <= B /*Illegal expression*/

then the compound operand A <= X is an operand of the second <= operator. Since the compound operand is itself the result of a relational operation, it is illegal. To get a legal expression with the desired meaning, we can write

A <= X AND X <= B /*Legal expression*/
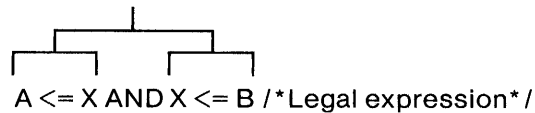
There are other cases where the restriction applies. For example, it might be desirable to write something like

A > B <> C /*Illegal expression*/
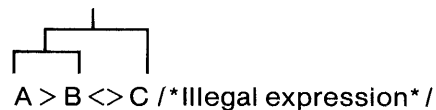
where C contains the BYTE result of some previous relational operation. To achieve this legally, we can write

(A > B) <> C /*Legal expression*/
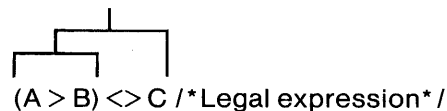
With the parentheses, (A > B) is no longer a compound operand but a subexpression. This is legal because the restriction applies only to compound operands.

Order of Evaluation of Operands

The third point to be made from the analytical example is that the binding of operators and operands is not the same thing as the order in which operands are evaluated. As we have just seen, the rules of analysis completely and unambiguously specify which operands are bound to each operator. In the expression

A + B*C

we know that B and C are the operands of the * operator, while A and the value of B*C are the operands of the + operator. Obviously, B and C must be evaluated before the * operation can be carried out. Also, the compound operand B*C must be evaluated before the + operation is carried out.

But it is not obvious whether B will be evaluated before C or vice versa. Indeed, A could be evaluated before either B or C, and its value stored until the + operation is performed.

The rules of PL/M *do not specify the order in which operands are evaluated.* The reason for this is to allow the compiler to optimize the code it produces.

In most cases this makes no difference. But in certain special cases the order of evaluation can affect the value of the expression. These cases arise only when an *embedded assignment* (see Section 4.6.3) or a *function reference* (see Section 9.2.2) is used as an operand, and the embedded assignment or function reference has the "side-effect" of changing the value of some other operand in the same expression.

Such cases must be avoided, as noted in Sections 4.6.3 and 9.2.2.

4.5.2 Choice of Arithmetic: Summary of Rules

We have already seen (Sections 3.2, 3.3, and 3.4) that PL/M-86 uses three distinct kinds of arithmetic: unsigned, signed, and floating-point. Whenever an arithmetic or relational operation is carried out, PL/M-86 uses one of these types of arithmetic, depending on the types of the operands.

Table 4-1 is a summary of the rules for which type of arithmetic is used in each case. The table also shows the type of the result in each case (for arithmetic operations). The notes following the table give additional information.

Table 4-1. Rules For Arithmetic And Relational Operations

	BYTE	WORD	INTEGER	REAL	Whole-Number Constant	POINTER
POINTER	ILLEGAL[6]	ILLEGAL[6]	ILLEGAL[6]	ILLEGAL[6]	ILLEGAL[6]	Relational operations only[7]
Whole-Number Constant	Unsigned arithmetic, BYTE or WORD result[3]	Unsigned arithmetic, WORD result[4]	Signed arithmetic, INTEGER result[5]	ILLEGAL[6]	Arithmetic and type of result depend on context[5]	
REAL	ILLEGAL[6]	ILLEGAL[6]	ILLEGAL[6]	Floating-point arithmetic, REAL result		
INTEGER	ILLEGAL[6]	ILLEGAL[6]	Signed arithmetic, INTEGER result			
WORD	Unsigned arithmetic, WORD result[2]	Unsigned arithmetic, WORD result				
BYTE	Unsigned arithmetic, BYTE or WORD result[1]					

Type of result is shown for arithmetic operators only. For relational operators, result is always BYTE.

1. Two BYTE operands: The result is of type BYTE for + and − operators or type WORD for *, /, and MOD operators.

2. BYTE and WORD: The BYTE operand is first extended with 8 high-order bits to convert it to a WORD value.

3. BYTE and whole-number constant: The constant is treated as a BYTE value if it is equal to or less than 255. Then see Note 1 above.

   If the constant is greater than 255, it is treated as a WORD value. Then see Note 2 above.

4. WORD and whole-number constant: The constant is treated as a WORD value, regardless of its magnitude.

5.   INTEGER and whole-number constant: The constant is treated as a positive INTEGER value.

6.   Note that PL/M-86 has a set of built-in procedures for converting a value from one type to another. See Chapter 12.

7.   POINTER values are compared according to the inherent ordering of 8086 locations. See also the discussion of the OPTIMIZE control in the *ISIS-II PL/M-86 Compiler Operator's Manual,* Chapter 3.

8.   This special case is described below.

Special Case: Constant Expressions

The rules already given explain expressions like

A + 3 * B

where we have a single whole-number constant. However, if we have an expression like

3 − 5 + A

we must consider which kind of arithmetic will be used to evaluate 3 − 5, since *both* operands are whole-number constants.

The answer, in this case, is that it depends on the type of the operand A. If A is of type BYTE or WORD, we say that 3 − 5 is in "unsigned context." Unsigned arithmetic is used to evaluate 3 − 5, giving a BYTE result of 254. Then unsigned arithmetic is used to add this to A.

If A is of type INTEGER, we say that 3 − 5 is in "signed context." Signed arithmetic is used to evaluate 3 − 5, giving an INTEGER result of −2. Then signed arithmetic is used to add this to A.

If A is of type REAL or POINTER, the expression is illegal.

Any compound operand, subexpression, or expression that contains only whole-number constants as primary operands is called a *constant expression*. (Note that this applies only to whole-number constants. Floating-point constants are of type REAL and are treated exactly like the values of REAL variables.)

In this expression
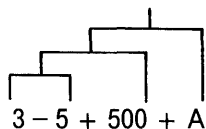
3 − 5 + 500 + A

3 − 5 is a constant expression which forms part of the larger constant expression 3 − 5 + 500.

If the constant expression is not the entire expression, then its value is an operand in the expression. The context is created by the other operand of the same operator.

If the other operand is of type BYTE or WORD, then each whole-number constant is treated as a BYTE value if it is equal to or less than 255, or a WORD value if it greater than 255. If it exceeds 65535 it is illegal. Unsigned arithmetic is used. In the example above, suppose the operand A has a BYTE value. Then the constant expression 3 − 5 + 500 is in *unsigned context*. The constants 3 and 5 are treated as BYTE

values, and 500 is treated as a WORD value. The operation 3 − 5 gives a BYTE result of 254, and this is extended to a WORD value of 254 before adding 500 to give a WORD result of 754. It is exactly as if the expression had been written as

754 + A

Now suppose that A has an INTEGER value. In this case, the constant expression 3 − 5 + 500 is in *signed context*, and all three constants are treated as INTEGER values. This time, signed arithmetic is used for the operation 3 − 5, for an INTEGER value of −2. Then 500 is added, and the INTEGER result is 498. It is as if the expression had been written as

498 + A

To summarize, if the context is created by a BYTE or WORD operand, the constant expression is in unsigned context. If the context is created by an INTEGER operand, the constant expression is in signed context. Note that if the context is created by a REAL or POINTER operand, the constant expression is *illegal*.

If the constant expression is the entire expression, then it is one of the following:

• Constant expression as right-hand part of an assignment statement: context is created by the variable to which the expression is being assigned. Rules are given below in Section 4.6.1.

• Constant expression as subscript of an array variable: evaluated as if being assigned to a WORD variable (see Section 4.6.1).

• Constant expression in the IF part of an IF statement: evaluated as if being assigned to a BYTE variable (see Sections 6.2 and 4.6.1).

• Constant expression in a DO WHILE statement: evaluated as if being assigned to a BYTE variable (see Sections 6.1.3 and 4.6.1).

• Constant expression as "start," "step," or "limit" expression in an iterative DO statement: evaluated as if being assigned to a variable of the same type as the index variable in the same iterative DO statement (see Sections 6.1.4 and 4.6.1).

• Constant expression in a DO CASE statement: evaluated as if being assigned to a WORD variable (see Sections 6.1.5 and 4.6.1).

• Constant expression as an actual parameter in a CALL statement or function reference: evaluated as if being assigned to the corresponding formal parameter in the procedure declaration (see Sections 9.2.1 and 4.6.1).

• Constant expression in a RETURN statement: evaluated as if being assigned to a variable of the same type as the typed procedure that contains the RETURN statement (see Sections 9.2.3 and 4.6.1).

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-86 *assignment statement* changes the value of a variable. Its simplest form is

variable = expression ;

The expression to the right of the equal sign may be any PL/M-86 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 = A + B;

the variable RESULT will have a new value, calculated by evaluating the expression A + B.

4.6.1 Implicit Type Conversions

If the variable on the left side of an assignment statement has a different type from that of the evaluated expression on the right, then either the assignment is invalid or an implicit type conversion occurs, as explained in the paragraphs below. (An invalid assignment can often be changed into a valid one via the explicit type- conversion procedures described in Section 12.2.)

Expression with a BYTE Value

WORD variable on the left: The BYTE value is extended by 8 high-order bits to convert it to a WORD value.

If the variable on the left is of any type except BYTE or WORD, the assignment is illegal.

Expression with a WORD Value

BYTE variable on the left: The 8 high-order bits of the WORD value are dropped to convert it to a BYTE value. Note that this is same as saying that the value of the expression is taken modulo 256.

If the variable on the left is of any type except BYTE or WORD, the assignment is illegal.

Expression with an INTEGER Value

No conversions are possible. If the variable on the left is of any type except INTEGER, the assignment is illegal.

Expression with a REAL Value

No conversions are possible. If the variable on the left is of any type except REAL, the assignment is illegal.

Expression with a POINTER Value

No conversions are possible. If the variable on the left is of any type except POINTER, the assignment is illegal.

Constant Expression

BYTE variable on the left: The constant expression is evaluated in unsigned context. If the resulting value is less than or equal to 255, it is treated as a BYTE value and no conversion is necessary. If the resulting value is greater than 255, it is converted to type BYTE by dropping all except its 8 low-order bits (in other words, it is taken modulo 256).

WORD variable on the left: The constant expression is evaluated in unsigned context. The resulting value is treated as a WORD value, and no conversion is necessary.

INTEGER variable on the left: The constant expression is evaluated in signed context to yield an INTEGER value. No conversion is necessary.

POINTER variable on the left: If the constant expression consists of nothing but a *single* whole-number constant, the constant is treated as a POINTER value. The whole-number constant must not be greater than 1048575. If the constant expression consists of anything more than a single whole-number constant, the assignment is illegal. This is one of the three cases in which a whole-number constant can be treated as a POINTER value. The other two cases are described in Sections 8.3 and 8.4.

REAL variable on the left: The assignment is illegal. However, the FLOAT procedure described in Section 12.2 can be used to convert the constant expression to a REAL value which can be assigned to that variable.

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-86 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;

The variables on the left-hand side of a multiple assignment must all be of the same type, with one exception: variables of types BYTE and WORD may be mixed. When this is done, the conversion rules given above are applied separately to each assignment.

CAUTION

The order in which the assignments are performed is not predictable. Therefore, if a variable on the left side of a multiple assignment also appears in the expression on the right side, the results are *undefined*.

4.6.3 Embedded Assignments

A special form of the assignment is used within PL/M-86 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 exam-
ple, 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
names for intermediate results can then be used at a later point in the program
without recalculating their values. The names must have been declared earlier (see
Chapter 3).

CAUTION

As mentioned at the end of Section 4.5.1, the order in which operands in an
expression are evaluated is not specified by the rules of PL/M. When
embedded assignments are used, this may become significant.

If an embedded assignment changes the value of a variable which also
appears elsewhere in the expression, it may be impossible to predict whether
the variable's value will be changed before or after it is referenced. In this
case, the value of the expression is *undefined*.

You can avoid such cases by removing the embedded assignment from the
expression and using a separate assignment statement to achieve the same
effect.

NOTE

It is **strongly** advised that you read Chapter 14 before using floating point
arithmetic in programs. See also Chapter 6 of the *PL/M-86 Compiler
Operating Instructions for 8080/8085-Based Development Systems
Manual*.

5.1 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" (actually, a value enclosed in parentheses as seen in Section 1.2.2). Such a group is called an array.

5.1.1 Array Declarations

An array is declared by using a "dimension specifier." The dimension specifier is a whole-number 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) REAL;

is equivalent to the following sequence:

        DECLARE WIDTH (100) REAL;
        DECLARE LENGTH (100) REAL;
        DECLARE HEIGHT (100) REAL;

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

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

5.1.2 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-86 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 whole-number constant, but can be another variable, or in fact any PL/M-86 expression that yields a BYTE, WORD, or INTEGER value. 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-86 permits an expression. Also, subscripted variables are permitted on the left side of an assignment statement.

5.2 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 REAL, ALTITUDE REAL);

This declares two REAL scalars, both associated with the identifier AIRPLANE. 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 *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual* for limits on the number of members allowed).

Note that *each* structure member must have its type given separately. The declaration given above cannot be rewritten as

DECLARE AIRPLANE STRUCTURE ((SPEED, ALTITUDE) REAL); /*Error!*/

Also, a structure member may not be based (see Section 5.4) and may not have any attributes (see Chapter 8).

5.2.1 Arrays of Structures

We have already seen arrays of scalars. PL/M-86 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 REAL, ALTITUDE REAL);

This declares twenty structures associated with the array identifier AIRPLANE, distinguished by subscripts from 0 to 19. Each of these structures consists of two REAL scalar members. Thus storage is allocated for 40 REAL scalars.

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

5.2.2 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,
                           AMOUNT REAL);

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 REAL scalar PAYCHECK.AMOUNT.

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

5.2.3 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). Note that the scalar elements of each "Y" array are stored contiguously, and the "Y" arrays themselves are elements of the "X" array and are stored contiguously.

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,
                                 AMOUNT REAL);
```

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, and amount 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.

5.3 Reference to Arrays and Structures

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."

5.3.1 Fully Qualified Variable References

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

```
DECLARE AVERAGE REAL;
DECLARE ITEMS (100) BYTE;
DECLARE RECORD STRUCTURE (KEY BYTE, INFO WORD);
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.

5.3.2 Unqualified and Partially Qualified Variable References

Unqualified and partially qualified variable references are allowed only in location references (see Section 3.5) and in the builtin procedures LENGTH, LAST, and SIZE (see Section 12.1).

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. @ITEMS is the location of the entire array ITEMS—that is, the location of its first byte. Similarly, @RECORD is the location of the first byte of the structure RECORD.

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 scalar. For example, NODE(15) and NODE(12).SUBLIST are partially qualified variable references, given the above declarations. @NODE(15) is the location of the first byte of the structure NODE(15), which is itself an element of the array NODE. Similarly, @NODE(12).SUBLIST is the location of the first byte of the array NODE(12).SUBLIST, which is itself a member of the structure NODE(12), which in turn is an element of the array NODE.

Note that @NODE.SUBLIST is not permitted: in a location reference referring to an array made up of structures, a subscript must be given before a member-identifier can be added to the reference. The rule is different for partially qualified variable references in connection with the builtin procedures LENGTH, LAST, and SIZE, as explained in Chapter 12.

5.4 Based Variables

Sometimes a direct reference to a PL/M-86 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-86 code to manipulate the locations of data elements rather than the data elements themselves, considering that the locations "point to" the data.

To permit this type of manipulation, PL/M-86 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 POINTER or WORD, and then declaring the based variable itself:

```
DECLARE ITEM$PTR POINTER;
DECLARE ITEM BASED ITEM$PTR 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$PTR. This means that the sequence

```
ITEM$PTR = 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 POINTER or WORD. Normal usage is for the base to be a POINTER variable; a base of type WORD will not always give correct results and is only allowed for compatibility with PL/M-80 programs (see Appendix E).

- 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)
  POINTER;
DECLARE AGE BASED AGE$PTR BYTE;
DECLARE (INCOME BASED INCOME$PTR, RATING BASED RATING$PTR)
  WORD;
DECLARE (CATEGORY BASED CATEGORY$PTR) (100) WORD;
```

In the first DECLARE statement, the POINTER variables AGE$PTR, IN-COME$PTR, RATING$PTR, and CATEGORY$PTR are declared. They are used as bases 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 POINTER variable AGE$PTR at that same time.

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

The fourth DECLARE statement defines a 100-element WORD 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 location 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.

5.4.1 Location References and Based Variables

An important use of location references is to supply values for bases. Thus the @ operator, together with the based variable concept, gives PL/M-86 a complete facility for manipulating pointers.

For example, suppose that we have three different REAL variables, NORTH$ER-ROR, EAST$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$ERROR) REAL;
DECLARE ERROR$PTR POINTER;
DECLARE ERROR BASED ERROR$PTR REAL;

...

ERROR$PTR = @NORTH$ERROR;
```

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

```
ERROR$PTR = @HEIGHT$ERROR;
```

Now a reference to ERROR will be, in effect, a reference to HEIGHT$ERROR. In the same way, we can cause the value of the pointer to be the location of EAST$ER-ROR, 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 locations to procedures as parameters. Examples are given in Chapter 9.

5.5 Contiguity of Storage

PL/M-86 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 location and the last element in the highest location. (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.

This chapter describes statements that affect the sequence of execution of statements in a PL/M-86 program and the grouping of PL/M-86 statements into blocks.

6.1 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.

6.1.1 Simple DO Blocks

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

    DO;
            statement-0;
            statement-1;
            ...
            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-86 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 6.1.5 and 6.2.

* A simple DO block delimits the scope of variables as explained in Chapter 10.

* As explained in Chapter 11, a program module is a simple DO block (with certain other requirements).

Each statement within a simple DO block may be any PL/M-86 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:

```
able:     DO;
                              statement-0;
                              statement-1;
          baker:              DO;
                                      statement-a;
                                      statement-b;
                                      statement-c;
                              END baker;
                              statement-2;
                              statement-3;
          END able;
```

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-86 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-86 Compiler. See *ISIS-II PL/M-86 Compiler Operator's Manual.*

6.1.2 "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 6.2). We have seen (Section 4.3) 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 WORD 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."

6.1.3 DO WHILE Blocks

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

```
DO WHILE expression;
        statement-0;
        statement-1;
        ...
        statement-n;
END;
```

The effect of this statement is as follows:

1. First the expression following the reserved word WHILE is evaluated as if it were being assigned to a BYTE variable. If the result is a quantity whose rightmost bit is 1, then the sequence of statements up to the END is executed.

2. 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.

3. 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 6.3.2) could transfer control out of the block without regard to the value of the expression 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.

Like a simple DO block, a DO WHILE block can be considered as a single PL/M-86 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.)

6.1.4 Iterative DO Blocks

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

Like a simple DO block, an iterative DO block can be considered as a single PL/M-86 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.)

The form of the iterative DO block is

```
DO index = start-expr TO limit-expr [BY step-expr];
        statement-0;
        statement-1;
        ...
        statement-n;
END;
```

where "index" is a reference to a scalar variable (not subscripted) of type BYTE, WORD, or INTEGER, called the "index variable." The "start-expr," "limit-expr," and "step-expr" are PL/M-86 expressions. The reserved word BY and the step- expr may be omitted, if a step value of 1 is desired (see below).

The operation of an iterative DO block depends on the type of the index variable.

Iterative DO Block With INTEGER Index Variable

When the index variable is of type INTEGER, the iterative DO block operates as follows:

1. The start-expr is assigned to the index variable.

2. The limit-expr and the step-expr are evaluated as if being assigned to variables of type INTEGER. If no step-expr has been supplied, a value of 1 is used.

3. If the value of the step-expr is negative and the value of the index variable is less than the value of the limit-expr, the iterative DO block terminates at this point and control passes to the statement following the END statement.

4. If the value of the step-expr is not negative and the value of the index variable is greater than the value of the limit-expr, the iterative DO block terminates at this point and control passes to the statement following the END statement.

5. If the iterative DO block has not been terminated in Step 3 or Step 4, the executable statements within the block are now executed.

6. When the last executable statement in the block has been executed, the index variable is incremented by the value of the step-expr. (The value of the step-expr was obtained in Step 2; it is not re-evaluated at this point.) Then we go back to Step 2.

NOTES

The start-expr is evaluated only once. The limit-expr and the step-expr are evaluated each time the block is repeated, before the executable statements are executed.

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 6.3.2) could transfer control out of the block without regard to the values of the index-variable and the limit-expr.

An example of an iterative DO block 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 INTEGER 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 next example uses a step-expr:

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

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

Iterative DO Block With BYTE or WORD Index Variable

When the index variable is of type BYTE or WORD, the iterative DO block operates as follows:

1. The start-expr is assigned to the index variable.

2. The limit-expr is evaluated as if it were being assigned to a variable of the same type as the index variable, and this 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, the iterative DO block terminates at this point and control passes to the statement following the END statement.

3. The executable statements in the block are executed.

4. The step-expr is evaluated as if it were being assigned to the index variable, and the index variable is incremented by this value. If this causes the new value to be *less* than the old value (because of "wrap-around" due to modulo arithmetic), the iterative DO block terminates at this point and control passes to the statement following the END statement. If the new value of the index variable is *not* less than the old value, we go back to Step 2 above.

NOTES

The start-expr is evaluated only once. The limit-expr is evaluated each time the block is repeated, before the executable statements are executed. The step-expr is evaluated each time the block is repeated, after the executable statements are executed.

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 a WORD index variable).

With an index variable of type BYTE or WORD, there is no such thing as a negative step. For example, if the step-expr is—5, it will be evaluated as 251.

Furthermore, 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 with a BYTE or WORD index variable will *always* terminate if the value of the index-variable is *greater* than the value of the limit-expr.

6.1.5 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-0;
          statement-1;
          ...
          statement-n;
END;
```

The expression in the DO CASE statement may yield a BYTE, WORD, or IN-TEGER value. If it is a constant expression, it is evaluated as if it were being assigned to a WORD variable. The value of the expression must lie between 0 and n. (Call this value K.) K is used to select *one* of the statements in the DO CASE block, which is then executed. The first case (statement-0) corresponds to K=0, the second case (statement-1) 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 6.3.2) could transfer control from the selected case to another case, or out of the DO CASE block.

CAUTION

If the run-time value of the expression in the DO CASE statement is less than 0 or greater than n (where n + 1 is the number of statements in the DO CASE block) then the effect of the DO CASE statement is *undefined*. This may have disastrous effects on program execution. Therefore if there is any possibility that this may occur, the DO CASE block should be contained within an IF statement that tests the expression to make sure that it has a value that will produce meaningful results.

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:

```
SELECTOR = COUNT−5;
IF SELECTOR <= 2 AND SELECTOR >= 0 THEN
DO CASE SELECTOR;

          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 */
ELSE CALL ERROR;
```

Here SELECTOR and COUNT are INTEGER variables; therefore negative values could occur. The DO CASE block is placed within an IF statement to guarantee that if the value of SELECTOR is less than 0 or greater than 2, execution of the DO CASE block will not be attempted. Instead, a procedure called ERROR (declared previously) will be activated. IF statements are discussed in the next section.

This example illustrates the use of a simple DO block as a single PL/M-86 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-86 statement. In the example above, this allows the entire DO CASE block to be written inside the THEN part of the IF 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.)

6.2 The IF Statement

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

    IF expression THEN statement-a ;
    [ELSE statement-b ;]

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 as if it were being assigned to a variable of type BYTE. If the result is "true" (see Section 6.1.2) then statement-a is executed. If the result is "false" then statement-b is executed. Following execution of the chosen alternative, control passes to the next statement following the IF statement. Thus of the two statements (statement-a and statement-b) 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-b is not needed, the ELSE part may be omitted entirely. Such an IF statement takes the form

    IF expression THEN statement-a;

Here statement-a 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-86 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.

6.2.1 Nested IF Statements

An IF statement (including the ELSE part, if any) may be considered a single PL/M-86 statement (although it is *not* a block). Thus the THEN part of an IF statement may contain another IF statement. This "nesting" of IF statements may be carried to several levels, without needing to enclose any of the nested IF statements in DO blocks, as in the following construction:

```
IF expression-1 THEN
        IF expression-2 THEN
                IF expression-3 THEN statement-a;
```

Here we have three levels of nesting. Note that statement-a will be executed only if the values of all three expressions are "true." Thus the above is equivalent to

```
IF (expression-1) AND (expression-2) AND (expression-3)
THEN statement-a;
```

Notice that the above example of nesting does not have an ELSE part. When using nested IF statements, it is important to understand the following important rule of PL/M-86:

• A set of nested IF statements may only have *one* ELSE part, and it belongs to the *innermost* (that is, the last) of the nested IF statements.

The same rule can be stated as follows:

• 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 expression-1 THEN
        IF expression-2 THEN statement-a;
        ELSE statement-b;
```

is legal and means that if the values of both expression-1 and expression-2 are "true," then statement-a will be executed. If the value of expression-1 is "true" and the value of expression-2 is "false," then statement-b will be executed. If the value of expression-1 is "false," neither statement-a nor statement-b will be executed, regardless of the value of expression-2.

The construction above is equivalent to

```
IF expression-1 THEN
        DO;
                IF expression-2 THEN statement-a;
                ELSE statement-b;
        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 expression-1 THEN
        DO;
                IF expression-2 THEN statement-a;
        END;
ELSE statement-b;
```

Note that the meaning of this construction is completely different from the previous one.

Finally, consider the following:

```
IF expression-1 THEN
        IF expression-2 THEN
                IF expression-3 THEN statement-a;
                ELSE statement-b;
        ELSE statement-c;                /*Illegal statement!*/
ELSE statement-d;                        /*Illegal statement!*/
```

This construction is *illegal*, because only one ELSE part is allowed. If the intention is for the ELSE parts to match the IF parts as indicated by the indenting, the nesting *must* be done with DO blocks:

```
IF expression-1 THEN DO;
        IF expression-2 THEN DO;
                IF expression-3 THEN statement-a;
                ELSE statement-b;
        END;
        ELSE statement-c;
END;
ELSE statement-d;
```

6.2.2 Sequential IF Statements

Consider the following example. An ASCII-coded character is stored in a BYTE variable named CHAR. If the character is an A, we want to execute statement-a. If the character is a B, we want to execute statement-b. If the character is a C, we want to execute statement-c. If the character is neither A, B, or C, we want to execute statement-x. The code for doing this could be written as follows, using IF statements that are completely independent of one another:

```
IF CHAR='A' THEN statement-a;
IF CHAR='B' THEN statement-b;
IF CHAR='C' THEN statement-c;
IF CHAR<>'A' AND CHAR<>'B' AND CHAR<>'C' THEN statement-x;
```

This is an inefficient way to write the tests. Note that the tests of all four IF statements (six tests in all) will be carried out in every case. This is wasteful in cases when one of the earlier tests succeeds.

We need to test for 'A' in all cases. But we need to test for 'B' only if the test for 'A' fails, and we need to test for 'C' only if both previous tests fail. Finally, if the tests for 'A', 'B', and 'C' all fail, no further tests are needed—we must execute statement-x. To improve the code, we rewrite it as follows:

```
IF CHAR='A' THEN statement-a;
ELSE IF CHAR='B' THEN statement-b;
ELSE IF CHAR='C' THEN statement-c;
ELSE statement-x;
```

Notice that this is *not* a case of "nested IF statements" as described in the preceding section. IF statements are said to be nested only when one IF statement is inside the THEN part of another. Here we have IF statements inside the ELSE parts of other IF statements. This construction is called "sequential IF statements." It is equivalent to the following:

```
IF CHAR='A' THEN statement-a;
ELSE DO;
        IF CHAR='B' THEN statement-b;
        ELSE DO;
                IF CHAR='C' THEN statement-c;
                ELSE statement-x;
        END;
END;
```

Sequential IF statements are useful whenever a set of tests is to be made, but you want to skip the remaining tests whenever one of the tests succeeds. This construction works in such cases because all the remaining tests are in the ELSE part of the current test.

6.3 Statement Labels and GOTOs

6.3.1 Labels and Label Definitions

PL/M-86 *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-86 identifier. Multiple labels may precede the statement. See *ISIS-II PL/M-86 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 (see Section 8.6) 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.

Labels may be used in conjunction with GOTO statements (see below), and may also be used simply to improve the readability of a program. When a label is not used in conjunction with a GOTO, it has no effect on the operation of the program.

6.3.2 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.

For reasons given in Section 10.3, there are certain restrictions on the action of GOTO statements. The only possible GOTO transfers are the following:

- From a GOTO statement in the outer level of some block to a labeled statement in the outer level of the *same* block.

- From a GOTO statement in an inner block to a labeled statement in the outer level 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 statement in the outer level of the main program module (see Chapter 11).

- From any point in one program module to a labeled statement in the outer level of the main program module (see Section 1.2.7 and Chapter 11). To do this, the label must have "extended scope" (see Section 8.2).

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 activation (see Chapter 9) is preferable. Indiscriminate use of GOTOs will result in a program that is difficult to understand, correct, and maintain.

6.4 The HALT Statement

The HALT statement has the form

HALT ;

It causes the 8086 to come to a halt with interrupts enabled (see Section 9.2.6).

6.5 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 9.

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 of control from the procedure to the point from which it was activated.

At this point, we have examined all of the constructions available in PL/M-86 except procedures, and we can now consider a complete PL/M-86 program.

7-1. Insertion Sort Algorithm

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-86 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 location) and the largest key at the end (highest location).

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 sample program and a detailed explanation follow. 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 WORD);

        DECLARE CURRENT STRUCTURE (KEY BYTE,
                                   INFO WORD);

        DECLARE (J, I) INTEGER;

        /*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 text 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 a WORD 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 11.

8.1 GENERAL

As we have seen in Chapter 1, a variable must be declared before it can be referred to by its identifier. This is done by means of a DECLARE statement. Chapters 3 and 5 provided some examples of simple DECLARE statements, without describing all the kinds of information that can be included in declarations. This chapter gives additional information on DECLARE statements.

Labels may also be declared in DECLARE statements, although this is usually not necessary, as has been seen in Chapter 6. Label declarations are covered in this chapter, as are "LITERALLY" declarations.

Procedures must also be declared. However, the declaration of procedures is treated as a separate topic in Chapter 9.

8.1.1 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 parameterless "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 5.5 for rules on contiguity of storage).

8.1.2 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-86 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 8.2).

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 10.

8.1.3 Where Declarations May Occur

Declarations may occur only at the head of a simple DO block (see Chapter 6) or procedure block (see Chapter 9)—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 11).

8.2 The PUBLIC and EXTERNAL Attributes: Extended Scope

The PUBLIC and EXTERNAL attributes permit the programmer to extend the scope of variables (see Chapter 10) so as to allow linkage between separate modules of a program (see Chapter 11). 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 10 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 11).
*   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 11).
*   The EXTERNAL attribute may only be used with a variable that is declared PUBLIC in another module (see Chapter 11) 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 the EXTERNAL item is actually a constant, the attribute DATA may be placed after EXTERNAL to indicate its membership in the constant section rather than the data section (see also the *ISIS-II PL/M-86 Compiler Operator's Manual*). This use of DATA for EXTERNAL constants is required if the ROM control is used. This use of the keyword DATA does not permit initialization values following it. Such values may appear in the "defining" declaration, i.e., where the item is declared PUBLIC.

- 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 9.

8.3 The AT Attribute

The AT attribute has the form

    AT (location)

where "location" may be either a location reference formed with the @ operator, or a single whole-number constant in the range 0 - 1048575.

If it is a location reference, it must refer to a non-based variable that has already been declared. If there is a subscript expression, it must be a constant expression containing no operators except + and −.

If the "location" is a whole-number constant, it represents an absolute 8086 storage location.

The following are examples of valid AT attributes:

    AT (4096)

    AT (@BUFFER)

    AT (@BUFFER(128))

    AT (@NAMES(INDEX + 1))

In the last example, INDEX represents a whole-number constant that has been previously declared with a "LITERALLY" declaration (see Section 8.7). The compiler replaces this name with the declared whole-number constant, thus satisfying the restrictions given above.

The effect of an AT attribute is to cause a variable to be located at the location specified by the "location." 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$A to be located at the location of the array BUFFER. The variables CHAR$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 location as a previously declared variable called DATA$BUFFER. The other scalars making up the structure will follow this location in logical order: T(0).X(1), T(0).X(2), and so on up to T(9).Z(2), the last scalar, which is located in the 89th byte after the location of DATA$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. However, the "location" in this case may *not* be a location reference that refers to a variable which is EXTERNAL.

- 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 WORD;
DECLARE ITEM BYTE AT (@DATUM);
```

causes ITEM to be declared a BYTE variable at the same location that has just been allocated for the WORD variable DATUM. The result is that any reference to ITEM is in effect a reference to the *low-order* byte of DATUM (because WORD values are stored with the low-order 8 bits preceding the high-order 8 bits).

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 location 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-86 Compiler does not optimize the machine code generated for accesses to that variable. This is useful in connection with memory-mapped I/O.

NOTE

For compatibility with programs written in PL/M-80, PL/M-86 allows the "location" in an AT attribute to be an expression containing a location reference formed with the "dot" operator. See Appendix E.

8.4 The INITIAL Initialization

Initializations are used to supply values for variables at compile time. Initialization is not automatic. If it is not specified as part of the compilation process, it must be done by the program itself when execution begins. Otherwise the values of un-initialized variables remain indeterminate and may lead to unintended and undesired results. There are two kinds of compile-time 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 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 11). 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.

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* or a string enclosed in apostrophes.

Values are taken one at a time from the value list and used to initialize the individual scalars being declared. Initialization of a scalar is performed in the same manner as an assignment. Initial values for members of an array or structure must be specified explicitly. See also section 12.5.

A *restricted expression* is any one of the following:
- A single floating-point constant, with no operator of any kind. Such a value must be used to initialize a REAL scalar.
- A constant expression containing no operators except + and −. The constant expression is evaluated as if being assigned to the scalar being initialized, according to the rules of Section 4.6.1.

- A location reference formed with the @ operator, which must refer to a variable that has already been declared. It must be used to initialize a POINTER scalar. If the location reference contains a subscript expression, the subscript expression must be a constant expression containing no operators except + and −.

NOTE

For compatibility with programs written in PL/M-80, PL/M-86 allows the restricted expression to be an expression containing a location reference formed with the "dot" operator. See Appendix E.

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) INTEGER INITIAL (1024, 0, −2);

declares the INTEGER scalars COUNTER, LIMIT, and INCR, and initializes COUNTER to a value of 1024, LIMIT to a value of 0, 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$BOUND WORD,
                            VALUE (3) BYTE,
                            LOW$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.

If a string appears in the value 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 WORD scalar. For example,

    DECLARE GREETING (5) BYTE AT (1600) 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 value lists that match up one-for-one with the scalars being declared. It is permissible for the value 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 value list, and the remainder of the array is left uninitialized. However, the value list may not have *more* elements than are being declared.

The Implicit Dimension Specifier

Often, when one initializes an array, one wants the array to have the same number of elements as the value 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 values in the value 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 value list—it is not restricted to use with strings.

8.5 The DATA Initialization

The DATA initialization has the form

    DATA (value list)

The DATA initialization is identical to the INITIAL initialization, except for the following differences:

*   The DATA initialization causes storage to be allocated in the program's constant data segment. Variables declared with a DATA initialization are "variables" in name only, and should never appear on the left-hand 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.

<div align="center">

NOTE

</div>

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 location, it may defeat the purpose of the DATA initialization.

Use of the keyword DATA with the attribute EXTERNAL does not permit initialization values to follow.

8.6 Label Declarations

A label is an identifier that is associated with a particular executable statement in a PL/M-86 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 6.3.1. Under certain circumstances, however, it may be desirable to declare a label explicitly in order to give it the PUBLIC or EXTERNAL attribute (see Section 8.2). The label declaration makes this possible.

8.6.1 Explicit Versus Implicit Label Declarations

As noted in Section 6.3.1, 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 10.3.

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

    DECLARE identifier LABEL [attribute];

The identifier is the PL/M-86 identifier being declared as a label. The attribute may be PUBLIC or EXTERNAL (see Section 8.2 above), or may be omitted, as indicated by the brackets.

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 10.3).

The number of labels that can be declared in a single factored label declaration is limited by the PL/M-86 Compiler. See *ISIS-II PL/M-86 Compiler Operator's Manual*.

8.6.2 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 11). The effect of these attributes is to give labels extended scope, just as with variables. The rules given in Section 8.2 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 10, both the explicit declaration and the label definition must be at the outer level of the "main program module" (see Chapter 11 for discussion of modules).

8.7 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-86 identifier, and the string is a sequence of arbitrary characters from the PL/M-86 set, not exceeding 255 in length. The following example illustrates the use of this facility.

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

    DECLARE ROUGH BYTE;
    DECLARE (X, Y, DELTA, FINAL) REAL;

    ...

    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-86 handles relational operators (see Section 4.3). This often makes a program more readable.

Another use of the LITERALLY 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) WORD;

    ...

    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.

8.8 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 INTEGER;

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

    DECLARE CHR BYTE INITIAL ('A'), COUNT INTEGER;

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 INTEGER 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.

9.1 General

A procedure is a section of PL/M-86 code which is *declared* without being executed, and then *activated* from other parts of the program. A function reference or CALL statement activates the procedure, causing the procedure code to be executed out of normal sequence: program control is transferred from the point of activation 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 activation.

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 activate procedures.

9.2 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 9.2.7).

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-86 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 10 for a detailed description). Within this scope, the procedure can be activated 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 10. 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.

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.

9.2.1 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 activated, 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 PTR, N, LOWER, and UPPER. It examines N contiguously stored BYTE variables. The parameter PTR is the location 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 activated.

```
RANGE$CHECK: PROCEDURE (PTR, N, LOWER, UPPER);
        DECLARE PTR POINTER;
        DECLARE (N, LOWER, UPPER, I) BYTE;
        DECLARE ITEM BASED PTR (1) BYTE;


        DO I = 0 TO N−1;
                IF (ITEM(I) < LOWER) OR (ITEM(I) > UPPER)
                THEN CALL ERRORSET;


                /*ERRORSET is a procedure declared elsewhere.*/


        END;
END RANGE$CHECK;
```

Notice that the array ITEM is declared to have only one element. Since it is a based array, a reference to any element of ITEM is really a reference to some location relative to the location represented by PTR. In writing the procedure RANGE$CHECK, we must supply some dimension specifier for ITEM so that references to ITEM can be subscripted. But it does not matter what the dimension specifier is. We arbitrarily use 1 here.

Having made this declaration, suppose that we have 25 variables stored contiguously in an array called QUANTS. 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 (@QUANTS, 25, LOW, HIGH);
```

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

- The four actual parameters in the CALL statement—@QUANTS, 25, LOW, and HIGH—are assigned to the formal parameters PTR, N, LOWER, and UPPER, which are declared within the procedure RANGE$CHECK. Since ITEM is based on PTR and the value of PTR is @QUANTS, any reference to an element of ITEM is a reference to the corresponding element of QUANTS.

* The executable statements of the procedure RANGE$CHECK are executed, and if any of the values are less than the value of LOW or greater than the value of HIGH, the procedure ERRORSET is activated.

- 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 activated.

When a procedure has more than one parameter, PL/M-86 does not guarantee the order in which actual parameters are evaluated when the procedure is activated. If an expression used as an actual parameter contains an embedded assignment or function reference which has the side-effect of changing the value of some variable which appears in another expression used as an actual parameter for the same procedure, the results are *undefined.* See also the caution in Section 9.2.2 below.

9.2.2 Typed Versus 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 activated by using its name in a CALL statement, as shown above and as explained in Section 9.3.

A typed procedure has a type—BYTE, WORD, INTEGER, REAL, or POINTER—in its PROCEDURE statement. It returns a value of this type. It is activated 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 9.2.1.

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

CAUTION

The body of a typed procedure may contain code (such as an assignment statement) that changes the value of some variable declared outside the procedure. This is called a "side effect."

Recall that PL/M-86 does not guarantee the order in which operands in an expression are evaluated. Therefore, if a function reference appears in an expression, and the typed procedure that it activates has the side effect of changing the value of another variable in the same expression, the value of the expression depends on whether the function reference or the variable is evaluated first. If the analysis of the expression does not force one of these operands to be evaluated before the other, then the value of the expression is *undefined*. This situation can be avoided by being careful about the use of any typed procedure that has a side effect, e.g., enclosing in parentheses the subexpression containing the affected operand.

9.2.3 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 11). 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 if it were being assigned to a variable of the type given in the PRO-CEDURE statement.

9.2.4 The Procedure Body

The statements within the procedure body may be any valid PL/M-86 statements, including CALL statements and nested procedure declarations.

Example 1

The following is a typed procedure declaration:

```
AVG: PROCEDURE (X, Y) REAL;
        DECLARE (X, Y) REAL;
        RETURN (X + Y)/2.0;
END AVG;
```

This procedure could be used as follows:

```
LOW = 3.0;
HIGH = 4.0;
MEAN = AVG (LOW, HIGH);
```

The effect would be to assign the value 3.5 to MEAN.

Example 2

The following is an untyped procedure:

```
AOUT: PROCEDURE (ITEM);
        DECLARE ITEM WORD;
        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 activated 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 POINTER,
                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 activated.

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);
```

9.2.5 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 8.2 and Chapter 11.

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 11).

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 11).

- 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 a discrepancy between the parameter lists in the defining declaration and a usage declaration will not be automatically detected.

- 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 9.2.4) by giving it the PUBLIC attribute as follows:

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

In another module, we can have a usage declaration:

     AVG: PROCEDURE (X, Y) REAL EXTERNAL;
        DECLARE (X, Y) REAL;
     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.

9.2.6 Interrupts and the INTERRUPT Attribute

Only an untyped procedure with no parameters, declared at the outer level of a program module (see Chapter 11), may have the INTERRUPT attribute. A procedure with this attribute is called an "interrupt procedure." An interrupt procedure may be desirable if you wish to provide non-default handling of exception conditions arising in floating-point arithmetic (see Chapter 14).

The INTERRUPT attribute has the form

     INTERRUPT n

where n is any whole-number constant from 0 to 255 (inclusive). The effect of this attribute is that the procedure will be activated whenever the 8086 interrupt corresponding to n occurs.

To explain this in more detail, we must first consider the 8086 interrupt mechanism and the PL/M-86 statements ENABLE and DISABLE.

The interrupt mechanism has two states, "enabled" and "disabled." The 8086 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 6.4).

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

     DISABLE ;

An interrupt is initiated by some peripheral device or processor in the 8086-based system, which sends an interrupt signal and an interrupt number to the 8086 CPU. The software command CAUSE$INTERRUPT (constant) can also initiate an interrupt signal. If the interrupt mechanism is in the disabled state, the signal is ignored. If it is enabled, the interrupt is processed as follows:

1. The machine instruction currently being executed is completed.

2. The interrupt mechanism is automatically disabled.

3. The interrupt procedure whose number corresponds to the number sent by the peripheral device is activated. If no such procedure has been declared, the results are undefined, since the vector which transfers control remains uninitialized. (See also Sections 12.6.8 and 12.6.9 of this manual, and Chapter 10 of the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.)

4. When the procedure terminates (by executing a RETURN or reaching the END of the procedure) the interrupt mechanism is automatically enabled, and control returns to the point where the interrupt occurred.

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.

The following is an example of an interrupt procedure for a hypothetical system where a peripheral device initiates an "interrupt 45" 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 45;
        CALL ANNUNCIATOR (1);
                /*This will result in an output
                from the 8086 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 be used with the PUBLIC attribute.)

- It may only be used in a PROCEDURE statement at the outer level of a program module.

- The whole-number constant in the INTERRUPT attribute may be any number from 0 to 255 (inclusive). Each number may be used only once within a program.

- The procedure must be untyped and may not have any parameters.

Activating an Interrupt Procedure with a CALL Statement

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 upon activation of the procedure. If interrupts are enabled when the CALL is executed, then unless the procedure has a DISABLE as its first executable statement, it will run with interrupts enabled and should have the REENTRANT attribute (see next section).

In every other respect, an interrupt procedure activated by a CALL statement is like any other procedure so activated.

NOTE

Unlike PL/M-80, PL/M-86 interrupt routines activated with a CALL statement do not alter the interrupt enable status. This means that termination of the procedure by means of a RETURN statement or the END statement will not automatically enable interrupts.

Section 6.8 of Chapter 12 discusses the builtin function INTERRUPT$PTR, which returns the interrupt entry point, given an interrupt procedure name, and also the builtin procedure SET$INTERRUPT, which sets an interrupt vector given the interrupt procedure name and number.

The CAUSE$INTERRUPT command causes a software interrupt to the vector specified in the statement

CAUSE$INTERRUPT (constant)

9.2.7 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 activated 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 activates itself (this is known as "direct recursion") or if it activates a second procedure with the result that the first procedure is activated 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 activated before it is declared, if it is activated from within the body of a reentrant procedure—possibly the same procedure.

This permits "direct recursion," where the procedure activates itself, and "indirect recursion," where the procedure activates a second procedure and the second procedure activates the first—or activates a third procedure, which activates a fourth, etc., with the result that the first procedure is activated 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.

  Note that this may apply to an interrupt procedure that runs with interrupts enabled, either because it contains an ENABLE statement or because it is activated by means of a CALL statement. If there is any possibility that it will be interrupted by its own interrupt, it should have the REENTRANT attribute. This situation is equivalent to recursion.

- Any procedure that is directly recursive (activates itself) should have the REENTRANT attribute.

- Any procedure that is indirectly recursive (activates another procedure and is activated itself as a result) should have the REENTRANT attribute.

- Any procedure that is activated by a reentrant procedure should also have the REENTRANT attribute.

In other words: if there is any possibility that a procedure can be activated while it is already running, it should be reentrant.

The following rules apply to the REENTRANT attribute.

- The REENTRANT attribute cannot be used in the same declaration as the EXTERNAL attribute. (It may be used with the PUBLIC attribute.)

- The REENTRANT attribute may only be used in a PROCEDURE statement at the outer level of a module (see Chapter 11).

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

9.3 Activating A Procedure—Function References and CALL Statements

As we have already seen, there are two forms of procedure activation, depending on whether the procedure is typed or untyped. An untyped procedure is activated 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 9.3.1 below.)

A typed procedure is activated 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$ARRAY is a previously declared typed procedure. The value of the operand SUM$ARRAY(@ITEMS, COUNT) is the value returned by SUM$AR-RAY. See the cautionary note in Section 9.2.2.

In both forms of procedure activation, 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 activation, 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-86 expression.

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

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

9.3.1 Indirect Procedure Activation

The CALL statement, in the form shown above, activates an untyped procedure by its name. It is also possible to activate an untyped procedure by its location, *if the procedure has extended scope* (see Section 9.2.5). 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 POINTER type variable reference, and its value is assumed to be the location of the entry point of the procedure being activated.

<div align="center">

NOTE

</div>

> For compatibility with programs written in PL/M-80, a reference to a WORD variable is also allowed. In this case, the procedure need not have extended scope. See Appendix E.

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 the CALL statement uses a *location*, 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 builtin type-conversion procedures described in Section 12.2 can be used to force the value of an expression to the desired type in certain cases.)

9.4 Sample Program #2

The example program of Chapter 7 is very limited in its application. It always sorts 128 records. Each record consists of a structure with one BYTE element and one WORD element, and the BYTE element of each record is used as the sort key. To sort records structured in any other way, or to sort a different number of records, you would have to rewrite the program.

Using the techniques discussed in this chapter, we can rewrite the sort program as a procedure. By using parameters to control the operation of the procedure, we can use it to sort any number of records, and we can use it on different kinds of records. The procedure can be used any number of times within a single program.

In the following sample program, we first declare a procedure called SORTPROC, which implements the same sorting method used in Sample Program #1. This procedure makes only the following assumptions about the records it is to sort:

* Each record occupies a contiguous set of storage locations. Therefore, by using based variables each record can be handled as a sequence of bytes, even though the parts of a record are not necessarily BYTE scalars.

* The records themselves are also stored contiguously, so the entire set of records can be regarded as a single sequence of bytes. The location of the first byte of the first record is specified by the POINTER parameter PTR.

* All records are the same size, that is, each occupies the same number of bytes. This size is specified by the INTEGER parameter RECSIZE, and may not exceed 128.

- In each record, the value of one byte is to be used as the sort key. Within each record, this byte is always in the same relative position—that is, the first byte in the record, or the third, etc. This relative position is specified by the INTEGER parameter KEYIND, which resembles an array subscript—that is, it is 0 if the key is the first byte in the record, 1 if the key is the second byte, etc.
- The number of records is specified by the INTEGER parameter COUNT.

The program is followed by a detailed explanation.

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

    SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX);
                 DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER;

    /*Parameters:
          PTR is pointer to first record.
          COUNT is number of records to be sorted.
          RECSIZE is number of bytes in each record— max is 128.
          KEYINDEX is byte position within each record of a BYTE scalar
                     to be used as sort key.*/

             DECLARE RECORD BASED PTR (1) BYTE,
                            CURRENT (128) BYTE,
                            (I, J) INTEGER;

    SORT:      DO J = 1 TO COUNT-1;
                  CALL MOVB(@RECORD(J*RECSIZE), @CURRENT, RECSIZE);
                  I=J;

    FIND:         DO WHILE I > 0
                        AND RECORD((I-1)*RECSIZE + KEYINDEX)
                        > CURRENT(KEYINDEX);
                     CALL MOVB(@RECORD((I-1)*RECSIZE),
                                 @RECORD(I*RECSIZE),
                                 RECSIZE);

                     I = I-1;
                  END FIND;

                  CALL MOVB(@CURRENT, @RECORD(I*RECSIZE), RECSIZE);
               END SORT;

             END SORTPROC;

    /*Program continues on next page.*/



    /*Program to sort two sets of records, using SORTPROC*/

        DECLARE SET1(50) STRUCTURE (ALPHA WORD,
                              BETA(12) BYTE,
                              GAMMA INTEGER,
                              DELTA REAL,
                              EPSILON BYTE);

    /*Key of Nth record in SET1 is SET1(N).BETA(0), the 3rd byte in the record.*/

        DECLARE SET2(500) STRUCTURE (ITEMS(21) INTEGER,
                              KEY BYTE);

    /*Key of Nth record in SET2 is SET2(N).KEY, the 43rd byte in the record.*/

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

        CALL SORTPROC(@SET1, INT(LENGTH(SET1)), INT(SIZE(SET1(1))), 2);
        CALL SORTPROC(@SET2, INT(LENGTH(SET2)), INT(SIZE(SET2(1))), 42);

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

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

After the PROCEDURE statement and the declaration of the parameters, we declare a based BYTE array called RECORD. This array is based on the parameter PTR, which points to the beginning of the first record to be sorted. Therefore, a reference to a scalar element of RECORD will be a reference to some byte within the set of records to be sorted, as long as the subscript used with RECORD is less than the total number of bytes in all the records.

Note that a dimension specifier of 1 is used in declaring RECORD. We need to use a dimension specifier here, so as to be able to use subscripts. However, the value of the dimension specifier is unimportant because RECORD is a based array and does not have any actual storage allocated to it. The value 1 is chosen arbitrarily.

Next we declare CURRENT, an array of 128 BYTE elements. Like the structure CURRENT in Sample Program #1, the array CURRENT will be used to store the "current" record. Note that the dimension (size) of the array CURRENT is what establishes the maximum size of record that this procedure can handle. We have chosen 128 here, but in principle any dimension could be specified.

As in Sample Program #1, the INTEGER variables I and J are used to control the DO WHILE and iterative DO blocks. They have the same meaning as before. However, here they are also used to calculate subscripts for the based array RECORD.

In the statement following the iterative DO, we introduce a *builtin procedure*, MOVB. A builtin procedure is a predefined procedure that need not be declared and may be activated anywhere in the program (see Chapter 12 for details).

MOVB has the effect of copying a sequence of byte values from one storage location to another. It takes three parameters. The first specifies the starting location of the byte sequence to be copied, the second specifies the starting location to copy the bytes to, and the third specifies the number of bytes to be copied.

In the first activation of MOVB, the parameter @RECORD(J*RECSIZE) is the location of the beginning of the Jth record and @CURRENT is the location of the beginning of the array CURRENT. Thus the effect of this CALL statement is to copy the Jth record into the array CURRENT.

To understand the DO WHILE statement, consider that RECORD((I−1)*REC-SIZE) would be the first byte of the (I−1)st record, so RECORD((I−1)*RECSIZE + KEYINDEX) is the byte that is to be used as the sort key of the (I−1)st record. Similarly, CURRENT(KEYINDEX) is the sort key of the "current" record. Therefore, this DO WHILE is logically equivalent to the corresponding DO WHILE in Sample Program #1.

The second CALL statement activates MOVB to copy the (I−1)st record into the position of the Ith record, and the third CALL on MOVB copies the "current" record into the position of the Ith record.

Thus the sorting method of this procedure is identical to that of Sample Program #1. To illustrate the way this procedure can be used, it is set in a program which declares two sets of records, SET1 and SET2, and sorts them. As in the previous sample program, comments are inserted in place of the code which would be used in a working program to read data into the records and write it out after they are sorted.

SET1 is a set of 50 structures, each of which represents one record. Each structure contains a WORD scalar, an array of 12 BYTE scalars, an INTEGER scalar, a REAL scalar, and another BYTE scalar. We want to sort the records using the first element of the 12-byte array as the sort key. Since the preceding WORD scalar occupies two bytes of storage, the key is to be found in the 3rd byte of each record. Accordingly, we will specify 2 for the parameter KEYINDEX.

SET2 is a set of 500 structures, each containing an array of 21 INTEGER scalars and a BYTE scalar which is to be used as the key. Since an INTEGER scalar occupies two bytes of storage, the key is to be found in the 43rd byte of each record and we will specify 42 for the parameter KEYINDEX.

In the two CALL statements used to activate SORTPROC, we introduce three more builtin procedures: INT, LENGTH, and SIZE. LENGTH and SIZE are WORD procedures. LENGTH takes as its only parameter the name of an array (without subscript) and returns the number of *elements* in the array. SIZE takes either the name of a structure (without member-identifier) or the name of an array (without subscript), and returns the number of *bytes* in the structure or array. Thus they are convenient for calculating the COUNT and RECSIZE parameters for an activation of SORTPROC—except that they return WORD values and SORTPROC requires INTEGER values. The builtin procedure INT solves this problem. INT accepts a WORD value as its parameter and returns the corresponding INTEGER value (always positive). See Chapter 12 for complete details.

PL/M-86 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 11 discusses modules and extended scope.

10.1 Blocks

There are two kinds of blocks in PL/M-86: every DO block is a block, and every procedure declaration is a block. As will be seen in Chapter 11, a PL/M-86 program is made up of modules, and a module is a particular kind of DO block. Thus everything in a PL/M-86 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-86 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 10-1.

- 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 10-2.

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.
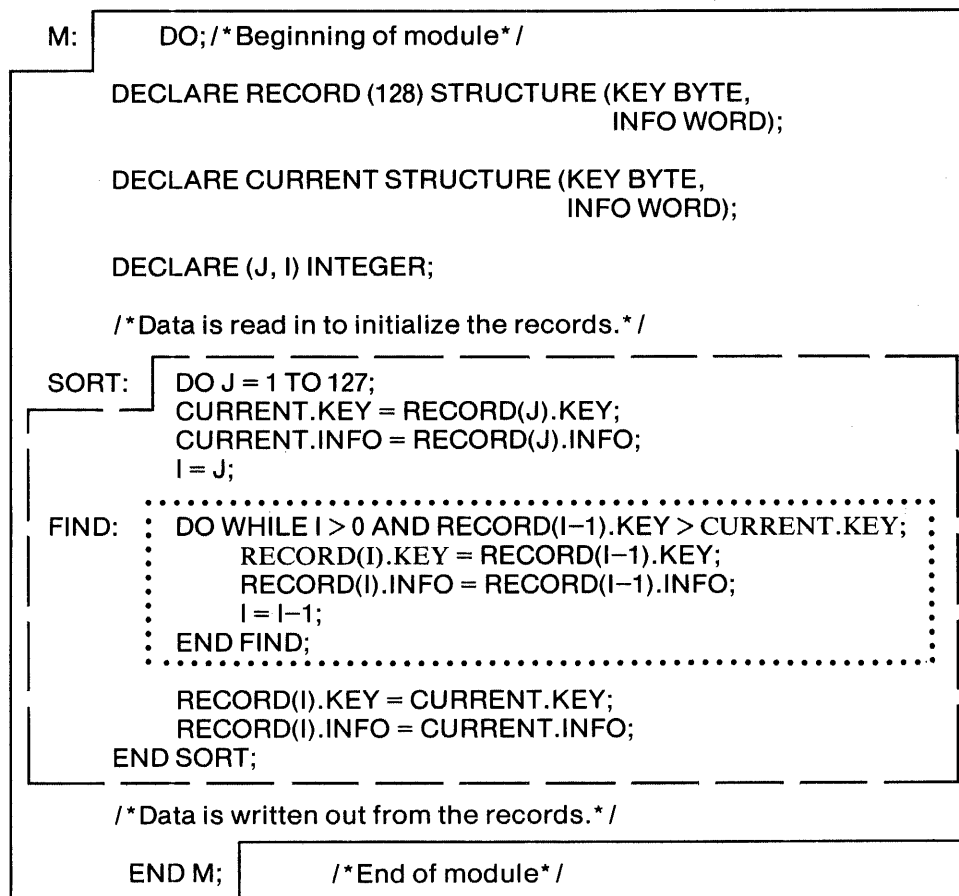
10.2 Scope

Every object that is declared in a PL/M-86 program has scope. This includes

- Variables

- Procedures

- Labels

- Macros

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.

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

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

        DECLARE CURRENT STRUCTURE (KEY BYTE,
                                   INFO WORD);

        DECLARE (J, I) INTEGER;

        /*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*/
```

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.

Figure 10-1. Inclusive Extent of a Block

- 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 9.2.7) 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.

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

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

          DECLARE CURRENT STRUCTURE (KEY BYTE,
                                     INFO WORD);

          DECLARE (J, I) INTEGER;

          /*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*/
```

The shaded area is the exclusive extent of block SORT.

Figure 10-2. Exclusive Extent of a Block

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.

10.3 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 10.1) 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 11):

- 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 9.2.3.

- 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.

11.1 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 10) of a module.

- A *main program module* is a module that contains executable statements at the module level.

11.2 Structure of a Compilation

A "compilation" is one module of PL/M-86 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-86 Compiler. See *ISIS-II PL/M-86 Compiler Operator's Manual.*

11.3 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.

11.4 Linkage Between Program Modules

The compiled modules produced by the PL/M-86 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 activation, and label reference is meaningful.

11.5 Example of Modular Program Structure

Consider once again the sort program used in Chapter 7 and Section 9.4. This program contains the procedure SORTPROC, which might be useful in other programs. Therefore, it might be desirable to compile SORTPROC as a separate module, so that it can be linked to this program and also to any other program that needs to sort records.

Broken into two modules, our example program appears as follows.

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

      SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) PUBLIC;
               DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER;

      /*Parameters:
           PTR is pointer to first record.
           COUNT is number of records to be sorted.
           RECSIZE is number of bytes in each record— max is 128.
           KEYINDEX is byte position within each record of a BYTE scalar
                    to be used as sort key.*/

               DECLARE RECORD BASED PTR (1) BYTE,
                       CURRENT (128) BYTE,
                       (I, J) INTEGER;

      SORT:    DO J = 1 TO COUNT-1;
               CALL MOVB(@RECORD(J*RECSIZE), @CURRENT, RECSIZE);
               I=J;

      FIND:            DO WHILE I > 0
                            AND RECORD((I-1)*RECSIZE + KEYINDEX)
                            > CURRENT(KEYINDEX);
                       CALL MOVB(@RECORD((I-1)*RECSIZE),
                                     @RECORD(I*RECSIZE),
                                     RECSIZE);
                       I = I-1;
                       END FIND;

               CALL MOVB(@CURRENT, @RECORD(I*RECSIZE), RECSIZE);
               END SORT;

      END SORTPROC;


               END SORTMODULE; /*End of 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 9.4, but a usage declaration of the SORTPROC procedure has been substituted for the defining declaration, which is now in the above module.

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

/*Program to sort two sets of records, using SORTPROC*/

SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) EXTERNAL;
            DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER;
          END SORTPROC; /*End of usage declaration*/


          DECLARE SET1(50) STRUCTURE (ALPHA WORD,
                                      BETA(12) BYTE,
                                      GAMMA INTEGER,
                                      DELTA REAL,
                                      EPSILON BYTE);

/*Key of Nth record in SET1 is SET1(N).BETA(0), the 3rd byte in the record.*/


          DECLARE SET2(500) STRUCTURE (ITEMS(21) INTEGER,
                                       KEY BYTE);

/*Key of Nth record in SET2 is SET2(N).KEY, the 43rd byte in the record.*/


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


          CALL SORTPROC(@SET1, INT(LENGTH(SET1)), INT(SIZE(SET1(1))), 2);
          CALL SORTPROC(@SET2, INT(LENGTH(SET2)), INT(SIZE(SET2(1))), 42);


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


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

Builtin procedures and 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 builtin procedure or variable is used in a declaration within the program, the scope of the builtin procedure or variable 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.

No builtin procedure may be used within a location reference. No builtin variable may be used within a location reference, except as specifically noted in the following sections.

12.1 Obtaining Information About Variables

PL/M-86 has three builtin procedures that take variable names as actual parameters and return information based on the declarations of the variables.

12.1.1 The LENGTH Procedure

LENGTH is a WORD procedure that returns the number of elements in an array. It is activated by a function reference, with the form

> LENGTH (variable-ref)

where

- "variable-ref" 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 12.6.2).

The WORD 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 5.3). For example, given the declaration

> DECLARE RECORD STRUCTURE (KEY BYTE,
> INFO (3) WORD);

then LENGTH(RECORD.INFO) is a valid function reference and returns a WORD value of 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) WORD);

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 when a member- identifier is supplied, since the arrays within the structures are all the same length. PL/M-86 allows a "shorthand" form of partially qualified variable reference in the LENGTH, LAST, and SIZE function references:

```
LENGTH(LIST.INFO)
```

12.2.2 The LAST Procedure

LAST is a WORD procedure that returns the subscript of the last element in an array. It is activated by a function reference, with the form

LAST (variable-ref)

where

● "variable-ref" 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 12.6.2).

The WORD 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.

12.1.3 The SIZE Procedure

SIZE is a WORD procedure that returns the number of bytes occupied by an array or structure. It is activated by a function reference, with the form

SIZE (variable-ref)

where

● "variable-ref" is a fully qualified, partially qualified, or unqualified reference to any scalar, array, or structure variable except the MEMORY array (see Section 12.6.2).

The WORD 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 the number of bytes required for the 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.

12.2 Type Conversions

12.2.1 The LOW, HIGH, and DOUBLE Procedures

LOW and HIGH are BYTE procedures that convert WORD values to BYTE values. They are activated by function references with the forms

> LOW (expression)
> HIGH (expression)

where

• "expression" has a WORD or BYTE value.

If "expression" has a WORD value, LOW returns the value of the low-order (least significant) byte of the expression value, whereas HIGH returns the value of the high-order (most significant) byte of the expression value.

If "expression" has a BYTE value, then LOW will return this value unchanged. However, HIGH will return 0.

DOUBLE is a WORD procedure that converts a BYTE value to a WORD value. It is activated by a function reference with the form

> DOUBLE (expression)

where

• "expression" has a BYTE or WORD value.

If "expression" has a BYTE value, the procedure appends 8 high-order 0-bits to convert it to a WORD value and returns this WORD value. If "expression" has a WORD value, the procedure returns this value unchanged.

12.2.2 The FLOAT Procedure

FLOAT is a REAL procedure that converts an INTEGER value to a REAL value. It is activated by a function reference, with the form

> FLOAT (expression)

where

• "expression" has an INTEGER value.

FLOAT converts the INTEGER value to the corresponding REAL value and returns this REAL value.

12.2.3 The FIX Procedure

FIX is an INTEGER procedure that converts a REAL value to an INTEGER value. It is activated by a function reference, with the form

> FIX (expression)

where

* "expression" has a REAL value.

FIX rounds the REAL value to the nearest INTEGER. If both INTEGERs are equally near, FIX rounds to the even one. The resulting INTEGER value is then returned. Thus FIX(1.4) would result in the INTEGER value 1, FIX(−1.8) in −2, FIX(3.5) in 4, an FIX(6.5) in 6.

If the result calculated by FIX is not within the implemented range of INTEGER values, i.e., is greater than 32767, the result is undefined.

NOTE

FIX is affected by your choice of rounding mode—see Chapter 14. The above examples assume the default mode, which is "round to nearest or even."

12.2.4 The INT Procedure

INT is an INTEGER procedure that converts a BYTE or WORD value to an INTEGER value. It is activated by a function reference, with the form

    INT (expression)

where

* "expression" has a BYTE or WORD value.

INT interprets the BYTE or WORD value as a positive number and returns the corresponding INTEGER value.

If the result calculated by INT is not within the implemented range of INTEGER values, the result is *undefined*.

12.2.5 The SIGNED Procedure

SIGNED is an INTEGER procedure that converts a WORD value to an INTEGER value. It is activated by a function reference, with the form

    SIGNED (expression)

where

* "expression" has a WORD or BYTE value. If it has a BYTE value, it will be extended by 8 high-order 0-bits to produce a WORD value.

SIGNED interprets the WORD value as a 16-bit twos-complement number and returns the corresponding INTEGER value.

This means that if the highest-order (most significant) bit of the WORD value is a 0, SIGNED interprets the WORD value as a positive number and returns the corresponding INTEGER value. For example,

    SIGNED (0000$0000$0000$0100B)

returns an INTEGER value of 4.

But if the highest-order bit of the WORD value is a 1, SIGNED returns a negative INTEGER value whose absolute magnitude is the twos complement of the WORD value. For example,

SIGNED (1111$1111$1111$1100B)

returns an INTEGER value of −4.

12.2.6 The UNSIGN Procedure

UNSIGN is a WORD procedure that converts an INTEGER value to a WORD value. It is activated by a function reference, with the form

UNSIGN (expression)

where

* "expression" has an INTEGER value.

UNSIGN converts the INTEGER value to a WORD value.

If the INTEGER value is positive, then the WORD value will be numerically the same as the INTEGER value. But if the INTEGER value is negative, then the WORD value will be the twos complement of the absolute magnitude of the INTEGER value. For example,

UNSIGN (−4)

returns a WORD value equal to 1111$1111$1111$1100B.

12.3 Shift and Rotate Procedures

In shift and rotate operations, a value is handled as a pattern of 8 bits (for a BYTE value) or 16 bits (for a WORD or INTEGER 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 0-bits move into the pattern from the other end (except in the case of SAR—see Section 12.3.3 below). In a rotate, bits moved off one end move onto the other end.

12.3.1 Byte Rotation Procedures, ROL and ROR

ROL and ROR are BYTE or WORD procedures. They are activated by function references, with the forms

ROL (pattern, count)
ROR (pattern, count)

where

* "pattern" is an expression with a BYTE or WORD value, and "count" is an expression with a BYTE value. If the value of "count" is 0, no rotation occurs.

The value of "pattern" is handled as an 8-bit or 16-bit binary quantity which is rotated to the left (by ROL) or to the right (by ROR). The number of bit positions by which it is rotated is specified by "count."

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.

ROR(1101011010011010B, 9) returns a value of 0100110101101011B.

12.3.2 Logical-Shift Procedures, SHL and SHR

SHL and SHR are procedures whose type depends on the type of the value of an expression given as an actual parameter. They are activated by function references, with the forms

SHL (pattern, count)
SHR (pattern, count)

where

- "pattern" and "count" are expressions with BYTE or WORD values. If "count" has a WORD value, the 8 high-order bits will be dropped to produce a BYTE value. If the value of "count" is 0, no shift occurs.

The value of "pattern" may be either a BYTE value or a WORD value and will not be converted. If it is a BYTE value, then the procedure will return a BYTE value. If it is a WORD value, then the procedure will return a WORD value.

The value of "pattern" is shifted left (by SHL) or right (by SHR), with the bit count given by "count".

Note that a shift of one bit position has the effect of multiplication by 2 for a left shift, or division by 2 for a right shift. For example, suppose that VAR is a BYTE variable with a value of 8. This is represented as 0000$1000. SHL(VAR,1) will return 0001$0000, which represents 16, while SHR(VAR,1) will return 0000$0100 which represents 4.

12.3.3 Algebraic-Shift Procedures, SAL and SAR

SAL and SAR are INTEGER procedures. They are activated by function references, with the forms

SAL (pattern, count)
SAR (pattern, count)

where

- "pattern" is an expression with an INTEGER value.
- "count" is an expression with a WORD or BYTE value. If "count" has a WORD value, the 8 high-order bits will be dropped to produce a BYTE value. If the value of "count" is 0, no shift occurs.

SAL and SAR treat the INTEGER value of "pattern" as a bit pattern. This pattern is shifted to the left or to the right.

In a left shift (SAL), 0-bits move into the pattern from the right (as in SHL and SHR).

In a right shift (SAR), either 0-bits or 1-bits move into the pattern from the left. If the original value of "pattern" is positive, the sign bit (leftmost bit) is a 0 and 0-bits move in from the left. If the original value is negative, the sign bit is a 1, and 1-bits move in from the left.

This means that just as with the logical shifts, an algebraic shift of one bit position has the effect of multiplication by 2 for a left shift or division by 2 for a right shift. For example, suppose that VAL is an INTEGER variable with a value of −8. This value is represented as 1111$1111$1111$1000. SAL(VAL,1) will return 1111$1111$1111$0000, which represents −16, while SAR(VAL,1) will return 1111$1111$1111$1100, which represents −4.

12.4 Input and Output

12.4.1 The INPUT and INWORD Procedures

INPUT is a BYTE procedure and INWORD is a WORD procedure. They are activated by function references, with the forms

        INPUT (expression)
        INWORD (expression)

where

*   "expression" has a BYTE or WORD value.

The value of "expression" specifies one of the input ports of the 8086 CPU. The value returned by INPUT is the BYTE quantity found in the specified input port. The value returned by INWORD is the WORD quantity found in the specified input port.

12.4.2 The OUTPUT and OUTWORD Arrays

The builtin variables OUTPUT and OUTWORD are arrays, each with 65536 elements. Each element corresponds to one of the output ports of the 8086 CPU.

OUTPUT is a BYTE array, and OUTWORD is a WORD array.

A reference to OUTPUT or OUTWORD may only appear as the *left* part of an assignment statement or embedded assignment. Anywhere else it is illegal. The right-hand side of the assignment must have a BYTE or WORD value.

The effect of an assignment to an element of OUTPUT is to place the BYTE value of the expression on the right side of the assignment into the corresponding output port. (Since OUTPUT is a BYTE array, the value of the expression will be automatically converted to type BYTE if necessary.)

The effect of an assignment to an element of OUTWORD is to place the WORD value of the expression on the right side of the assignment into the corresponding output port.

12.5 String Manipulaltion Procedures

The string-manipulation procedures (with the exception of XLAT) are available in pairs. One of each pair is for BYTE strings and the other is for WORD strings.

Note that the term "string" is used here in a broader sense than previously. The "character strings" mentioned previously (Sections 2.5, 3.6, and 8.4) are BYTE strings. More broadly, a string is any contiguously stored set of BYTE values or WORD values. We can regard a string as if it were a BYTE or WORD array, and refer to the BYTE or WORD values as "elements."

We will use the word "index" to refer to the position of a given element within a string. The index is like the subscript of an array reference. Thus the index of the first element of a string is 0, the index of the second element is 1, etc.

In the following descriptions, the "location" of a string is always the location of its first element. In each string-manipulation procedure, the location of a string is specified by a parameter called "source" or "destination," which is an expression with a POINTER value. Thus the source points to the lowest element. For MOVB and MOVW, this is the first element to be processed. For MOVRB and MOVRW, it is the last element to be processed.

The "length" of a string is the number of elements it contains. In each string-manipulation procedure, the length of a string is specified by a parameter called "count," which is an expression with a WORD or BYTE value representing the number of elements.

These procedures can also be used to initialize arrays.

12.5.1 The MOVB Procedure

MOVB is an untyped procedure that copies a BYTE string from one location to another. It is activated by a CALL statement with the form

        CALL MOVB (source, destination, count) ;

where

*   "source" and "destination" are expressions with POINTER values.

*   "count" is an expression with a BYTE or WORD value.

The value of "source" is the location of the string to be copied, and the value of "destination" is the location to which the string is to be copied.

The string elements are copied in *ascending* order—that is, element 0 is copied first, then element 1, etc. This is significant if the source string and the destination string overlap. If the value of "destination" is higher than the value of "source," and the two strings overlap, elements in the overlap area will be overwritten before they are copied. This can be avoided by using MOVRB instead of MOVB (see Section 12.5.3 below).

If the two strings overlap, but the value of "source" is higher than the value of "destination," then elements in the overlap area will not be overwritten until after they have been copied.

A procedure such as the following can be used to make the correct choice between MOVB and MOVRB.

```
MOVBYTES: PROCEDURE(SRC, DST, CNT);
      DECLARE (SRC, DST) POINTER, CNT WORD;
      IF SRC > DST THEN CALL MOVB(SRC, DST, CNT);
      ELSE CALL MOVRB(SRC, DST, CNT);
END MOVBYTES;
```

This procedure can be activated without the need to consider whether overlap may occur or whether "source" or "destination" is higher.

12.5.2 The MOVW Procedure

MOVW is the same as MOVB (see Section 12.5.1 above), except that it copies a WORD string instead of a BYTE string.

12.5.3 The MOVRB Procedure

MOVRB is the same as MOVB (see Section 12.5.1 above), except that the elements in the source string are copied to the destination string in *descending order*. This is significant when the two strings overlap. If the value of "destination" is higher than the value of "source," and there is overlap, elements in the overlap area will not be overwritten until *after* they have been copied. However, if the value of "source" is higher than the value of "destination," then elements in the overlap area will be overwritten before they are copied.

12.5.4 The MOVRW Procedure

MOVRW is the same as MOVRB (see Section 12.5.3 above), except that it copies a WORD string instead of a BYTE string.

12.5.5 The CMPB Procedure

CMPB is a WORD procedure that compares two BYTE strings. It is activated by a function reference with the form

    CMPB (source1, source2, count)

where

*   "source1" and "source2" are expressions with POINTER values.
*   "count" is an expression with a BYTE or WORD value.

CMPB compares two BYTE strings of length "count," whose locations are "source1" and "source2."

If every element in the string at "source1" is equal to the corresponding element in the string at "source2," CMPB returns a WORD value of 0FFFFH. Otherwise, it returns the index (position within the strings) of the first pair of elements found to be unequal.

12.5.6 The CMPW Procedure

CMPW is the same as CMPB (see Section 12.5.5), except that it compares two WORD strings instead of two BYTE strings.

12.5.7 The XLAT Procedure

XLAT is an untyped procedure that "translates" a BYTE string to produce another BYTE string, using a translation table. It is activated by a CALL statement of the form

        CALL XLAT (source, destination, count, table) ;

where

* "source," "destination," and "table" are expressions with POINTER values.
* "count" is an expression with a BYTE or WORD value.

XLAT "translates" the BYTE elements in the source string, placing the translated elements in the destination string. The value of "table" is assumed to be the location of a BYTE string of up to 256 elements. This string is used as a *translation table*.

The value of an element in the source string is used as an index for the translation table. The index selects one element from the translation table, and this element is then copied into the destination string.

For example, if the fifth element in the source string is 202, then 202 is used as an index for the translation table. The 203rd element of the table is copied into the fifth position in the destination string.

The elements of the source string are translated into the destination string in *ascending* order.

12.5.8 The FINDB Procedure

FINDB is a WORD procedure that searches a BYTE string to find an element that has a specified value. It is activated by a function reference of the form

        FINDB (source, target, count)

where

* "source" is an expression with a POINTER value.
* "target" is an expression with a BYTE or WORD value. If "target" has a WORD value, the 8 high-order bits will be dropped to produce a BYTE value.
* "count" is an expression with a BYTE or WORD value.

FINDB examines each element of the source string (in ascending order) until it finds an element whose value is equal to the BYTE value of "target"—or until "count" elements have been searched without any of them matching "target." If the search is successful, FINDB returns the index of the first element of the string that matches "target." If the search is unsuccessful, FINDB returns a WORD value of 0FFFFH.

12.5.9 The FINDW Procedure

FINDW is the same as FINDB (see Section 12.5.8 above), except that it searches a WORD string instead of a BYTE string. If the "target" parameter has a BYTE value, it is extended by 8 high-order 0-bits to produce a WORD value.

12.5.10 The FINDRB Procedure

FINDRB is the same as FINDB (see Section 12.5.8 above), except that it searches the source string in *descending* order. Thus if the search is successful, FINDRB returns the index of the *last* element that matches the BYTE value of "target."

12.5.11 The FINDRW Procedure

FINDRW is the same as FINDRB (see Section 12.5.10 above), except that it searches a WORD string instead of a BYTE string (in descending order).

12.5.12 The SKIPB Procedure

SKIPB is a "converse" of FINDB (see Section 12.5.8 above). Instead of searching for the first element in the BYTE source string that matches the BYTE value of "target," SKIPB searches for the first element that does *not* match.

In every other respect, the operation is exactly the same as FINDB.

12.5.13 The SKIPW Procedure

SKIPW is a "converse" of FINDW (see Section 12.5.9 above). Instead of searching for the first element in the WORD source string that matches the WORD value of "target," SKIPW searches for the first element that does *not* match.

In every other respect, the operation is exactly the same as FINDW.

12.5.14 The SKIPRB Procedure

SKIPRB is a "converse" of FINDRB (see Section 12.5.10 above). Instead of searching for the last element in the BYTE source string that matches the BYTE value of "target," SKIPB searches for the last element that does *not* match.

In every other respect, the operation is exactly the same as FINDRB.

12.5.15 The SKIPRW Procedure

SKIPRW is a "converse" of FINDRW (see Section 12.5.11 above). Instead of searching for the last element in the WORD source string that matches the WORD value of "target," SKIPB searches for the last element that does *not* match.

In every other respect, the operation is exactly the same as FINDRW.

12.5.16 The SETB Procedure

SETB is an untyped procedure that sets each element of a BYTE string to a single specified value. It is activated by a CALL statement with the form

CALL SETB (newvalue, destination, count) ;

where

- "newvalue" is an expression with a BYTE or WORD value. If it has a WORD value, the 8 high-order bits are dropped to produce a BYTE value.
- "destination" is an expression with a POINTER value.
- "count" is an expression with a BYTE or WORD value.

SETB assigns the BYTE value of "newvalue" to each element in the destination string.

12.5.17 The SETW Procedure

SETW is the same as SETB, except that it assigns a single WORD value to each element of a WORD string instead of a BYTE string.

If "newvalue" has a BYTE value, it will be extended by 8 high-order 0-bits to produce a WORD value.

12.6 Miscellaneous Builtins

12.6.1 The MOVE Procedure

MOVE is an untyped procedure that is provided for compatibility with PL/M-80 programs. It is activated by a CALL statement with the form

CALL MOVE (count, source, destination) ;

where

- "count," "source," and "destination" are expressions with WORD or BYTE values. If any of these parameters has a BYTE value, it will be extended by 8 high-order 0-bits to produce a WORD value.

The values of "source" and "destination" are assumed to be the WORD-type addresses of the source string and the destination string. The operation differs from MOVB (see Section 12.5.1) as follows:

- All three parameters must have either BYTE or WORD values, and will be converted to WORD values if they are BYTE values. POINTER values for "source" and "destination" are not allowed and therefore the values cannot be supplied by means of the @ operator. Thus MOVE can only handle strings whose locations can be expressed as WORD addresses.
- Note that the parameters are in a different order.
- If the source and destination strings overlap, the results are always *undefined*.

12.6.2 The MEMORY Array

MEMORY is a BYTE array of *unspecified* length which represents an uninitialized (free) segment of 8086 storage. 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 location of the beginning of free memory space, i.e., byte 0 of the memory segment.

A reference to MEMORY may not be used as an actual parameter for the LENGTH, LAST, and SIZE procedures.

12.6.3 The TIME Procedure

The untyped procedure TIME causes a time delay specified by its actual parameter. It is activated by a CALL statement, with the form

        CALL TIME (expression);

where the expression is converted, if necessary, to a WORD 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 statement

        CALL TIME (45);

causes a delay of 4.5 milliseconds. Since the maximum delay offered by the procedure is about 6.55 seconds, longer delays must be obtained by repeated activations. The following block takes one second to execute:

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

The TIME procedure is based on 8086 CPU cycle times, and assumes that the system is running at 5 MHz without interruptions.

12.6.4 STACKPTR and STACKBASE

STACKPTR and STACKBASE are builtin WORD variables that provide access to the 8086 hardware stack pointer and stack base registers.

Care must be exercised in setting these registers (that is, using STACKPTR or STACKBASE 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.

12.6.5 The ABS Procedure

ABS is a REAL procedure that returns the absolute value of a REAL value. It is activated by a function reference with the form

        ABS (expression)

where

• "expression" has a REAL value.

If the value of "expression" is positive, ABS returns it unchanged. If the value of "expression" is negative, ABS returns −(expression).

12.6.6 The IABS Procedure

IABS is an INTEGER procedure that returns the absolute value of an INTEGER value. It is activated by a function reference with the form

IABS (expression)

where

- "expression" has an INTEGER value.

If the value of "expression" is positive, IABS returns it unchanged. If the value of "expression" is negative, IABS returns −(expression).

12.6.7 The LOCKSET Procedure

The LOCKSET procedure permits a programmer to implement a simple software lock. It is a BYTE procedure called by a function reference with the form

LOCKSET(lockptr, newvalue)

where

- "lockptr" is an expression with a POINTER value.

- "newvalue" is an expression with a BYTE or WORD value. If "newvalue" has a WORD value, the 8 high-order bits will be dropped to produce a BYTE value.

The action of LOCKSET is as follows: The "lockptr" parameter is used as a pointer to a BYTE variable. The value of "newvalue" is assigned to this variable, and LOCKSET returns the original value of the variable. During this transaction, a hardware lock is set on the memory bus to prevent any interference from another processor. However, the hardware lock is released before LOCKSET returns.

To see how this facility can be used, consider a system having more than one 8086 processor using the same memory, and consider a program in one of these processors. Suppose that this program uses memory locations that are also used by other processors in the system.

Within certain "critical" regions of our program, we want to ensure that no other processor will access the shared memory locations. To achieve this, we declare a BYTE variable called LOCK and establish a convention that if LOCK=0, any processor in the system may access the shared memory locations. But if LOCK=1, no processor may access the shared memory locations unless it was the one that set LOCK to 1.

Now if we write the function reference LOCKSET(@LOCK, 1), the value 1 is assigned to LOCK. Furthermore, if the value returned by LOCKSET is 0, then LOCK was not already set and so this processor is the one that set it. We are now allowed, by convention, to enter the critical region of our program and access the shared memory locations. At the end of the critical region, we must release the lock by writing LOCK=0.

But if LOCKSET returns a value of 1, then LOCK was already set and this processor was not the one that set it. By convention, we must wait until a LOCKSET(@LOCK, 1) function reference returns a value of 0 before accessing the shared memory locations.

Thus our program could contain the following construction:

```
/*Begin critical region*/

DO WHILE LOCKSET(@LOCK, 1);
        /*Do nothing but repeat until LOCKSET returns 0*/
END;

/*Now LOCK has been set to 1 by this processor*/


...
/*Critical region of program, where shared memory
  locations are accessed*/
...

LOCK=0;
/*End critical region*/
```

In the simple case just described, only one software lock is used. It is represented by the variable LOCK. But if more than one set of memory locations needed protection at different times, we could establish as many different software locks as necessary, each using a different BYTE variable.

Also, note that a software lock can be used for other purposes than protecting memory locations. LOCKSET provides a mechanism that can be used to implement various types of synchronization in a multiprocessor system.

12.6.8 The SET$INTERRUPT Procedure

This procedure permits a program in execution to set an interrupt vector to point to the interrupt entry point of a separately compiled interrupt handling routine, or to alter such vectors dynamically. See also section 9.2 of this manual and Chapter 10 of the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.

The procedure is invoked by a CALL of the form

    CALL SET$INTERRUPT (constant, name)

where

• "name" is the interrupt procedure name

and

• "constant" is an interrupt number, i.e., a whole-number constant between 0 and 255.

12.6.9 The INTERRUPT$PTR Procedure

This builtin function returns the interrupt entry point. Its form is INTERRUPT$PTR (name). It is typically used in an assignment statement, e.g.,

    INT$ARRAY (4) = INTERRUPT$PTR (HANDLER__PROC__4)

The interrupt entry point is not readily accessible without using this function, since the @ operator refers to the procedure entry point instead. These differences are discussed in greater detail in Chapter 10 of the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.

12.6.10 The CAUSE$INTERRUPT Procedure

This procedure causes a software interrupt to the vector specified in the statement

CAUSE$INTERRUPT (constant)

The PL/M-86 features described in this chapter make use, directly or indirectly, of the 8086 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-86 program to modules containing code to perform the same functions, but written in assembly language.

13.1 Optimization and the 8086 Hardware Flags

In order to produce an efficient machine-code program from a PL/M-86 source, the PL/M-86 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-86 source statements cannot be predicted.

Consequently, the state of the 8086 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-86 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-86 feature that is dependent on 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.

13.2 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 8086 CPU hardware CARRY flag in performing the operation.

13.3 Carry-Rotation Builtin Procedures

SCL and SCR are built-in rotation procedures whose type depends on the type of the value of an expression given as an actual parameter. They are activated 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 0, no rotation occurs.

The value of "pattern" may be either a BYTE value or a WORD value and will not be converted. If it is a BYTE value, then the procedure will return a BYTE value. If it is a WORD value, then the procedure will return a WORD value.

The value of "pattern" is rotated left (by SCL) or right (by SCR), with the bit count given by "count," just as with the ROL and ROR procedures described in Chapter 12. But with SCL and SCR, the rotation includes the CARRY flag: the bit rotated off one end of "pattern" is rotated into CARRY, and the old value of CARRY is rotated into the other end of "pattern." In effect, SCL and SCR perform 9-bit rotations on 8-bit values, and 17-bit rotations on 16-bit values.

13.4 The DEC Procedure

DEC is a built-in BYTE procedure which uses the value of the hardware CARRY flag internally. It is activated 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.

13.5 CARRY, SIGN, ZERO, and PARITY Builtin Procedures

There are four built-in BYTE procedures that return the logical values of the 8086 hardware flags. These procedures take no parameters, and are activated by function references with the following forms:

```
CARRY
ZERO
SIGN
PARITY
```

An occurrence of one of these activations (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.

This chapter covers the general aspects of the design of the REAL math facility used to support REAL arithmetic in PL/M-86, plus REAL error control and the use of REALs by interrupting programs. This facility operates as described herein, whether performed by the INTEL 8087 chip or INTEL emulators. The discussions therefore make no distinction as to that environment, except in Chapter 6 of the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*.

<div align="center">NOTE</div>

If your program performs ANY floating-point arithmetic or assignments, it must first initialize the REAL math facility by calling the procedures INIT$REAL$MATH$UNIT (Section 14.5) and, optionally, SAVE$REAL$STATUS (Section 14.8), and SET$REAL$MODE (Section 14.8). (However, it must be noted that PL/M-86 programs using REAL assignments and arithmetic cannot be accommodated on an SDK-86 board.)

The use of REAL functions within REAL expressions can lead to stack overflow because PL/M-86 does not clear the stack before the function call. The recommended practice is to use an assignment statement first to store the functions value in a REAL variable, and then use that variable in the longer expression.

14.1 Representation of REAL Values

This section describes Intel's standard single-precision format for floating-point arithmetic. All PL/M-86 REAL values use this format.

A REAL value occupies four contiguous memory bytes, which may be viewed as 32 contiguous bits. The bits are divided into fields as follows:

sign (1 bit)	exponent (8 bits)	fraction (23 bits)

where
- The byte with the *lowest* address contains the least significant 8 bits of the fraction field, and the byte with the *highest* address contains the sign bit and the most significant 7 bits of the exponent field.
- The *sign* bit is 0 if the REAL value is positive or zero, or 1 if the REAL value is negative.
- The *exponent* field contains a value "offset" by 127—in other words, the actual exponent can be obtained from the exponent field value by subtracting 127. This field is all 0's if the REAL value is zero.
- The *fraction* field contains the binary digits of the fractional part of the REAL value, when it is represented in "binary scientific" notation (see below). This field is all 0's if the REAL value is zero.

The following examples illustrate these concepts.

Consider the following binary number (which is equivalent to the decimal value 10.25):

1010.01B

(The "." in this number is a *binary point*.) The same number can be represented as

$1.01001B * 2^3$

This is "binary scientific" notation, with the binary point immediately to the right of the most significant digit. The digits 01001 are the fractional part, and 3 is the exponent. This value would be represented as follows:

- The sign bit would be 0, since the value is positive.
- The exponent field would contain the binary equivalent of 127 + 3=130.
- The leftmost digits of the fraction field would be 01001, and the remainder of this field would be all 0's.

The complete 32-bit representation would be

0 10000010 01001000000000000000000

and the contents of the four contiguous memory bytes would be as follows:

highest address:	01000001
	00100100
	00000000
lowest address:	00000000

Note that the most significant digit is not actually represented, since by definition it is a "1" unless the REAL value is zero. If the REAL value is zero, the entire 32-bit representation is all 0's.

For a second example, consider the fraction 1/16, or 0.0625. In binary, this is

0.0001B

In "binary scientific" we would have

$1.0000B * 2^{-4}$

The actual exponent, −4, would be represented as 123 (127-4), and the fraction field would contain all 0's.

The largest possible value for a valid exponent field is 254, which corresponds to an actual exponent of 127. The largest possible absolute value for a positive or negative REAL value is therefore

$1.11111111111111111111111B * 2^{127}$

or approximately $3.37*10^{38}$.

The lowest permissible exponent field value for a non-zero REAL value is 1, which corresponds to an actual exponent of −126. The smallest possible absolute value for a positive or negative REAL value is therefore

$1.0B * 2^{-126}$

or approximately $1.17*10^{-38}$.

The utility of the REAL data type is extended by the PL/M-86 compiler's practice of holding intermediate results in the 8087's temporary-real format, preserving 64 bits of precision and the full range of representable numbers. The exponent in this format is 15 bits instead of 11 or 8 in the long- and short-real formats, respectively.

This greater range of exponent greatly reduces the likelihood of underflow and overflow, and eliminates roundoff as a source of error until the final assignment of the result is performed. These advantages arise because underflow, overflow, and roundoff errors are more probable for intermediate computations than for the final result. For example, an intermediate underflow result might later be multiplied by a very large factor, providing a final result of acceptable magnitude.

14.2 REAL-Parameter Passing and Stack Conventions

The first seven REAL parameters of a procedure or function are passed by value, pushed onto the 8087 stack in the order in which they are specified in the CALL. (Thus the seventh is on top.) The values of any remaining parameters after the seventh, plus all non-REAL parameters, are pushed onto the 8086 stack, last on top.

The 8087 stack is organized and used with top-relative addressing and operations, permitting different routines to call a common subroutine without observing a convention for passing parameters in dedicated registers. Only the order, type, and number of the parameters need be consistent. Results from procedures typed REAL are returned on the top of the REAL stack.

14.3 The REAL Math Facility

From the program's point of view, the facility consists of the following:

* The REAL stack, used to hold operands and results during REAL operations
* The REAL Error Byte, consisting of 8 bits initialized to all 0's.

    The first six bits in this byte correspond to the possible errors that can arise during REAL operations (see Section 14.4 below). When an error occurs, the facility sets the corresponding bit to 1. There is a built-in procedure described in Section 14.7 that a program can invoke to read and clear the REAL Error Byte.

    The exception/error categories are discussed below in Section 14.4.
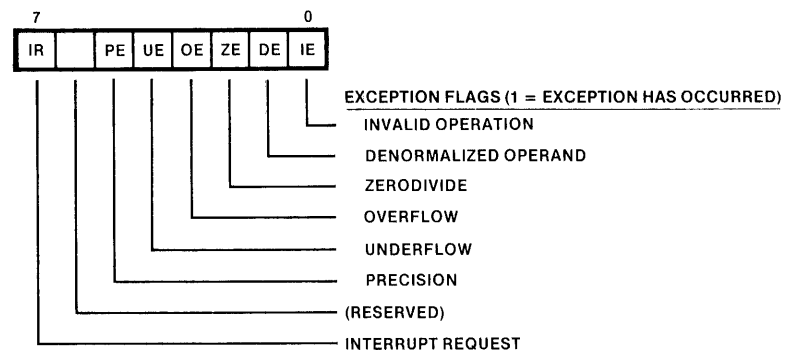
Figure 14-1. The REAL Error Byte

(1) Interrupt-Enable Mask:
   0 = Interrupts Enabled
   1 = Interrupts Disabled (Masked)
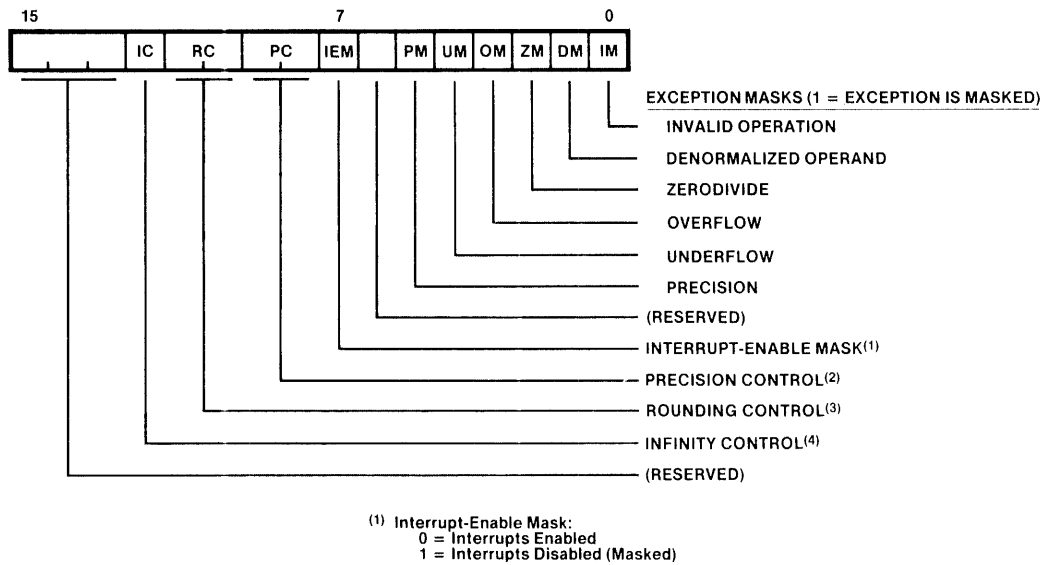
Figure 14-2. The REAL Mode Word

• The REAL mode word, consisting of 16 bits initialized to 03BFH.

Bits 0-5 determine whether the corresponding error condition is to be handled by using the default recovery described below or by using the programmer-supplied exception procedure (see Secton 14.9 for details on writing these). When the bit is 1, the default is used. When it is 0, the user routine is used. In either case the facility records the error by setting the corresponding bit of the REAL Error Byte. For most uses, the default recovery is appropriate, and less work.

This mode word is often called a mask, i.e., it lets some signals through (to interrupt processing) and not others. If one of the bits 0-5 is a 0, the corresponding error is said to be unmasked. (See Section 14.6 on how to set the mode word.)

If one of these bits is 0 and the corresponding error occurs during floating point processing, the REAL math facility interrupts the host CPU, e.g., an 8086, with interrupt 16. The exception condition is thus reported, and control passed, to the user-written error handling routine. This situation is called an un-masked error. Sections 12.6.8, 12.6.9, and 9.2.6 discuss aspects of interrupt procedures.

Conversely, a "masked error" means the mode bit corresponding to that error is 1. Masked errors do not cause an interrupt 16, but are handled as described in Section 14.4, Exception Conditions.

Bits 13, 14, 15 are reserved and not for PL/M-86 use. Bits 8-12 provide options for controlling precision, rounding, and infinity representation, as follows:

BITS	SET TO	MEAN
(1) 9,8	00	24-bit precision
	11	64-bit precision [default]
(2) 11,10	00	Round to nearest (or even) [default]
	01	Round Down (toward −infinity)
	10	Round Up (toward +infinity)
	11	Chop (truncate toward 0)

<table>
<tr><td>12</td><td>0</td><td>Projective Infinity: +infinity=−infinity [default]</td></tr>
<tr><td></td><td>1</td><td>Affine Infinity: +infinity > −infinity
(see Figure 14-3)</td></tr>
</table>
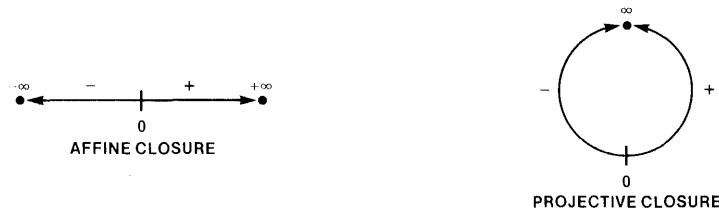
Figure 14-3. Projective Versus Affine Closure

(1) All intermediate results are held in an internal format of 64-bit precision. The most-significant 24 bits of the final result are returned (plus sign and 7-bit exponent) as the PL/M-86 answer, rounded if needed according to the user-specified control. The default precision setting preserves extended precision and operates slightly faster than the other.

(2) Rounding introduces an error of less than one unit in the last place to which the result is rounded. The default provides the statistically most accurate and unbiased estimate of the "true result", i.e., the 64-bit result. In all rounding modes except "round down", subtracting a number from itself yields +0; round down yields −0.

The full extent of the 8087's numeric and operational capabilities are discussed in the *8086 Family User's Manual Supplement for the 8087 Numeric Data Processor*.

14.4 Exception Conditions In REAL Arithmetic

As indicated in Figure 14-1, there are six exception conditions that apply to normal numeric operations:

- invalid operation
- denormalized operand
- zero divide
- overflow
- underflow
- precision

These are discussed in order below. In each case, only a few of the possible causes are described, because most are not likely in common PL/M-86 usage. Sophisticated numeric processing of extreme precision and flexibility may be performed. For full information at that level, see the *8086 Family User's Manual Supplement for the 8087 Numeric Data Processor*.

As the sections following indicate, the masked, default response to most exceptions will provide the least abrupt, most appropriate action for most PL/M-86 applications. Infrequency of exception conditions is almost guaranteed by the extreme range of the temporary-real format (64-bit precision) used to hold intermediate results. The "soft" recovery of gradual underflow, described under the denormal-exception, also extends the range of permissible execution rather than reporting a hard-failure condition.

14-5

Programmers who use the recommended setting of the REAL Mode Word (see Section 14.6) need to handle only the invalid-exception. Study of the information from the end of Section 14.4.1 to Section 14.5 is advised in that it provides a general overview of the meaning of the other exception conditions. Section 14.9 describes writing the exception handler.

14.4.1 Invalid Operation Exception

This exception generally indicates a program error. It could be caused by referencing an uninitialized REAL variable or a location that does not contain a REAL value (as might occur with an out-of-range subscript for a REAL array). Attempting to take the square root of a negative number or to store a number too large for integer format would also generate this exception.

Another interpretation of this exception is stack error. This may be caused by failing to restore the math unit status before returning from an interrupt routine that had saved the status. Another cause is the generation of more than 8 intermediate results during REAL arithmetic, which can arise if REAL procedure function calls are nested too deeply. The compiler ensures that no single procedure does this, but cannot check what may occur only at run time. This exception can also occur when REAL functions (typed procedures) are used as operands within longer REAL expressions, for example,

DELTA$1 = ALPHA * (BETA / GAMMA) + CHI (PSI, RHO, PI)

where all these names are typed REAL and CHI is some previously declared REAL function of three parameters.

The following is less likely to cause an exception condition:

EPS = CHI (PSI, RHO, PI)
DELTA$1 = ALPHA * (BETA / GAMMA) + EPS

Here's why:

All REAL arithmetic is performed using the 8087 stack (actual or emulated), which has eight registers. The first seven REAL parameters supplied in procedure calls are placed on this stack. If the procedure is typed, that is, invoked as a function, it can be imbedded as one operand within a longer REAL expression.

Since the evaluation of such an expression also involves the use of this stack for prior and subsequent arithmetic operations, stack overflow can occur. This overflow amounts to unpredictable destruction of original parameters or intermediate results. It becomes more likely as you increase the complexity of REAL expressions containing REAL functions. Thus you are safer using an assignment statement first, to store the function's value in a real variable, and using that variable in the larger expression.

If stack error might apply to your program, modify the code for the affected procedures to call the built-in procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS as their first and last operations, respectively.

The masked (default) response is to set the result to one of the special bit patterns called Not-A-Number (NANs), usually the indefinite value, the smallest number representable in the specified precision. It also sets Bit 0 of the REAL Error Byte.

If Bit 0 of the REAL Mode Word is 0 (invalid-exception unmasked), an interrupt 16 occurs, transferring control to the user-written interrupt handler.

14.4.2 Denormal Operand Exception

This condition arises when numeric operations have resulted in a number whose exponent is literally zero and whose significand, while non-zero, does not begin with a 1, as it would if it were normalized.

The masked, default response is to adjust the significand, moving significant digits off to the right and raising the exponent until the latter becomes non-zero. For example, a 24-bit significand of .01 with an exponent of zero implies the number 1×2^{-129}, since a zero exponent in this format means -127. This would be adjusted into a significand of .00001 with an exponent of 1, i.e., 0.00001×2^{-126}. Then this number would be available for use in subsequent REAL operations, which might well yield valid results. For example, a small denormal multiplied by a large normal REAL number can give an acceptable, in-range answer. In practice, since intermediate results are kept in temporary real format (15-bit exponent), denormals are very rare.

This condition causes Bit 1 of the REAL Error Byte to be set to 1. If Bit 1 of the REAL Mode Word is 1, the response described above occurs; if 0, an interrupt 16 occurs, transferring control to the user-written interrupt handler.

14.4.3 Zero Divide Exception

This condition arises when, in the course of some REAL computation, a divisor turns out to be zero. The masked response, when Bit 2 of the REAL Mode Word is 1, is to return infinity, appropriately signed if need be. If that bit is 0, interrupt 16 occurs, giving control to the user-written interrupt handler. In either case, Bit 2 of the REAL Error Byte is set to 1.

14.4.4 Overflow Exception

This error occurs when a real result is too large for the format in use, i.e., for REAL assignment, greater than about 3.37×10^{38}, or for intermediate REAL computations using the extended format, greater than about 10^{4932}. It can arise during assignment, addition, subtraction, multiplication, division, or conversion to integer.

The masked, default response (Bit 3 of REAL Mode Word = 1) is to return infinity (signed if Affine) and set Bit 3 of the REAL Error Byte to 1. Unmasked overflow must go through a user-written interrupt 16 handler.

14.4.5 Underflow Exception

This exception is caused by an exponent too small for the format in use, i.e., for REAL assignments, less than -127, and for intermediate results, less than -16383. Underflow can be caused by the same type of REAL operations as overflow.

The masked, default response (Bit 4 of REAL Mode Word = 1) is to use the denormal number created by adjusting the very small result (see denormal discussion above), or to return zero if that is the rounded result. Bit 4 of the REAL Error Byte is set to 1. Unmasked underflow must go through a user-written interrupt 16 handler.

14.4.6 Precision Exception

This error occurs when the result of an operation is inexact, i.e., rounded, and as a result of an overflow exception. No special action is performed by a masked response (Bit 5 of REAL Mode Word = 1) other than setting Bit 5 of the REAL Error Byte. Unmasked response is as chosen by the user.

Table 14-1. Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN, return it; if both are NANs, return NAN with larger absolute value; if neither is NAN, return *indefinite* NAN.	Request interrupt.
Zerodivide	Return ∞ signed with "exclusive or" of operand signs.	Request interrupt.
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then re-evaluate for exceptions.	Request interrupt.
Overflow	Return properly signed ∞.	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt.

* On overflow, 24,576 decimal is *subtracted* from the true result's exponent; this forces the exponent back into range and permits a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is *added* to the true result's exponent.

14.5 The INIT$REAL$MATH$UNIT Procedure

INIT$REAL$MATH$UNIT is a built-in untyped procedure activated by a CALL statement, as follows:

    CALL INIT$REAL$MATH$UNIT;

This call is required as the first access to the REAL math facility, irrespective of whether the 8087 chip or its software emulator will be used. That decision can be deferred until link time, and the proper controls are described in the *PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems Manual*

The effect of this call is to initialize the REAL math unit for subsequent operation. This includes setting a default value into the control word, namely 03BFH or 0000001110111111 in binary. The effect of this setting is to mask all exceptions and interrupts, set precision to 64 bits, and cause rounding to even (as described in Section 12.6.8). This means no interrupts will occur from the REAL Math Facility regardless of what errors are detected. See also Section 14.6 below.

Procedures which are activated after this call has taken effect do not need to do such initialization. See also Section 14.8.

14.6 The SET$REAL$MODE Procedure

SET$REAL$MODE is a built-in untyped procedure, activated by a CALL statement with the following form:

    CALL SET$REAL$MODE (modeword) ;

where

• modeword is an expression with a WORD value.

The value of modeword becomes the new contents of the REAL mode word. If bits 6,7,13,14, and 15 are not 0 in modeword then the results are undefined. The suggested value for modeword is 033EH, that is, 0000001100111110 in binary. This value provides maximum precision, default rounding, and masked handling of all exception conditions except invalid, which can alert you to errors of initialization or stack usage. See Section 14.9 for facts and references on writing an interrupt handling procedure.

14.7 The GET$REAL$ERROR Procedure

GET$REAL$ERROR is a built-in BYTE procedure activated by a function reference with the following form:

    GET$REAL$ERROR

The BYTE value returned is the current contents of the REAL error byte. This procedure also clears the error byte in the REAL math facility.

14.8 Saving and Restoring REAL Status

If any interrupt procedure performs any floating point operation, it will change the REAL status. If such an interrupt procedure is activated during a floating point operation, the program will be unable to continue the interrupted operation correctly after return from the interrupt procedure. Therefore, it is necessary for any interrupt procedure that performs a floating-point operation to first save the REAL status and subsequently restore it before returning. The built-in procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS make this possible.

These procedures can also be used in a multi-tasking environment, where a running task using the 8087 may be preempted by another task that also uses the 8087. The preempting task must call SAVE$REAL$STATUS before it executes any statements that affect the 8087. This means before calling INIT$REAL$MATH$UNIT and SET$REAL$MODE, and before any arithmetic or assignment of REALs (other than GET$REAL$ERROR, if needed).

New vectors will be required for the interrupt handlers appropriate to each new task, e.g., to handle unmasked exception conditions. These vectors can be placed in the correct locations via the SET$INTERRUPT procedure described in Section 12.6.8. Multitasking must be disabled during this operation.

After its processing is complete and the preempting task is ready to terminate, it must call RESTORE$REAL$STATUS to reload the state information that applied at the time the former running task was preempted. This enables that task to resume execution from the point where it relinquished control.

NOTE
The 8087 Emulator is not supported for multitasking by the RMX/86 Real-Time Operating System, due to the requirement for dynamic storage allocation (i.e., memory reallocation during execution) in such an environment.

WARNING

The use of real functions without real parameters within real expressions may result in loss of processor synchronization. Do not call GET$REAL$ERRORS or SAVE$REAL$STATUS from such a function before executing at least one floating point instruction.

14.8.1 The SAVE$REAL$STATUS Procedure

SAVE$REAL$STATUS is a built-in untyped procedure activated by a CALL statement with the form

    CALL SAVE$REAL$STATUS (location) ;

where

• location is a pointer to a memory area of 100 bytes where the REAL status information will be saved.

The REAL status is saved at the specified location, and the REAL stack, mode word, and error byte are reinitialized.

If the state of the REAL math unit is unknown to this procedure when it is called, as in the case mentioned above for preempting tasks, then you don't want to do an initialization because that will destroy existing error flags, masks, and control settings. The action appropriate to these circumstances (except for error-recovery routines, discussed later) is to issue

    CALL SAVE$REAL$STATUS (location__1)

before any REAL math usage and, prior to the procedure's return, a CALL RESTORE$REAL$STATUS (location__1), as described below. The save automatically reinitializes the math unit and the error byte.

This protects the status of preempted tasks or prior procedures and establishes a known initialization state for the current procedure's actions. 8086 interrupts are disabled during the save.

14.8.2 Deadlock

When you use an actual 8087 Numeric Data Processor chip (NDP), a problem called "deadlock" can arise if 8086 interrupts are disabled:

    The 8086 processor can have interrupts disabled when it enters a WAIT state. The wait is to allow the NDP to complete a current operation so as to synchronize the two processors for subsequent instructions.

    If an 8087 exception now occurs, the NDP sends an interrupt signal to the 8086 chip. Until that exception is cleared by a call to GET$REAL$ERROR or SAVE$REAL$STATUS, the NDP remains in a "busy" state. But that interrupt signal never arrives because 8086 interrupts were disabled. Therefore each processor indefinitely awaits the other.

When you use the Emulator, it ignores a WAIT condition, so that if 8086 interrupts are disabled it simply executes a return. If 8086 interrupts are enabled, then the Emulator checks the NDP's interrupt mask: if it is zero (meaning 8087 interrupts are enabled), the Emulator then interrupts the 8086; if the NDP's interrupt mask is one (8087 interrupts disabled), the Emulator simply executes a return.

14.9 Writing a Procedure to Handle REAL Interrupts

This section partially summarizes advice, notes, and warnings from Chapters 9, 12, and 14 pertaining to interrupts, floating-point usage, and procedures.

(It does not duplicate all of the information to be found there as to additional capabilities which may be permitted or disallowed, e.g., the attributes PUBLIC or REENTRANT may be applied to an interrupt procedure, but the attribute EXTERNAL may not. INTERRUPT may only be used in an untyped PROCEDURE statement at the outer level of a program module, and may not have any parameters.)

The procedure must begin by declaring its name and nature:

```
HANDLER: PROCEDURE INTERRUPT 16;
```

This alerts the compiler to create a code prologue appropriate to a routine which will, in general, be invoked by interrupts. It also provides the number of the interrupt, used during linkage and locating to create the correct vector to this routine's absolute location during execution.

If HANDLER will do any REAL arithmetic or assignments, its first executable statements should be of the form

```
ERR$INFO = GET$REAL$ERROR ; /* must earlier declare ERR$INFO BYTE */
```

or

```
CALL SAVE$REAL$STATUS (local_save_area) ; /* also declare earlier */
```

Each procedure clears the error byte. The latter procedure also clears out the REAL stack. Thus, after either procedure is used, the REAL Error Byte no longer contains the flagged cause of the exception condition that invoked HANDLER.

(Using SAVE$REAL$STATUS is a way of avoiding possible stack errors from cumulative usage. This permits errors in HANDLER to be detected independently of the originating exception condition, and allows HANDLER to restore the state of the interrupted procedure despite HANDLER's own use of the REAL facility. SAVE$REAL$STATUS also makes available all the information as to the state of the 8087 exceptions, stack and operations, as shown below.)

Thus the beginning of a typical routine to handle REAL exception conditions could look like this:

```
HANDLER: PROCEDURE INTERRUPT 16;
         DECLARE ERR$INFO BYTE;
         DECLARE LOCAL_SAVE_AREA (100) BYTE;
         ERR$INFO = GET$REAL$ERROR;
```

or like this:

```
HANDLER: PROCEDURE INTERRUPT 16;
         DECLARE ERR$INFO BYTE;
         DECLARE LOCAL_SAVE_AREA (100) BYTE;
         CALL SAVE$REAL$STATUS (LOCAL_SAVE_AREA);
         ERR$INFO = SAVE_AREA.STATUS(0) ;
                  /*(see structure defined below)*/
```

(If you used GET$REAL$ERROR prior to the above call, e.g., the sequence

```
ERR$INFO = GET$REAL$ERROR ;
CALL SAVE$REAL$STATUS (LOCAL_SAVE_AREA) ;
```

the error byte of the status word saved in this local area would not reflect the exceptions that invoked HANDLER, because the byte would have been zero'd by the prior use of GET$REAL$ERROR. The actual exceptions would be in ERR$INFO.)

If you won't need the extra information gained by the SAVE, i.e., if you need only the exceptions, use the GET$REAL$ERROR beginning shown first.

Conversely, if you use the SAVE, GET$REAL$ERROR is unnecessary because the SAVE supplies the exceptions as part of the 8087 status (see Figure 14-4).

The rest of HANDLER can perform any actions deemed appropritate. This is an application-dependent decision. Among the possibilities:

*   incrementing an exception counter for later display

*   printing diagnostic data, e.g., the contents of local_save_area

*   aborting further execution of the calculation causing exception

*   aborting all further execution

The format of the local_save_area as it is filled by the save procedure is

INCREASING ADDRESSES

15		0	
	CONTROL WORD		+0
	STATUS WORD		+2
	TAG WORD		+4
	IP15-0		+6
IP19-16	0	OPCODE	+8
	OP15-0		+10
OP19-16		0	+12
	SIGNIFICAND 15-0		+14
	SIGNIFICAND 31-16		+16
	SIGNIFICAND 47-32		+18
	SIGNIFICAND 63-48		+20
S	EXPONENT 14-0		+22
	SIGNIFICAND 15-0		+24
	SIGNIFICAND 31-16		+26
	SIGNIFICAND 47-32		+28
	SIGNIFICAND 63-48		+30
S	EXPONENT 14-0		+32
	SIGNIFICAND 15-0		+84
	SIGNIFICAND 31-16		+86
	SIGNIFICAND 47-32		+88
	SIGNIFICAND 63-48		+90
S	EXPONENT 14-0		+92

INSTRUCTION POINTER: (IP15-0 and IP19-16 | 0 | OPCODE)

OPERAND POINTER: (OP15-0 and OP19-16 | 0)

TOP STACK ELEMENT:ST: (SIGNIFICAND 15-0 through EXPONENT 14-0, +14 to +22)

NEXT STACK ELEMENT:ST(1): (SIGNIFICAND 15-0 through EXPONENT 14-0, +24 to +32)

LAST STACK ELEMENT:ST(7): (SIGNIFICAND 15-0 through EXPONENT 14-0, +84 to +92)

NOTES:
S = Sign
Bit 0 of each field is rightmost. least significant bit of corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

Figure 14-4. Memory Layout of REAL Save Area

If you might later perform more extensive manipulations on that area, you could declare a structure permitting you to access its component parts by name and/or byte:

```
DECLARE SAVE_AREA STRUCTURE
            ( CONTROL(2) BYTE,
            STATUS(2) BYTE,
        TAG        WORD,
        INSTR_PTR  WORD,
    IP_OPCODE      WORD,
 OPERAND_PTR(2)    WORD,
    STACK_TOP(5)   WORD,
    STACK_ONE(5)   WORD,
    STACK_TWO(5)   WORD,
    STACK_3  (5)   WORD,
    STACK_4  (5)   WORD,
    STACK_5  (5)   WORD,
    STACK_6  (5)   WORD,
    STACK_7  (5)   WORD
    ) AT(@LOCAL_SAVE_AREA) ;
```

NOTE

To make use of the words from TAG through IP__OPCODE, you must employ masks and shifts to access the individual fields shown in Figure 14-4.

The final action prior to returning (if desired) to the interrupted procedure is to restore the status of the REAL math unit:

```
CALL RESTORE$REAL$STATUS (LOCAL__SAVE__AREA);
```

However, if you did not use GET$REAL$ERROR prior to the SAVE$REAL$STATUS call, the local save area will contain the original contents of the error byte. Under these circumstances, you must first clear the lower byte of the saved status word before the above RESTORE so as to avoid retriggering the same exception that invoked HANDLER to begin with.

To do so, you can use a command of the form

```
LOCAL__SAVE__AREA (2) = 0 ; (should precede restore)
```

or

```
SAVE__AREA.STATUS (0) = 0 ;
```

This appendix lists the entire BNF syntax of the PL/M-86 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 the 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>
          I  <non-quote character>

<non-quote character>::= <letter>
              I <decimal digit>
              I $
              I <special character>
              I blank

<letter>::= <upper case letter>
       I  <lower case letter>

<upper case letter>::= AIBICIDIEIFIGIHIIIJIKILIMINIOIPIQIRI SITIUIVIWIXI YIZ
<lower case letter>::= alblcldlelflglhliljlklllmlnlolplqlrlsltlulvlwlxlylz
<decimal digit>::= 0I1I2I3I4I5I6I7I8I9
<special character>::= +I-I*I/I<I>I=I:I;I.I,I(I)I@I_
<apostrophe>::= '
```

A.1.2 Tokens

```
<token>::= <delimiter>
       I  <identifier>
       I  <reserved word>
       I  <numeric constant>
       I  <string>
```

A.1.3 Delimiters

```
<delimiter>::= <simple delimiter>
            | <compound delimiter>

<simple delimiter>::= +I–I*I/I<I>I=I:I;I.I,I(I)I@
<compound delimiter>::= <>
                      | <=
                      | >=
                      | :=
```

A.1.4 Identifiers and Reserved Words

```
<identifier>::= <letter>  ⌈<letter>         ⌉
                          |<decimal digit>  |  ...
                          |     $            |
                          ⌊     —            ⌋
```

`<reserved word>` see list, Appendix C

A.1.5 Numeric Constants

```
<numeric constant>::= <binary number>
                    | <octal number>
                    | <decimal number>
                    | <hexadecimal number>
                    | <floating point 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
                                         ⌊       $            ⌋
```

```
<floating point number>::= <digit string> <fractional part> [<exponent part>]
<fractional part>::= . [<digit string>]
<exponent part>::= E [+ I –] <digit string>
<digit string>::= <decimal digit>  ⌈<decimal digit>⌉   ...
                                   ⌊     $          ⌋
```

```
<binary digit>::= 0I1
<octal digit>::= <binary digit> I2I3I4I5I6I7
<decimal digit>::= <octal digit> I8I9
<hexadecimal digit>::= <decimal digit>IAIBICIDIEIF
```

A.1.6 Strings

```
<string>::= ' <string body element>   ...  '
<string body element>::= <non-quote 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 the Main Program

<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>
     [<array specifier>] <variable type>
     [<variable attributes>]

<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>][<initialization>]
                      I [EXTERNAL][<constant attribute>]

<locator>::= AT (<expression>)

<constant attribute>::= DATA

<array specifier>::= <explicit dimension>
                      I <implicit dimension>

<explicit dimension>::= (<numeric constant>)
<implicit dimension>::= (*)

<variable type>::= <basic type>
                      I <structure type>

<basic type>::= INTEGER
               I REAL
               I POINTER
               I BYTE
               I WORD

A.3.3 Label Element

<label element>::= <identifier> LABEL $\begin{bmatrix} \text{PUBLIC} \\ \text{EXTERNAL} \end{bmatrix}$

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>]

A.3.6 Factored Label Elements

<factored label element>::= (<identifier> [,<identifier>]...) LABEL $\begin{bmatrix} \text{PUBLIC} \\ \text{EXTERNAL} \end{bmatrix}$

A.3.7 The Structure Type

<structure type>::= STRUCTURE (<member element> [,<member element>]...:)
<member element>::= <factored member>
                      I <unfactored member>

<factored member>::= (<member name> [,<member name>]...)
     [<explicit dimension>] <basic type>

<unfactored member>::= <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>::= INTEGER
            I  REAL
            I  POINTER
            I  BYTE
            I  WORD

<formal parameter list>::= ( <formal parameter> [,<formal parameter>]... )
<formal parameter>::= <identifier>
<procedure attributes>::=      ( INTERRUPT <numeric constant>      ...
                               { <linkage>
                               ( REENTRANT

<linkage>::= PUBLIC
         I  EXTERNAL
```

A.3.9 Attributes

A.3.9.1 AT

```
<locator>::= AT ( <expression> )
```

A.3.9.2 INTERRUPT

```
<interrupt>::= INTERRUPT <numeric constant>
```

A.3.9.3 Initialization

```
<initialization>::=    ( INITIAL)    ( <initial value> [,<intial value>]   ... )
                       ( DATA  )

<initial value>::= <expression>
               I  <string>
```

A.4 Units

```
<unit>::= <conditional clause>
      I  <do block>
      I  <basic statement>
      I  <label definition> <unit>
```

```
<basic statement>::= <assignment statement>
               I  <call statement>
               I  <goto statement>
               I  <null statement>
               I  <return statement>
               I  <8086 dependent statement>

<scoping statement>::= <simple do statement>
                  I  <do-case statement>
                  I  <do-while statement>
                  I  <iterative do statement>
                  I  <end statement>
                  I  <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>
                 I  <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>
                  I  <untyped return>

<typed return>::= RETURN <expression> ;
<untyped return>::= RETURN ;
```

A.4.1.6 8086 Dependent Statements

```
<8086 dependent statement>::= <disable statement>
                              |  <enable statement>
                              |  <halt statement>
                              |<cause interrupt statement>
```

```
<disable statement>::= DISABLE ;
<enable statement>::= ENABLE ;
<halt statement>::= HALT ;
<cause interrupt statement>::= CAUSE$INTERRUPT (numeric constant)
```

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>] ;
```

A.4.3 Conditional Clause

\<conditional clause\>::= \<if condition\> \<true unit\>
         I \<if condition\> \<true element\> ELSE \<false element\>

\<if condition\>::= IF \<expression\> THEN
\<true element\>::= [\<label definition\> ...] \<do block\>
        I [\<label definition\> ...] \<basic statement\>

\<false element\>::= \<unit\>
\<true unit\>::= \<unit\>

A.4.4 DO Blocks

\<do block\>::= \<simple do block\>
    I \<do-case block\>
    I \<do-while block\>
    I \<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\>
      I \<variable reference\>
      I \<location reference\>
      I \<subexpression\>

\<subexpression\>::= (\<expression\>)

A.5.1.1 Constants

\<constant\>::= \<numeric constant\>
      I \<string\>

A.5.1.2 Variable References

```
<variable reference>::= <data reference>
                     I   <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>
                           { . }

                    I      { @ }      <variable reference>
                           { . }
```

```
<constant list>::= (<constant>[,<constant>]...)
```

A.5.2 Operators

```
<operator>::= <logical operator>
           I  <relational operator>
           I  <arithmetic operator>
```

```
<logical operator>::= AND
                   I  OR
                   I  NOT
                   I  XOR
```

```
<relational operator>::= < I > I <= I >= I <> I =
<arithmetic operator>::= + I – I PLUS I MINUS I * I / I MOD
```

A.5.3 Structure of Expressions

```
<expression>::= <logical expression>
             I  <embedded assignment>
```

```
<embedded assignment>::= <variable reference> := <logical expression>
<logical expression>::= <logical factor>
                     I  <logical expression> <or operator> <logical factor>
```

```
<or operator>::= OR
              I  XOR
```

```
<logical factor>::= <logical secondary>
                 I  <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>::= < I > I <= I >= I <> I =


<arithmetic expression>::= <term>
                          I   <arithmetic expression> <adding operator> <term>


<adding operator>::= + I - I PLUS I MINUS
<term>::= <secondary>
        I   <term> <multiplying operator> <secondary>


<multiplying operator>::= * I / I MOD
<secondary>::=⌈<unary minus>⌉<primary>
             ⌊<unary plus>  ⌋


<unary minus>::= -
<unary plus>::= +
```

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
<array specifier>	A.3.2
<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
<digit string>	A.1.5

SYMBOL	NAME	USE
$	dollar sign	number and identifier spacer
=	equal sign	Two distinct uses:
		(1) assignment operator
		(2) relational test operator
:=	assign	embedded assignment operator
@	at-sign	location reference operator
.	dot	Three distinct uses:
		(1) decimal point
		(2) structure member qualification
		(3) address operator
/	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
__	underscore	significant character in identifier

These are the reserved words of PL/M-86. They may not be used as identifiers.

ADDRESS
AND
AT
BASED
BY
BYTE
CALL
CASE
DATA
DECLARE
DISABLE
DO
ELSE
ENABLE
END
EOF
EXTERNAL
GO
GOTO
HALT
IF
INITIAL
INTEGER
INTERRUPT
LABEL
LITERALLY
MINUS
MOD
NOT
OR
PLUS
POINTER
PROCEDURE
PUBLIC
REAL
REENTRANT
RETURN
STRUCTURE
THEN
TO
WHILE
WORD
XOR

These are the identifiers for the builtin procedures and predeclared variables. If one of these identifiers is declared in a DECLARE statement, the corresponding builtin procedure or predeclared variable becomes unavailable within the scope of the declaration.

ABS	MOVW
CARRY	OUTPUT
CMPB	OUTWORD
CMPW	PARITY
DEC	ROL
DOUBLE	ROR
FINDB	SAL
FINDRB	SAR
FINDRW	SCL
FINDW	SCR
FIX	SETB
FLOAT	SETW
HIGH	SHL
IABS	SHR
INPUT	SIGNED
INT	SIZE
INWORD	SKIPB
LAST	SKIPRB
LOCKSET	SKIPRW
LENGTH	SKIPW
LOW	STACKBASE
MEMORY	STACKPTR
MOVB	TIME
MOVE	UNSIGN
MOVRB	XLAT
MOVRW	ZERO

E.1 General Comparison

PL/M-86 may be regarded as an extension of the PL/M-80 language, described in Intel document 9800268. PL/M-86 differs from PL/M-80 in three principal respects:

* Floating-point arithmetic and signed integer arithmetic are provided. These are supported by two new data types, REAL and INTEGER.

* The extended addressing capability of the 8086 is supported by a new data type, POINTER, for storage of 8086 locations, and a new location reference operator, @.

* The set of builtin procedures is greatly expanded.

In addition, the PL/M-80 reserved word ADDRESS is replaced by the PL/M-86 reserved word WORD. Thus where PL/M-80 has only the two data types BYTE and ADDRESS, PL/M-86 has five: BYTE, WORD, INTEGER, REAL, and POINTER.

The PL/M-86 rules for expression evaluation are more complete than those of PL/M-80, to make proper use of the extended capabilities. There are also various other differences which stem from the ones noted here. In particular, an iterative DO block operates differently if its index variable is an INTEGER variable.

E.2 Compatibility of PL/M-80 Programs and the PL/M-86 Compiler

PL/M-80 programs which operate correctly on an 8080 can be recompiled with the PL/M-86 compiler to produce code which will run on an 8086. The PL/M-80 source code must first be edited as follows:

* All identifiers in the PL/M-80 source code must be examined and changed if they are PL/M-86 reserved words. The PL/M-86 reserved words which might occur as identifiers in a PL/M-80 source program are WORD, INTEGER, REAL, and POINTER (since these are not reserved words in PL/M-80).

* It is not necessary to change ADDRESS to WORD; ADDRESS is a PL/M-86 reserved word with the same meaning as WORD.

Note that where PL/M-86 programs would normally have POINTER variables and location references formed with the @ operator, PL/M-80 programs have ADDRESS (WORD) variables and location references formed with the "dot" operator. PL/M-80 usage is therefore slightly less restricted than normal PL/M-86 usage, since arithmetic operations are allowed on WORD values. To provide upward compatibility, the PL/M-86 compiler in general supports PL/M-80 usage. However, some restrictions are imposed, as explained in the *ISIS-II PL/M-86 Compiler Operator's Manual*, Intel document number 9800478. These restrictions affect the types of expressions allowed in the AT attribute, the INITIAL and DATA initializations, and location references. See also the discussions of the dot and @ operators in this manual.

Likewise, the base of a based variable may be a WORD variable instead of a POINTER variable. When a procedure is activated by its location a WORD variable may be used instead of a POINTER variable. In this case, the procedure is not required to have extended scope.

(In fact, all of these constructions are formally permitted by the PL/M-86 language, but their use in PL/M-86 programs is not recommended for most purposes, since they will not always produce correct results in a program where POINTER values also appear.)

ASCII CHARACTER	HEX	PL/M-86 CHARACTER?	ASCII CHARACTER	HEX	PL/M-86 CHARACTER?
NUL	00	no	@	40	yes
SOH	01	no	A	41	yes
STX	02	no	B	42	yes
ETX	03	no	C	43	yes
EOT	04	no	D	44	yes
ENQ	05	no	E	45	yes
ACK	06	no	F	46	yes
BEL	07	no	G	47	yes
BS	08	no	H	48	yes
HT	09	no	I	49	yes
LF	0A	no	J	4A	yes
VT	0B	no	K	4B	yes
FF	0C	no	L	4C	yes
CR	0D	no	M	4D	yes
SO	0E	no	N	4E	yes
SI	0F	no	O	4F	yes
DLE	10	no	P	50	yes
DCI	11	no	Q	51	yes
DC2	12	no	R	52	yes
DC3	13	no	S	53	yes
DC4	14	no	T	54	yes
NAK	15	no	U	55	yes
SYN	16	no	V	56	yes
ETB	17	no	W	57	yes
CAN	18	no	X	58	yes
EM	19	no	Y	59	yes
SUB	1A	no	Z	5A	yes
ESC	1B	no	[	5B	no
FS	1C	no	\	5C	no
GS	1D	no	]	5D	no
RS	1E	no	∧ (↑)	5E	no
US	1F	no	—	5F	yes
space	20	yes	`	60	no
!	21	no	a	61	yes
"	22	no	b	62	yes
#	23	no	c	63	yes
$	24	yes	d	64	yes
%	25	no	e	65	yes
&	26	no	f	66	yes
'	27	yes	g	67	yes
(	28	yes	h	68	yes
)	29	yes	i	69	yes
*	2A	yes	j	6A	yes
+	2B	yes	k	6B	yes
,	2C	yes	l	6C	yes
−	2D	yes	m	6D	yes
.	2E	yes	n	6E	yes
/	2F	yes	o	6F	yes
0	30	yes	p	70	yes
1	31	yes	q	71	yes
2	32	yes	r	72	yes
3	33	yes	s	73	yes
4	34	yes	t	74	yes
5	35	yes	u	75	yes
6	36	yes	v	76	yes
7	37	yes	w	77	yes
8	38	yes	x	78	yes
9	39	yes	y	79	yes
:	3A	yes	z	7A	yes
;	3B	yes	{	7B	no
<	3C	yes	|	7C	no
=	3D	yes	}	7D	no
>	3E	yes	≃	7E	no
?	3F	no	DEL	7F	no

.

intel®