# PL/M-86 USER'S GUIDE

Order Number: 121636-002

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

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.



The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

| | | | |
|---|---|---|---|
| BXP | Intel | Library Manager | Plug-A-Bubble |
| CREDIT | int$_e$l | MCS | PROMPT |
| i | Intelevision | Megachassis | RMX/80 |
| ICE | Intellec | Micromainframe | System 2000 |
| iCS | iRMX | Micromap | UPI |
| im | iSBC | Multibus | |
| Insite | iSBX | Multimodule | |



A486/1181/75K DD

| REV. | REVISION HISTORY | DATE |
|---|---|---|
| -001 | Original issue. | 11/80 |
| -002 | Added information on DWORD and SELECTOR data types, new built-in functions, and segmentation control extensions. | 11/81 |

# HOW TO USE THIS MANUAL

This manual gives instructions for programming in PL/M-86 and for using the PL/M-86 compiler to prepare programs for iAPX 86 and iAPX 88 microcomputer systems. It is primarily a reference manual for use when you are writing or compiling PL/M-86 programs; however, it also contains some introductory information to help you familiarize yourself with PL/M-86 as you start to use it.

The manual assumes you are familiar with basic programming concepts, including structured programming. This manual, however, does define the language completely, assuming no prior knowledge of PL/M-86.

Following the description of the language, this manual provides instructions for compiling your PL/M-86 programs, linking and locating the compiled code, and executing the final program. It explains how to interpret compiler output, including error messages.

Finally, appendixes provide quick reference information, plus supplementary instructions for interfacing PL/M-86 modules to modules in other languages and to your own operating system software.

## Manual Organization

This manual contains four kinds of information:

- Introductory and general reference information, including installation instructions
- Language information, for use when you are programming in PL/M-86
- Operating instructions for the compiler and descriptions of compiler controls
- Interfacing information you need if you supply some of your own systems software in place of that supplied by Intel (e.g., a non-Intel operating system or your own real arithmetic error handler), or if you are interfacing PL/M-86 modules to modules written in other languages such as 8086/8087/8088 Macro Assembly Language or Pascal-86

If you are a manager evaluating PL/M-86 to determine whether it fits your needs, you will find most of the information you need in Chapter 1, which is an overview of the product.

To get started with PL/M-86, first read this Preface (How to Use this Manual) and Chapter 1. (If you are familiar with assembly languages but not with high-level languages, see section 1.2 for a discussion of the advantages of a high-level language such as PL/M.) Then install the compiler (see instructions in specific host-system appendix) and try compiling, linking, locating, and running the sample program at the end of Chapter 1 to verify that the software operates correctly.

After that, if PL/M is a new language for you, study and run the sample programs in Chapters 8 and 16. Finally, skim through the manual from Chapter 2 to the end, and try writing and running a few programs of your own. Once you have become familiar with PL/M-86, you will find this manual useful as a complete reference. For quick reference, see the *PL/M-86 Pocket Reference* (order no. 121622).

If you wish to transport existing PL/M programs to your iAPX 86 or iAPX 88 application system, refer to Appendix E for a list of the differences between PL/M-80 and PL/M-86. This appendix indicates the areas of your programs that may require modification.

Once you have coded your programs, you are ready to compile, link, locate, and run them. Refer to Chapter 15 for the use of compiler controls and to your specific host-system appendix for compiler operating instructions. Chapter 18 helps you interpret error messages you may receive when compiling or running your programs. For a detailed explanation of the linking and locating process, refer to the *iAPX 86, 88 Family Utilities User's Guide* (order no. 121616).

If you are coding some of your application software in another language such as 8086/8087/8088 Macro Assembly Language or Pascal-86, refer to Appendix H for the information you need. If you are interfacing to your own operating system or providing your own file/device drivers, refer to Appendix I for instructions.

# Notational Conventions

## Section Numbers

All chapters and appendixes are section-numbered for easy cross-referencing: for instance, the heading number 5.3 denotes Chapter 5, section 3. When the text of one section refers to another section, the reference is made by number, e.g., "as described in 7.1." Figures and tables are also numbered to aid in cross-referencing—e.g., "in table 3-1," "see figure 14-1."

## Syntax Notation

In the syntax notation for this manual, the following conventions apply:

- Keywords, letter symbols, and punctuation symbols that you use verbatim in your programs—the *terminal symbols* of the language—are represented in monospace type, in which every character has the same width, just as they do in output media such as CRT console displays and printouts. All letters in terminal symbols are shown in upper case in the notation; however, you may use either upper case or lower case for these symbols in your programs. For example:

  ```
  E       WHILE     PROCEDURE
  (       TO        LITERALLY
  :=      DO        END
  ```

  are all terminal symbols.

- Terms standing for language elements or constructs that are defined elsewhere in this notation—in other words, *nonterminal symbols*—are represented in italicized lower-case letters in non-monospace type, in which the width of a character varies. For example:

  *digits* *variable*
  *sign* *expression*
  *binary-digit* *statement*

  are all nonterminal symbols.

- When two adjacent items must be concatenated, they appear with no space between them. A blank space between two items indicates that the two items may be separated by one or more *logical-blank*s. For example:

  .*digits*[E[*sign*]*digits*]

specifies that the first set of *digits*, the . symbol, and the second set of *digits* must be concatenated, with no blanks between them. Likewise, the E symbol, the *sign* if included, and the third set of *digits* must be concatenated. Blanks are permitted only between the second set of *digits* and the E symbol.

- Optional constructs are enclosed in square brackets. For example, in the construct represented by

  *digits*.*digits*[E[*sign*]*digits*]

  the first and second sets of *digits* and the . symbol are required, and the entire part following the second set of *digits* is optional. If this optional part is included, the *sign* may still be omitted.

- Optional constructs that can be repeated a number of times are marked by a three-dot ellipsis following the closing square bracket. For example:

  *binary-digit*[*binary-digit*]...B

  stands for a concatenated sequence of one or more *binary-digit*s followed immediately by a B symbol.

- Alternative constructs are represented as vertically adjacent items separated by extra vertical spacing and enclosed between curly braces that are taller than a single line of type. When these braces appear, choose any one of the constructs enclosed between the braces. For example:

  *digits*

  *binary-digit*[*binary-digit*]...B

  *octal-digit*[*octal-digit*]...Q

  *hex-digit*[*hex-digit*]...H

  indicates that the construct described may have any one of the four forms listed between the large braces.

- Text enclosed between the character sequence /* and the sequence */, when these symbols are in light, non-monospace type, is a prose definition of the given construct. Such definitions are used when symbolic definitions would be more cumbersome. For example:

  /* any upper-case or lower-case letter of the alphabet */

  is used to avoid listing 52 separate characters vertically between braces.

- The start of a new line in the notation does not mean you must start a new line at that point in your program; however, you may do so for readability. For example, when you use the construct:

  ```
  DO variable= expression TO expression;
    END statement list
  ```

  you need not include a carriage return after the second *expression*, but in many programs doing so makes the statement more readable.

(See Appendix A for the BNF notation for the PL/M-86 language.)

# CONTENTS

# CONTENTS (Cont'd.)



# TABLES

# ILLUSTRATIONS

# CHAPTER 1
# OVERVIEW

This chapter introduces the PL/M-86 language and explains the process of developing software for your iAPX 86 and iAPX 88 application system using PL/M-86.

## 1.1 Product Definition

PL/M is a high-level language for programming various families of microprocessors. It was designed by Intel Corporation to meet the software requirements of computers in a wide variety of systems and applications work.

The PL/M-86 compiler is a software tool that translates your PL/M-86 source programs into relocatable iAPX 86 object modules. You can then link these to other modules coded in PL/M, assembly, or other high-level languages.The compiler provides a listing output, error messages, and a number of compiler controls to aid in program development and debugging. The compiler runs on an Intel microcomputer development system.

To perform the steps following compilation, use the iAPX 86-based software development utilities—LINK86, LIB86, CREF86, LOC86, and OH86. Debug your programs using an applications debugger and the ICE-86 or ICE-88 In-Circuit Emulator. For firmware systems, you then use the Universal Prom Programmer (UPP) with its Universal Prom Mapper (UPM) software to transfer your programs to PROM.

## 1.2 The PL/M-86 Language

### Using a High-Level Language

High-level languages more closely model the human thought process than lower-level languages such as assembly language. They therefore are easier and faster to write, since one fewer translation step is required from concept to code. High-level language programs are also more likely to be correct, since there is less occasion to introduce error.

Programs in a high-level language are easier to read and understand, and thus easier to modify. As a result, you can develop high-level language programs in a much shorter period of time, and they are easier to maintain throughout the life of the product. Thus, high-level languages result in lower costs for both development and maintenance of programs.

In addition, programs in a high-level language are easily transferred from one processor to another. Programs that can be transferred between processors without modification are said to be portable.

If PL/M-86 is your first high-level language, you probably want to know how programming in a high-level language differs from assembly-language programming. When you use a high-level language:

- You do not need to know the instruction set of the processor you are using.
- You need not be concerned with the details of the target processor, such as register allocation or assigning the proper number of bytes for each data item—the compiler takes care of these things automatically.

- You use keywords and phrases that are closer to natural English.
- You can combine many operations (including arithmetic and Boolean operations) into expressions; thus you can perform a whole sequence of operations with one statement.
- You can use data types and data structures that are closer to your actual problem; for instance, in PL/M-86 you can program in terms of Boolean variables, characters, and data structures rather than bytes and words.

The introductory example at the end of this chapter (section 1.7) illustrates these points. Compare this PL/M program with an assembly-language program you might write to solve the same problem.

Coding programs in a high-level language involves thinking differently from coding in assembly language. This level is actually closer to the level of thinking you use when you are planning your overall system design.

## Why PL/M?

Many high-level programming languages are available today; some of them have been around far longer than PL/M. So once you have decided to use a high-level language, you might ask: How does PL/M differ from other high-level languages? What advantages does it have? When is it the right language to use?

Here are some of the characteristics of PL/M:

- It has a block structure and control constructs that aid—in fact, encourage and enforce—structured programming.
- It includes facilities for such data structures as structured arrays and pointer-based dynamic variables.
- It is a typed language—that is, the compiler does data type compatibility checking and range checking to help you detect logic errors in your programs at compile time.
- Its data structuring facilities and control statements are designed in a logically consistent way. Thus, PL/M is a good language for expressing algorithms for systems programming.
- Its control constructs make program correctness relatively easy to verify.
- It is a standard language used on Intel microcomputers, so PL/M programs are portable across Intel's processors.

For iAPX 86 and 88 systems, Intel offers several languages besides PL/M, such as Pascal and Fortran. Your choice among these should depend on your implementation. PL/M-86 allows you to program at a level closer to your microprocessor hardware. Thus, it is generally more suitable for systems programming.

PL/M was designed for programmers (generally systems programmers) who need access to the microprocessor's features, such as indirect addressing and direct I/O for optimum use of all system resources.

What about the differences between PL/M and older, more established languages like FORTRAN, BASIC, AND COBOL? PL/M has many more features than BASIC and is a more general-purpose language than either FORTRAN (best suited for scientific applications) or COBOL (tailored for business data processing). Additionally, PL/M differs from these other languages in its typing and block structure.

## 1.3 Categories of PL/M-86 Statements

There are two types of statements in PL/M-86: declarations and executable instructions. A simple example of a declare statement is:

```
DECLARE WIDTH BYTE;
```

This introduces the identifier WIDTH and associates it with the contents of one byte (8 bits) of memory. The programmer need not know the location of the byte, i.e., its actual address in memory. He will simply refer to the contents of this byte by using the name WIDTH.

An example of an executable statement is:

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

Here, we have two identifiers, CLEARANCE and WIDTH. Both must be declared prior to this executable statement, which produces machine code to retrieve the WIDTH value from memory, add 2 to it, and store the sum in the memory location for CLEARANCE.

For most purposes, 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.

A group of statements intended to perform a function, i.e., a subprogram or subroutine, can be given a name by declaring it to be a procedure:

```
ADDER_UPPER: PROCEDURE (BETA);
```

The statements that define the procedure then follow. This block of PL/M-86 statements is invoked from other points in the program and may involve passing parameters to it and returning a value. When a procedure has finished executing, control is returned immediately to the main program. This capability is the major feature permitting modular program construction.

## 1.4 The Structure of a PL/M-86 Program

PL/M-86 is a block-structured language: every statement in a program is part of at least one block, i.e., a well-defined group of statements that begins with a DO statement or a procedure declaration and ends with an END statement.

A module is a labeled simple DO-block; that is, a module must begin with a labeled DO statement and end with an END statement. Between those end points, i.e., within that DO-block, other statements provide the definitions of data and processes that make up the program. These statements are said to be part of the block, or contained within the block, or nested within the block. A module can contain other blocks but is never itself contained within another block.

(The reason for saying "simple" DO-block is that there are three other varieties of DO-blocks, explained in later chapters.)

Every PL/M-86 program consists of one or more modules, separately compiled, each consisting of one or more blocks. There are two kinds of blocks: DO-blocks and procedure definition blocks.

A procedure definition block is a set of statements beginning with a procedure declaration (as shown above) and ending with an END statement. Other declarations and executable statements can go between these endpoints, to be used later when the procedure is actually invoked or called into execution. The definition block is really a further declaration of everything the procedure will use and do. Since it is only executed later, the definition block is considered simply another form of declaration rather than being viewed as immediately executable.

## Block Nesting and Scope of Variables: An Introduction

Some blocks contain other blocks entirely, as in the following examples:

Example 1

```
start: DO;
       DECLARE (A,B,C,D,E,F,G,H,L) BYTE;
       A = 17;
       C = B + D;

       middle: DO;
               DECLARE (J,K) BYTE;
               E = F + G;
               H = J + K + A;
       END middle;

       last: L = H + C;

END start ;
```

Example 2

```
start: DO;
       DECLARE (A,B,C,D,E,F,G) BYTE;
       A = 17;
       C = B + D;

       middle: DO;
               DECLARE(H,J,K,L)BYTE;
               E = F + G;
               H = J + K + A;
       END middle;

       last: B = H + C; /*H undeclared at outer level */
END start;
```

(Multiple names of the same type can be declared in one statement, as above, meaning all the names within the parentheses are of the same type.)

The block called middle is completely contained inside the block labeled start: middle is said to be nested within the start block.

The start block is called an *outer* block. The phrase *outer level* is used to refer to statements that are in start but not in middle. For example, the statements beginning with A=, C=, B= are all in at the outer level in the blocks shown above.

PL/M-86 permits each block to be independent of other blocks, in that any names declared at an outer level can be redeclared, with new meanings and values, inside a nested block. If they are not redeclared, they keep their original locations and present contents.

Thus A will still be 17 inside middle unless we added a new declaration to make it have a new, *local* meaning there. Variables that *are* declared inside a nested block have *only* that local meaning while statements in that block are being executed. They lose that local meaning as soon as execution passes to statements outside that block.

Therefore, if H is declared only inside middle, as it is in example 2, its value will be unknown in the statement labeled "last"; the statement will be invalid and the compiler will say so. If H is declared also in start, then the value used in "last" will be the outer level meaning, unrelated to the one created in middle because that H is unique. They will only be the same if the sole declaration is in start and not in middle, as in example 1.

The effect of these rules is that, when writing a block and declaring objects solely for use inside that block, you need not worry about whether the same identifier has already been used in another block. Even if the same name is used elsewhere, it refers to a different object. This subject is dealt with in detail in Chapter 9.

This notion of nested blocks, inner and outer levels, is central and basic to successful PL/M-86 programming. For example, the modules of a program must conform to a rule that only one module may have executable statements at the outermost level. That module is called the main module (or sometimes, the main program). The outermost level of all other modules must contain only procedure definition blocks and other declarations, as discussed in the sections following.

Most of the rules discussed in this book, including the above, relate to creating and preserving unambiguous meanings, addresses, and values for each name you use. This uniqueness must be true in every block and in communicating values between blocks.

## 1.5 Executable Statements

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

| | |
|---|---|
| Assignment Statement | Chapter 5 |
| GOTO Statement | Chapter 7 |
| IF Statement | Chapter 7 |
| Simple DO Statement | Chapter 7 |
| Iterative DO Statement | Chapter 7 |
| DO WHILE Statement | Chapter 7 |
| DO CASE Statement | Chapter 7 |
| END Statement | Chapter 7 |
| CALL Statement | Chapter 10 |
| RETURN Statement | Chapter 10 |
| HALT Statement | Chapter 7 |
| ENABLE Statement | Chapter 10 |
| DISABLE Statement | Chapter 10 |
| Null Statement | Chapter 7 |
| CAUSE$INTERRUPT Statement | Chapter 7 |

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

## Assignment Statement

The assignment statement has already been introduced. It is fundamental to PL/M-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 5.

The simplest form of the assignment statement is:

identifier = expression ;

where the identifier represents a variable. The expression is evaluated, and the resulting value becomes the value of the variable. Variations of this form are given in Chapter 5.

## 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 four lines, with indentations, 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 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 less 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 is optional. Chapter 7 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 DO 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 (these *types* are discussed in detail in Chapters 3, 4, and 5.) VECTOR must be a previously declared array having at least 10 elements (a list, as discussed in Chapters 2 and 6). 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 the *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 variables. I is first assigned a value of 0, then used as a subscript for TABLE. It is incremented after each execution of 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 the subscript of the first element of TABLE that was not greater than LEVEL.

Finally, there is the *DO CASE block*, introduced by a *DO CASE statement,* which uses the value of the given expression 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.

## Built-in Procedures and Variables

PL/M-86 provides a large repertoire of built-in procedures and variables. These procedures provide such functions as shifts and rotations, data type conversions, and string manipulation. The built-in procedures and variables are described in Chapter 11.

## 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 4 and 5). 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 4 and 5 for details).

## Input and Output

PL/M-86 does not provide formatted I/O capabilities like those of FORTRAN, BASIC, or COBOL. PL/M-86 does provide built-in functions for direct I/O that do not require operating system run-time support. These I/O functions are INPUT, INWORD, OUTPUT, and OUTWORD.

INPUT causes the program to read the 8-bit quantity found in one of the 64K input ports of the iAPX 86. A reference to OUTPUT causes the program to place an 8-bit quantity into one of the 64K output ports of the iAPX 86.

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. (For more information, see Chapter 11.)

## 1.6 The Program Development Process

The PL/M-86 compiler and run-time libraries are part of an integrated set of tools that make up the total iAPX 86 or iAPX 88 development solution for your microcomputer system.

The steps in the software development processes are as follows:

1. Define the problem completely.
2. Outline the proposed solution in terms of hardware plus software. Once this step is done, you may begin designing your hardware.
3. Design the software for your system. This important step may consist of several sub-steps, including breaking down the task into modules, choosing the programming language, and selecting the algorithms to be used.
4. Code your programs and prepare them for translation using a text editor, such as the CRT-based text editor, CREDIT.
5. Translate your PL/M program code using the PL/M-86 compiler.
6. Using the text editor, correct any compile-time errors; then recompile.
7. Using iAPX 86-based LINK86 (and LOC86 if needed), link the resulting relocatable object module to the necessary support libraries supplied with PL/M-86, and locate your object code. The use of LINK86 and LOC86 depends on your application; for detailed instructions, see the *iAPX 86,88 Family Utilities User's Guide.*

## 1.7 Sample Program

Figure 1-1 shows a sample PL/M-86 program divided into two modules. This program contains many undefined words and constructs that will be explained in the coming chapters. It is here only to show what a small PL/M-86 program looks like separated into two modules for ease of comprehension.

The main program, to be compiled as a module named "M," does little but define some data and then call the procedure named SORTPROC. This procedure is defined in the other module, which is to be compiled with the name SORTMODULE.

```
SORTMODULE:DO;/*Beginning of module*/
   SORTPROC:PROCEDURE(PTR,COUNT,RECSIZE,KEYINDEX)PUBLIC;
            DECLARE PTR POINTER,(COUNT,RECSIZE,KEYINDEX)WORD;

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

Figure 1-1. Sample Program

```
            DECLARE RECORD BASED PTR(1) BYTE,
                    CURRENT (128) BYTE,
                    (I,J)WORD;
            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 follows.

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

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

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

Figure 1-1. Sample Program (Cont'd.)

# CHAPTER 2
# BASIC CONSTITUENTS OF A PL/M-86 PROGRAM

PL/M-86 programs are written free-form, meaning there is no significance to where a statement is placed on an input line, and blanks can 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 the ASCII character set, as follows:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789

along with the special characters

= . / ( ) + − ' * , < > : ; @ $ _

and the blank or space; plus the tab, carriage-return, and line-feed characters.

The rules in this section apply to everything in a PL/M-86 program except character string constants, discussed in section 2.4, and comments, discussed in section 2.5.

If a PL/M-86 program contains any character that is not in the set above, the compiler treats it as an error (or, in some cases, a warning only: see Chapter 18).

Upper- and lower-case letters are not distinguished from each other except in string constants. For example, xyz and XYZ are interchangeable. In this manual, all PL/M-86 code is in upper-case letters to help distinguish it from explanatory text.

Blanks are not distinguished from each other except in string constants. The compiler treats any unbroken sequence of blanks as a single blank.

Special characters and combinations of them have particular meanings in a PL/M-86 program, as described in the remainder of this manual.

Here is a concise glossary of special characters and combinations:

Table 2-1. PL/M-86 Special Characters

| Symbol | Name | Use |
|---|---|---|
| = | equal sign | Two distinct uses:<br>(1) assignment operator<br>(2) relational test operator |
| := | assign | embedded assignment operator |
| @ | at-sign | location reference operator |
| . | dot | Three distinct uses:<br>(1) decimal point<br>(2) structure member qualification<br>(3) address operator |
| / | slash | division operator |
| /* | | beginning-of-comment delimiter |

Table 2-1. PL/M-86 Special Characters (Cont'd.)

| Symbol | Name | Use |
|---|---|---|
| */ | | end-of-comment delimiter |
| ( | left paren | left delimiter of lists, subscripts, and some expressions |
| ) | right paren | right delimiter of lists, subscripts, and some expressions |
| + | plus | addition operator or unary plus operator |
| - | minus | subtraction or unary minus operator |
| ' | apostrophe | string delimiter |
| * | asterisk | multiplication operator, implicit dimension specifier |
| < | less than | relational test operator |
| > | greater than | relational test operator |
| <= | less or equal | relational test operator |
| >= | greater or equal | relational test operator |
| = | equal | relational test operator |
| <> | not equal | relational test operator |
| : | colon | label delimiter |
| ; | semicolon | statement delimiter |
| , | comma | list element delimiter |
| _ | underscore | significant character in identifier |
| $ | dollar sign | non-significant character in numbers or identifiers |

## 2.2 Identifiers and Reserved Words

*Identifiers* are used to name variables, procedures, symbolic constants, and statement labels. Identifiers 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 freely to improve the readability of an identifier or constant (although the $ may not be the first character). An identifier or constant containing a dollar sign is exactly equivalent to the same identifier with the dollar sign deleted.

Examples of valid identifiers are:

```
INPUT_COUNT
X
GAMM
LONGIDENTIFIERWITHNUMBER3
LONG$$$IDENTIFIER$$$NUMBER$$$3
INPUT$COUNT
INPUTCOUNT
```

The long identifiers are identical (to the compiler). The last two examples are interchangeable, but are different from the first.

Certain *reserved words* must not be used as identifiers because they are actually part of the PL/M-86 language. These are listed in Appendix C.

There is also a set of predeclared identifiers naming built-in procedures and variables. You are permitted to declare these names for your own purposes, but when you do so the built-in value or procedure becomes inaccessible. Appendix D lists these identifiers.

## 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 combinations of two special characters:
  <> <= >= := /* */
- Numeric constants (discussed below)
- Character string constants (discussed below)

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.

Sometimes a simple or compound delimiter does not occur between two identifiers, reserved words, or numeric constants, e.g., DECLAREABYTE. In these cases 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 section 2.5) 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 Constants

A constant is a value that does not change during your program's execution. There are three types of constants, all discussed in this chapter.

### Whole-Number Constants

Whole-number constants can be binary, octal, decimal, or hexadecimal. The compiler recognizes these by a suffix of B, O (or Q), D, or H, respectively. Numbers without a suffix are considered decimal. If a constant contains characters invalid in the designated number base, it will be flagged as an error.

For example, the maximum whole-number word constant is:

$2^{16}-1$ = 1111$1111$1111$1111 = 177777Q = 65535D = 0FFFFH

The first character of a hexadecimal number must be a numeric digit to avoid looking like an identifier. For example, the hexadecimal representation for 163 must be written 0A3H rather than A3H, which would be taken as an identifier.

Examples of valid whole-number constants:

12AH 2 33Q 1010B 55D 0BF3H 65535 777O 3EACH 0F76C05H

Examples of invalid whole-number constants:

| | |
|---|---|
| 12AF | Hexadecimal digits used without an H suffix, hence invalid in the default decimal interpretation. |
| 12AD | Here the final D could be a suffix but the A is not a decimal digit. If hexadecimal is intended, a final H is needed. |
| 11A2B | 'A' and '2' are not valid binary digits. If hexadecimal is intended, a final H is necessary. |
| 2ADGH | 'G' is not a valid hexadecimal digit. |

A whole-number constant can be a BYTE, WORD, DWORD, INTEGER, POINTER, or SELECTOR value depending on its size and context, as explained in Chapters 3 and 4. POINTER context means the three restricted cases that allow an actual numeric address; the maximum value for these is 1048575 (see section 4.4). INTEGER context means a signed value from −32768 to +32767.

Note that a minus sign in front of a constant is not part of the constant. An INTEGER value may be negative, but the range of whole-number constants is non-negative.

## Floating-Point Constants

The presence of a decimal point in a decimal constant creates a floating-point constant, i.e., a number of type REAL. Only decimal REALs are allowed.

At least one decimal digit (e.g., 0) must precede the decimal point. A fractional part is optional after the decimal point, as is the base-ten exponent, indicated by the letter E. This exponent must have at least one digit. Note that no fractional exponents are possible. The largest REAL constant value is $3.37 \times 10^{+38}$, and the smallest REAL is $1.17 \times 10^{-38}$.

You should read Chapter 13 before using REAL arithmetic or assignments.

Examples of valid real constants:

| | | | | | | |
|---|---|---|---|---|---|---|
| 5.3 | 176.0 | 1.88 | 3.14159 | 0.15 | 16. | 222.2 |
| 53.0E-1 | 1.760E2 | 0.188E1 | 314159E-5 | 1.5E-1 | 1.6E+1 | 2.222E+2 |

The exponents in the third and sixth examples are the same; plus signs don't change the meaning.

Examples of invalid REAL constants:

| | |
|---|---|
| 6 | No decimal point |
| 1.3AH | Hexadecimal not allowed in REALs, |
| 10.011B | nor binary, |
| 7.52Q | nor octal |
| 4.8E1AH/2 | Only decimal constants in exponents—no hexadecimal, no expressions, no fractions |

### 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, and those of length 3 or 4 translate to double-word values. For example:

'A' is equivalent to 41H
'AG' is equivalent to 4147H
'AGR' is equivalent to 414752H
'AGRX' is equivalent to 41475258H

(See ASCII code table in Appendix F.)

Therefore, character strings can be used only as BYTE, WORD, or DWORD values, since strings longer than 4 characters would exceed the 32-bit capacity of a DWORD value. As constants, however, longer character strings are stored as a sequence of bytes and can be used in a PL/M-86 program (see sections 3.1, 3.2, 3.3 and 4.4).

The maximum length of a string constant is 255 characters. It can be used only as an initialization for an array or as part of a location reference pointing to where that string constant is stored. See references above.

## 2.5 Comments

Explanatory *comments* may be interleaved with PL/M-86 program text, to improve readability and provide program documention. 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, i.e., it will become part of the string and the compiler won't recognize it. 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, to help distinguish them visually from program code, which is always presented in upper case.

# CHAPTER 3
# DATA DECLARATIONS

There are four types of objects that can be declared to have symbolic names: variables, constants, labels, and procedures. There must be exactly one declaration available for each name used in a block, no more, no less. This declaration may appear at the beginning of the block, or in an outer block. Multiple declarations of the same name in the same block are invalid.

Only after being declared and defined can names for the four elements above be used in executable statements. For variables, constants, and labels, such usage is in essence an operational definition. For a procedure, the set of statements between its declaration and its end statement constitutes its definition, in fact, its full declaration.

In addition to the item's name, a declaration tells its type, attributes, and/or location. These terms will be clarified in the course of this chapter.

Table 3-1 shows the general appearance of declarations, e.g., what elements are required or optional.

**Table 3-1. Declaration Elements**

| Declaration Statements For | Must Use | Can Use |
|---|---|---|
| Variable Names | type:<br>BYTE, WORD, DWORD, INTEGER, POINTER, SELECTOR, REAL, or STRUCTURE | linkage attributes:<br>PUBLIC or EXTERNAL<br><br>location attributes:<br>AT (location reference)<br><br>variable initialization attribute:<br>INITIAL (value-list) |
| Execution Constant Names | type, as above, and constant initialization attribute: DATA (value-list) | |
| Label Names | LABEL | linkage attributes as above |
| Compilation Constant Names | LITERALLY 'string' | |

## 3.1 Variable Declaration Statements

Simple DECLARE statements have appeared in Chapter 1. A DECLARE statement is a non-executable statement that introduces some object or collection of objects, associates names (and sometimes values) 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*.

A scalar variable is a single object whose value is not necessarily known at compile time and may change during the execution of the program. You therefore refer to it by declaring a name to be used in the program: an identifier.

The term "variable" has a more general meaning: a variable may be a scalar variable, or it may be a list of scalars referred to by a single identifier.

An array is such a list of scalars all named by the same identifier, differentiated from each other by the use of *subscripts,* e.g., A(0), A(1), A(123), etc.

A structure is a list of scalars and/or arrays which all use the same main identifier and can be differentiated from each other by their own *member-identifiers* (field names). For example, EMPLOYEES. NAME could refer to the NAME field within the structure EMPLOYEES.

Such variables ("arrays" and "structures") are discussed in greater detail in Chapter 6.

Examples of each of these categories appear below, after a brief introduction to the meaning of types.

## Types

A scalar always has a type: BYTE, WORD, DWORD, INTEGER, REAL, POINTER, or SELECTOR.

- A BYTE scalar is an 8-bit quantity occupying one byte of memory (two bytes when passed as a parameter on the stack, discussed in Appendix I). The value of a BYTE scalar is an unsigned whole number that ranges from 0 to 255.
- A WORD scalar is a 16-bit quantity occupying two contiguous bytes of memory, with the least significant 8 bits stored in the first byte (lower address). The value of a WORD scalar is an unsigned whole number that ranges from 0 to 65535.
- A DWORD scalar is a 32-bit quantity occupying two contiguous words of memory, with the least significant 16 bits stored in the first word (lower address). The value of a DWORD scalar is an unsigned whole number that ranges from 0 to 4,294,967,295.
- An INTEGER scalar is a 16-bit quantity occupying two contiguous bytes of memory, with the least significant 8 bits stored in the first byte (lower address). The value of an INTEGER scalar is a signed whole number that ranges from −32768 to +32767.
- A REAL scalar is a 32-bit quantity occupying two contiguous words of memory, with the least sigificant 16 bits stored in the first word (lower address). The value of a REAL scalar is a signed floating-point number whose specific features are discussed fully in Chapter 13.
- A POINTER scalar is a location address in the iAPX 86 memory that is made up of a base portion and an offset portion. POINTER scalars are usually four-byte quantities, but may be two bytes depending on the program size control specified for compilation. (This is discussed further in Chapter 17 and Appendix H.) In the four-byte representation, the offset portion occupies the first word (lower address). The two-byte representation is offset only.
- A SELECTOR scalar is a 16-bit quantity that is equivalent to the base portion of a POINTER. The value of a SELECTOR scalar is an unsigned whole number that ranges from 0 to 65535 and represents a paragraph number on the iAPX 86.

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

Arithmetic and other expressions using the different types are discussed in detail in Chapter 5.

## Examples

The following statements declare scalars:

```
DECLARE APPROX REAL;
DECLARE (OLD, NEW) BYTE;
DECLARE POINT WORD, VAL12 BYTE;
```

The first example above declares a single scalar variable of type REAL, with the identifier (name) APPROX.

The second example declares two scalars, OLD and NEW, both of type BYTE. This kind of statement is called a "factored declaration." It is equivalent to the sequence:

```
DECLARE OLD BYTE;
DECLARE NEW BYTE;
```

except that the factored declaration guarantees the bytes will be contiguous. Separated declaration statements do not.

The third declares two scalars of different types: POINT is of type WORD, and VAL12 is of type BYTE.

The following statements declare arrays:

```
DECLARE DOMAIN (128) BYTE;
DECLARE GAMMA (19) DWORD;
```

The first statement declares the array DOMAIN, with 128 scalar elements, each of type BYTE. These elements are distinguishable by subscripting the name DOMAIN, using the range 0 to 127 for the subscripts. For example, the third element of DOMAIN can be referred to as DOMAIN(2). The first element of every array has subscript 0.

The second statement declares the array GAMMA, with 19 scalar elements of type DWORD. The subscripts for this array can range from 0 to 18.

The next 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. The word named by RECORD.INFO is the second and third bytes of this structure.

Further discussion of structures appears in sections 4.4, 4.8, and Chapter 6.

## Results

The two results of a valid variable declaration are:

1. The name is given a unique address.
2. It is considered to have the attributes declared.

This means all subsequent uses of the variable in this block will refer to the same address (except for based variables, discussed in section 4.6).

It also requires all references to the variable to conform to the rules for the current attributes, i.e., those having priority in the current block. This allows the compiler to flag a large variety of errors of inconsistency, i.e., incompatibility of declarations with later usage (at this level of the block).

## 3.2 Initializations

Initialization is necessary for every constant and variable name that is used (read) before it is filled (written) during execution, since there are no default initial values. Either kind of name can, of course, be initialized by an assignment statement such as:

```
PI = 3.1415927; /*PI must first be declared REAL.*/
VAR13 = 10; /*VAR13 must be declared earlier, not real.*/
```

However, the PL/M-86 language provides a means for having the compiler set up these values during the compilation rather than using both instruction space and execution time in your program to do so.

There are two kinds of compile-time initilizations: INITIAL, used with variables, and DATA, used for constants. (DATA is explained in greater detail later in this section.) In each case the initialization attribute is placed after the type in the declaration, for example:

```
DECLARE FAMILY WORD INITIAL (2);
```

INITIAL causes initialization to occur during program loading, for variables that have storage allocated for them. Such variables can subsequently be changed during execution, like any other variable. (They will *not* be reinitialized on a program restart.)

The following rules apply to *both* INITIAL and DATA:

- INITIAL and DATA may not be used together in the same declaration.
- INITIAL may only appear in declarations at the outer level of a module. DATA, however, may appear in declarations at any level.
- No initializations are permitted with based variables (discussed in section 4.6) or with the EXTERNAL attribute (discussed in section 9.2).
- Either initialization may follow use of the AT attribute discussed in section 4.8, but if this causes multiple initializations, the result cannot be predicted.

The general form of the INITIAL attribute is:

```
INITIAL (value-list)
```

where *value-list* is a sequence of values separated by commas.

Values are taken one at a time from the value list and used to initialize the individual scalars being declared. The initialization 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 11.5 for built-in procedures you can use to initialize BYTE and WORD strings at run-time.)

Each value may be a string of up to four characters (e.g., 'A', 'NO') or a *restricted expression* as explained below. (Byte arrays can accommodate longer strings since each element can represent one character.)

A restricted expression is one of the following three possibilities:

- For REAL variables only: a single floating-point constant, with no operator of any kind, to be used *only* to initialize a REAL scalar.
- For POINTER variables only: a location reference formed with the @ operator, which must refer to a variable that has already been declared. (Location references are discussed in Chapter 4.)
- For all other types: a constant expression containing no operators except + or −. A constant expression has only whole number constants as operands, e.g., 2048−256+5, as explained in Chapter 5. It is evaluated as if being assigned to the scalar being initialized, using the rules of that chapter.

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, and four in each DWORD 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 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 four elements of the array DATUM are initialized with the four 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 you initialize an array, you want the array to have the same number of elements as the value list. This can be done conveniently by using the *implicit dimension specifier* in place of an ordinary dimension specifier (a parenthesized constant) and has the form

```
(*)
```

This may also be used to define an external array whose precise number of elements is either unknown or insignificant. Thus the declaration:

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

declares a public 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. To reference this array in another program module, you can declare it as follows:

```
DECLARE FAREWELL (*) BYTE EXTERNAL;
```

(See Chapter 9 for more information about PUBLIC and EXTERNAL attributes.) Note that the INITIAL and DATA lists must not be present when the implicit dimension specifier is used with an external array; otherwise, these lists are required. Also, the LENGTH, LAST, and SIZE built-ins (see section 11.1) may not be used on an external array that was declared with the implicit dimension specifier.

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 that is a member of a structure

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

## Names for Execution Constants: The Use of DATA

As discussed above, a *variable* is the name of a single data item intended to be used and altered by your program. If it isn't altered during execution, it's a *constant*.

For example, the formula for circumference of a circle as the product of radius and two pi could be written in PL/M-86 as:

```
C = R * 2.0 * 3.14159;
```

in which C and R would be variables. Their declarations would of course have to precede the above executable statement, and could appear as:

```
DECLARE (C,R) REAL;
```

If pi were used often enough, you might wish to simplify the writing of statements using it by declaring a symbolic name with that value:

```
DECLARE PI REAL DATA (3.1415927);
```

An array of constants would require a list of values, for example:

```
DECLARE FIBONACCI(9) BYTE DATA (0,1,1,2,3,5,8,13,21);
```

The form and use of the DATA initialization is *identical* to that of INITIAL except for these four facts:

- DATA causes storage to be allocated in the program's constant data segment. The content and meaning of the name cannot be changed during execution. The name should never appear on the left-hand side of an assignment statement. (This is not the case with INITIAL.)
- Data initializations can be used in declarations at any block level in the program. (INITIAL can only appear at the module level, that is, inside the DO-block which is the module itself, but outside any sub-blocks that the module may contain.)
- If the keyword DATA is used in a PUBLIC declaration when compiling with the ROM option (see section 15.4), DATA must also be used in the EXTERNAL declaration of program modules that reference it. However, no value list is then permitted, since the data is defined elsewhere. (INITIAL cannot be combined with EXTERNAL.)
- Use of the AT attribute, as explained in Chapter 4. This forces a name to be associated with a specific memory location, which can defeat the purpose of the DATA initialization (not so with INITIAL unless you explicitly redefine your own variables and location using multiple AT's).

## 3.3 Compilation Constants (Text Substitution): The Use of LITERALLY

If your program were large enough to have many declarations, you might choose to declare a compilation constant to save time at the keyboard:

```
DECLARE DC LITERALLY 'DECLARE';
```

Thereafter, during compilation, every time DC appears alone (not as part of a word), the full string DECLARE will be substituted by the compiler. Subsequent declarations can thus be written:

```
DC AREA REAL;
DC SIZE WORD;
```

A declaration using the reserved word LITERALLY defines a parameterless "macro" for expansion at compile-time. You declare an identifier to represent a character string, which will then be substituted for each occurrence of the identifier in subsequent text. This expansion will not take place in strings or constants. The form of the declaration is:

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

where *identifier* is any valid PL/M-86 identifier, and *string* is a sequence of arbitrary characters from the PL/M-86 set, not exceeding 255 in length. The following example illustrates another 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;
. . .
```

This example of a LITERALLY declaration defines the boolean values TRUE and FALSE in a manner consistent with the way PL/M-86 handles relational operators (see section 5.3). This kind of literal substitution for fixed values often makes a program more readable.

Another use of LITERALLY declaration is the declaration of quantities that 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$SIZE) WORD;
. . .
PRINT$BUFFER(BUFFER$SIZE - 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 the program for the occurrences of "32" that are buffersize references and not some other 32's.

## 3.4 Declarations of Names for Labels

A *label* marks the location of an instruction as opposed to a data item. Labels are permitted only on executable statements, not on declarations.

There are two ways of declaring a name to be a label. The *explicit* kind of label declaration is used mainly to allow for module-to-module references, which are discussed in detail in Chapter 9. The three possible forms for explicit label declarations look like this:

```
DECLARE PART3 LABEL;
DECLARE START1 LABEL PUBLIC; /*for intermodule reference */
DECLARE PHASE2 LABEL EXTERNAL; /*for intermodule reference */
```

The rules for the latter two are discussed in Chapter 9.

The more common kind of label declaration is termed *implicit* and is even simpler: the name is placed at the very beginning of the executable statement it is supposed to point to, for example:

```
START2: ALPHA = 127;
```

This statement *defines* the label START2 as pointing to the location of the PL/M-86 instruction shown. If this block has no explicit declaration of START2, i.e., no statement like:

```
DECLARE START2 LABEL;
```

then the compiler takes the definition above as an implicit declaration as well as a definition. It is as if the declaration had occurred at the start of the last simple DO or procedure statement. (If there *is* an explicit declaration, then the actual placement of the label remains simply a definition.)

Labels are used to indicate significant instructions or the starting point of instruction sequences. They can be useful reference points for understanding the parts of a program, or as targets for the transfer of control during execution (as discussed under GOTO and CALL in Chapters 9 and 10).

### Results

The results of a valid label declaration are:

1. The declared name can be used to point to an executable instruction.
2. The use of that name as a variable in this block is disallowed.
3. If the label is also *defined* in this block, by appearing on an executable statement, then the address of that statement is assigned as the value of the label.

## 3.5 Combining DECLARE Statements

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

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

we may write both declarations in a single DECLARE statement, as follows:

```
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.

Previously, most 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.

Another way of combining declaration elements is called a factored declaration. For example:

```
DECLARE A BYTE, B BYTE;
DECLARE C WORD, D WORD;
DECLARE E DWORD, F DWORD;
```

can be combined as:

```
DECLARE (A,B)BYTE,(C,D) WORD, (E,F) DWORD;
```

In each factored declaration, the allocated locations will be contiguous. Elements declared in a non-factored declaration statement may not be.

Subscripted variables are not allowed in factored declarations.

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

## 3.6 Declarations for Procedures

As illustrated earlier, the declaration of a procedure begins by giving its name, with a statement of the form:

*name*: `PROCEDURE`

followed optionally by parameters, type, and/or attributes. The definition of the procedure then follows, i.e., the set of statements declaring items used in the procedure (including any parameters) and the executable statements of the procedure itself. The definition ends with an END statement, optionally including the procedure name from the declaration.

The complete declaration of a procedure includes all the statements from the PROCEDURE statements through the END statement. This whole definition/declaration must appear before the procedure name is used in an executable statement, just as variable and constant names must be declared before their use.

The only exceptions arise when the full definition appears in another module where it is declared PUBLIC or when a procedure has been declared REENTRANT. In the first case, if a separate module intends to make use of that public definition, the using module is required to:

1. Declare the procedure as having the attribute EXTERNAL (so LINK86 will search for it.
2. Declare each formal parameter the procedure uses, so the compiler can verify correct usage when this module invokes the procedure.
3. End this local declaration with an END statement.

Example

```
SUMMER: PROCEDURE (A,B) EXTERNAL;
   DECLARE A WORD, B BYTE;
END SUMMER;
```

The full details for intermodule references appear in Chapter 9, and the discussion of procedure definition and usage appears in Chapter 10.

# CHAPTER 4
# DATA TYPES AND BASED VARIABLES

PL/M-86 performs calculations using three different kinds of arithmetic: unsigned, signed, or floating-point, depending on the data types involved.

## 4.1 BYTE, WORD, and DWORD 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 iAPX 86 memory. The value of a WORD variable is a 16-bit binary number ranging from 0 to 65535 and occupying two contiguous bytes of iAPX 86 memory. The value of a DWORD variable is a 32-bit binary number ranging from 0 to 4,294,967,295 and occupying two contiguous words of iAPX 86 memory. Values of DWORD, WORD, and BYTE variables are treated as *unsigned binary integers*.

NOTE

Support for DWORD is located in the PLM86.LIB (see Chapter 14). You must link to this library if you use the DWORD data type in *, /, or MOD operations.

Unsigned integer arithmetic is used in performing any arithmetic operation upon DWORD, WORD, and BYTE variables. All of the PL/M-86 operators may be used with them (see Chapter 5). Arithmetic and logical operations on such variables yield a result of type BYTE, WORD, or DWORD, depending on the operation and the operands. Relational operations always yield a "true" or "false" result of type BYTE.

With unsigned arithmetic, if a large value is subtracted from a smaller one, the result is the two's complement of the absolute difference between the two values. For example, if a BYTE value of 1 (00000001 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 is supposed to hold or produce a non-integer, it *must* be REAL. This will avoid unexpected incorrect results from arithmetic operations (see Chapters 5 and 13).

## 4.2 INTEGER Variables: Signed Arithmetic

INTEGER variables represent signed integers ranging from −32768 to 32767. Internally, an INTEGER value is represented in two's-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. (See also the OVERFLOW control in section 15.4.) Relational operations are signed arithmetic comparisons to yield a "true" or "false" result of type BYTE.

However, as with BYTE, WORD, and DWORD operands, division produces only an integer result. The result is rounded toward zero, i.e., down if it is positive, up if it is negative.

Only the arithmetic and relational operators may be used with INTEGER operands. Logical operators are not allowed. See Chapter 5.

## 4.3 REAL Variables: Floating-Point Arithmetic

The value of a REAL variable is a signed floating-point number whose size limits and other properties are discussed in Chapter 13. 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. Further discussion appears in Chapters 5 and 13; the latter also describes the binary representations of REAL values and error handling in floating-point arithmetic.

## 4.4 POINTER Variables and Location References

The value of a POINTER variable is the address of an iAPX 86 storage location and is made up of a base portion and an offset portion. Among other uses, POINTER variables are important as bases for based variables (see section 4.6).

Only the relational operators may be used with POINTER operands, yielding a "true" or "false" result of type BYTE. No arithmetic or logical operations are allowed. See Chapter 5, and the OPTIMIZE(3) discussion in section 15.4.

There are only a few ways to create or change the value of a POINTER variable, that is, the address that the variable points to:

1. The variable can be initialized when declared, using INITIAL or DATA with a whole-number constant as described in section 3.2.
2. It can be assigned a whole-number constant as described in section 5.6.
3. It can be initialized to or assigned an address created via the @ operator described below. This is the safest and most used method.
4. It can be assigned a value generated by the BUILD$PTR function described in section 11.8.
5. It can be assigned the value of a POINTER variable or function.

A POINTER variable may be placed at a specific address using the AT clause described in section 4.8. The address may be a whole-number constant or a location reference formed via the @ operator and a variable name.

## The @ Operator

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

The basic form of a location reference is:

@ *variable-ref*

where *variable-ref* is the name of some variable. The value of this location reference is the actual location at run time of the variable.

*Variable-ref* may also refer to an unqualified array or structure name. The pointer value 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);
```

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

The RECORD structure declares a byte called KEY followed by 25 bytes called INFO(0), INFO(1), etc. After these comes the POINTER variable named HEAD. Since these are all declared part of the RECORD structure, their contents must be referred to as RECORD.KEY, RECORD.INFO(0),...,RECORD.INFO(24), and RECORD.HEAD.

Their addresses can be referred to using the @ operator. @RECORD.HEAD is the location of the POINTER scalar RECORD.HEAD, while @RECORD is the location of the structure, which is the same as that of the BYTE scalar RECORD.KEY. @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, i.e., the KEY of ***which*** LIST.

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, i.e., the location of the array itself.

A special case exists when the identifier used as "variable-ref" is the name of a procedure. The procedure must be declared at the outer level of the program module. 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. (Further discussion appears in Chapter 10, and Appendixes H and I.

### Storing Strings and Constants via Location References

Another form of location reference is:

@(*constant list*)

where *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.

## 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 that represents an offset to the current data segment (for variables) or the current code segment (for procedures). This address should be used with caution, since it will not always produce correct results in a PL/M-86 program that contains more than one data segment or more than one code segment. See section 4.6 for indirect variable references and section 10.2 for indirect procedure calling. See also Appendix E.

# 4.5 SELECTOR Variables

The value of a SELECTOR variable is an iAPX 86 paragraph number and, as such, is the complete address of an iAPX 86 storage location. It is equivalent to the base portion of a POINTER and may also be used as the base of a based variable (see section 4.6).

Only the relational operators may be used with SELECTOR operands, yielding a "true" or "false" result of type BYTE. No arithmetic or logical operations are allowed (see Chapter 5).

There are only three ways to create or change the value of a SELECTOR variable:

1. The variable can be initialized when declared, using INITIAL or DATA with a whole-number constant as described in section 3.2.
2. It can be assigned a whole-number constant as described in section 5.6.
3. It can be assigned a SELECTOR variable or function, or the built-in function SELECTOR$OF (see section 11.8).

The results of the @ and dot operators may not be assigned to SELECTOR variables directly. They must first be converted to SELECTOR type with the built-in function SELECTOR$OF (see section 11.8).

## 4.6 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. Since the locations "point to" the data, you can later access the data elements themselves.

To permit this type of manipulation, PL/M-86 uses "based variables." A based variable is one that is pointed to by another variable, called its "base." This means the base contains the address of the desired (based) variable. We saw its declaration charted at the end of Chapter 3.

A based variable is not allocated storage by the compiler. At different times during the program run it may actually refer to 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, SELECTOR, 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, SELECTOR, or WORD. However, use a base of type WORD with caution, since it does not contain a full iAPX 86 address. WORD-based variables are addressed relative to the current DS register (see Chapter 17).
- 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$PTR, INCOME$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 the 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$PTR. This means that when any element of CATEGORY is referenced at run time, the value of CATEGORY$YPTR at that same time is the location of the array CATEGORY, i.e., its first element.

The other elements follow contiguously. The parentheses around the tokens CATEGORY BASED CATEGORY$PTR are optional. They help to make the statement more readable, but may be omitted.

### 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 very powerful facility for manipulating pointers.

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

```
DECLARE (NORTH$ERROR, EAST$ERROR, HEIGHT$ERROR) REAL;
DECLARE ERROR$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$ERROR. In the same way, we can cause the value of the pointer to be the location of EAST$ERROR, and a reference to ERROR will be a reference to EAST$ERROR.

This kind of technique is useful for manipulating complicated data structures and for passing locations to procedures as parameters. Examples are given in Chapter 10.

## 4.7 Contiguity of Storage

PL/M-86 only 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.

## 4.8 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 to 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 iAPX 86 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. The compiler replaces this name with the declared whole-number constant, thus satisfying the restrictions given above.

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.

The effect of an AT attribute is to cause the addess of a variable to be the location specified by the *location*. The first scalar in the declaration will refer to the *location*. Other scalars in the same declaration will, in sequence, refer to successive locations thereafter.

For example, the declaration:

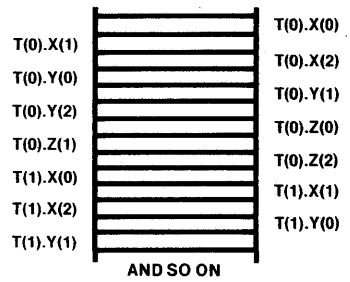
```
DECLARE (CHAR$A, CHAR$B, CHAR$C) BYTE AT (@BUFFER);
```

causes the BYTE variable CHAR$A to refer to the location of the array BUFFER. The variables CHAR$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);
```

sets up structure references to 90 bytes. They are organized such that each of the ten members of T refers to 9 bytes, the first three using the name X, the second three Y, the last three Z. The following diagram may help you visualize this structure. It shows the names for successive byte references.




The declaration above, using the AT attribute, 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 at the AT attribute:

- The AT attribute cannot be used with based variables.
- It can be used with the PUBLIC attribute, in which case it must immediately follow the word PUBLIC. However, the *location* in this case may *not* be a location reference 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.


This optimization may interfere with the operation of memory-mapped I/O. (If you pass one byte of memory as a parameter, two bytes may be accessed when the parameter is pushed on the stack.)

If a variable is located in memory by means of the AT attribute (see section 4.8), access to that variable is optimized only at the OPTIMIZE(3) level.

# CHAPTER 5
# EXPRESSIONS AND ASSIGNMENTS

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 arithmetic 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.

## 5.1 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.

### Constants

Any numeric constant may be used as an operand in an expression. Its type must be appropriate, as discussed below.

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.

A whole-number constant may be found in either *signed context* or *unsigned context* (see section 5.6). 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, as a WORD value if it is greater than 255 and equal to or less than 65,535, and as a DWORD value if it is greater than 65,535 and equal to or less than 4,294,967,295. A single whole-number constant may also be treated as a POINTER value (three special cases are discussed in sections 3.1, 4.8, and 5.7) or a SELECTOR value (see section 4.5).

A string constant containing not more than four 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.

If it is a three- or four-character string, it is treated as a DWORD constant whose value is formed by stringing together the ASCII codes for all of the characters. In a three-character string, the most significant 16 bits of the 32-bit number are formed of 8 high-order zeroes, then the code for the first character. In a four-string character, the code for the first two characters forms the most significant 16 bits of the number.

Strings of more than four characters are called string constants. They are illegal as operands in expressions, and may appear only in two contexts: as initialization values for an array (see Chapter 3) or as part of a location reference pointing to where that string constant is stored (see Chapter 4).

## Variable and Location References

As we have seen, a fully qualified variable reference refers unambiguously to a single scalar value. (Partially qualified references, discussed in Chapter 6, have very restricted uses). 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 example, in the statement:

```
I = J + ABS(L);
```

the absolute value of L will be returned by the function ABS and then added to the value of J before being stored in I. If L is −27, the result is exactly as if you had written:

```
I = J + 27;
```

For a complete discussion of procedures and function references, see Chapter 10.

Location references have already been described in Chapter 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.

## 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 5.5

## 5.2 Arithmetic Operators

There are five principal arithmetic operators in PL/M-86 (two others are described in Chapter 12). 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, DWORD, INTEGER, or REAL value (except that REAL operands are not allowed with the MOD operator). Arithmetic operations on POINTER and SELECTOR variables are not allowed.

### The +, −, *, and / Operators

The operators +, −, *, and / perform addition, subtraction, multiplication, and division on operands of any type except POINTER and SELECTOR. The following rules govern these operations (see also table 5-2).

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 operands, as discussed in Chapter 4.
2. Only three combinations of mixed operand types are allowed. A BYTE operand can be combined with a WORD operand, a BYTE operand can be combined with a DWORD operand, and a WORD operand can be combined with a DWORD operand. In the first two cases, the BYTE operand is extended by 8 high-order zero bits to produce a WORD value, or by 24 high-order zero bits to produce a DWORD value. Similarly, a WORD operand is extended by 16 high-order zero bits to produce a DWORD value. Then the operation is performed as though both operands are of the same type.
3. If one operand is a whole-number constant, and the other is a DWORD, WORD, or BYTE operand, the whole-number constant is treated as a BYTE value if it is equal to or less than 255, a WORD value if it is greater than 255 and equal to or less than 65,535, and a DWORD value if it is greater than 65,535. Then the operation is performed under rule 1 or rule 2 above. If the whole-number constant exceeds 4,294,967,295, the operation is invalid.
4. If one operand is a whole-number constant and the other 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 32,767, the operation is invalid.
5. If one operand is a whole-number constant and the other is of type REAL, POINTER, or SELECTOR, the operation is invalid.
6. If both operands are whole-number constants, the operation depends on the context in which it occurs, as explained in section 5.6.

The result of division by 0 is undefined, except for REAL values—see Chapter 13.

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, WORD, or DWORD, 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 5.6.

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).

### The MOD Operator

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

- REAL operands are not allowed—only BYTE, WORD, DWORD, 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, and –A MOD B yields –3.

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

## 5.3 Relational Operators

Relational operators are used to compare operands of the same type. They work with 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, as follows:

If both operands are of the same type, unsigned arithmetic is used to compare two BYTE values, two WORD values, or two DWORD values; signed arithmetic is used to compare two INTEGER values or two REAL values; and POINTER or SELECTOR values are compared according to the ordering of iAPX 86 locations (see OPTIMIZE(3) in section 14.4).

As with the arithmetic operators, the only legal mixed combinations of operand types are BYTE/WORD, BYTE/DWORD, and WORD/DWORD. Whole-number operands are treated as BYTE, WORD, DWORD, or INTEGER values as explained in the rules of section 5.2.

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

## 5.4 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, DWORD, or whole-number constant operands only, to perform logical operations on 8, 16, or 32 bits in parallel.

NOT is a unary operator, taking one operand only. It produces a result of the same type as its operand: each bit of the result is the ones complement of the corresponding bit of the original value.

The remaining operators each take 2 operands, and perform bitwise "and," "or," and "exclusive or," respectively. The bits of an AND result are 0 only where the corresponding bit in each operand is 1. The bits of an OR result are 1 where the corresponding bit of either operand was a 1, and 0 only where both operands have a 0. The bits of an XOR result are 0 only where the corresponding bits of the operand are the same, i.e., both 1 or both 0; the result has a 1 wherever one operand has a 1 and the corresponding bit of the other operand is 0.

If both operands are of the same type, the result is the same type as the operands.

As with the arithmetic and relational operators, the only legal mixed combinations of operand types are BYTE/WORD, BYTE/DWORD, and WORD/DWORD (see rule 2 under section 5.2). Whole-number operands are treated as BYTE, WORD, or DWORD values as explained in rule 3 of section 5.2.

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

| | |
|---|---|
| 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." |

# 5.5 Expression Evaluation

## Precedence of Operators: Analyzing an Expression

Operators in PL/M-86 have an implied order (stated below) which determines how operands and operators are grouped and analyzed during compilation.

The PL/M operators are listed in table 5-1 from highest to lowest precedence, meaning those which take effect first are listed first. Operators in the same line are of equal precedence and are evaluated as encountered in a left to right reading of an expression.

The order of evaluation in an expression is controlled first by parentheses, then by operator precedence, and finally by left to right order.

The compiler first evaluates operands and operators enclosed in paired parentheses as subexpressions, working from innermost to outermost pairs of parentheses. The value of the subexpression is then used as operand in the remainder of the expression as a whole.

(Parentheses are also used around both subscripts and the parameters of function or procedure references. These are not subexpressions, but they too must be evaluated before the remainder of the expressions or references can be evaluated at a higher level.)

When you have more than one operator in an expression, you can evaluate the results by beginning with the one having the highest precedence. If the operators are of equal precedence, evaluate them left to right.

| EXAMPLE | REASON |
|---|---|
| (A + B)*C is not the same as A+B*C | Parentheses form subexpressions |
| A + B * C means the same as A + (B*C) | Operator precedence |
| A/B*C means the same as (A/B)*C | Left to right, equal precedence |

Table 5-1. Operators' Precedence

| Operator Class | Operator | Interpretation |
|---|---|---|
| Precedence | ( ) | Controls order of evaluation: expressions within parentheses are evaluated before the action of any outside operator on the parenthesized items |
| Unary | +,- | Single positive operator, single negative operator |
| Arithmetic | *,/,MOD<br>+,- | Multiplication, division, modulo (remainder) division<br>Addition, subtraction |
| Relational | <, <=, <>, =, >=, > | Less than, less than or equal to, not equal to, equals, greater than or equal to, greater than |
| Logical | NOT<br>AND<br>OR, XOR | Logical negation<br>Logical conjunction<br>Logical inclusion disjunction,<br>Logical exclusive disjunction |

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

| | | |
|---|---|---|
| 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 above, 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 for operators of equal precedence makes no difference in the value of the expression. But in the fourth example, the left-to-right rule is critical.

A further example will show the action of the rules of precedence on a longer expression

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

We will assume 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, the parameter is the expression B*B−4.0*A*C. Floating point constants (4.0; 2.0) are used rather than whole-number constants (4;2) because it is invalid to combine whole-number constants with REAL variables.

As the full expression is analyzed below, association of operands with operators is indicated by brackets drawn over each operator and its operand(s).

The compiler analyzes first the portions of the expressions within the innermost parentheses: the procedure parameter above and the subexpression 2.0 * A, also called a compound operand since its result is used in evaluating the whole expression.

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

In a left to right scan, 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 results of the second evaluation (i.e., the compound operand 4.0*A) 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 still contains a minus (−) operator that has not been analyzed. The operands of this operator are the result of evaluating B*B and the result of evaluating 4.0*A*C. Once this is done, the parameter expression is analyzed and its value can be calculated.

This value does not become an operand in the overall expression. It is passed to the procedure SQRT, which returns the square root of the parameter. This returned value then becomes an operand in the remainder of the full original expression

(−B + "returned value") / 2.0 * A

Now that the innermost subexpressions have been analyzed and evaluated, what remains is a division whose left operand must be evaluated further. This outer subexpression is −B + the returned square root: there are two operators. The first is a unary minus (−) and its operand is the value of B. The second is the binary plus (+) operator, with two operands: the value of −B and the value of SQRT(B*B−4.0*A*C). −B has the same meaning as 0−B, which is to be added to

the now-known value of the square root indicated. The final operator is division (/), whose two operands are fully known: the value of (−B+SQRT(B*B−4.0*A*C)) and the value of (2.0*A).

Three important points must be emphasized about expression evaluation, as discussed in the next three sections.

## Compound Operands Have Types

The first point is that compound operands (as shown between brackets above) have types just as primary operands do. All of the primary operands used in the example above were of type REAL, causing the resulting compound operands to be of type REAL also. It is always valid for all the operands in an arithmetic expression to be of the same type, and the result is of that type. (The only exception is that combining BYTE values can validly create a WORD value.)

But in an expression containing mixed data types, most combinations are not allowed. Again the only exception involved BYTE, WORD, and DWORD values; these may be mixed as operands in expressions, whether constants or variables.

Mixing any other types in arithmetic, logical, or relational expressions is invalid. For example, if F and G are INTEGER variables and H and K are REAL variables, then the expressions F > K, H + G are invalid.

Due to operator precedence, some combinations can validly occur in the same instruction without being directly combined. In the following logical expression

```
F > G AND H < K
```

the subexpression F > G yields a byte value, as does the subexpression H < K. Then the byte values are ANDed together. This expression is legal despite an apparent mixing of types, as follows:

G and H could not be the operands for two reasons:

1. The relational operators are of higher precedence than the AND operator.
2. Only BYTE, WORD, or DWORD operands are legal with logical operators.

```
F > G AND H < K
```

## Relational Operators Are Restricted

The second point is: in the absence of parentheses denoting a subexpression, the result of a relational operation (comparison) is not allowed to become an operand in another relational operation.

The algebraic meaning of A ⩽ X ⩽ B is well-defined on paper, but in PL/M-86 the expression

```
A <= X <= B
```

is invalid because the second <= operator would have to use the result of the first <= operator as one of its operands.

The valid PL/M-86 way to achieve the desired meaning is

```
A <= X AND X <= B
```

Parentheses could have created a valid expression; for example:

```
(A <= X) <= B
```

but the result does not have the desired meaning: A <= X becomes a byte of value 0 if A is greater than X, 0FFH if not. Thus if A is 0, X is 1, and B is 2

```
(0 <= 1) <= 2
becomes (0FFH) <= 2
yields FALSE, contrary to the original intention
```

### 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 + 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-86 do not specify the order in which subexpressions or operands are evaluated in each statement. This flexibility allows the compiler to optimize the object code it produces, as discussed in Chapter 15. In most cases the order of evaluation makes no difference.

However, certain embedded assignments (section 5.7) or function references (section 10.2) have the side effect of changing the value of an operand in the same expression. Due to the variability of evaluation order, this side effect can lead to undesired results. You should avoid such usage. (See the sections mentioned above.)

## 5.6 Choice of Arithmetic: Summary of Rules

As discussed in Chapter 4, 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 5-2 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 5-2. Summary of Expression Rules

| Variable Type | Kind of Arithmetic | Operand Type | Arithmetic Operation | Result | Notes |
|---|---|---|---|---|---|
| DWORD, WORD, and BYTE | Unsigned | BYTE w/ BYTE | + or –<br>* or / or MOD | BYTE<br>WORD | range: 0-255<br>range: 0-65535 |
| | | BYTE w/ WORD becomes WORD w/ WORD | any | WORD | A byte operand is first extended with 8 high-order zeroes to a word value. |
| | | BYTE w/ DWORD becomes DWORD w/ DWORD | any | DWORD | A byte operand is first extended with 24 high-order zeroes to a dword value. |
| | | WORD w/ DWORD becomes DWORD w/ DWORD | any | DWORD | A word operand is first extended with 16 high-order zeroes to a dword value. |
| | | BYTE w/ whole-number constant < 256 | + or –<br>* or / or MOD | BYTE<br>WORD | Constant treated as byte.<br>Constant treated as word. |
| | | BYTE or WORD w/ whole-number constant < 65,536 | any | WORD | Constant treated as word. |
| | | BYTE or WORD w/ whole-number constant > 65,535 | any | DWORD | Constant treated as dword. |
| | | DWORD w/ whole-number constant<4,294,967,295 | any | DWORD | Constant treated as dword. |
| INTEGER | Signed | INTEGER w/ INTEGER | any | INTEGER | range: –32768 to +32767 |
| | | INTEGER w/ whole number constant becomes INTEGER w/ INTEGER w/ positive INTEGER | any | INTEGER | Constant treated as a positive INTEGER value.<br>Note: unary minus may be applied to this positive INTEGER value. |
| REAL | Floating Point | REAL w/ REAL | + or – or * or / | REAL | — |

POINTER and SELECTOR variables can appear only in relational expressions, e.g., PTR11 < PTR22, which result in a BYTE value of 0 for FALSE or 0FFH for TRUE. POINTER values are compared as full iAPX 86 addresses except under the OPTIMIZE (3) control, discussed in Chapter 15. SELECTOR values are compared as 16-bit paragraph numbers.

NOTE: The above are the only permitted combinations of operations and operands. All other combinations are invalid; for example, INTEGER with BYTE, REAL with WORD, and so forth. However, explicit conversions may be coded in-line using the built-in PL/M-86 functions described in Chapter 11.

## 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, depends on the type of operand A. If A is of type BYTE, WORD, or DWORD, 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, POINTER, or SELECTOR, 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 less than 65,536 (all DWORD arithmetic is performed at run-time). 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 that 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, WORD, or DWORD, then each whole-number constant is treated as a BYTE value if it is equal to or less than 255, as a WORD value if it is greater than 255 and equal to or less than 65,535, and as a DWORD value if it is greater than 65,535. If the constant exceeds 4,294,967,295 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, resulting in a WORD value 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, WORD, or DWORD 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, POINTER, or SELECTOR 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 5.7.
- Constant expression as subscript of an array variable: evaluated as if being assigned to a WORD variable (see section 5.7).
- Constant expression in the IF part of an IF statement: evaluated as if being assigned to a BYTE variable (see sections 7.2 and 5.7).
- Constant expression in a DO WHILE statement: evaluated as if being assigned to a BYTE variable (see sections 7.1 and 5.7).

- 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 7.1 and 5.7).
- Constant expression in a DO CASE statement: evaluated as if being assigned to a WORD variable (see sections 7.1 and 5.7).
- 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 10.2 and 5.7).
- 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 section 5.7).

## 5.7 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 ;*

where *expression* is 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) *variable*. 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.

### Implicit Type Conversions

In an assignment statement, if the type of the value of the right-hand expression is not the same as the type of the variable on the left side of the equal sign, then either the assignment is illegal or an *implicit* type conversion occurs. Except for constant expressions, only byte, word, or dword values are converted automatically. Chapter 11 presents eight built-in functions you can invoke to perform *explicit* conversions for use in expressions or assignments. The following paragraphs spell out the rules for the implicit conversions:

Expression with a BYTE value. *WORD variable on the left*: the BYTE value is extended by 8 high-order zero bits to convert it to a WORD value. *DWORD variable on the left*: the BYTE value is extended by 24 high-order zero bits to convert it to a DWORD value.

If the variable on the left is of any type except BYTE, WORD, or DWORD, 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. *DWORD variable on the left*: the WORD value is extended by 16 high-order zero bits to convert it to a DWORD value.

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

Expression with a DWORD value. *BYTE variable on the left*: the 24 high-order bits of the DWORD value are dropped to convert it to a BYTE value. *WORD variable on the left*: the 16 high-order bits of the DWORD value are dropped to convert it to a WORD value.

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

Expression with an INTEGER value. No implicit conversions are performed. If the variable on the left is of any type except INTEGER, the assignment is illegal.

Expression with a REAL value. No implicit conversions are performed. If the variable on the left is of any type except REAL, the assignment is illegal.

Expression with a POINTER value. No implicit conversions are performed. If the variable on the left is of any type except POINTER, the assignment is illegal.

Expression with a SELECTOR value. No implicit conversions are performed. If the variable on the left is of any type except SELECTOR, 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.

*WORD variable on the left*: The constant expression is evaluated in unsigned context. If the resulting value is equal to or less than 65,535, it is treated as a WORD value, and no conversion is necessary. If the resulting value is greater than 65,535, it is converted to type WORD by dropping all except its 16 low-order bits.

*DWORD variable on the left*: The constant expression is evaluated in unsigned context. 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 1,048,575. 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 3.1 and 4.8.

*SELECTOR variable on the left*: If the constant expression consists of nothing but a single whole-number constant, it is treated as a SELECTOR value. The whole-number constant must not exceed 65,535. If the constant expression consists of anything more than a single whole-number, the assignment is illegal. This is one of two cases where a whole-number constant can be treated as a SELECTOR value. The other case is described in section 3.1.

*REAL variable on the left*: The assignment is illegal unless all values on the right are REAL. However, the FLOAT function described in section 11.2 can be used to convert the constant expression to a REAL value which can be assigned to that variable.

## 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 be all of the same type, with one exception: variables of types BYTE, WORD, and DWORD may be mixed. When this is done, the conversion rules given above are applied separately to each assignment.


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.

## Embedded Assignments

A special form of assignment can be 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* (everything to the right of the := assignment symbol) is evaluated and stored into the *variable* on the left. Parentheses are strongly recommended, therefore, to specify the limits of an embedded assignment within an assignment statement. The value of the embedded assignment is the same as that of its right half. For example, the expression:

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

results in exactly the same value as:

```
ALT + (TCORR + PCORR) - (HT/SCALE)
```

The only difference is the side-effect of storing the intermediate results TCORR + PCORR and HT/SCALE into CORR and ELEV, respectively. These names for intermediate results can then be used at a later point in the program without calculating their values. The names must have been declared earlier.


The rules of PL/M-86 do not specify the order in which subexpressions or operands are evaluated. When an embedded assignment changes the value of a variable that also appears elsewhere in the same expression, the results cannot be predicted: they depend on too many factors, e.g., the optimization level you specify to the compiler (as discussed in Chapter 15).

As an example of this ambiguity, if you write:

```
A =(X:=X+4) + Y*Y + X;
```

you could mean either of the following interpretations:

```
A1 = (X+4) + Y*Y + (X+4);
A2 = (X+4) + Y*Y + X;
```

You should avoid this ambiguity by removing the embedded assignment from the expression and using a separate assignment statement to achieve the desired effect. For example, each of the above is unambiguously achieved by the following:

```
(1)  X = X + 4;
     A1 = X + Y*Y + X;

(2)  X = X + 4;
     A2 = X + Y*Y + X - 4;
```

**NOTE**

Before using floating-point (REAL) arithmetic or assignments, you should study Chapter 13, which explains many features and restrictions affecting your results.

# CHAPTER 6
# STRUCTURES AND ARRAYS

## 6.1 Arrays

As mentioned briefly in Chapter 3, it is often desirable to use a single identifier to refer to a whole group of scalars, and distinguish the individual scalars by means of a "subscript," i.e., a value enclosed in parentheses. The scalars are all the same type. Such a list is called an array.

It is declared by using a "dimension specifier." The dimension specifier is a non-zero 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.

### Subscripted Variables

To refer to a single element of a previously declared array, you use the array name followed by a subscript enclosed in parentheses. This construct is called a "subscripted variable."

For example, given the DECLARE statement:

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

you can refer to each byte as an individual item using 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. Thus the subscript of the last element is 1 less than the dimension specifier.

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 it is depends on the expression ITEMS(3) + 2. This value 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(2) 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" named 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 array variables are permitted anywhere PL/M-86 permits an expression. They may also appear on the left side of an assignment statement.

## 6.2 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 as AIRPLANE.ALTITUDE. These names are also called the "members" of this structure.

A structure may have up to 64 members.

Individual structure members may not be based and may not have any attributes, as discussed in Chapters 4 and 3, respectively.

## Arrays of Structures

We have already seen arrays of scalars. PL/M-86 also allows arrays of structures. The following DECLARE statement creates 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, each 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.

## 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).

## Arrays of Structures with Arrays Inside the Structures

Given that an array can be made up of structures, and a structure can have arrays as members, we can combine the two constructions to write:

```
DECLARE FLOOR (30) STRUCTURE (OFFICE (55) BYTE);
```

The identifier FLOOR refers to an array of 30 structures, each of which contains one array of 55 BYTE scalars. This could be thought of as a 30-by-55 matrix of BYTE scalars. To reference a particular scalar value—say element 46 of structure 25—we would write FLOOR(25).OFFICE(46). Note that the scalar elements of each OFFICE array are stored contiguously, and the OFFICE arrays themselves are elements of the FLOOR 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-1).FIRST$NAME(K-1)
```

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.

## 6.3 References 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."

### 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. The compiler flags violations of these rules as errors.

### Unqualified and Partially Qualified Variable References

Unqualified and partially qualified variable references are allowed only in location references, as discussed in Chapter 4, and in the built-in procedures LENGTH, LAST, and SIZE, as discussed in Chapter 11.

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 fails to refer uniquely to a single scalar even using a subscript and/or member-identifier with an identifier. For example, NODE(15) and NODE(12).SUBLIST are partially qualified variable references, given the above declarations.

When used with the @ operator, such references are taken to mean the first byte that could fit the description. Thus @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 because it is completely ambiguous: 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 built-in procedures LENGTH, LAST, and SIZE, as explained in Chapter 11.

# CHAPTER 7
# FLOW CONTROL STATEMENTS

This chapter describes statements that alter the sequence of execution of PL/M-86 statements, and the group statements into blocks.

## 7.1 DO and END Statements: DO Blocks

Procedures and DO blocks are the basic building units of modular programming in PL/M-86. (Procedures are discussed in Chapter 10.)

The present chapter discusses all four kinds of DO-blocks. Each begins with a DO statement and includes all subsequent statements through the closing END statement. The four kinds are

- The *simple DO block*; for example:

```
DO;  /* all statements executed, each in order */
 statement-0;
 statement-1;
 statement-2;
 .
 .
END;
```

- The *DO CASE block*; for example:

```
DO CASE select_expression;
 case-0-statement;     /* exactly one statement executed- */
 case-1-statement; /* selected by the expression's value */
 .
 .
END;
```

- The *DO WHILE block*; for example:

```
DO WHILE expression_true;
 statement-0; /* all executed if expression is true, */
 statement-1;           /* none if expression false. */
 .
 .
END;
```

- The *iterative DO block*; for example:

```
DO counter = start-expr TO limit-expr BY step-expr;
 statement-0;     /* all statements executed a number */
 statement-1;     /* of times depending on comparison */
 .                     /* of counter with limit_expr. */
 .
 .
END;
```

The last two blocks are also referred to as DO-loops because the executable statements within them may be executed repeatedly (in sequence) depending on the expressions in the DO statement.

As discussed with earlier charts, any DO statement may have multiple labels on it, and the last (only) of these may appear between the word END and the next semicolon. For example:

```
A: B: C: D: EM: DO;
                .
                .
                .
                END EM ; /* indicates end of block EM; */
                         /* A, B, C, D also end here. */
```

As mentioned in Chapters 1 and 3, the placement of declarations is restricted. Except for use in procedures, declarations are permitted only at the top of a simple DO block, before any executable statements of the block. (This DO can, of course, be nested within other DOs or procedures. Chapter 9 discusses the scope of declared names.)

Each DO block can contain any sequence of executable statements, including other DO blocks. Each block is considered by the compiler as a unit, as if it were a single executable statement. This fact is particularly useful in the DO CASE block and the IF statement, both discussed in this chapter.

The discussions that follow describe the normal flow of control within each kind of DO block. The normal exit from the block passes through the END statement to the statement immediately following. These discussions assume that none of the statements in the block causes control to bypass that process. A GOTO statement with the target outside the block would be one such bypass. (GOTOs are discussed later in this chapter.)

## Simple DO Blocks

A simple DO block merely groups, as a unit, a set of statements that will be executed sequentially (except for the effect of GOTOs or CALLs):

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

For example:

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

This simple DO block adds the value of TEMP to the value of OLD$VALUE and stores it in NEW$VALUE. It then increments the value of COUNT by one.

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

```
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 permitted up to 18 levels.

A simple DO block can delimit the scope of variables, as discussed in Chapter 9.

## 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 maximum number of cases is 255. The form of the DO CASE block is

```
DO CASE select_expression;
         statement-0;
         statement-1;
         . . .
         statement-n;
END;
```

In the DO CASE statement, *expression* must yield a BYTE, WORD, or INTEGER value. If it is a constant expression, it is evaluated as if it were being assigned to a WORD variable. The value of *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 (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.


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 out-of-range condition 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;
    ;                                /* case 0 */
    CONVERSIONS=CONVERSIONS+1;       /* case 1 */
    SAFETIES = SAFETIES + 1;         /* case 2 */
    FIELDGOALS = FIELDGOALS + 1;     /* case 3 */
    ;                                /* case 4 */
    ;                                /* case 5 */
    TOUCHDOWNS=TOUCHDOWNS+1;         /* case 6 */
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 statement is executed, causing the corresponding variable to be incremented.

A more complex DO CASE block is the following:

```
SELECT = COUNT-5;
IF SELECT <=2 AND SELECT >=0 THEN
   DO CASE SELECT;

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

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

        DO I = LAST$HI+1 TO TOP-6 ;  /* Begin Case 2 */
                Z(I) = X*Y+1      ;
                W(I) = Z(I)*Z(I)  ;
                V(I) = W(I)-Z(I)  ;
        END;                        /* End Case 2 */

   END;                             /* End DO CASE block */
ELSE CALL ERROR;
```

If we assume SELECT and COUNT are INTEGER variables, negative values could occur. The DO CASE block is placed within an IF statement to guarantee that if the value of SELECT 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 section 7.2.

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 or Case 2 and cause multiple statements to be executed. This is only possible because they are grouped as a simple DO block, which acts as a single statement.

## DO WHILE Blocks

DO WHILE and IF statements examine only the least significant bit of the value of the expression. If the expression is relational, e.g., A<B, the result will have a value of 00H or 0FFH, but this is incidental; 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."

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

```
DO WHILE expression; /*expression must yield */
                        /* BYTE or WORD value*/
          statement-0;
          statement-1;
          . . .
          statement-n;
END;
```

The effect of this statement is as follows:

1. First the BYTE or WORD *expression* following the reserved word WHILE is evaluated. If the rightmost bit of result is 1, then the sequence of statements up to the END is executed.
2. When the END is reached, *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 *expression* has a value whose rightmost bit is 0. Execution then skips the statements in the block and passes to the statements following the END 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.

## Iterative DO Blocks

An iterative DO block begins with an iteration statement and executes each statement in order in the block, repeating the entire sequence as described in this section. The form of the iterative DO block is:

```
DO counter = start-expr TO limit-expr BY step-expr ;
 statement-0 ;
 statement-1 ;
 .
 .
 .
END ;
```

The BY *step-expr* phrase is optional; if omitted, a step of 1 is used.

The *counter* must be a non-subscripted variable of type BYTE, WORD, or INTEGER. The *start-expr*, *limit-expr*, and *step-expr* may be any valid PL/M-86 expressions also of these types. However, if *counter* or any of these expressions has type INTEGER, then all must be INTEGER, as explained in Chapter 5.

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;
```

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

The next example uses *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;
```

The type of counter affects two important factors in the execution flow of iterative DOs:

a. When *step-expr* is evaluated

b. What causes execution to exit the DO block.

The following steps constitute the general execution sequence of an iterative DO block, with BYTE, WORD, or INTEGER variables and expressions in the DO itself. Type is mentioned only for steps where it matters, i.e., where the actions or consequences are different for different types. Where the INTEGER case is different, it is described in parentheses. The discussion following this description summarizes all the rules and their results.

1. The *start-expr* is evaluated and assigned to *counter*.
2. The *limit-expr* is evaluated and compared with *counter*. (If these are INTEGER type, then *step-expr* is also newly evaluated at this time.)
   a. If *counter* is greater than *limit-expr*, execution exits the DO and passes to the statement following the next END (unless *step-expr* is a negative INTEGER: if so, the exit occurs only if *counter* is less than *limit-expr*.)
   b. Otherwise, the statements within the DO block are executed in order until the END statement is reached.
   c. At the END, a *step-expr* of type BYTE or WORD is newly evaluated.
3. The *counter* is incremented by the value of *step-expr*. For BYTE or WORD *counter*s, if the new value is less than the old value (due to modulo arithmetic as explained below), the loop is exited immediately.

   Otherwise control returns to step 2 above.

(An 8-bit byte can represent numbers no larger than 11111111B (255 decimal). The largest number a 16-bit word can represent is 1111111111111111B, which is 65535 decimal. If you add 1 to these values, you get 0. Thus the new *counter* can be less than the old.)

These rules and their consequences can be summarized in two broad cases:

A. If you start with a non-negative *step-expr*, then the loop is exited as soon as any one of the following becomes true:
   1. The new *counter* is greater than the new *limit-expr*.
   2. An INTEGER *step-expr* becomes negative AND the new *counter* is still less than the new *limit-expr*.

3. A BYTE or WORD *step-expr* causes a lower *counter* than the one just used.

B. If you start with a negative INTEGER *step-expr*, then the loop is exited as soon as either of the following two conditions occurs:

1. The new *counter* is less than the new *limit-expr*.
2. The new *step-expr* becomes non-negative AND the new *counter* is greater than the new *limit-expr*.

Upon exit from the iterative DO block:

In all cases *step-expr* has been reevaluated.

In all but one case *limit-expr* has been reevaluated: when a non-INTEGER *counter* has just "gone over" and become smaller, *limit-expr* is unchanged from its value during the last loop.

In all cases *counter* has been changed, but the step value that was added to it varies: if INTEGER, *counter* has been incremented by the former step value, before it was reevaluated. For BYTE or WORD *counter*s, the newer step has been used.

The following distinctions can be important:

- In every case, *start-expr* is evaluated only once and *limit-expr* is evaluated before any execution.
- An INTEGER *step-expr* is evaluated in step 2; other *step-expr*s are evaluated in step 3.
- With a *counter* of BYTE or WORD, there is no such thing as a negative step. For example, if *step-expr* is −5, 251 is used. Furthermore, stepping down to a *limit-expr* that is less than *start-expr* is not possible because the loop will be exited immediately.

## 7.2 The IF Statement

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

```
IF expression THEN statement-a;
    ELSE statement-b; /*optional*/
```

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

The IF statement has the following effect: first *expression* is evaluated as if it were being assigned to a variable of type BYTE. If the result is "true" (rightmost bit 1) then *statement-a* is executed. If the result is "false" (rightmost bit 0), 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, but 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 if the value of *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.

## Nested IF Statements

Any 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 statement to be executed in a THEN or an ELSE clause may in fact be another IF statement.

An IF statement inside a THEN clause is called a "nested" IF. Nesting 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.

This rule could also be restated as:

> When 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;
```

This construction 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 differs completely 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, as follows:

```
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;
```

## 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, nor 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 sequence is inefficient because all four IF statements (six tests in all) will be carried out in every case, which is wasteful 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 sequence 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.

## 7.3 GOTO Statements

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

```
GOTO label;
```

For example:

```
GOTO ABORT;
```

The appearance of *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 discussed in Chapter 9, GOTO statements are restricted. 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.
- From any point in one program module to a labeled statement in the outer level of the main program module. To jump to such a label, you must declare the label to have "extended scope," i.e., declare it PUBLIC in the main module and EXTERNAL in the module containing the GOTO.

The use of GOTOs is necessary in some situations. However, in most situations where control transfers are desired, the use of an iterative DO, DO WHILE, DO CASE, IF, or a procedure activation (see Chapter 10) is preferable. Indiscriminate use of GOTOs will result in a program that is difficult to understand, correct, and maintain.

## 7.4 The HALT Statement

The HALT statement has the form:

```
HALT;
```

It generates an STI instruction followed by a HLT, causing the iAPX 86 to come to a halt with interrupts enabled (see section 10.2).

## 7.5 The CAUSE$INTERRUPT Statement

The CAUSE$INTERRUPT statement causes a software interrupt to be generated. It takes the form:

```
CAUSE$INTERRUPT (constant);
```

where *constant* is a whole-number constant in the range 0 to 255. It generates an INT instruction with the constant as the interrupt type, causing the iAPX 86 to transfer control to the appropriate interrupt vector. (See section 10.5 and Appendix I.)

## 7.6 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 10.

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.

# CHAPTER 8
# SAMPLE PROGRAM 1

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.

## 8.1 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.
- The algorithm 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) WORD;

   /*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 1.

# CHAPTER 9
# BLOCK STRUCTURE AND SCOPE

This chapter is intended to clarify the meaning of *outer level* and the concept of *scope*, including the use of the linkage attributes, PUBLIC and EXTERNAL.

The outer level of a block means statements (or labels) contained in the block but not contained in any nested blocks. The term "exclusive extent" also has this meaning. The inner level, or "inclusive extent," includes this outer level and all nested blocks as well.

A block "at the same level" as another block means both are contained by exactly the same outer blocks.

The scope of an object means those parts of a program where its name, type, and attributes are recognized, i.e., handled according to a given declaration. An object means a variable, label, procedure, or symbolic (named) constant (i.e., a compilation constant or execution constant as discussed in Chapter 3). A program is the complete set of modules that are ultimately linked together and located as a unit.

These definitions are explained further by the text and examples that follow:

## 9.1 Names Recognized Within Blocks

Throughout this manual we have seen that PL/M-86 is a block-structured language, enabling you to implement your design for solving a problem, processing data, or controlling hardware.

You create blocks of code containing declarations followed by executable statements. You order and nest the blocks in such a way as to simplify and clarify the flow of data and control. (The maximum nest is 18 blocks deep.) A collection of these blocks that performs a single function, or a small set of related functions, is usually compiled as one module, as discussed in Chapter 1.

Beyond the advantages of modularity, simplicity, and clarity, the nesting of blocks serves another very basic purpose: names declared at an outer level are known to all statements of all nested blocks as well.

You can always declare a new meaning for any such name within a nested simple-DO or procedure block, thereby cutting off its earlier meaning for this block. But if you don't choose this option, its meaning is established by a single declaration at an outer level. (The only objects that don't require declarations prior to use are labels and reentrant procedures.)

In figure 9-1, everything inside the solid line constitutes the inclusive extent of block MMM (in this case, *module* MMM). KK is known throughout this block, including within all nested blocks.

Everything inside the dashed line constitutes the inclusive extent of block SORT. JJ and II are known throughout this block, but not outside it, that is, not before the label SORT or after the END SORT statement.

Everything inside the dotted line constitutes the inclusive extent of block FIND. Since this is not a simple-DO or procedure block, declarations are not allowed. All prior declarations shown are available for use within FIND.

See also figure 9-2.

```
MMM:  DO; /*Beginning of module*/
      DECLARE RECORD (128) STRUCTURE
              (KEY BYTE,
              INFO WORD);
      DECLARE CURRENT STRUCTURE
              (KEY BYTE,
              INFO WORD);
      DECLARE KK BYTE:
      KK = 127;
      /*Instructions here would read in data.*/

      SORT:   DO;
              DECLARE (JJ,II) INTEGER;
              DO JJ = 1 TO 127;
                      CURRENT.KEY = RECORD(JJ).KEY;
                      CURRENT.INFO = RECORD(JJ).INFO;
                      II = JJ;
              FIND:   DO WHILE II > 0 AND
                         RECORD(II-1).KEY > CURRENT.KEY;
                      RECORD(II).KEY = RECORD(II-1).KEY;
                      RECORD(II).INFO = RECORD(II-1).INFO;
                      II = II-1;
                      END FIND;
              RECORD(II).KEY = CURRENT.KEY;
              RECORD(II).INFO = CURRENT.INFO;
              END;
              END SORT;

      /*Instructions here would write out
                  data from the records.*/
END MMM;  /*End of module*/
```

Figure 9-1. Inclusive Extent of Blocks

```
MMM:    DO; /*Beginning of module*/
        DECLARE RECORD (128) STRUCTURE
                (KEY BYTE,
                INFO WORD);
        DECLARE CURRENT STRUCTURE
                (KEY BYTE,
                INFO WORD);
        DECLARE KK BYTE:
        KK = 127;
        /*Instructions here would read in data.*/

        SORT:   DO;
                DECLARE (JJ,II) INTEGER;
                DO JJ = 1 TO 127;
                        CURRENT.KEY = RECORD(JJ).KEY;
                        CURRENT.INFO = RECORD(JJ).INFO;
                        II = JJ;
                FIND:   DO WHILE II > 0 AND
                           RECORD(II-1).KEY > CURRENT.KEY;
                        RECORD(II).KEY = RECORD(II-1).KEY;
                        RECORD(II).INFO = RECORD(II-1).INFO;
                        II=II-1;
                        END FIND;
                RECORD(II).KEY = CURRENT.KEY;
                RECORD(II).INFO = CURRENT.INFO;
                END;
                END SORT;

        /*Instructions here would write out
                        data from the records.*/
    END MMM;    /*End of module*/
```

Figure 9-2. Outer Level of Block SORT

The shaded area is the exclusive extent (the outer level) of block SORT. The unshaded area *within* SORT is the exclusive (and inclusive) extent of block FIND. To the instructions within the FIND block, SORT's exclusive extent is an outer level. The *outermost* level (or module level) is the area *outside* the solid lines enclosing the SORT block.

## Restrictions on Multiple Declarations

In any given block, a known name cannot be redeclared at the same level as its original declaration. A new declaration is permitted inside a nested simple-DO or procedure block, where it automatically identifies a new object despite the existence of the same name at a higher level. The new object will be the only one known by this name within its block, and it will be unknown outside its block, where the prior name maintains its meaning. These observations also apply when a name is redeclared in another block at the same level as the block containing the original declaration.

When a name is declared *only* in a separate block at the same level, there is no way to access it except in that block where it is declared. The definition is not at an outer level to the block in which you are now programming. Any local declaration you supply will establish a new separate object, whose values bear no relation to those of the other.

The reason for these rules, as for many in programming, is that there must be no ambiguity about what address/location is meant by each name in the program. The declaration rules above give you freedom to choose whatever names seem appropriate within a given block, without interfering with exterior uses of them. But when you redeclare a name, its outer-level meaning is inaccessible until execution exits the block containing the new declaration. For example:

```
A:  DO;
    DECLARE X, Y, Z BYTE;

L1: X=2;
    Y=X;
    Z=X;

B:    DO;
      DECLARE X, Y BYTE;
      X=3;
      Y=X;
L2:   Z=X;

END B;

L3: /*At this point, X=2, Y=2, Z=3, because the value
      of the redeclared X was used to fill Z*/

    /*If statement L2 were outside the loop labeled B,
      then Z would be 2 because the outer X value would
      be used.*/
```

## 9.2 Extended Scope: The PUBLIC and EXTERNAL Attributes

These attributes permit you to extend the scope of names for all objects except modules; a module name may not be declared with either attribute.

By "extend the scope," we mean make the names available for use in modules other than the one where they are defined. (The names are already available to nested blocks in *this* module.) To be specific, this includes names for variables, labels, procedures, and execution constants.

For example, the statement:

```
DECLARE FLAG BYTE PUBLIC;
```

causes a byte to be allocated, named FLAG, and its address made known to any other module where the following declaration occurs:

```
DECLARE FLAG BYTE EXTERNAL;
```

Similarly, if one module has a procedure declaration block that begins:

```
SUMMER: PROCEDURE (A,B) WORD PUBLIC;
        DECLARE (A,B) BYTE;
        . /*other declarations can go here*/
        . /*executable statements go here,
            defining the procedure*/
        .
END SUMMER;
```

then any other module may invoke SUMMER if it first declares:

```
SUMMER: PROCEDURE (A,B) WORD EXTERNAL;
                  /*A,B can be any names*/
        DECLARE (A,B) BYTE;
                 /*but these names must match them,*/
    /*and each type must match its public definition*/
END SUMMER;
```

Since no ambiguity of location or definition is permissible, the use of PUBLIC and EXTERNAL must follow a strict set of rules, as follows:

1. These attributes may only be used in a declaration at the outermost level of a module, i.e., never in a nested block.
2. Only one may appear on any declaration, and only once. Thus:

   ```
   DECLARE ZETA BYTE PUBLIC EXTERNAL; /*error*/
   DECLARE RHO WORD PUBLIC PUBLIC; /*error*/
   ```

   and similar constructs are all invalid.
3. Names may be declared PUBLIC at most once. The PUBLIC declaration is the defining declaration: the address it creates is used in each procedure or module where the same name is declared EXTERNAL. Clearly you must not create more than one PUBLIC address for any name.
4. Names may only be declared EXTERNAL if they are also declared PUBLIC in a different module of the program. The EXTERNAL attribute is essentially a request to use a PUBLIC address. An EXTERNAL without a PUBLIC is a dead letter. Lack of a definition elsewhere will result in a link-time error.
5. Where the name is declared EXTERNAL, it must be given the same type as where it is declared PUBLIC. Any contradiction of type would violate the intention to use the location(s) and content(s) defined elsewhere.
6. Similarly, names declared EXTERNAL must not be given a location, i.e.,with the AT phrase, or an initialization, i.e., using DATA or INITIAL value. Such usage would again contradict the fact of being defined in another module.

   However, in that other module, where this name is declared PUBLIC, the use of AT, DATA, or INITIAL is allowed with it.
7. Neither PUBLIC nor EXTERNAL may be applied to a name that is based. For example:

   ```
   DECLARE PTR1 POINTER;
   DECLARE V1 BASED PTR1 PUBLIC;
   ```

   is invalid. The reason: by definition, V1 has no home of its own; its location is always determined by PTR1. Thus to declare V1 PUBLIC or EXTERNAL does not permit the correct assignment of addresses. PTR1, on the other hand, always contains the current address of V1. Declaring the base, PTR1 in this case, to be PUBLIC and EXTERNAL is always permissible, since it permits valid results.

(Four additional restrictions on the use of PUBLIC and EXTERNAL procedures appear in Chapter 10.)

Following these rules will permit consistent and reliable execution of programs using names with extended scope. A PUBLIC definition occurring in one module will then

be used by all related references to that name in separate modules, that is, references which declare the name EXTERNAL. The diagram below illustrates this:

```
MOD1:   DO
          DECLARE V1 BYTE PUBLIC;
          .
          .
          .
        END MOD1;
```

```
MOD2:   DO;
          DECLARE V1 BYTE EXTERNAL;
          QQ4: PROCEDURE PUBLIC;
          .
          .
          .
          END QQ4;
          END MOD2;
```

Both references to V1 will use the same definition (location) for V1, namely that in module MOD1. Similarly, if any module needed to call procedure QQ4, it would first need a declaration like

```
QQ4: PROCEDURE EXTERNAL ;
END QQ4 ;
```

so that a subsequent CALL QQ4 would correctly pass control to that procedure in module MOD2.

## 9.3 Scope of Labels and Restriction on GOTOs

Labels are subject to exactly the same rules of scope discussed above.

One consequence is that a label is unknown outside the block where it is declared. As discussed in Chapter 1, a label is either declared explicitly at the beginning of a simple-DO or procedure block, or the compiler *considers* it to be declared there as soon as it is defined by use anywhere in the block. Therefore the discussion of what names are known in which blocks applies directly to labels as well as other names.

The label on a block is not part of the block it names. For example, the name on the DO enclosing the module itself is not part of that DO; it merely names it. For nested blocks, a label is again not part of the block it names, but belongs instead to the outer level, as part of that first enclosing block.

If a name used as a label on a block is defined inside that block, it will name a new thing, be it label, variable, or constant. There will be no confusion with the outer label name. This fact leads to important restictions on use of the GOTO statement:

1. It is impossible for a GOTO to transfer control from an outer block to a labeled statement inside a nested block.

2. Moreover, a GOTO can transfer control from one block to another in the same module *only* if the target block encloses the one containing the GOTO (and only if the name of that target label is not declared in the nested block.)

Furthermore, a label with the PUBLIC attribute is permitted only in the main module. (This has the interesting consequence of forcing all other transfers of control, that is, not involving a return to the main module, to use procedure calls. This favors the development of orderly, modularized, traceable programs.)

In fact, only three GOTO transfers are possible, as follows:

1. From one point in the outer level of a block to another statement also in the outer level of that block
2. From an inner, nested DO-block (not a nested procedure) to a statement in the outer level of *any* enclosing block
3. To a main-program label that is declared PUBLIC, from *any* point in any module that declares that label EXTERNAL

(Recall that *only* labels at the outer level of a main program may be declared PUBLIC.)

Given the program structure and declarations shown in figure 9-3, the only valid GOTO transfers are shown in figure 9-4. A single-headed arrow means the transfer is valid only in the direction shown. A double-headed arrow means both directions are valid, i.e., a GOTO may be used from either label to the other label.

```
MAIN:   DO;
        DECLARE (LAB33, LAB77) LABEL PUBLIC;
        DECLARE IT BYTE;
        .
        .
        .
    LAB33: . . . ;
                DO;
                .
                .
                .
                END;
        .
        .
        .
    LAB77: . . .
                DO WHILE IT > 0;
                .
                .
                .
                END;
        .
        .
        .
        END MAIN;
```

**Figure 9-3. Sample Program Modules Illustrating Valid GOTO Usage**

```
MOD1: DO;
        P1: PROCEDURE;
                DECLARE (LAB33,LAB77) LABEL EXTERNAL;
                L1: . . . ;
                        DO
                        DECLARE KO BYTE;
                        P2: PROCEDURE
                                                .
                                                .
                                                .
                                                L2: . . .;

                                                .
                                                .
                                                .
                                                END P2;
                L3: . . . ;
                        .
                        .
                        .
                        END P1;
      END MOD1;
```

```
MOD2: DO;
        .
        .
        .
        P4: PROCEDURE;
                DECLARE (LAB33,LAB77) LABEL EXTERNAL;
                .
                .
                .
                L4: . . . ;
                .
                .
                .
                L5: . . . ;
                        DO;
                                                L6: . . . ;
                                                .
                                                .
                                                .
                                                END ;
                .
                .
                .
                L7: . . . ;
                .
                .
                .
                END P4;
            L8: . . . ;
        END MOD2;
```

Figure 9-3. Sample Program Modules Illustrating Valid GOTO Usage (Cont'd.)

Figure 9-4 illustrates legal GOTO transfers, in fact, the only transfers permitted among the given labels in figure 9-3.

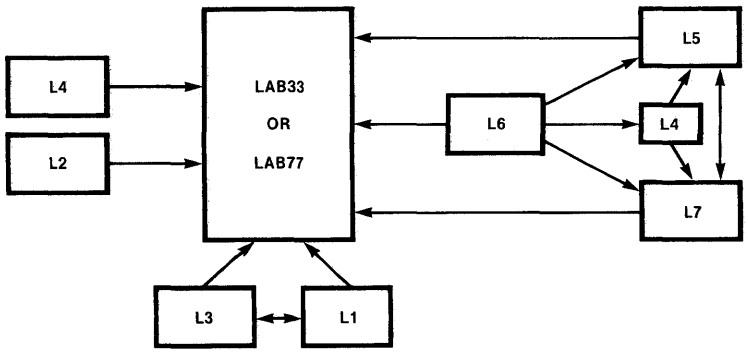



**Figure 9-4. Valid GOTO Transfers** 121636-1

# CHAPTER 10
# PROCEDURES

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 (even if it is physically located elsewhere): 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 the statement immediately after the point of activation.

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

## 10.1 Procedure Declarations

You must declare procedures, just as you must declare variables. Thereafter, any reference to a procedure must occur within the scope defined by the procedure declaration. Also, a procedure may not be used (called, or invoked in an expression) until *after* the END statement of the procedure declaration (unless reentrant—see section 10.5).

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

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

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.

## 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. However, procedure 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.

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.

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 a dimension specifier above zero 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 were declared within the procedure RANGE$CHECK. Since ITEM is based on PTR and the value of PTR is @QUANTS, every reference to an element of ITEM becomes 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 will be evaluated when the procedure is activated. If one actual parameter changes another actual parameter, the results are *undefined*. This can occur if an expression used as an actual parameter contains an embedded assignment or function reference which changes another actual parameter for the same procedure. See also the caution below.

## 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 10.2.

A *typed procedure*, also called a *function*, has a type in its PROCEDURE statement: BYTE, WORD, DWORD, INTEGER, REAL, POINTER, or SELECTOR. Such a procedure returns a value of this type, to be used in an expression or stored as the value of a variable. The procedure is activated by using its name as an operand in an expression,as a special kind of variable reference called a "function reference."

When the expression is processed at run time, 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 containing the function reference is then evaluated, and program execution continues in normal sequence.

Like an untyped procedure, a typed procedure may have parameters. They are handled as described above.

The body of a typed procedure must always contain a RETURN statement with an expression, as explained later in this chapter.


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 used in an expression 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 using parentheses to segregate any typed procedure that has a side effect, or using it in an assignment statement first to create an unambiguous sequence.

## 10.2 Activating a Procedure—Function References and CALL Statements

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;
```

or:

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

An example is the following:

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

(An alternate form of the CALL statement is discussed later.)

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

*name*

or

*name(parameter list)*

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

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

where SUM$ARRAY is a previously declared typed procedure. The value added to SUBTOTAL will be the value returned by SUM$ARRAY using the actual parameters (@ITEMS, COUNT). See the cautionary note in section 10.1.

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 activation, each actual parameter is evaluated and the result assigned to the corresponding formal parameter in the procedure declaration. Then the procedure body is executed. Any PL/M-86 expression may be an actual parameter if its type is the same as that of the corresponding formal parameter.

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

As in expression evaluation and assignment statements (see Chapter 5), a few type conversions are performed automatically when necessary in activating and returning from a procedure. The built-in explicit type conversion procedures of Chapter 11 can also be used to force the value of an expression to a desired type.

### 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. This is done by means of a CALL statement with the form:

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

The identifier may not be subscripted, though it may be a structure reference. It must be a fully qualified POINTER or WORD type variable reference, and its value is assumed to be the location of the entrypoint of the procedure being activated.

NOTE

Calls through 32-bit POINTERs will be translated into "long" calls while those through 16-bit WORDs or POINTERs (in the SMALL case) will be translated into "short" calls (relative to the current code segment). The compiler will issue a warning if the wrong addressing type is used to gain a procedure address for later indirect calls.

A normal CALL uses the *name* of the procedure, and 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.

## 10.3 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 (see Chapter 9).

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 *expression* becomes the value returned by the procedure. It is evaluated as if it were being assigned to a variable of the same type as used on the PROCEDURE statement.

## 10.4 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 = OFFH THEN COUNTER = COUNTER + 1;
         RETURN;
END AOUT;
```

Here COUNTER is some variable declared outside the procedure, i.e., it 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 (with a non-zero dimension) 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);
```

## 10.5 The Attributes: PUBLIC and EXTERNAL, INTERRUPT, REENTRANT

The PUBLIC and EXTERNAL attributes can be included in PROCEDURE statements to give procedures extended scope. Extended scope is discussed in Chapter 9.

A procedure declaration with the PUBLIC attribute is called a "defining declaration." A procedure declaration with the EXTERNAL attribute is called a "usage declaration." Most of the rules for PUBLIC and EXTERNAL appeared in Chapter 9. The following additional rules apply to the use of the EXTERNAL attribute in a procedure declaration:

1. The EXTERNAL attribute may not be used in the same PROCEDURE statement as a PUBLIC or REENTRANT attribute (see below). Note, however, that the defining declaration of a procedure may have the REENTRANT attribute.
2. A usage (EXTERNAL) declaration of a procedure should have the same number of parameters as the defining (PUBLIC) 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. (See Chapter 15 for a description of the TYPE control to detect such an error at module linkage time.)
3. 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.
4. No labels may appear in a usage declaration.

For example, we can alter the procedure AVG (from Example 1 above) by giving it the PUBLIC attribute:

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

thereby activating the procedure AVG as declared in the first module.

## Interrupts and the INTERRUPT Attribute

The INTERRUPT attribute allows you to define a procedure to handle some condition signaled by an iAPX 86 interrupt, e.g., from a peripheral device. A procedure with this attribute is activated when the corresponding interrupt signal is received in the iAPX 86 based system. The PL/M-86 statement CAUSE$INTERRUPT (*constant*) can also be used to initiate an interrupt signal.

The INTERRUPT attribute can be used only in declaring an untyped procedure with no parameters, at the outermost level of a program module. It may be declared PUBLIC, EXTERNAL, and/or REENTRANT. The form is:

```
INTERRUPT n
```

where *n* is any decimal number from 0 to 255. Each number can only be used once in a program. Each such procedure is then referred to as an interrupt procedure. Such a procedure is necessary if you wish to provide non-default handling of exception conditions arising in floating-point arithmetic (see Chapter 13).

How they work is clarified in the following discussion of the iAPX 86 interrupt mechanism and the PL/M-86 instructions ENABLE and DISABLE.

The iAPX 86 interrupt mechanism has two states: *enabled* or *disabled*. The ENABLE statement permits interrupts to take effect and the DISABLE prevents them from having any effect. The HALT statement also enables interrupts. (The iAPX 86 CPU always starts with interrupts disabled, but the compiler enables them in the code it inserts at the beginning of the main program, the "prologue.")

When some peripheral device sends an interrupt to the iAPX 86 CPU, it is ignored if the interrupt mechanism is disabled. If interrupts are enabled, the interrupt is processed as follows:

1. The CPU completes any instruction currently in execution.
2. The CPU sends an "acknowledge interrupt" signal and then the interrupting device sends its interrupt number.
3. The interrupt mechanism is disabled. This prevents any other device from interfering.

4. Control passes to the interrupt procedure whose number matches the number sent by the peripheral device. If no such procedure has been established, the results are undefined, since the vector which transfers control remains uninitialized.
5. When the procedure is through (by executing a RETURN or reaching the END of the procedure), the interrupt mechanism is enabled so other devices may be serviced, and control returns to the point where the interrupt occurred.

   It is 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 posi-
                tions of ALERT, which contains a bit
                pattern representing current alerts.*/
END HITEMP ;
```

## 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 7 of Chapter 11 discusses the built-in function INTERRUPT$PTR, which returns the interrupt entry point, given an interrupt procedure name, and also the built-in procedure SET$INTERRUPT, which sets an interrupt vector given the interrupt procedure name and number.

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

```
CAUSE$INTERRUPT(constant);
```

where *constant* is in the range 0 to 255.

## Reentrancy and the REENTRANT Attribute

The REENTRANT attribute allows a procedure to suspend execution temporarily, restart with new parameters, and then later complete the original execution successfully as if there had been no interruption.

This ability is desirable in two circumstances: (1) if your procedure (PROC1) activates itself (called *direct recursion*), or (2) activates another procedure PROC2 that will reactivate PROC1 before PROC1 has finished its original processing (called *indirect recursion*).

Without the REENTRANT attribute, storage for procedure variables is allocated statically, in fixed locations within the Data Segment of the object module. Re-entering such a procedure would write over the earlier contents of such locations, making it impossible to complete the original, suspended execution.

When you use the attribute REENTRANT in declaring a procedure, its variables are stored in the Stack Segment of the object module, separately from all earlier variables. Thus preserved, each set can be used independently by each invocation of the procedure.

Multiple sets of variables may therefore need storage on the stack during recursive use of such procedures. You must be sure to specify (at relocation and link time) a stack size large enough for all such storage needed by all multiple invocations that may be active at one time.

A procedure with the REENTRANT attribute may be activated before it is declared.

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 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 (discussed in Chapter 9).
- A procedure declaration with the REENTRANT attribute may not have another procedure declaration nested inside it.

# CHAPTER 11
# BUILT-IN PROCEDURES, FUNCTIONS, AND VARIABLES

Built-in procedures, functions, and variables act as if they were declared in an all-encompassing global block invisible to the programmer.

These identifiers are subject to the rules of scope. This means the name of a built-in procedure or variable can be declared to have a local meaning within the program. Within the scope of such a declaration, the built-in is unavailable. This distinguishes these identifiers from reserved words, listed in Appendix C, which cannot be used as identifiers in declarations.

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

## 11.1 Obtaining Information About Variables

PL/M-86 has three built-in procedures that take variable names as actual parameters and return information based on the declarations of the variables: LENGTH, LAST, and SIZE.

### The LENGTH Function

LENGTH is a WORD function 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, discussed in section 11.6, or an EXTERNAL array using the implicit dimension specifier (see section 3.2).

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 must be an unqualified variable reference. If the array is a structure member, then the reference is a partially qualified variable reference (see section 6.3). 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. For example:

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

is a valid function reference and returns the value 3.

### The LAST Function

LAST is a WORD function 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 11.6), or an EXTERNAL array using the implicit dimension specifier (see section 3.2).

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 function, 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.

### The SIZE Function

SIZE is a WORD function 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 except the MEMORY array (see section 11.6). It may not be an EXTERNAL declaration that uses the implicit dimension specifier (see section 3.2).

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 that is an array, or to an array element that is a structure. The value is the number of bytes required for the array or structure.

As in the LENGTH function, a "shorthand" form of partially qualified variable reference is allowed in the case where the array or scalar is a member of a structure and the structure is an array element.

## 11.2 Explicit Type and Value Conversions

The ten functions in this section provide explicit conversion from one type to another and from signed values to or from absolute magnitudes.

Explicit type-conversion functions are invoked

*function-name (expression)*

where the function has a type and the expression a value.

In table 11-1, each function name is followed by its type, the expression type expected, the purpose of the function, and the nature of the value it returns to the expression that invoked it.

Table 11-1. Explicit Type and Value Conversion

| Procedure Name | Type | Parameter Expression Type | Function | Result Returned |
|---|---|---|---|---|
| LOW | BYTE | BYTE<br>WORD | Converts WORD value to BYTE value | BYTE value unchanged<br>Low-order byte of WORD |
| | WORD | DWORD | Converts DWORD value to WORD value | Low-order word of DWORD |
| HIGH | BYTE | BYTE<br>WORD | Converts WORD value to BYTE value | 0 (zero)<br>High-order byte of WORD |
| DOUBLE | WORD | DWORD | Converts DWORD value to WORD value | High-order word of DWORD |
| | WORD | BYTE | Converts BYTE value to WORD value | WORD value, by appending 8 high-order zero bits |
| | DWORD | WORD | Converts WORD value to DWORD value | DWORD value, by appending 16 high-order zero bits |
| | | DWORD | | DWORD value unchanged |
| FLOAT | REAL | INTEGER | Converts INTEGER value to REAL value | Same value of type REAL |
| FIX | INTEGER | REAL | Converts REAL value to INTEGER value (rounds toward zero) | INTEGER value modulo 32768, i.e., within range ±32767 |
| INT | INTEGER | BYTE<br>WORD | Converts BYTE or WORD to INTEGER; interprets parameter as positive | Corresponding INTEGER value within range 0 to 32767 |
| SIGNED | INTEGER | BYTE<br>WORD | Converts a WORD value to an INTEGER value | BYTE value is extended with 8 high-order zeros; WORD value unchanged. |
| UNSIGN | WORD | INTEGER | Converts an INTEGER value to a WORD value | The bit pattern is unchanged but can now be used in WORD expressions. |
| ABS | REAL | REAL | Converts negative value to positive | Absolute value of expression supplied. If positive, returned unchanged; if negative, −(expression) is returned. |
| IABS | INTEGER | INTEGER | Converts negative value to positive | Absolute value of expression supplied. If positive, returned unchanged; if negative, −(expression) is returned. |

## The LOW, HIGH, and DOUBLE Functions

LOW and HIGH are BYTE functions that convert WORD values to BYTE values, or WORD functions that convert DWORD values to WORD values. They are activated by function references with the forms:

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

where *expression* has a DWORD, WORD, or BYTE value.

If *expression* has a DWORD value, LOW returns the value of the low-order (least significant) word of the expression value, whereas HIGH returns the value of the high-order (most significant) word of the expression 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 function that converts a BYTE value to a WORD value or a DWORD function that converts a word value to a DWORD value. It is activated by a function reference with the form:

```
DOUBLE (expression)
```

where *expression* has a BYTE, WORD, or DWORD value.

If *expression* has a BYTE value, the function 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 function appends 16 high-order zero bits to convert it to a DWORD value and returns this value. If *expression* has a DWORD value, the function returns it unchanged.

## The FLOAT Function

FLOAT is a REAL function 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.

## The FIX Function

FIX is an INTEGER function 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, and FIX(6.5) in 6.

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

NOTE

FIX is affected by your choice of rounding mode—see Chapter 13. The above examples assume the default mode, which is "round to nearest or even."

## The INT Function

INT is an INTEGER function 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*.

## The SIGNED Function

SIGNED is an INTEGER function that converts a BYTE or 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 two's-complement number and returns the corresponding integer value.

This means that if the highest-order (most significant) bit of the WORD valued 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.

### The UNSIGN Function

UNSIGN is a WORD function 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 two's complement of the absolute magnitude of the INTEGER value. For example:

```
UNSIGN(-4)
```

returns a WORD value of

```
1111$1111$1111$1100B
```

### The ABS and IABS Functions

ABS is a REAL function 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*).

IABS is an INTEGER function 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*).

## 11.3 Shift and Rotate Functions

In shift and rotate operations, a value is handled as a pattern of 8 bits (for a BYTE value), 16 bits (for a WORD or INTEGER value), or 32 bits (for a DWORD 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 below). In a rotate, bits moved off one end move onto the other end.

## Rotation Functions, ROL and ROR

ROL and ROR are functions whose type depends on the type of the expression given as an actual parameter. They are activated by function references, with the forms:

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

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

The value of *pattern* is handled as an 8-bit, 16-bit, or 32-bit binary quantity that is rotated to the left (by ROL) or to the right (by ROR). The type of *pattern* determines whether a byte, word, or dword rotate is performed. 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.
```

## Logical-Shift Functions, SHL and SHR

SHL and SHR are functions whose type depends on the type of the 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, WORD, or DWORD values. If count has a WORD or DWORD value, all but the 8 low-order bits will be dropped to produce a BYTE value. If the value of *count* is 0, no rotation occurs.

The value of *pattern* may be either a BYTE, WORD, or DWORD value and will not be converted. If it is a BYTE value, the function will return a BYTE value. If pattern is a WORD value, the function will return a WORD value; if it is a DWORD value, the function will return a DWORD value.

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

A shift operation can force a 1-bit out of the pattern. For example:

```
SHL(1000$0001B,1)
```

becomes 0000$0010, losing the former high-order bit, and:

```
SHR(1000$0001B,1)
```

becomes 0100$0000, losing the former low-order bit.

If the specified pattern and count do not cause such a "lost information" shift, then 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.

### Algebraic-Shift Functions, SAL and SAR

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

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

where *pattern* is an expression with an INTEGER value, and *count* is an expression with a BYTE, WORD, or DWORD value. If *count* has a WORD or DWORD value, all but the 8 low-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 in some instances, that just as with the logical shifts, an algebraic shift of one bit position can have 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$1000B. SAL(VAL,1) will return 1111$1111$1111$0000B, which represents -16, while SAR(VAL,1) will return 1111$1111$1111$1100B, which represents -4.

## 11.4 Input and Output

Byte or word input is performed as a function invocation in an expression on the right-hand side of an assignment statement. Byte or word output is achieved by filling the appropriate element of the output array corresponding to the desired output port of the iAPX 86 CPU.

### The INPUT and INWORD Functions

INPUT is a BYTE function and INWORD is a WORD function. 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 iAPX 86 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.

### The OUTPUT and OUTWORD Arrays

The built-in variables OUTPUT and OUTWORD are arrays, each with 65536 elements. Each element corresponds to one of the output ports of the iAPX 86 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 a 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.

## 11.5 String Manipulation Procedures

The term "string" is used here in a broader sense than previously. The "character strings" mentioned in sections 2.4, 3.1, and 4.4 are BYTE strings.

Considered 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 always means 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,as discussed below.

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

NOTE

If the source or destination string address is in SELECTOR or WORD form, the built-in function BUILD$PTR can be used to construct the pointer-parameter for the string built-in.

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.

### The MOVB and MOVW Procedures

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

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.

MOVW is the same as MOVB except that it copies a WORD string instead of a BYTE string.

## The MOVRB and MOVRW Procedures

MOVRB is the same as MOVB, 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.

MOVRW is the same as MOVRB except that it copies a WORD string instead of a BYTE string.

NOTE

If two strings overlap, a procedure such as the following can be used to make the correct choice between MOVB and MOVRB elements in the overlap area will not be overwritten until after they have been copied.

```
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.

## The CMPB and CMPW Functions

CMPB is a WORD function 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, and *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.

CMPW is the same as CMPB except that it compares two WORD strings instead of two BYTE strings.

## The FINDB/FINDW and FINDRB/FINDRW Functions

FINDB is a WORD function 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.

FINDW is the same as FINDB 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.

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

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

## The SKIPB/SKIPW and SKIPRB/SKIPRW Functions

SKIPB is a "converse" of FINDB (see 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.

SKIPW is a "converse" of FINDW (see 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.

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

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

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

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

### 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, and *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.

### The SETB and SETW Procedures

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 of a WORD string instead of a BYTE string.

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.

## 11.6 Miscellaneous Built-Ins

### 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 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 than the one used by the other built-in string functions.
- If the source and destination strings overlap, the results are always *undefined*.

## 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 iAPX 86 CPU cycle times, and assumes that the system is running at 5 MHz without interruptions.

## The MEMORY Array

MEMORY is a BYTE array of *unspecified* length which represents an uninitialized (free) segment of iAPX 86 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. 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. However, some systems provide service routines you can use to learn the size of free memory.

## STACKPTR and STACKBASE

STACKPTR and STACKBASE are built-in WORD variables that provide access to the iAPX 86 hardware stack pointer and stack base registers, i.e., SP and SS.

Care must be exercised in setting these registers (that is, using STACKPTR or STACKBASE on the left side of an assignment). To do so takes control of the stack away from the compiler and can invalidate the compile-time checks on stack overflow and the compiler's assumptions about the run-time status of the stack.

## The LOCKSET Function

The LOCKSET function 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 iAPX 86 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 assessing 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 simplest 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.

## 11.7 Interrupt-Related Procedures

The two capabilities described in this section permit programs to set interrupt vectors and learn the entry point of an interrupt-handling procedure.

### 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 10.3 and Appendix J.

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.

### The INTERRUPT$PTR Function

This built-in function returns the interrupt entry point. Its form is:

```
INTERRUPT$PTR (name)
```

It is typically used in an assignment statement, for example:

```
INT$ARRAY(4) = INTERRUPT$PTR (HANDLER_PROC_4)
```

The interrupt entry point is not accessible without using this function, since the @ operator refers to the procedure entry point instead. These differences are discussed in greater detail in Appendix J.

## 11.8 Pointer and Selector-Related Functions

The following built-in functions permit programs to manipulate POINTER and SELECTOR values that serve as location addresses in iAPX 86 memory. (For more information on POINTER and SELECTOR, see sections 4.4 and 4.5.)

### The BUILD$PTR Function

BUILD$PTR is a POINTER function that takes a SELECTOR value (the base portion) and a WORD value (the offset portion), and returns a POINTER. It is activated by a function reference with the form:

`BUILD$PTR` (*base, offset*)

where *base* has a SELECTOR value and *offset* has a WORD value.

### The SELECTOR$OF Function

SELECTOR$OF is a SELECTOR function that returns the SELECTOR portion of a POINTER. It is activated by a function reference with the form:

`SELECTOR$OF` (*pointer*)

where *pointer* has a POINTER value.

### The OFFSET$OF Function

OFFSET$OF is a WORD function that returns the offset portion of a POINTER. It is activated by a function reference with the form:

`OFFSET$OF` (*pointer*)

where *pointer* has a POINTER value.

# CHAPTER 12
# PL/M-86 FEATURES INVOLVING iAPX 86 HARDWARE FLAGS

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.

## 12.1 Optimization and the iAPX 86 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 iAPX 86 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.

## 12.2 The PLUS and MINUS Operators

In addition to the arithmetic operators described in section 5.2, 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 iAPX 86 CPU hardware CARRY flag in performing the operation.

## 12.3 Carry-Rotation Built-in Functions

SCL and SCR are built-in rotation functions 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 *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 function will return a BYTE value. If it is a WORD value, then the function 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 functions described in Chapter 11. 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.

## 12.4 The DEC Function

DEC is a built-in BYTE function 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 *expression* will be converted, if necessary, to a BYTE value. The procedure performs a decimal adjust operation on the actual parameter value and returns the result of this operation.

## 12.5 CARRY, SIGN, ZERO, and PARITY Built-in Functions

There are four built-in BYTE functions that return the logical values of the iAPX 86 hardware flags. These functions 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.

# CHAPTER 13
# FLOATING-POINT ARITHMETIC: THE REAL MATH FACILITY

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 implemented by the Intel 8087 chip or Intel 8087 emulators. The discussions therefore make no distinction as to that environment, except where noted. All math conforms to the proposed IEEE Standard for Floating Point Arithmetic (Intel's REALMATH standard).

NOTE

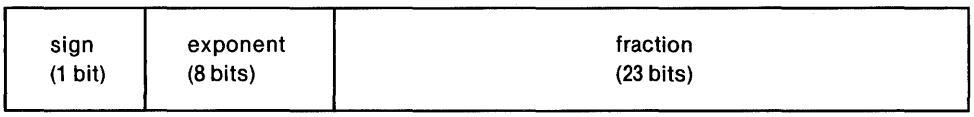
If your program performs ANY floating-point arithmetic or assignments, it must first initialize the REAL math facility by calling the procedures INIT$REAL$MATH$UNIT (section 13.5) or SAVE$REAL$STATUS, (section 13.8), and optionally, SET$REAL$MODE (section 13.8). However, it must be noted that PL/M-86 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 function value in a REAL variable, and then use that variable in the longer expression.

## 13.1 Representation of REAL Values

This section describes the 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:




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.

## 13.2 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 iAPX 86 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.

## 13.3 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 (see figure 13-1), consisting of 7 exception flags initialized to all 0's. (The reserved bit is set to 1 by the 8087.)

  The first six bits in this byte correspond to the possible errors that can arise during REAL operations (see section 13.4). When an error occurs, the facility sets the corresponding bit to 1. There is a built-in procedure described in section 13.7 that a program can invoke to read and clear the REAL Error Byte.

  The exception/error categories are discussed in section 13.4

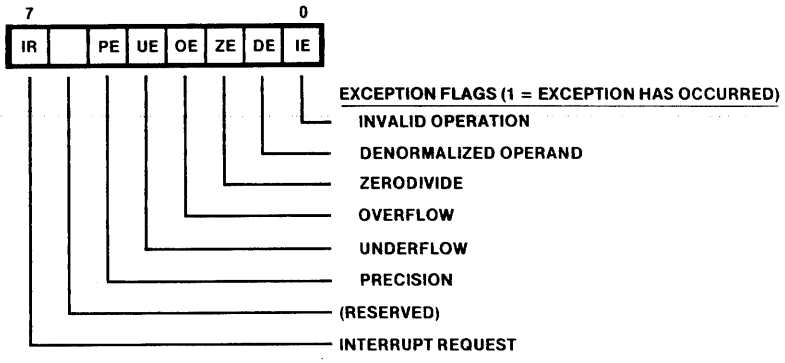



**Figure 13-1. The REAL Error Byte** 121636-2

- The REAL mode word (see figure 13-2), consisting of 16 bits initialized to 03FFH

  1. Bits 0-7 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 13.9 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 13.6 on how to set the mode word.)

     If the interrupt is enabled (IEM = 0), one of the masked bits is 0, and the corresponding error occurs during floating point processing, the REAL math facility interrupts the host CPU. The 8087 Emulator executes an interrupt 16; the 8087 interrupt is dependent on the internal configuration. The exception condition is thus reported and control passed, to the user-written error handling routine. This situation is called an unmasked error. Sections 10.2, 11.6, and Appendix J 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, but are handled as described in section 13.4, Exception Conditions.

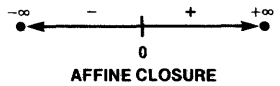
     Bits 13, 14, 15 are reserved and not for PL/M-86 use.

     Bits 8-12 provide options for controlling precision, rounding, and infinity representation. (See figure 13-2.)

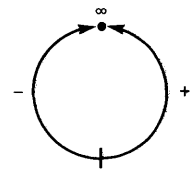
  2. All intermediate results are held in an internal format of 64-bit precision. The most-significant 24 bits of the final result are returned (plus sign and 7-bit exponent) as the PL/M-86 answer, rounded if needed according to the user-specified control. The default precision setting preserves extended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

  3. 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.

  4.



AFFINE CLOSURE



PROJECTIVE CLOSURE

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*.

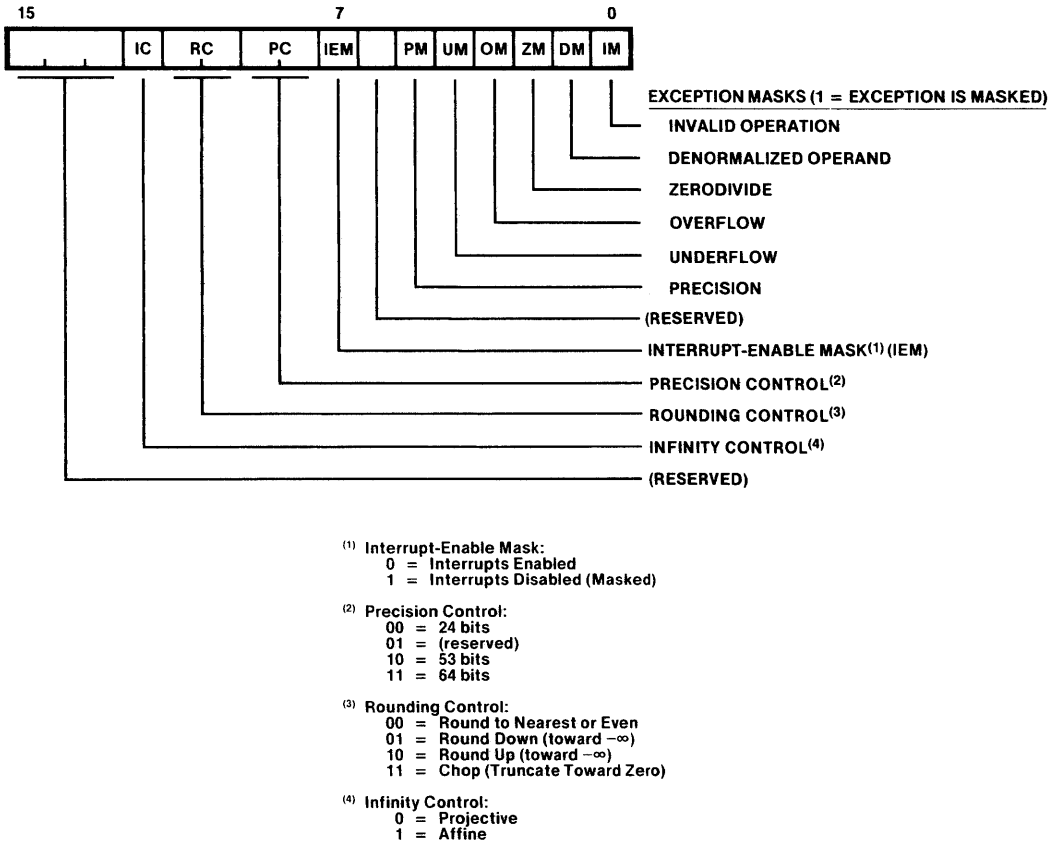



Figure 13-2. The REAL Mode Word 121636-3

## 13.4 Exception Conditions In REAL Arithmetic

As indicated in figure 13-1, 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.

Programmers who use the recommended setting of the REAL Mode Word (see section 13.6) need to handle only the invalid-exception. Study of the information from the end of the next section up to section 13.5 is advised in that it provides a general overview of the meaning of the other exception conditions. Section 13.9 describes writing the exception handler.

## 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
```

This is because 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 NAN representable in the specified precision. It also sets Bit 0 of the REAL Error Byte.

If Bits 0 and 6 of the REAL Mode Word are 0 (invalid-exception unmasked), an interrupt occurs, transferring control to the user-written interrupt handler.

## Denormal Operand Exception

This condition arises when numeric operations have resulted in a number whose exponent is literally zero and whose significand is non-zero, or a number whose significand does not begin with a 1. Denormals usually arise in response to masked underflow. Gradual underflow is the masked, default response to underflow. A small denormal added to a large normal REAL number can give an acceptable, in-range answer if the denormal exception is masked. 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 Bits 1 and 6 are 0, an interrupt occurs, transferring control to the user-written interrupt handler.

## 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 Bits 1 and 6 are 0, an interrupt occurs, giving control to the user-written interrupt handler. In either case, Bit 2 of the REAL Error Byte is set to 1.

## 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 $3.37 \times 10^{38}$, 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.

## 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. It adjusts 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 $1 \times 2^{-129}$, since a zero exponent in this format means −127. If the denormal exception is masked, this would be adjusted into a significand of .001 with an exponent of 1, i.e., $0.001 \times 2^{-126}$ prior to the operation. Then this number would be available for use in subsequent REAL operations, which might well yield valid results. Zero is returned 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.

### 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 13-1. 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. (8087 stack unchanged.) |
| Zerodivide | Return ∞ signed with "exclusive or" of operand signs. | Request interrupt. (8087 stack unchanged.) |
| Denormalized | Memory operand: proceed as usual.<br>Register operand: convert to valid unnormal, then re-evaluate for exceptions. | Request interrupt. (8087 stack unchanged.) |
| Overflow | Return properly signed ∞. | Register destination: adjust exponent,* store result, request interrupt.<br>Memory destination: request interrupt. |
| Underflow | Denormalize result. | Register destination: adjust exponent,* store result, request interrupt.<br>Memory destination: request interrupt. |
| Precision | Return rounded result. | Return rounded result, request interrupt. |

## 13.5 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 Section 13.10.

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 03FFH or 0000001111111111 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 11.6). This means no interrupts will occur from the REAL Math Facility regardless of what errors are detected. See also section 13.6 below.

Procedures that are activated after this call has taken effect do not need to do such initialization. See also section 13.8.

## 13.6 The SET$REAL$MODE Procedure

This procedure should only be invoked if you wish to change the default mode word, as for example to unmask the invalid exception.

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. 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 13.9 for facts and references on writing an interrupt handling procedure.

## 13.7 The GET$REAL$ERROR Function

GET$REAL$ERROR is a built-in BYTE function 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 function also clears the error byte in the REAL math facility.

## 13.8 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. SAVE$REAL$STATUS initializes the 8087.

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 SET$REAL$MODE, and before any arithmetic or assignment of REALs (other than GET$REAL$ERROR, if needed).

**NOTE**

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

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 11.6. 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

REAL functions without REAL parameters should not call GET$REAL$ERRORS or SAVE$REAL$STATUS before executing at least one floating point instruction. To do so may result in loss of processor synchronization.

### 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 94 bytes where the REAL status information will be saved.

The REAL status is saved at the specified location, and the REAL stack 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. iAPX 86 interrupts are disabled during the save.


The iAPX 86 processor must be able to acknowledge 8087 interrupts or loss of synchronization will occur.

## 13.9 Writing a Procedure to Handle REAL Interrupts

This section partially summarizes advice, notes, and warnings from Chapters 10, 11, and 13 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. For example, if you were using the 8087 emulator, it would begin:

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

This alerts the compiler to create a code prologue appropriate to a routine that 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 (94) BYTE;

         ERR$INFO = GET$REAL$ERROR;
```

or like this:

```
HANDLER: PROCEDURE INTERRUPT 16;

         DECLARE ERR$INFO BYTE;
         DECLARE LOCAL_SAVE_AREA (94) 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 zeroed 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 13-4).

The rest of HANDLER can perform any actions deemed appropriate. 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 shown in figure 13-3.

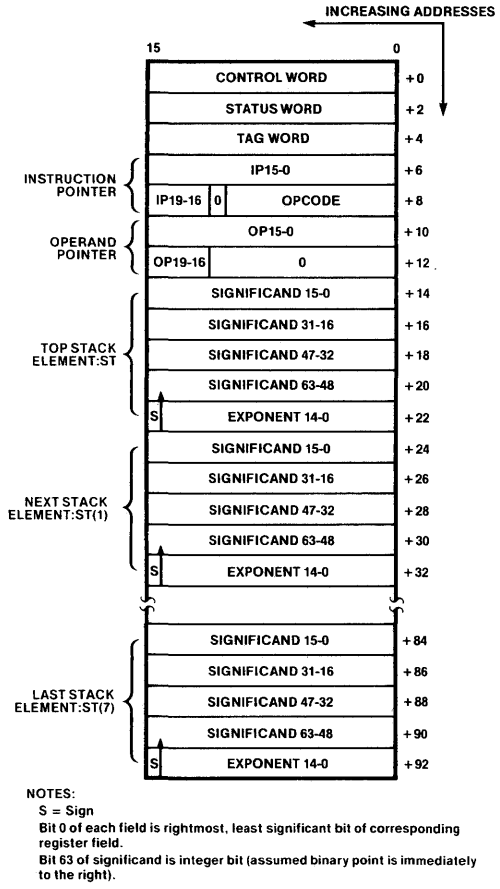



Figure 13-3. Memory Layout of REAL Save Area 121636-5

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 13-4.

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;
```

## 13.10 Floating-Point Linkage

This section deals with the issues of choosing the linkage specifications appropriate to your use of the REAL math facility: no use, PL/M-86 use only, or use of routines not written in PL/M-86. What is appropriate also depends on whether execution will use an actual 8087 chip or an emulator.

These linkage specifications make available to your program the libraries of floating-point functions. The circumstances determining which library is appropriate are given in table 13-2. The libraries themselves are discussed briefly below the table.

**Table 13-2. Linkage Choices for REAL-Math Usage**

| Use of REAL Math Facility | Emulator or Actual Chip Used | Link-List Should Include the Specifications Below (Not Necessarily in the Order Shown) After Object Modules |
|---|---|---|
| NONE | NEITHER | (none) |
| All Floating Point in PL/M-86 ONLY | EMULATOR | E8087.LIB, PE8087 |
| With Some Modules That use Floating Point NOT in PL/M-86 | EMULATOR | E8087.LIB, E8087 |
| ANY | ACTUAL 8087 CHIP | 8087.LIB |

The interface libraries do the following:

- 8087.LIB resolves external references inserted by the translator of an iAPX 86 program so that floating-point instructions will correctly invoke the 8087 chip. 8087.LIB is the library of floating-point functions written for the chip itself rather than for the Emulator.
- E8087.LIB resolves such references to invoke the Emulator software instead of the actual 8087 chip.

Emulation is performed by E8087 or PE8087.

- E8087 is the actual library of emulation routines, which provide every function and feature of an actual 8087 chip except speed. Emulation is invoked automatically as needed, using interrupts 20 through 31. The full Emulator occupies about 16K bytes of code space.
- PE8087 is a subset of E8087. The REAL arithmetic performed by PL/M-86 programs does not require the complete set of routines in the full Emulator. The full Emulator occupies about 8K bytes of code space.

NOTE

The 8087 Emulator processes exceptions exactly as the 8087 does. However, if your iAPX 86/8087 implementation includes some external interrupt masking device such as an 8259A, the effect of this external device cannot be simulated by the 8087 Emulator. With the Emulator, an Interrupt 16 will occur after the execution of any instruction when the (emulated) interrupt is active and the iAPX 86 interrupt is enabled, even if the 8259A is disabled.

(For examples of how to link interface libraries with your program, see specific host-system appendix.)

To locate the 8087 Emulator at a specified memory location:

- Locate the read-only code by referring to class "AQMCODE" in the LOC86 invocation.
- Locate the read-write data area by referring to class "AQMDATA."

NOTE

The 8087 Emulator uses iAPX 86 interrupts 20 through 31. Therefore if your program uses any REAL variables, absolute memory locations 50H through 7FH in the iAPX 86 final located module will contain the necessary code for these interrupts in the iAPX 86 interrupt vector. These memory locations should *not* be used for any other purpose.

# CHAPTER 14
# SUPPORT LIBRARY: PLM86.LIB

The PLM86.LIB library supports *, /, and MOD operations on the DWORD data type. You must link to this support library (using LINK86) before executing any program that performs these operations on DWORD types.

**NOTE**

The DWORD routines in this library form only one segment; a code segment called LQ__PLM86__LIB__CODE. Since it is read-only code, it may be burned into ROM.

```
8086/8087/8088 MACRO ASSEMBLER     PLM86.LIB 32-BIT MULTIPLY ARITHMETIC ROUTINE                    08/15/81  PAGE    1

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LQ_DWORD_MUL
OBJECT MODULE PLACED IN :F5:LODWML.OBJ
NO INVOCATION LINE CONTROLS

 LINE     SOURCE

    1 +1  $TITLE(' PLM86.LIB 32-BIT MULTIPLY ARITHMETIC ROUTINE
    2                   NAME    LQ_DWORD_MUL
    3     ;
    4     ; PL/M-86 32-BIT MULTIPLY ARITHMETIC ROUTINE
    5     ;
    6     ;
    7     ;
    8     ;
    9     ;
   10     ;
   11     ASSUME  CS:LQ_PLM86_LIB_CODE
   12     LQ_PLM86_LIB_CODE       SEGMENT PUBLIC 'CODE'
   13                     PUBLIC  LQ_DWORD_MUL
   14
   15
   16     ;
   17     ; DX:AX => OPERAND 1
   18     ; DI:CX => OPERAND 2
   19     ; DX:AX => RESULT
   20     ;
   21     ; THE ALGORITHM IS VERY SIMPLE, IT CONVERTS
   22     ;
   23     ;       1)  (HIGH(OP1) * LOW(OP1)) * (HIGH(OP2) * LOW(OP2)) =
   24     ;
   25     ;       2)  (HIGH(OP1) * (HIGH(OP2))+(HIGH(OP1)*LOW(OP2)) +
   26     ;           (LOW(OP1) * (HIGH(OP2)) + (LOW(OP1) * LOW(OP2)) =
   27     ;
   28     ;       3)  (HIGH(OP1) * LOW(OP2)) + (LOW(OP1) * HIGH(OP2)) + (LOW(OP1)*LOW(OP2))
   29     ;           SINCE  HIGH(OP1) * HIGH(OP2) YIELDS A RESULT THAT IS TOO LARGE TO FIT
   30     ;           IN 32 BITS, IT CAN BE IGNORED
   31     ;
   32     ;       4)  HIGH(RESULT) = LOW( (HIGH(OP1) * LOW(OP2)) ) +
   33     ;                          LOW( (LOW(OP1) * HIGH(OP2)) ) +
   34     ;                          HIGH( (LOW(OP1) * LOW(OP2)) )
   35     ;           LOW(RESULT) =  LOW( (LOW(OP1) * LOW(OP2)) )
   36     ;
   37     ;           IT SHOULD BE NOTED FURTHER THAT THE HIGH RESULT FROM ANY MULTIPLICATION
   38     ;           INVOLVING THE HIGH PART OF AN OPERAND CAN BE IGNORED SINCE IT WILL ALSO
   39     ;           EXCEED THE 32 BITS THAT ARE AVAILABLE FOR THE RESULT
   40     ;
   41     LQ_DWORD_MUL    PROC    FAR
   42                     MOV BX,AX     ; SAVE LOW WORD OF OP1
   43                     MOV AX,DX     ; SET UP TO MULTIPLY HIGH(OP1) * LOW(OP2)
   44                     MUL CX
   45                     MOV SI,AX     ; HIGH PART OF RESULT IS MEANINGLESS. SAVE LOW PART AS
   46                                   ; FIRST COMPONENT OF HIGH(RESULT)
   47                     MOV AX,DI     ; SET UP TO MULTIPLY HIGH(OP2) * LOW(OP1)
   48                     MUL BX
   49                     ADD SI,AX     ; AS BEFORE, HIGH PART IS MEANINGLESS. ADD LOW PART AS
   50                                   ; A COMPONENT OF HIGH(RESULT)
   51                     MOV AX,CX     ; SET UP FOR LOW(OP1) * LOW(OP2)
   52                     MUL BX        ; LOW PART OF THIS RESULT IS THE SOLE COMPONENT OF THE
   53                                   ; LOW WORD OF THE RESULT
   54                     ADD DX,SI     ; THE OTHER COMPONENTS OF HIGH(RESULT) ARE ADDED TO THIS
   55                                   ; COMPONENT OF HIGH(RESULT)
   56                     RET           ; RETURN TO INTERFACE ROUTINE
   57     LQ_DWORD_MUL    ENDP
   58
   59     LQ_PLM86_LIB_CODE       ENDS
   60                     END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

**Figure 14-1. Listing of PLM86.LIB Multiplication Routine**

```
8086/8087/8088 MACRO ASSEMBLER     PLM86.LIB 32-BIT DIVISION/MOD ARITHMETIC ROUTINE                08/15/81  PAGE    1


SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE
OBJECT MODULE PLACED IN :F5:LODWDV.OBJ
NO INVOCATION LINE CONTROLS


   LINE     SOURCE

      1 +1  $TITLE(' PLM86.LIB 32-BIT DIVISION/MOD ARITHMETIC ROUTINE
      2                     NAME    LO_DWORD_DIV
      3     ;
      4     ; PL/M-86 32-BIT DIVISION/MOD ARITHMETIC ROUTINE
      5     ;
      6     ;
      7     ;
      8     ;
      9     ;
     10     ;
     11     ASSUME  CS:LO_PLM86_LIB_CODE
     12     LO_PLM86_LIB_CODE       SEGMENT PUBLIC 'CODE'
     13                     PUBLIC  LO_DWORD_DIV
     14
     15
     16     ;
     17     ; DX:AX => NUMERATOR AND RESULT
     18     ; DI:CX => DENOMINATOR
     19     ; SI:DI => REMAINDER
     20     ;
     21     ; DX:AX PAIR IS USED TO HOLD TWO VALUES,
     22     ; THE NUMERATOR AND THE PARTIAL RESULT (SHIFTED IN 1 BIT AT A TIME).
     23     ; TWO SPECIAL CASES ARE TESTED FOR (TO REDUCE TIME SPENT IN THIS ROUTINE)
     24     ;       1) DI = 0 AND CX > DX   IN THIS CASE, THE RESULT WILL BE < 65536 SO
     25     ;          WILL FIT IN AX AND NOT CAUSE OVERFLOW
     26     ;       2) DI = 0 AND CX <= DX  DIVISION OF THE HIGH(NUMERATOR) BY BX YIELDS THE
     27     ;          COMPONENT OF THE HIGH(RESULT) AND A REMAINDER IN
     28     ;          DX THAT OBEYS THE RULES FOR CASE #1 ABOVE, ALLOWING
     29     ;          US TO DIVIDE FOR THE LOW(RESULT)
     30     ;
     31     LO_DWORD_DIV    PROC    FAR
     32                     OR DI,DI          ; TEST FOR HIGH OF DENOMINATOR = 0
     33                     JNZ DWORD_DWORD_DIV
     34                     CMP CX,DX         ; IF BX > DX THEN CAN DO SIMPLE DIVIDE
     35                     JBE DWORD_WORD_LONG
     36                     DIV CX            ; RESULT FITS IN A WORD SO CAN DO SIMPLE DIV
     37                     XOR SI,SI         ; SET UP STANDARD RESULT/REMAINDER REGS
     38                     MOV DI,DX         ; REMAINDER IN SI:DI
     39                     XOR DX,DX         ; RESULT IN DX:AX
     40                     RET
     41     DWORD_WORD_LONG:
     42                     MOV DI,AX         ; KNOW DI = 0 SO USE DI AS A TEMP
     43                     MOV AX,DX         ; SET UP TO DIVIDE HIGH PART
     44                     XOR DX,DX
     45                     DIV CX            ; DIVIDE HIGH PART BY LOW OF DENOMINATOR
     46                                       ; (HIGH PART KNOWN TO BE 0)
     47                     XCHG DI,AX        ; SAVE RESULT AND MOVE LOW OF NUMERATOR INTO  AX
     48                                       ; DX STILL CONTAINS REMAINDER OF PREVIOUS DIV SO
     49                                       ; IS ALREADY SET UP FOR THIS NEXT DIV
     50                     DIV CX            ; DIVIDE (REMAINDER OF HIGH) + LOW
     51                     XCHG DX,DI        ; SET UP RESULT AND REMAINDER REGS
     52                     XOR SI,SI
     53                     RET
     54     DWORD_DWORD_DIV:
     55                     PUSH BP           ; BP USED AS A GENERAL REG
     56                     MOV BP,CX
     57                     MOV BX,DI
     58                     XOR     SI,SI     ; INITIALIZE REMAINDER TO 0
     59                     MOV     DI,SI
     60                     MOV     CX, 32    ; INITIALIZE COUNTER
     61     DIV_MOD_LOOP:
     62                     SHL     DI, 1
     63                     RCL     SI, 1     ; REMAINDER*2
     64                     SHL     AX, 1
     65                     RCL     DX, 1     ; NUMERATOR*2
     66                     ADC     DI, 0     ; ADD IN ANY CARRY FROM THE SHIFT
     67                     SUB     DI, BP    ; SUBTRACT DENOM FROM REMAINDER
     68                     SBB     SI, BX
     69                     INC     AX        ; TURN ON RESULT BIT FOR NOW
     70                                       ; INC DOESN'T AFFECT CARRY FLAG
     71                     JNC     INC_REMAINDER
     72     ; CAN'T SUBTRACT THE DENOMINATOR FROM THE REMAINDER, ADD IT BACK
     73                     ADD     DI, BP
     74                     ADC     SI, BX
     75                     DEC     AX        ; TURN RESULT BIT OFF
     76     INC_REMAINDER:
     77                     LOOP    DIV_MOD_LOOP
     78                     POP     BP        ; RESTORE BP
     79                     RET               ; RETURN TO INTERFACE ROUTINE
     80     LO_DWORD_DIV    ENDP
     81
     82     LO_PLM86_LIB_CODE       ENDS
     83                     END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 14-2. Listing of PLM86.LIB Division/Mod Arithmetic Routine

# CHAPTER 15
# COMPILER CONTROLS

## 15.1 Introduction to Compiler Controls

The exact operation of the compiler may be controlled by a number of *controls* that specify options such as the type of listing to be produced and the destination of the object file. Controls may be specified as part of the command invoking the compiler, or as *control lines* appearing as part of the source input file.

A *control line* is a source line containing a dollar sign ($) in the left margin. Normally, the left margin is set at column one, but this may be changed with the LEFTMARGIN control. Control lines are introduced into the source to allow selective control over sections of the program. For example, it may be desirable to suppress the listing for certain sections of the program, or to cause page ejects at certain places.

A line is considered a control line by the compiler if there is a dollar sign in the left margin, even if it appears to be part of a PL/M-86 comment or character string constant.

On a control line, the dollar sign is followed by zero or more blanks and then by a sequence of controls. The controls must be separated from each other by one or more blanks.

Examples of control lines

```
$NOCODE XREF
$ EJECT CODE
```

There are two types of controls: *primary* and *general*. Primary controls must occur either in the invocation command or in a control line which precedes the first non-control line of the source file. Primary controls may not be changed within a module. General controls may occur either in the invocation command or on a control line located anywhere in the source input, and may be changed freely within a module.

There are a large number of available controls, but you may only need to specify a few of them for most compilations, because a set of defaults is built into the compiler. The controls are summarized in alphabetic order in table 15-1.

A *control* consists of a control-name which, depending on the particular control, may be followed by a parenthesized *control parameter*.

Examples of controls

```
LIST
NOXREF
OBJECT(PROG2.OJB)
```

Table 15-2 lists the controls in the order they are discussed in this chapter. This is approximately in order of importance or usage. Primary controls have an asterisk attached. Examples of system-dependent controls are included on fold-out pages in your specific host-system appendix.

## Table 15-1. Compiler Controls

| Primary Control Names | Abbreviation | Default | Discussed in Section |
|---|---|---|---|
| DEBUG/NODEBUG | DB | NODEBUG | 15.4 |
| INTVECTOR/NOINTVECTOR | IV | INTVECTOR | 15.4 |
| OBJECT/NOOBJECT | OB | OBJECT(*source-file*.OBJ) | 15.4 |
| OPTIMIZE | OT | OPTIMIZE(1) | 15.4 |
| PAGING/NOPAGING | PI | PAGING | 15.6 |
| PAGELENGTH | PL | PAGELENGTH(60) | 15.6 |
| PAGEWIDTH | PW | PAGEWIDTH(120) | 15.6 |
| PRINT/NOPRINT | PR | PRINT(*source-file*.LST) | 15.5 |
| RAM/ROM | none | RAM | 15.4 |
| SMALL/COMPACT/MEDIUM/LARGE | SM/CP/MD/LA | SMALL | 15.4 |
| SYMBOLS/NOSYMBOLS | SB | NOSYMBOLS | 15.5 |
| TITLE | TT | *module name* | 15.6 |
| TYPE/NOTYPE | TY | TYPE | 15.4 |
| WORKFILES | WF | WORKFILES(:WORK:,:WORK:) | 15.2 |
| XREF/NOXREF | XR | NOXREF | 15.5 |
| **General Control Names** | **Abbreviation** | **Default** | **Discussed in Section** |
| CODE/NOCODE | CO | NOCODE | 15.5 |
| COND/NOCOND | none | COND | 15.8 |
| EJECT | EJ | — | 15.6 |
| IF/ELSEIF/ELSE/ENDIF | none | — | 15.8 |
| INCLUDE | IC | — | 15.7 |
| LEFTMARGIN | LM | LEFTMARGIN(1) | 15.3 |
| LIST/NOLIST | LI | LIST | 15.5 |
| OVERFLOW/NOOVERFLOW | OV | NOOVERFLOW | 15.4 |
| SAVE/RESTORE | SA/RS | — | 15.7 |
| SET/RESET | none | RESET (0) | 15.8 |
| SUBTITLE | ST | no subtitle | 15.6 |

## Table 15-2. Controls by Categories

| Section | | |
|---|---|---|
| 15.2 | Compiler Resources | *WORKFILES |
| 15.3 | Input Format | LEFTMARGIN |
| 15.4 | Object File | *INTVECTOR/NOINTVECTOR<br>OVERFLOW/NOOVERFLOW<br>*OPTIMIZE<br>*OBJECT/NOOBJECT<br>*DEBUG/NODEBUG<br>*TYPE/NOTYPE<br>*SMALL/COMPACT/MEDIUM/LARGE<br>*ROM/RAM |
| 15.5 | Listing Content | *PRINT/NOPRINT<br>LIST/NOLIST<br>CODE/NOCODE<br>*XREF/NOXREF<br>*SYMBOLS/NOSYMBOLS |
| 15.6 | Listing Format | *PAGING/NOPAGING<br>*PAGELENGTH<br>*PAGEWIDTH<br>*TITLE<br>SUBTITLE<br>EJECT |

* Denotes primary control.

Table 15-2. Controls by Categories (Cont'd.)

| Section | | |
|---|---|---|
| 15.7 | Source Inclusion and Control Status | INCLUDE<br>SAVE/RESTORE |
| 15.8 | Conditional Compilation | IF/ELSEIF/ELSE/ENDIF<br>SET/RESET<br>COND |

* denotes primary control.

## 15.2 The WORKFILES Control

The WORKFILES control is a primary control, with the form:

```
WORKFILES(directory-name,[directory-name])
Default:  WORKFILES(:WORK:,:WORK:)
```

Each *directory-name* represents a direct access device such as a disk drive.

During compilation, the compiler creates work files that are deleted at the end of compilation. If the WORKFILES control is not used, these files will be the system default. The WORKFILES control allows you to specify any two devices for storage of these files. (See examples in specific host-system appendix.)

As a rule of thumb, the space required for work files on *each* device is roughly equal to the total space required for the PL/M-86 source (including "included" source files—see section 15.7). If only one device is used for work files, it should have twice this amount of space available.

## 15.3 The LEFTMARGIN Control

This is the only control for specifying the format of the source input. It is a general control with the form:

```
LEFTMARGIN(column)
Default: LEFTMARGIN(1)
```

where *column* is a non-zero, unsigned integer specifying the left margin of the source input. All characters to the left of this position on subsequent input lines are not processed by the compiler (but do appear on the listing).

The new setting of the left margin takes effect on the next input line. It remains in effect for all input from the source file and any INCLUDE files until it is reset by another LEFTMARGIN control.

Note that a control line is one that contains a dollar sign in the column specified by the most recent LEFTMARGIN control.

## 15.4 Object File Controls

These controls determine what type of object file is to be produced and on which device it is to appear. The controls are discussed in the following order:

INTVECTOR/NOINTVECTOR
OVERFLOW/NOOVERFLOW
OPTIMIZE
OBJECT/NOOBJECT
DEBUG/NODEBUG
TYPE/NOTYPE
Program Size Controls
RAM/ROM

### INTVECTOR/NOINTVECTOR

These are primary controls. They have the form

```
INTVECTOR
NOINTVECTOR
```
Default: `INTVECTOR`

Under the INTVECTOR control, the compiler creates an interrupt vector consisting of a 4-byte entry for each interrupt procedure in the module. For Interrupt *n*, the interrupt vector entry is located at absolute location 4**n*. See Chapter 10 and Appendix I for further discussion.

Alternatively, it may be desirable to create the interrupt vector independently, using either PL/M-86 or assembly language. In this case, the NOINTVECTOR control is used and the compiler does not generate any interrupt vector. The implications of this are discussed in Appendix I.

### OVERFLOW/NOOVERFLOW

These are general controls. They have the form:

```
OVERFLOW
NOOVERFLOW
```
Default: `NOOVERFLOW`

These controls specify whether overflow is to be detected in performing signed (INTEGER) arithmetic. If the NOOVERFLOW control is specified, no overflow detection is implemented in the compiled module and the results of overflow in signed arithmetic are undefined. If the OVERFLOW control is specified, overflow in signed arithmetic results in a nonmaskable Interrupt 4, and it is the programmer's responsibility to provide an interrupt procedure to handle the interrupt. Failure to provide such a procedure may result in unpredictable program behavior when overflow occurs.

If this control is nested within a program statement, overflow detection will begin when the next complete statement is evaluated.

Note that the use of the OVERFLOW control results in some expansion of the object code.

## OPTIMIZE

This is a primary control. It has the form:

`OPTIMIZE(`*n*`)`
Default: `OPTIMIZE(1)`

where *n* may be 0, 1, 2, or 3.

This control governs the kinds of optimization to be performed in generating object code.

### OPTIMIZE(0)

OPTIMIZE(0) specifies only "folding" of constant expressions.

Folding means recognizing, during compilation, operations that are superfluous or combinable, and removing or combining them so as to save memory space or execution time. Examples include addition with a zero operand, multiplication by one, and logical expressions with "true" or "false" constants. Also, in the statement:

```
A = 6 + 3 + A;
```

the compiler will add 6 and 3, producing code to add 9 to A.

### OPTIMIZE(1)

OPTIMIZE(1) specifies strength reduction, elimination of common subexpressions, and short-circuit evaluation of some Boolean expressions, in addition to the above optimizations of level (0).

Strength reduction means substituting quick operations in place of longer operations, e.g., shifting left by 1 instead of multiplying by 2. This requires less space for the instruction as well as executing faster. The addition of identical subexpressions may also result in generation of left shift instructions.

Elimination of common subexpressions means that if an expression reappears in the same block, its value is re-used rather than being recomputed. The compiler also recognizes commutative forms of subexpressions, e.g., A+B and B+A are seen to be the same. Intermediate results during expression evaluation are saved in registers and/or on the stack for later use, for example:

```
A = B + C*D/3;
C = E + D*C/3;
```

The value of C*D/3 will not be recomputed for the second statement.

Optimizing the evaluation of Boolean expressions uses the fact that in certain cases some of the terms are not needed to determine the value of the expression. For example, in the expression:

```
( A > B     AND     I > J )
```

if the first term (A > B) is false, the entire expression is false, and it is not necessary to evaluate the second term. The use of PL/M-86 built-in procedures does not change this optimization. However, if a user-function or an embedded assignment is part of the expression, this short evaluation is not done. For example:

```
( A > B     AND     ( UFUN (A) > J ))
```

is evaluated in full.

## OPTIMIZE(2)

OPTIMIZE(2) specifies all of the above, plus the following:

- Machine code optimizations (e.g., short jumps, moves)
- Elimination of superfluous branches
- Re-use of duplicate code
- Removal of unreachable code and reversal of branch-condition

Optimizing machine code means using shorter forms for identical machine instructions, to save space. This is possible because the iAPX 86 has multiple forms for some of its instructions. For example:

```
MOV RESLT1, AX; /* move accumulator value to location RESLT1 */
```

can be generated in 3 bytes as A30800, or in 4 bytes as B9060800. The former choice saves a byte of storage for the program. Similarly, jumps that the compiler can recognize as within the same segment or even closer, within 127 bytes, permit the use of fewer-byte instructions.

Elimination of superfluous branches means optimizing consecutive or multiple branches into a single branch example. For example:

```
          JZ       LAB1;      /* Jump on zero to LAB1 */
          JMP      LAB2;      /* unconditional jump to LAB2 */
LAB1:     .....
          ...
          ...
LAB2:     .....
```

will be transformed into:

```
          JNZ      LAB2;      /* Jump on non-zero to LAB2 */
LAB1:     .....
          ...
          ...
LAB2:     .....
```

Similarly, multiple branches like the following are eliminated:

```
LAB0:     JMP      LAB1
          ...
          ...
LAB1:     JMP      LAB2
          ...
          ...
LAB2:     .....
```

and transformed into:

```
LAB0:   JMP     LAB2
        ...
        ...
LAB1:   JMP     LAB2
        ...
        ...
LAB2:   .....
```

Reuse of duplicate code can refer to identical code at the end of two converging paths. In such a case the code is inserted in only one path, and a jump to that path is inserted in the other path, for example:

```
DECLARE A BYTE, SPOT POINTER;
DECLARE S BASED SPOT STRUCTURE (B BYTE, C BYTE);
IF A = 1 THEN
       S.C = INPUT (0F7H) AND 07FH;
ELSE
       S.C = INPUT (0F9H) and 07FH;
```

**Before**

```
      CMP   A,1H
      JZ    @+5H
      JMP   @1
      IN    0F7H
      AND   AL, 7FH
      MOV   BX, SPOT
      MOV   S [BX+1H], AL
      JMP   @2
@1:   IN    0F9H
      AND   AL, 7FH
      MOV   BX, SPOT
      MOV   S [BX+1H], AL
@2:
```

**After**

```
      CMP   A,1H
      JNZ   @1

      IN    0F7H
      JMP   @2


@1:    IN   0F9H
@2:   AND   AL, 7FH
      MOV   BX, SPOT
      MOV   S [BX+1H], AL
```

Reuse of duplicate code can also refer to machine instructions immediately preceding a loop being identical to those ending the loop. A branch can be generated to re-use the code generated at the beginning of the loop, for example:

**Before**

```
      ADD   AX, BX
      MOV   ANS, AX
LAB0: MOV   AL, DUM1
      CMP   AL, DUM2
      JNZ   LAB1
      ...
      ...
      ADD   AX, BX
      MOV   ANS, AX
      JMP   LAB0
LAB1: .....
```

**After**

```
LAB0: ADD     AX, BX
      MOV     ANS, AX
      MOV     AL, DUM1
      CMP     AL, DUM2
      JNZ     LAB1
      ...
      ...
      JMP     LAB0
LAB1: ....
```

This is safe so long as LAB0 is not the target of a jump instruction. The compiler normally handles a whole procedure at a time, and is thus aware of such a condition. The optimization cannot be safely applied to labels in the outer level of the main program module. This optimization will not change your program and will save code space.

The optimization that removes unreachable code takes a second look at the generated object code, finding areas which can never be reached due to the control structures created earlier.

For example, if the following code were generated before optimization:

```
    MOV     AX, A
    RCR     AL, 1
    JB      @1
    JMP     @2

@1: MOV     AX, 0FFFFH
    OUTW    0F6H
    JMP     @2
    MOV     AX, B
    ADD     A, AX
    JMP     @3
@2: ....
    ....
    ....
@3: ......
    ......
```

then the removal of unreachable code would produce:

```
    MOV     AX, A
    RCR     AL, 1
    JB      @1
    JMP     @2

@1: MOV     AX, 0FFFFH
    OUTW    0F6H
    JMP     @2
@2: ...
    ...
@3: .....
```

This can be further optimized by reversing the branch condition in the third instruction and removing the unnecessary JMP @2:

```
    MOV     AX, A
    RCR     AL, 1
    JNB     @2

@1: MOV     AX, 0FFFFH
    OUTW    0F6H
@2: ...
    ...
@3: ......
```

## OPTIMIZE(3)

OPTIMIZE(3) includes all of the above optimizations. It also optimizes indeterminate storage operations (e.g., those using based variables or variables declared with the AT attribute) and pointer comparisons. The two assumptions validating these new optimizations are that based variables (or AT variables) are not overlaying other user-declared variables, and that segments are not overlapped.

On this optimization level, all Boolean expressions are short-circuited except those containing embedded assignments. (For a description of how this optimization occurs, see OPTIMIZE(1).)

The benefits of this optimization level include more efficient use of code space because the user guarantees he has not caused an overlay of needed values. Faster execution of pointer comparisons is a consequence of the user guaranteeing there are no overlapped segments.

The first guarantee is a consequence of user caution in variable-declaration and usage. For example, the sequence:

```
DECLARE (I,J) WORD;
DECLARE THETA (19) AT (@I);
DECLARE A BASED J (10)
        STRUCTURE (F1 BYTE,F2 WORD);
..
J=.I;
....
..
A(I).F1 = 7;
A(I).F2 = 99;
THETA(I) = 31;
..
..
```

violates this caution and guarantee because it causes the values being used as pointers and subscripts to be overlaid. The compiler normally takes steps to avoid the difficulties implied here, but in OPTIMIZE(3) these steps are omitted due to the implicit user guarantee that such situations have been avoided.

OPTIMIZE(3) also changes the way POINTER values are compared. The normal case in comparing PTR_1 and PTR_2 is this: for each pointer, the segment word is effectively multiplied by 16 and the offset word is added, giving a full 20-bit address. The two 20-bit addresses are compared and the correct result is returned.

These manipulations are not needed under this level of optimization due to the implicit guarantee that no segments overlap. Thus it is sufficient to compare the segment parts bit for bit in order to determine which is a lower number. Only if the segment parts are equal is it necessary to compare the offset parts. Pointer comparisons are therefore faster under this level of optimization.

The second guarantee mentioned above required no special action unless the AT attribute and the segment-locating controls of LINK86 and LOC86 are invoked. Users exercising these controls must consider carefully their full effects. If segments are overlapped and pointer comparison is used in the program, this optimization level must not be used.

Figures 15-1 through 15-4 illustrate the levels of optimization described above.

```
system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F1:EXMPLE.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:EXMPLE.SRC NOPAGING COMPACT CODE
                      OPTIMIZE(0)


  1          EXAMPLES_OF_OPTIMIZATIONS: DO;
  2    1        DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1, PTR_2) POINTER,
                       ABASED BASED PTR_1 (10) WORD;
  3    1        DO WHILE D(A+B) < D(A+B+1);
  4    2           IF PTR_1 = PTR_2 THEN DO;
  6    3              A = A * 2;
  7    3              ABASED(A) = ABASED(B);
  8    3              ABASED(B) = ABASED(C);
  9    3              END;
 10    2           ELSE A = A + 1;
 11    2        END;
 12    1     END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 3


          0000  8BEC            MOV     BP,SP
          0002  FB              STI
                          @1:
          0003  8B1E0000        MOV     BX,A
          0007  031E0200        ADD     BX,B
          000B  D1E3            SHL     BX,1
          000D  8B360000        MOV     SI,A
          0011  03360200        ADD     SI,B
          0015  D1E6            SHL     SI,1
          0017  8B870600        MOV     AX,D[BX]
          001B  3B840800        CMP     AX,D[SI+2H]
          001F  7203            JB      $+5H
          0021  E97700          JMP     @2
                                                    ; STATEMENT # 4
          0024  C406CE00        LES     AX,PTR_1
          0028  06              PUSH    ES        ; 1
          0029  C416D200        LES     DX,PTR_2
          002D  8CC7            MOV     DI,ES
          002F  5E              POP     SI        ; 1
          0030  B104            MOV     CL,4H
          0032  8BD8            MOV     BX,AX
          0034  D3EB            SHR     BX,CL
          0036  03F3            ADD     SI,BX
          0038  8BDA            MOV     BX,DX
          003A  D3EB            SHR     BX,CL
          003C  03FB            ADD     DI,BX
          003E  3BF7            CMP     SI,DI
          0040  7507            JNE     $+9H
          0042  240F            AND     AL,0FH
          0044  80E20F          AND     DL,0FH
          0047  3AC2            CMP     AL,DL
          0049  7403            JZ      $+5H
          004B  E94100          JMP     @3
                                                    ; STATEMENT # 6
          004E  8B060000        MOV     AX,A
          0052  D1E0            SHL     AX,1
          0054  89060000        MOV     A,AX
                                                    ; STATEMENT # 7
          0058  8B360200        MOV     SI,B
          005C  D1E6            SHL     SI,1
          005E  8B3E0000        MOV     DI,A
          0062  D1E7            SHL     DI,1
          0064  C41ECE00        LES     BX,PTR_1
          0068  268B00          MOV     AX,ES:[BX].ABASED[SI]
          006B  C41ECE00        LES     BX,PTR_1
          006F  268901          MOV     ES:[BX].ABASED[DI],AX
                                                    ; STATEMENT # 8
          0072  8B360400        MOV     SI,C
          0076  D1E6            SHL     SI,1
          0078  8B3E0200        MOV     DI,B
          007C  D1E7            SHL     DI,1
          007E  C41ECE00        LES     BX,PTR_1
          0082  268B00          MOV     AX,ES:[BX].ABASED[SI]
          0085  C41ECE00        LES     BX,PTR_1
          0089  268901          MOV     ES:[BX].ABASED[DI],AX
                                                    ; STATEMENT # 10
          008C  E90900          JMP     @4
                          @3:
          008F  8B060000        MOV     AX,A
          0093  40              INC     AX
          0094  89060000        MOV     A,AX
                                                    ; STATEMENT # 11
                          @4:
          0098  E968FF          JMP     @1
                          @2:
                                                    ; STATEMENT # 13


MODULE INFORMATION:

     CODE AREA SIZE     = 009BH    155D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-1. Sample Program Showing the OPTIMIZE(0) Control

```
system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F1:EXMPLE.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:EXMPLE.SRC NOPAGING COMPACT CODE
                      OPTIMIZE(1)


  1          EXAMPLES_OF_OPTIMIZATIONS: DO;
  2    1        DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1, PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
  3    1        DO WHILE D(A+B) < D(A+B+1);
  4    2           IF PTR_1 = PTR_2 THEN DO;
  6    3              A = A * 2;
  7    3              ABASED(A) = ABASED(B);
  8    3              ABASED(B) = ABASED(C);
  9    3              END;
 10    2           ELSE A = A + 1;
 11    2        END;
 12    1     END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 3


          0000  8BEC            MOV     BP,SP
          0002  FB              STI
                          @1:
          0003  8B1E0000        MOV     BX,A
          0007  8B060200        MOV     AX,B
          000B  03D8            ADD     BX,AX
          000D  D1E3            SHL     BX,1
          000F  8B8F0600        MOV     CX,D[BX]
          0013  3B8F0800        CMP     CX,D[BX+2H]
          0017  7203            JB      $+5H
          0019  E96800          JMP     @2
                                                    ; STATEMENT # 4
          001C  C406CE00        LES     AX,PTR_1
          0020  06              PUSH    ES          ; 1
          0021  C416D200        LES     DX,PTR_2
          0025  8CC7            MOV     DI,ES
          0027  5E              POP     SI          ; 1
          0028  B104            MOV     CL,4H
          002A  8BD8            MOV     BX,AX
          002C  D3EB            SHR     BX,CL
          002E  03F3            ADD     SI,BX
          0030  8BDA            MOV     BX,DX
          0032  D3EB            SHR     BX,CL
          0034  03FB            ADD     DI,BX
          0036  3BF7            CMP     SI,DI
          0038  7507            JNE     $+9H
          003A  240F            AND     AL,0FH
          003C  80E20F          AND     DL,0FH
          003F  3AC2            CMP     AL,DL
          0041  7403            JZ      $+5H
          0043  E93700          JMP     @3
                                                    ; STATEMENT # 6
          0046  8B060000        MOV     AX,A
          004A  D1E0            SHL     AX,1
          004C  89060000        MOV     A,AX
                                                    ; STATEMENT # 7
          0050  8B360200        MOV     SI,B
          0054  D1E6            SHL     SI,1
          0056  D1E0            SHL     AX,1
          0058  C41ECE00        LES     BX,PTR_1
          005C  268B08          MOV     CX,ES:[BX].ABASED[SI]
          005F  89C6            MOV     SI,AX
          0061  268908          MOV     ES:[BX].ABASED[SI],CX
                                                    ; STATEMENT # 8
          0064  8B360400        MOV     SI,C
          0068  D1E6            SHL     SI,1
          006A  8B3E0200        MOV     DI,B
          006E  D1E7            SHL     DI,1
          0070  C41ECE00        LES     BX,PTR_1
          0074  268B00          MOV     AX,ES:[BX].ABASED[SI]
          0077  268901          MOV     ES:[BX].ABASED[DI],AX
                                                    ; STATEMENT # 10
          007A  E90400          JMP     @4
                          @3:
          007D  FF060000        INC     A
                                                    ; STATEMENT # 11
                          @4:
          0081  E97FFF          JMP     @1
                          @2:
                                                    ; STATEMENT # 13


MODULE INFORMATION:

     CODE AREA SIZE     = 0084H    132D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-2. Sample Program Showing the OPTIMIZE(1) Control

```
system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F1:EXMPLE.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:EXMPLE.SRC NOPAGING COMPACT CODE
                      OPTIMIZE(3)


   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2    1        DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1, PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
   3    1        DO WHILE D(A+B) < D(A+B+1);
   4    2           IF PTR_1 = PTR_2 THEN DO;
   6    3              A = A * 2;
   7    3              ABASED(A) = ABASED(B);
   8    3              ABASED(B) = ABASED(C);
   9    3              END;
  10    2           ELSE A = A + 1;
  11    2        END;
  12    1     END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 3


          0000  8BEC            MOV     BP,SP
          0002  FB              STI
                          @1:
          0003  8B1E0000        MOV     BX,A
          0007  A10200          MOV     AX,B
          000A  03D8            ADD     BX,AX
          000C  D1E3            SHL     BX,1
          000E  8B8F0600        MOV     CX,D[BX]
          0012  3B8F0800        CMP     CX,D[BX+2H]
          0016  7343            JNB     @2
                                        ; STATEMENT # 4
          0018  C406CE00        LES     AX,PTR_1
          001C  06              PUSH    ES      ; 1
          001D  C416D200        LES     DX,PTR_2
          0021  8CC7            MOV     DI,ES
          0023  5E              POP     SI      ; 1
          0024  3BF7            CMP     SI,DI
          0026  7502            JNE     $+4H
          0028  3BC2            CMP     AX,DX
          002A  7529            JNZ     @3
                                        ; STATEMENT # 6
          002C  A10000          MOV     AX,A
          002F  D1E0            SHL     AX,1
          0031  A30000          MOV     A,AX
                                        ; STATEMENT # 7
          0034  8B3E0200        MOV     DI,B
          0038  D1E7            SHL     DI,1
          003A  D1E0            SHL     AX,1
          003C  C41ECE00        LES     BX,PTR_1
          0040  268B09          MOV     CX,ES:[BX].ABASED[DI]
          0043  96              XCHG    AX,SI
          0044  268908          MOV     ES:[BX].ABASED[SI],CX
                                        ; STATEMENT # 8
          0047  8B360400        MOV     SI,C
          004B  D1E6            SHL     SI,1
          004D  268B08          MOV     CX,ES:[BX].ABASED[SI]
          0050  268909          MOV     ES:[BX].ABASED[DI],CX
                                        ; STATEMENT # 10
          0053  EBAE            JMP     @1
                          @3:
          0055  FF060000        INC     A
                                        ; STATEMENT # 11
          0059  EBA8            JMP     @1
                          @2:
                                        ; STATEMENT # 13


MODULE INFORMATION:

     CODE AREA SIZE     = 005BH     91D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-3. Sample Program Showing the OPTIMIZE(2) Control

```
system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F1:EXAMPLE.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:EXMPLE.SRC NOPAGING COMPACT CODE
                      OPTIMIZE(3)


   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2    1        DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1, PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
   3    1        DO WHILE D(A+B) < D(A+B+1);
   4    2           IF PTR_1 = PTR_2 THEN DO;
   6    3              A = A * 2;
   7    3              ABASED(A) = ABASED(B);
   8    3              ABASED(B) = ABASED(C);
   9    3              END;
  10    2           ELSE A = A + 1;
  11    2        END;
  12    1     END EXAMPLES_OF_OPTIMIZATIONS;
                                                  ; STATEMENT # 3


            0000  8BEC            MOV     BP,SP
            0002  FB              STI
                          @1:
            0003  8B1E0000        MOV     BX,A
            0007  A10200          MOV     AX,B
            000A  03D8            ADD     BX,AX
            000C  D1E3            SHL     BX,1
            000E  8B8F0600        MOV     CX,D[BX]
            0012  3B8F0800        CMP     CX,D[BX+2H]
            0016  7343            JNB     @2
                                                  ; STATEMENT # 4
            0018  C406CE00        LES     AX,PTR_1
            001C  06              PUSH    ES      ; 1
            001D  C416D200        LES     DX,PTR_2
            0021  8CC7            MOV     DI,ES
            0023  5E              POP     SI      ; 1
            0024  3BF7            CMP     SI,DI
            0026  7502            JNE     $+4H
            0028  3BC2            CMP     AX,DX
            002A  7529            JNZ     @3
                                                  ; STATEMENT # 6
            002C  A10000          MOV     AX,A
            002F  D1E0            SHL     AX,1
            0031  A30000          MOV     A,AX
                                                  ; STATEMENT # 7
            0034  8B3E0200        MOV     DI,B
            0038  D1E7            SHL     DI,1
            003A  D1E0            SHL     AX,1
            003C  C41ECE00        LES     BX,PTR_1
            0040  268B09          MOV     CX,ES:[BX].ABASED[DI]
            0043  96              XCHG    AX,SI
            0044  268908          MOV     ES:[BX].ABASED[SI],CX
                                                  ; STATEMENT # 8
            0047  8B360400        MOV     SI,C
            004B  D1E6            SHL     SI,1
            004D  268B08          MOV     CX,ES:[BX].ABASED[SI]
            0050  268909          MOV     ES:[BX].ABASED[DI],CX
                                                  ; STATEMENT # 10
            0053  EBAE            JMP     @1
                          @3:
            0055  FF060000        INC     A
                                                  ; STATEMENT # 11
            0059  EBA8            JMP     @1
                          @2:
                                                  ; STATEMENT # 13


MODULE INFORMATION:

     CODE AREA SIZE     = 005BH     91D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-4. Sample Program Showing the OPTIMIZE(3) Control

## OBJECT/NOOBJECT

These are primary controls. They have the form:

```
OBJECT
OBJECT (pathname)
NOOBJECT
Default: OBJECT (source-file.OBJ)
```

The OBJECT control specifies that an object module is to be created during the compilation. The *pathname* is a standard operating system pathname that specifies the file to receive the object module. If the control is absent or if an OBJECT control appears without a pathname, the object module is directed to the same file name as used for source input, but with the extension OBJ. (See example in specific host-system appendix.)

The NOOBJECT control specifies that an object module is not to be produced.

## DEBUG/NODEBUG

These are primary controls. They have the form:

```
DEBUG
NODEBUG
Default: NODEBUG
```

The DEBUG control specifies that the object module is to contain the statement number and relative address of each source program statement, information about each local symbol including based symbols and procedure parameters, and block information for each procedure. This information may be used later for symbolic debugging by an ICE-86 or ICE-88 emulator.

The NODEBUG control specifies that this information is not to be placed in the object module.

## TYPE/NOTYPE

These are primary controls. They have the form:

```
TYPE
NOTYPE
Default: TYPE
```

The TYPE control specifies that the object module is to contain information on the types of the variables output in symbols records. This information may be used later for type checking by LINK86, or an ICE-86 and ICE-88 emulator.

The NOTYPE control specifies that such type definitions are not to be placed in the object module.

## Program Size Controls

These controls specify the memory size requirements of the program that is to contain the module being compiled. They affect the operation of the compiler in various ways and impose certain constraints on the source module being compiled, as explained in detail in Chapter 17.

Note that for maximum efficiency of the object code, the smallest usable size should be used for any given program. Also note that all modules of a program must be compiled with the same size control. These are primary controls. They have the form:

```
SMALL
COMPACT
MEDIUM
LARGE
```
Default: `SMALL`

See Chapter 17 for discussions of the output of the compiler and of programming restrictions under each size control. Extensions to these controls are discussed in Appendix G.

### SMALL

The SMALL control provides for programs with the following space requirements:

- Not more than 64K bytes total for code sections from all modules
- Not more than 64K bytes total for constant, data, stack, and memory sections from all modules.

See Chapter 17 for details.

Note that the SMALL size is the most likely choice for compiling modules originally written in PL/M-80.

### COMPACT

The COMPACT control provides for programs with the following space requirements:

- Not more than 64K bytes total for code sections from all modules
- Not more than 64K bytes total for data and constant sections for all modules
- Not more than 64K bytes total for stack sections from all modules
- Not more than 64K bytes total for memory sections from all modules

See Chapter 17 for details.

### MEDIUM

The MEDIUM control provides for programs with the following space requirements:

- Not more than one megabyte total for code sections from all modules
- Not more than 64K bytes total for constant, data, stack, and memory sections from all modules.

Note that no one code section (compiled from one module) may exceed 64K bytes. See Chapter 17 for details.

### LARGE

The LARGE control provides for programs with the following space requirements:

- Not more than one megabyte total for code sections from all modules
- Not more than one megabyte total for data sections from all modules
- Not more than 64K bytes total for stack sections from all modules
- Not more than 64K bytes total for memory sections from all modules

In the LARGE case, no constant section is produced. Instead, the program constants are placed in the code section of each module.

Note that no one code or data section may exceed 64K bytes.

See Chapter 17 for details.

### RAM/ROM Control

This primary control directs the object-module placement of all constants, both user-defined and compiler-generated. Its form is:

```
RAM
ROM
```
Default: `RAM`

The default setting places the CONSTANT section within the DATA segment in all segmentation models (sizes) except LARGE, in which constants are placed in the CODE segment instead.

The ROM setting places constants in the CODE segment. Under this setting, the INITIAL attribute on a variable produces a warning message. The dot operator is not advised for variable references under the ROM option, since constants and variables will be relative to different segment registers. If SMALL is also specified, then pointers will be four bytes instead of two. (See also Appendix H.)

If the keyword DATA is used in a PUBLIC declaration when compiling with the ROM control, DATA must also be used in the EXTERNAL declaration of program modules that reference it. However, no value list is then permitted, since the data is defined elsewhere.

## 15.5 Listing Selection and Content Controls

These controls determine what types of listings are to be produced and on which device they are to appear. The controls are discussed in the following order:

PRINT/NOPRINT
LIST/NOLIST
CODE/NOCODE
XREF/NOXREF
IXREF/NOIXREF
SYMBOLS/NOSYMBOLS

A sample listing is discussed at the end of this section.

## PRINT/NOPRINT

These are primary controls. They have the form:

```
PRINT
PRINT (pathname)
NOPRINT
Default: PRINT (source-file.LST)
```

The PRINT control specifies that printed output is to be produced. *Pathname* is a standard operating system pathname that specifies the file to receive the printed output. Any output-type device, including a disk file, may also be given. If the control is absent, or if a PRINT control appears without a pathname, printed output is sent to a file that has the same name as the source file but with the extension LST. (See example in specific host-system appendix.)

The NOPRINT control specifies that no printed output is to be produced, even if implied by other listing controls such as LIST and CODE.

## LIST/NOLIST

These are general controls. They have the form:

```
LIST
NOLIST
Default: LIST
```

The LIST control specifies that listing of the source program is to resume with the next source line read.

The NOLIST control specifies that listing of the source program is to be suppressed until the next occurrence, if any, of a LIST control.

When LIST is in effect, all input lines (from the source file or from an INCLUDE file), including control lines, are listed. When NOLIST is in effect, only source lines associated with error messages are listed.

Note that the LIST control *cannot* override a NOPRINT control. If NOPRINT is in effect, no listing whatsoever is produced.

## CODE/NOCODE

These are general controls. They have the form:

```
CODE
NOCODE
Default: NOCODE
```

The CODE control specifies that listing of the generated object code in standard assembly language format is to begin. This listing is placed at the end of the program listing on the listing file.

The NOCODE control specifies that listing of the generated object code is to be suppressed until the next occurrence, if any, of a CODE control.

Note that the CODE control cannot override a NOPRINT control.

### XREF/NOXREF

These are primary controls. They have the form:

```
XREF
NOXREF
Default: NOXREF
```

The XREF control specifies that a cross-reference listing of source program identifiers is to be produced on the listing file.

The NOXREF control suppresses the cross-reference listing.

Note that the XREF control cannot override a NOPRINT control.

### SYMBOLS/NOSYMBOLS

These are primary controls. They have the form:

```
SYMBOLS
NOSYMBOLS
Default: NOSYMBOLS
```

The SYMBOLS control specifies that a listing of all identifiers in the PL/M-86 source program and their attributes is to be produced on the listing file.

The NOSYMBOLS control suppresses such a listing.

Note that the SYMBOLS control cannot override a NOPRINT control.

## 15.6 Listing Format Controls

Format controls determine the format of the listing output of the compiler. The controls are discussed in the following order:

PAGING/NOPAGING
PAGELENGTH
PAGEWIDTH
TITLE
SUBTITLE EJECT

### PAGING/NOPAGING

These are primary controls. They have the form:

```
PAGING
NOPAGING
Default: PAGING
```

The PAGING control specifies that the listed output is to be formatted onto pages. Each page carries a heading identifying the compiler and a page number, and possibly a user specified title.

The NOPAGING control specifies that page ejecting, page heading, and page numbering are not to be performed. Thus, the listing appears on one long "page" as would be suitable for a slow serial output device. If NOPAGING is specified, a page eject is not generated if an EJECT control is encountered.

## PAGELENGTH

This is a primary control. It has the form:

`PAGELENGTH(`*length*`)`
Default: `PAGELENGTH(60)`

where *length* is a non-zero, unsigned integer specifying the maximum number of lines to be printed per page of listing output. This number is taken to include the page headings appearing on the page.

The minimum value for *length* is 5.

## PAGEWIDTH

This is a primary control. It has the form:

`PAGEWIDTH(`*width*`)`
Default: `PAGEWIDTH(120)`

where *width* is a non-zero, unsigned integer specifying the maximum line width, in characters, to be used for listing output.

The minimum value for *width* is 60; the maximum value is 132.

## TITLE

This is a primary control. It has the form:

`TITLE('`*title*`')`
Default: *module name*

where *title* is a sequence of printable ASCII characters that are enclosed in quotes.

The sequence, truncated on the right if necessary to fit, is placed in the title line of each page of listed output.

The maximum length allowed for *title* is 60 characters, but a narrow pagewidth may restrict this number further, for example:

```
TITLE('TEST PROGRAM 4')
```

## SUBTITLE

This is a general control. It has the form:

`SUBTITLE('`*subtitle*`')`
Default: no subtitle

where *subtitle* is a sequence of printable ASCII characters that are enclosed in quotes.

The sequence, truncated on the right if necessary to fit, is placed in the subtitle line of each page of listed output.

The maximum length allowed for *subtitle* is 60 characters, but a narrow pagewidth may restrict this number further, for example:

```
SUBTITLE('TEST PROGRAM 4')
```

When a SUBTITLE control appears before the first noncontrol line in the source file, it causes the specified subtitle to appear on the first page and all subsequent pages until another SUBTITLE control appears.

A subsequent SUBTITLE control causes a page eject, and the new subtitle appears on the next page and all subsequent pages until the next SUBTITLE control.

## EJECT

This is a general control. It has the form:

```
EJECT
```

It causes printing of the current page to terminate and a new page to be started. The control line containing the EJECT control is the first printed (following the page heading) on the new page.

# Sample Program Listing

During the compilation process a listing of the source input is produced. Each page of the listing carries a numbered page-header that identifies the compiler, prints a time and date as designated by the host operating system, and optionally gives a title and a subtitle, and/or a date (see figure 15-5).

```
system-id PL/M-86 V2.0 COMPILATION OF MODULE STACK
OBJECT MODULE PLACED IN :F1:STACK.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:STACK.SRC NOPAGING CODE XREF
                      TITLE(STACK MODULE)


  1          STACK: DO;
             /* This module implements a BYTE stack with push and pop */
  2    1        DECLARE S(100) BYTE, /*Stack Storage*/
                        T BYTE PUBLIC INITIAL(-1);  /*Stack Index*/

  3    1        PUSH: PROCEDURE (B) PUBLIC;  /*Pushes B onto the stack*/
  4    2           DECLARE B BYTE;
  5    2           S(T:=T+1) = B; /*Increment T and store B*/
  6    2        END PUSH;

  7    1        POP: PROCEDURE BYTE PUBLIC; /*Returns value popped from stack*/
  8    2           RETURN S((T:=T-1)+1); /*Decrement T, return S(T+1)*/
  9    2        END POP;
 10    1     END STACK;  /*Module ends here*/
                                              ; STATEMENT # 3


                     PUSH      PROC NEAR
0000  55                       PUSH    BP
0001  8BEC                     MOV     BP,SP
                                              ; STATEMENT # 5
0003  8A066400                 MOV     AL,T
0007  FEC0                     INC     AL
0009  B400                     MOV     AH,0H
000B  88066400                 MOV     T,AL
000F  89C3                     MOV     BX,AX
0011  8A4E04                   MOV     CL,[BP].B
0014  888F0000                 MOV     S[BX],CL
                                              ; STATEMENT # 6
0018  5D                       POP     BP
0019  C20200                   RET     2H
                     PUSH      ENDP
                                              ; STATEMENT # 7
                     POP       PROC NEAR
001C  55                       PUSH    BP
001D  8BEC                     MOV     BP,SP
                                              ; STATEMENT # 8
001F  8A066400                 MOV     AL,T
0023  FEC8                     DEC     AL
0025  B400                     MOV     AH,0H
0027  88066400                 MOV     T,AL
002B  89C3                     MOV     BX,AX
002D  8A870100                 MOV     AL,S[BX+1H]
0031  5D                       POP     BP
0032  C3                       RET
                                              ; STATEMENT # 9
                     POP       ENDP
                                              ; STATEMENT # 11
```

Figure 15-5. Program Listing

The first part of the listing contains a summary of the compilation, beginning with the compiler identification and the name of the source module being compiled. The next line names the file receiving the object code. Then the command line used to invoke the compiler is reproduced. The listing of the program itself is shown in figure 15-5.

The listing contains a copy of the source input plus additional information. To the left of the source image appear two columns of numbers. The first column provides a sequential numbering of PL/M-86 statements. Error messages, if any, refer to these statement numbers. The second column gives the block nesting depth of the current statement.

Lines included with the INCLUDE control are marked with = just to the left of the source image. If the included file contains another INCLUDE control, lines included by this nested INCLUDE are marked with =1. For yet another level of nesting, =2 is used to mark each line, and so forth up to the compiler's limit of five levels of nesting. These markings make it easy to see where included text begins and ends.

Should a source line be too long to fit on the page in one line it will be continued on the following line. Such continuation lines are marked with "-" just to the left of the source image.

The CODE control may be used to obtain the iAPX 86 assembly code produced in the translation of each PL/M-86 statement. This code listing appears interspersed in the source text in six columns of information in a pseudo-assembly language format:

1. Location counter (hexadecimal notation)
2. Resultant binary code (hexadecimal notation)
3. Label field
4. Opcode mnemonic
5. Symbolic arguments
6. Comment field

Not all six of these columns will appear on any one line of the code listing. Compiler generated labels (e.g., those which mark the beginning and ending of a DO WHILE loop) are preceded by @. The comments appearing on PUSH and POP instructions indicate the stack depth associated with the stack instruction.

## Symbol and Cross-Reference Listing

If specified by the XREF or SYMBOLS control, a summary of all identifier usage appears following the program listing.

Depending on whether the SYMBOLS or XREF control was used to request the identifier usage summary, six or seven types of information are provided in the symbol or cross-reference listing. These are as follows:

1. Statement number where identifier was defined
2. Relative address associated with identifier
3. Size of object identified (in bytes)
4. The identifier
5. Attributes of the identifier (including expansion for LITERALLYs and scoping information for local variables and parameters)

6. Statement numbers where identifier was referenced (XREF control only)
7. Statement numbers where identifier was assigned a value (XREF control only)

Notice that a single identifier may be declared more than once in a source module (i.e., an identifier defined twice in different blocks). Each such unique object, even though named by the same identifier, appears as a separate entry in the listing.

The address given for each object is the location of that object relative to the start of its associated section. Which section is applicable depends upon the attributes of the object.

Identifiers in the SYMBOLS or XREF listing are given in alphabetical order with the following exception: members of structures are listed, in order of declaration, immediately following the entry for the structure itself. Indentation is used to differentiate between these entries.

The XREF listing differentiates between items 6 and 7 by adding the * character to statement numbers where a value is assigned. For example, if statement 17 read:

```
I = I + 1;
```

the list of statement numbers would include 17 and 17*, indicating a reference and an assignment in statement 17.

The AUTOMATIC attribute indicates that the identifier was declared as a parameter or as a local variable in a REENTRANT procedure, and therefore is allocated dynamically on the stack.

Figure 15-6 is an example of the cross-reference listing.

```
PL/M-86 COMPILER     STACK MODULE                    09/28/81
                     CROSS-REFERENCE LISTING

  DEFN  ADDR    SIZE  NAME, ATTRIBUTES, AND REFERENCES
 -----  ------ -----  --------------------------------

    4  0004H     1  B. . . . . .    BYTE IN PROC (PUSH) PARAMETER AUTOMA
                                      -TIC        4     5
    7  001CH    23  POP. . . . . .  PROCEDURE BYTE PUBLIC STACK=0002H
    3  0000H    28  PUSH . . . . .  PROCEDURE PUBLIC STACK=0004H
    2  0000H   100  S. . . . . .    BYTE ARRAY(100)          5*    8
       0000H        STACK. . . . .  PROCEDURE STACK=0000H
    2  0064H     1  T. . . . . .    BYTE PUBLIC INITIAL           8*
                                      8
```

Figure 15-6. Cross-Reference Listing

## Compilation Summary

Following the listing (or appearing alone if NOLIST is in effect) is a compilation summary. Seven pieces of information are provided:

- *Code area size* gives the size in bytes of the *code section* of the output module.
- *Constant area size* gives the size in bytes of the *constant section* of the output module.
- *Variable area size* gives the size in bytes of the *data section* of the output module.
- *Maximum stack size* gives the size in bytes of the *stack section* allocated for the output module.
- *Lines read* gives the number of source lines processed during compilation.

- *Dictionary required* gives the amount of dictionary space (if any) that was spilled to disk (see Appendix B).
- *Program warnings* give the number of warning messages issued during compilation.
- *Program errors* gives the number of error messages issued during compilation.

Figure 15-7 is an example of the compilation summary. Refer to Chapter 17 for an explanation of the various object module sections.

```
MODULE INFORMATION:

     CODE AREA SIZE     = 0033H     51D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0065H    101D
     MAXIMUM STACK SIZE = 0004H      4D
     14 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-7. Compilation Summary

## 15.7 Source Inclusion Controls

These controls allow the input source to be changed to a different file. The controls are:

INCLUDE
SAVE/RESTORE

### INCLUDE

INCLUDE is a general control, with the form:

INCLUDE(*pathname*)

where *pathname* is a standard operating system pathname specifying a file. (See example in specific host-system appendix.)

An INCLUDE control must be the rightmost control in a control line or in the invocation command.

The INCLUDE control causes subsequent source lines to be input from the specified file. Input will continue from this file until an end-of-file is detected. At that time, input will be resumed from the file which was being processed when the INCLUDE control was encountered.

An included file may itself contain INCLUDE controls. Note that such nesting of included files may not exceed a depth of *five*.

### SAVE/RESTORE

These are general controls. They have the form:

```
SAVE
RESTORE
```

These controls allow the settings of certain general controls to be saved on a stack before an INCLUDE control switches the input source to another file, and then restored after the end of the included file. However, SAVE and RESTORE can be used for other purposes as well. The controls whose settings are saved and restored are:

LIST/NOLIST
CODE/NOCODE
OVERFLOW/NOOVERFLOW
LEFTMARGIN
COND/NOCOND

The SAVE control saves all of these settings on a stack. This stack has a maximum capacity of five sets of control settings, which corresponds to the maximum nesting depth of five for the INCLUDE control.

The RESTORE control restores the most recently saved set of control settings from the stack.

## 15.8 Conditional Compilation Controls

These controls allow selected portions of the source file to be skipped by the compiler if specified conditions are not met. Figure 15-8 shows an example program using the conditional compilation controls, while figure 15-9 shows the same example with NOCOND being used.

The controls are:

IF/ELSEIF/ELSE/ENDIF
SET/RESET
COND/NOCOND

### IF / ELSE / ELSEIF / ENDIF

These controls provide the actual conditional capability, using conditions which are based on the values of switches.

These controls cannot be used in the invocation of the compiler, and each must be the only control on its control line.

An IF control and an ENDIF control are used to delimit an "IF element," which can have several different forms. The simplest form of IF element is:

```
$IF condition
text
$ENDIF
```

where

- *condition* is a limited form of PL/M expression, in which the only operators allowed are OR, XOR, NOT, AND, <, <=, =, >=, and >, and the only operands allowed are switches and whole-number constants in the range 0 to 255. If the switch did not appear in a set control, a value of "false" (0) is assumed. Parenthesized subexpressions are not allowed. Within these restrictions, *condition* is evaluated according to the PL/M-86 rules for expression evaluation. Note that *condition* ends with a carriage return.

- *text* is text which will be processed normally by the compiler if the least significant bit of the value of *condition* is a 1, or skipped if the bit is a 0. Note that text may contain any mixture of PL/M-86 source and compiler controls. If the text is skipped, any controls within it are not processed.

The second form of IF element contains an ELSE element:

```
$IF condition
text 1
$ELSE
text 2
$ENDIF
```

In this construction, *text 1* will be processed normally if the least significant bit of the value of *condition* is a 1, while *text 2* will be skipped. If the bit is a 0, *text 1* will be skipped and *text 2* will be processed normally.

Note that only one ELSE element is allowed within an IF element.

The most general form of IF element allows one or more ELSEIF elements to be introduced *before* the ELSE element (if any):

```
$IF condition 1
text 1
$ELSEIF condition 2
text 2
$ELSEIF condition 3
text 3

.
.
.

$ELSEIF condition n
text n
$ELSE
text n+1
$ENDIF
```

where any of the ELSEIF elements may be omitted, as may the ELSE element.

The conditions are tested in sequence. As soon as one of them yields a value with a 1 as its least significant bit, the associated text is processed normally. All other text in the IF element is skipped. If none of the conditions yields a least significant bit of 1, the text in the ELSE element (if any) is processed normally and all other text in the IF element is skipped.

## SET/RESET

These are general controls. The SET control has the general form:

`SET(`*switch assignment list*`)`

where *switch assignment list* consists of one or more switch assignments separated by commas. A switch assignment has the form:

```
SWITCH
```

or:

```
SWITCH = VALUE
```

where

- *switch* is a name which is formed according to the PL/M-86 rules for identifiers. Note that a switch name exists only at the compiler control level, and therefore you may have a switch with the same name as an identifier in the program; no conflict is possible. However, note that a PL/M-86 reserved word may *not* be used as a switch name.
- *value* is a whole-number constant in the range 0 to 255. This value is assigned to the switch. If the *value* and the = sign are omitted from the switch assignment, the default value 0FFH ("true") is assigned to the switch.

The following is an example of a SET control line:

```
$SET(TEST,ITERATION=3)
```

This example sets the switch TEST to "true" (0FFH) and the switch ITERATION to 3. Note that switches do not need to be declared.

Figure 15-8 is an example of a program that was compiled using the SET control.

```
PL/M-86 COMPILER    EXAMPLE

system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:CEX.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:CEX.SRC SET(DEBUG=3)


   1          EXAMPLE: DO;

   2    1     DECLARE BOOLEAN LITERALLY 'BYTE', TRUE LITERALLY '0FFH',
                  FALSE LITERALLY '0';
   3    1     PRINT$DIAGNOSTICS: PROCEDURE (SWITCHES, TABLES) EXTERNAL;
   4    2     DECLARE (SWITCHES, TABLES) BOOLEAN;
   5    2     END PRINT$DIAGNOSTICS;

   6    2     DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8    2     AWAIT$CR: PROCEDURE EXTERNAL; END AWAIT$CR;

              $IF DEBUG = 1
                  CALL PRINT$DIAGNOSTICS (TRUE, FALSE);
              $   RESET (TRAP)
              $ELSEIF DEBUG = 2
                  CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
              $   RESET (TRAP)
              $ELSEIF DEBUT = 3
                  CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
                  CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
              $   SET (TRAP)
              $ENDIF

              $IF TRAP
                  CALL DISPLAY$PROMPT;
                  CALL AWAIT$CR;
              $ENDIF

  10    1     END EXAMPLE;
```

Figure 15-8. Sample Program Showing the SET(DEBUG=) Control

```
MODULE INFORMATION:

     CODE AREA SIZE      = 0000H      0D
     CONSTANT AREA SIZE  = 0000H      0D
     VARIABLE AREA SIZE  = 0000H      0D
     MAXIMUM STACK SIZE  = 0000H      0D
     30 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

**Figure 15-8. Sample Program Showing the SET(DEBUG=) Control (Cont'd.)**

The RESET control has the form:

RESET (*switch list*)

where ***switch list*** consists of one or more switch names that have already occurred in SET controls.

Each switch in the ***switch list*** is set to "false" (0).

## COND/NOCOND

These controls determine whether text within an IF element will appear in the listing if it is skipped. They are general controls with the form:

COND
NOCOND
Default: COND

The COND control specifies that any text that is skipped is to be listed (without statement or level numbers). Note that a COND control cannot override a NOLIST or NOPRINT control, and that a COND control will not be processed if it is within text which is skipped.

The NOCOND control specifies that text within an IF element which is skipped is not to be listed. However, the controls that delimit the skipped text *will* be listed, providing an indication that something has been skipped. Note that a NOCOND control will not be processed if it is within text that is skipped.

Figure 15-9 is an example of a program that was compiled using the NOCOND control.

```
PL/M-86 COMPILER    EXAMPLE

system-id PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:CEX.OBJ
COMPILER INVOKED BY:  :F1:PLM86.86 :F1:CEX.SRC SET(DEBUG=3) NOCOND


   1          EXAMPLE: DO;

   2   1      DECLARE BOOLEAN LITERALLY 'BYTE', TRUE LITERALLY 'OFFH',
                    FALSE LITERALLY '0';
   3   1      PRINT$DIAGNOSTICS: PROCEDURE (SWITCHES, TABLES) EXTERNAL;
   4   2      DECLARE (SWITCHES, TABLES) BOOLEAN;
   5   2      END PRINT$DIAGNOSTICS;

   6   2      DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8   2      AWAIT$CR: PROCEDURE EXTERNAL; END AWAIT$CR;

              $IF DEBUG = 1
              $ENDIF

              $IF TRAP
              $ENDIF

  10   1      END EXAMPLE;


MODULE INFORMATION:

     CODE AREA SIZE     = 0000H      0D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0000H      0D
     30 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Figure 15-9. Sample Program Showing the NOCOND Control

# CHAPTER 16
# SAMPLE PROGRAM 2

The sample program of Chapter 8 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 the previous chapters, we can rewrite the sort program as a procedure. By using parameters to control the operation of the procedure, we can 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 the sample program of Chapter 8. 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 WORD parameter RECSIZE and may not exceed 128.
- In each record, the value of one byte is 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 (or offset) is specified by the WORD parameter KEYINDEX, 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 WORD parameter COUNT.

The program is followed by a detailed explanation.

```
SORT$MODULE: DO; /* Beginning of module. */

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

/*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) WORD;
```

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

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

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

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

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

    CALL SORTPROC (@SET1, LENGTH(SET1), SIZE(SET1(0)),
           SIZE(SET1(0).ALPHA));
        CALL SORTPROC (@SET2, LENGTH(SET2), SIZE(SET2(0)),
           .SET2(0).KEY - .SET2(0));

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

END SORT$MODULE; /* 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 (i.e., the subscript must be less than COUNT * RECSIZE).

Note that a dimension specifier of 1 is used in declaring RECORD. We need to use a nonzero dimension specifier here, in order to use subscripts later in the procedure. 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 the records that this procedure can handle. We have chosen 128 here, but in principle any dimension could be specified.

As in sample program 1, the WORD variables I and J are used to control the DO WHILE and iterative DO loops. 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 used the built-in procedure, MOVB to copy a sequence of byte values from one storage location to another. (See Chapter 11 for details.)

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)*RECSIZE) 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 that 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 that 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 key is the second element within the structure, its offset is just the number of bytes occupied by the first element of the structure. Therefore we will use the built-in function SIZE to calculate the offset of the key and use that as the value of the parameter KEYINDEX.

SET2 is a set of 500 structures, each containing an array of 21 INTEGER scalars, a REAL scalar, and a BYTE scalar that is to be used as the key. This time, the key is deep within the structure. We can calculate the offset of the key manually, but this is both error prone and inflexible. If the elements within the structure are changed, then we must remember to recalculate the offset of the key. A better method is to let the compiler calculate the offset for us. This can be accomplished by subtracting the location of the key element from the location of the start of the record. The result of this subtraction is the offset of the key, which becomes the value of the parameter KEYINDEX.

In the two CALL statements used to activate SORTPROC, we used the built-in function LENGTH to determine the number of records that are to be sorted, and we used the built-in function SIZE to calculate the number of bytes occupied by each record. These two values are used for the COUNT and RECSIZE parameters for an activation of SORTPROC. See Chapter 11 for complete details on built-in functions.

# CHAPTER 17
# OBJECT MODULE SECTIONS AND PROGRAM SIZE CONTROL

The allocation and arrangement of runtime program memory (via relocation and linkage) depend on the size-control (SMALL, COMPACT, MEDIUM, or LARGE) that you specified when compiling the modules of the program. All modules of a program must be compiled with the same controls.

These controls also influence how locations are referenced in the compiled program, which leads to certain programming restrictions for each size-control.

A PL/M-86 programmer need not normally be concerned about memory addressing concepts on the iAPX 86, as the size-controls transparently handle the mechanics of program segmentation. The simple rules are

- For programs with less than 64K bytes of code and less than 64K bytes of data (for a maximum program size of 128K bytes), use either the default (SMALL) or COMPACT, and observe the restrictions given in sections 17.3 and 17.4, respectively. (SMALL is recommended as a migration path from PL/M-80.)
- If your code exceeds 64K bytes, but all your data fits in 64K bytes, use MEDIUM and observe the restrictions in section 17.5.
- If you also need more than 64K bytes of data, use the LARGE control and observe the restrictions in section 17.6.

The material in this chapter explains some underlying concepts and restrictions that pertain to these guidelines. (See also Appendix H.)

## 17.1 iAPX 86 Memory Concepts

iAPX 86 memory space has an extent of one megabyte, but a 16-bit word can only address 64K locations. A complete physical address requires 20 bits. Therefore, two separate words are used in a special way to form this 20-bit address, as follows:

A *segment* is defined as contiguous memory locations, beginning at a 16-byte boundary, up to 64K bytes in length. The 20-bit address for the beginning of every segment thus ends in 0, e.g., 00000H, 00010H, . . . 12340H, . . . . This definition permits a 16-bit word to represent any segment starting address since the extra four bits not included are always 0. A 16-bit word used in this way is called a *segment address* (and corresponds to the values taken on by SELECTOR variables). Four CPU registers (CS, DS, SS, and ES) are used automatically to hold segment addresses.

The second word used in forming the full 20-bit address identifies the specific location within the segment, i.e., which of the 64K bytes starting at the segment address. This 16-bit quantity is called the *offset*.

The offset is often manipulated in PL/M-86 programs using WORD variables. References through these offset (WORD) values are relative to the current CS for indirect calls or the current DS for indirect (based) variable references.

To form a 20-bit address, the iAPX 86 CPU shifts a segment address left four bits and adds an offset to it.

## 17.2 Object Module Sections

The output of the compiler is an object file containing the compiled module. This object module may be linked with other object modules and located using LINK86 and LOC86. A knowledge of the makeup of an object module is not necessary for PL/M-86 programming, but it can aid you in understanding the controls for program size, linkage, and location.

The object module output by the compiler contains five *sections*.

- Code Section
- Constant Section (absent in LARGE case and in ROM—see below)
- Data Section
- Stack Section
- Memory Section

As explained later in this chapter, these sections can be combined in various ways into "memory segments" for execution, depending on the size of the program (SMALL, COMPACT, MEDIUM, or LARGE).

### Code Section

This section contains the object code generated by the source program. If either the LARGE control or the ROM control is used, this section also contains the information that would otherwise be in the constant section.

In addition, the code section for the main program module contains a "main program prologue" generated by the compiler. This code precedes the code compiled from the source program, and sets the CPU up for program execution by initializing various registers and enabling interrupts.

### Constant Section

This section contains all variables initialized with the DATA attribute, as well as all REAL constants and all constant lists. If the LARGE or ROM controls are used, this information is placed in the code section and no constant section is produced.

### Data Section

All variables are allocated space in this section except parameters, based variables, and variables that are located with an AT attribute, initialized with the DATA attribute, or local to a REENTRANT procedure.

Further, if a nested procedure refers to any parameter of an enclosing procedure, then during execution all parameters of the enclosing procedure will be placed in the data segment. The compiler reserves enough space during compilation to prepare for this.

### Stack Section

The stack section is used in executing procedures, as explained in Appendixes H and I. It is also used for any temporary storage used by the program but not explicitly declared in the source module (such as temporary variables generated by the compiler).

The exact size of the stack is automatically determined by the compiler except for possible multiple incarnations of reentrant procedures. The user can override this computation of stack size and explicitly state the stack requirement during the relocation process.

NOTE

When using reentrant procedures or interrupt procedures the user must be careful to allocate a stack section large enough to accommodate all possible storage required by multiple incarnations of such procedures. The stack size can be explicitly specified during the relocation and linkage process.

The stack space requirement of each procedure is shown in the listing produced by the SYMBOLS or XREF control. This information can be used to compute the additional stack space required for reentrant or interrupt procedures.

### Memory Section

This is the area of memory referenced by the built-in PL/M-86 identifier MEMORY. Its maximum allowable size depends on the size control used in compilation (SMALL, COMPACT, MEDIUM, or LARGE) as explained below.

The compiler generates a memory section of length zero, and it is your responsibility to specify the actual (run-time) space required during the linkage and relocation process. The *iAPX 86,88 Family Utilities User's Guide* can assist you in this.

## 17.3 The SMALL Case

When modules compiled with the SMALL control are linked, the code sections from all modules are combined and are allocated space within one segment. The segment address for this segment is kept in the CS register. The constant, data, stack, and memory sections from all modules are allocated space within a second segment. The segment address for this second segment is kept in the DS register, with an identical copy in the SS register.

Therefore, the SMALL control may be used if the total size of all code sections does not exceed 64K, and the total size of all constant, data, stack, and memory sections does not exceed 64K.

Since there is only one segment for code, the segment address for this segment (CS register) is never updated during program execution. Likewise, since there is only one segment for constants, data, stack, and memory sections, the segment address for this segment (DS and SS registers) is never updated (except when an interrupt occurs, as explained in Appendix I). Therefore, when any location is referenced, only a 16-bit offset is calculated and then used in conjunction with the appropriate segment address. POINTER values are therefore 16-bit values in the SMALL case, and this leads to the following restrictions.

1. POINTER variables may not be initialized with, or assigned, whole-number constants, for example:

```
DECLARE RR POINTER INITIAL (2277); /*invalid under SMALL*/
DECLARE SS POINTER;
SS = 100; /*invalid under SMALL*/
```

2. The @ operator must not be used with a variable that was located at an absolute address that was specified by a whole-number constant, for example:

```
DECLARE JO BYTE AT (100), PO POINTER;
PO = @JO; /*invalid under SMALL*/
```

This restriction does not apply if the absolute address in the declaration is *created* by the @ operator with a variable, for example:

```
DECLARE UKE BYTE, NAGE POINTER;
DECLARE SKI BYTE AT (@UKE);
NAGE = @SKI; /*valid*/
```

3. The PUBLIC attribute must not be used with a variable located at an absolute address specified by a whole-number constant. As above, this restriction does not apply when @ is used:

```
DECLARE SHOMEN BYTE PUBLIC AT (100); /*invalid*/

DECLARE IKYO BYTE;
DECLARE SANKYO BYTE PUBLIC AT (@IKYO); /*valid*/
```

   This restriction arises because external variables are assumed to be in the DATA segment.

4. Restrictions 1 and 2 apply also to WORD variables when used as offset pointers and to the use of the dot operator.
5. The @ and dot operators may not be used with variables based on SELECTOR, for example:

```
DECLARE SEL SELECTOR;
DECLARE R BASED SEL BYTE;
DECLARE PO POINTER;
PO = @R /* invalid under SMALL */
```

6. If the built-in function SELECTOR$OF is used, it will always return the value of the DS register. If BUILD$PTR is used, it will ignore the SELECTOR expression (see section 11.8).

### PL/M-80 Compatibility

The SMALL control is the most likely choice when compiling a program written in PL/M-80. The compiler produces error messages to flag violations of any of the restrictions or to flag the use of the new reserved keyboards (DWORD, INTEGER, REAL, POINTER, SELECTOR, and CAUSEINTERRUPT) as programmer-defined identifiers. Otherwise, complete upwards compatibility is provided by PL/M-86.

NOTE

Care must be used with the dot operator under conditions other than SMALL and RAM.

## 17.4 The COMPACT Case

A program compiled with the COMPACT control has four segments: code, data, stack, and memory. Each of these is the result of combining the same-type sections from all modules, and each has a maximum size of 64K bytes. The constant sections from all modules are grouped with the data segment unless the ROM control is used, which causes all constant sections to be merged into the CODE segment instead. Since the code, data, and stack segments are fully defined by the time the program is loaded, the segment base addresses in the CS and SS registers are never changed. (The DS register may change when an interrupt occurs, as explained in Appendix H.)

All code and a few prologue constants are addressed relative to CS. All data except absolute data (declared with the AT (*constant*) attribute) are addressed relative to DS. The stack is addressed relative to SS. ES is not initialized and can change during execution. References to any location require only a 16-bit offset address against these segment base addresses.

The sole programming restriction in the COMPACT case is that PUBLIC variables may not be declared AT an absolute location, for example:

```
DECLARE ANVIL BYTE PUBLIC AT (100);
```

is not allowed. This restriction does not apply when the "location" within the AT attribute is formed with the @ operator, i.e., DECLARE ANVIL BYTE PUBLIC AT (@HAMR); is valid. However, the phrases "@ MEMORY" and ".MEMORY" are not allowed in defining a PUBLIC variable.

### Programming Restrictions in the COMPACT Case

The following restrictions must be observed:

1. When an exported procedure is indirectly activated, a POINTER variable must be used in the CALL statement, for example:

```
$COMPACT(SUBSYS HAS MOD1, MOD2, MOD3; EXPORTS PROC)
MOD1: DO
   DECLARE P POINTER, W WORD;
   PROC: PROCEDURE PUBLIC;
   .
   .
   .
   END PROC;
   P=@PROC; CALL P; /* POINTER must be used */
   W=.PROC; CALL W; /* not allowed */

END MOD1;
```

2. When a procedure that is not exported is indirectly activated, a WORD variable must be used. Note that WORD variables do not range over the entire iAPX 86 address space but are restricted to offsets within the current code segment, for example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE; /* local */
.
.
.
END LPROC;
P=@LPROC; CALL P; /* not allowed */
W=.LPROC; CALL W; /* WORD must be used */
```

## 17.5 The MEDIUM Case

In a program compiled with the MEDIUM control, a separate segment is used for the code section of each compiled module. Therefore, the total space required for code may exceed 64K, although the maximum size of any one code section is still limited to 64K.

The constant, data, stack, and memory sections of all modules are combined and are allocated space within a single segment.

At any moment during program execution, one segment of code is the "current" segment, and its segment address is kept in the CS register. This segment address is updated whenever a PUBLIC or EXTERNAL procedure is activated, since this may involve a new code segment.

The segment address for the segment containing constants, data, stack, and memory sections is kept in the DS register (with an identical copy in the SS register) and is never changed (except when an interrupt occurs, as explained in Appendix I).

With the MEDIUM option, a POINTER value is a four-byte quantity containing a segment address and an offset. Therefore, the first three restrictions of the SMALL case do not apply. However, the MEDIUM case introduces two minor restrictions on indirect procedure activation.

## Programming Restrictions in the MEDIUM Case

The following restrictions must be observed:

1. When a PUBLIC or EXTERNAL procedure is indirectly activated, a POINTER variable must be used in the CALL statement, for example:

```
DECLARE P POINTER, W WORD;
PROC: PROCEDURE PUBLIC;
.
.
.
END PROC;


P=@PROC; CALL P;  /*recommended where an indirect
                    call must be used*/


W=.PROC; CALL W;  /*not allowed*/
```

2. When a procedure that is not PUBLIC or EXTERNAL is indirectly activated, a WORD variable must be used. This is consistent with PL/M-80, and is not recommended in PL/M-86 programs because WORD variables do not range over the entire iAPX 86 address space (but are restricted to offsets within an assumed segment), for example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE;     /*local*/
.
.
.
       END LPROC;
P=@LPROC; CALL P;     /*not allowed*/


W=.LPROC; CALL W;     /*not recommended, but allowed*/
```

3. A variable that is absolutely located (by using the AT attribute with a numeric constant) may not have the PUBLIC attribute. For example:

```
DECLARE B BYTE PUBLIC AT(100);
```

   is not allowed.

Restrictions 1 and 2 arise from the fact that the code segment address may change during program execution. Restriction 3 is the same as Restriction 3 in the SMALL case, and arises for the same reason.

## 17.6 The LARGE Case

In a program compiled with the LARGE control, a separate segment is used for the code section (with constants) from each compiled module. Thus the total space required for code and constants may exceed 64K, but the total for the code section (with constants) from any one module is limited to 64K.

A separate segment is used for the data section from each compiled module. Thus the total space required for data sections may exceed 64K, although the size of any one data section is limited to 64K.

The stack sections from all modules are combined in one segment, and the memory sections for all modules are combined in another segment. Thus the total space required for stack is limited to 64K, and the total space required for memory is also limited to 64K.

At any moment during program execution, one code segment and one data segment are "current." Code and data segments are paired, so that the current code and data segments are always from the same module. The compiler implements this pairing by placing the segment address for the data segment in a reserved location in the code section. During program execution, the segment addresses for the current code and data segments are kept in the CS and DS registers, respectively, and are updated whenever a PUBLIC or EXTERNAL procedure is activated, as this may involve new code and data segments.

The stack segment address is kept in the SS register.

### Programming Restrictions in the LARGE Case

These first two are the same as Restrictions 1 and 2 in the MEDIUM case, and arise for the same reason.

1. When a PUBLIC or EXTERNAL procedure is indirectly activated, a POINTER variable must be used in the CALL statement, for example:

```
DECLARE P POINTER, W WORD;
PROC: PROCEDURE PUBLIC;
.
.
.
END PROC;

P=@PROC; CALL P;  /*recommended where an indirect
                    call must be made*/

W=.PROC; CALL W;  /*not allowed*/
```

2. When a procedure that is not PUBLIC or EXTERNAL is indirectly activated, a WORD variable must be used. This is consistent with PL/M-80, and is not recommended in PL/M-86 programs because WORD variables do not range over the entire iAPX 86 address space (but are restricted to offsets within an assumed segment), for example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE;      /*local*/
.
.
.
END LPROC;

P=@LPROC; CALL P;      /*not allowed*/

W=.LPROC; CALL W;      /*not recommended, but allowed*/
```

# CHAPTER 18
# ERROR MESSAGES

The compiler may issue five varieties of error messages:

- Source PL/M-86 errors
- Fatal command tail and control errors
- Fatal input/output errors
- Fatal insufficient memory errors
- Fatal compiler failure errors

The source errors are reported in the program listing; the fatal errors are reported on the console device.

## 18.1 Source PL/M-86 Errors

Nearly all of the source PL/M-86 errors are interspersed in the listing at the point of error and follow the general format:

```
***ERROR #mmm, STATEMENT #nnn, NEAR ''aaa'', message
```

or:

```
***WARNING #mmm, STATEMENT #nnn, NEAR ''aaa'', message
```

where

- *mmm* is the error number from the list below.
- *nnn* is the source statement number where the error occurs.
- *aaa* is the source text near where the error is detected.
- *message* is the error explanation from the list below.

A prefix of W means the message is a warning only, and object code was not suppressed. Errors not so prefixed prevent object code generation.

Source error message list:

1. INVALID CONTROL

   See Chapter 15. Example:

   ```
   $NXCODE; /*probably intended NOCODE*/
   ```

2. ILLEGAL USE OF PRIMARY CONTROL AFTER NON-CONTROL LINE

   Primary controls may appear as control lines in your source program, but they must come first. No other statements may precede them. See Chapter 15.

3. MISSING CONTROL PARAMETER

   Certain controls, e.g., INCLUDE, require you to specify a parameter. See Chapter 15.

4. INVALID CONTROL PARAMETER

   One example is an illegal pathname for a control like OBJECT. See Chapter 15.

5. INVALID CONTROL FORMAT

   See Chapter 15 for correct formatting of control lines. An example that could cause this error is:

```
$LIST (MYPROG.LST);
```

   because no pathname is expected on this control. It could also be caused if an INCLUDE was followed by another control on the same line.

(W) 6. ILLEGAL PRINT CONTROL, IGNORED

   PRINT (:CI:) would be an example, since you cannot print to the console input device. See Chapter 15.

7. INVALID PATHNAME

   See the operating instructions for your specific host system.

(W) 8. ILLEGAL PAGELENGTH, IGNORED

   See section 15.6.

(W) 9. ILLEGAL PAGEWIDTH, IGNORED

   See section 15.6.

(W) 10. RESPECIFIED PRIMARY CONTROL, IGNORED

   See section 15.1.

11. MISPLACED ELSE OR ELSEIF CONTROL

   See section 15.8.

12. MISPLACED ENDIF CONTROL

   See section 15.8.

13. MISSING ENDIF CONTROL

   See section 15.8.

14. SWITCHNAME TOO LONG (31), TRUNCATED

See section 15.8.

15. MISSING OPERATOR

See section 15.8.

(W) 16. INVALID CONSTANT, ZERO ASSUMED

See section 15.8.

17. INVALID OPERAND

See section 15.8.

19. LIMIT EXCEEDED: SAVE NESTING (5)

See section 15.8.

20. LIMIT EXCEEDED: INCLUDE NESTING (5)

For example, if you include a file named A, which includes a file named B, and so on, this error will arise when the limit is exceeded.

21. MISPLACED RESTORE CONTROL

RESTORE can only work if there has been a prior SAVE. See section 15.7.

22. UNEXPECTED END OF CONTROL
A segmentation control was expecting a continuation line or a ')'.

23. SYMBOL EXISTS IN MORE THAN ONE HAS LIST

A module name may occur in only one HAS list.

24. SUBSYSTEM ALREADY DEFINED

The subsystem name has already been defined.

25. HAS OR EXPORT LIST RESPECIFIED

You cannot have more than one of each in a sub-system control.

26. ILLEGAL PL|M IDENTIFIER

Identifier does not meet the rules for PL/M identifiers. See section 2.2.

(W) 28. INVALID PLM86 CHARACTER, IGNORED

Look near the text flagged for an invalid character, or one that is inappropriate in context. Edit it out or retype the statement.

(W) 29. UNPRINTABLE CHARACTER, IGNORED

Retype the line in question using valid characters.

30. NAME OR STRING TOO LONG, TRUNCATED

Match your intended variable type with the length of the flagged item. For the correct maximum lengths, see Chapters 2 and 13.

31. ILLEGAL CONSTANT TYPE

This might reflect missing operators, e.g., A=4T instead of 4 + T. For the list of valid types, see Chapter 2.

32. INVALID CHARACTER IN CONSTANT

For example, 107B and 0ABCD will cause this error because neither can be valid in any PL/M-86 interpretation: 7 is not a binary numeral, B may not occur in decimal or octal, and neither string ends in H. See Chapter 2.

33. RECURSIVE MACRO EXPANSION

Here is an example causing this error:

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'A';
.
.
.
B=4; /* error discovered here */
```

No type can be assigned to variables declared circularly, i.e., solely in terms of each other.

34. LIMIT EXCEEDED: MACRO NESTING (5)

This error occurs when too many DECLARE statements refer back through each other to the one that actually supplies a type, for example:

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'C';
   .    .     .      .
   .    .     .      .
DECLARE Y LITERALLY 'Z';
DECLARE Z BYTE INTITIAL (77);
.
A=7;  /*  error discovered here */
```

35. LIMIT EXCEEDED: SOURCE LINE LENGTH (128)

A source line longer than 128 characters was read.

36. CONSTANT TOO LARGE FOR TYPE

DECLARE A BYTE INTITIAL (259) would be an example, because the maximum byte constant is 255.

37. INVALID REAL CONSTANT

Examples: 1.7F or 1.7. See Chapter 13.

(W) 38. REAL CONSTANT UNDERFLOW

An underflow occurred when conversion into floating-point was attempted.

(W) 39. REAL CONSTANT OVERFLOW

An overflow occurred when conversion into floating-point was attempted.

40. NULL STRING NOT ALLOWED

Strings of length zero (e.g., " ") are not supported.

41. DELETED: "<tokens>"

The compiler deleted the following tokens while attempting to recover from a syntax error.

42. INSERTED: "<tokens>"

The compiler inserted the following tokens while attempting to recover from a syntax error.

43. STATEMENTS FOLLOW MODULE END

Information follows logical end-of-module.

(W) 45. MISMATCHED BLOCK IDENTIFIER

See section 7.1. If a label is supplied in an END statement, the label must match that of a prior DO statement, in fact the first unmatched DO above the END. Sometimes the error involves a confusion of module name with procedure name. See also Chapters 1 and 9.

46. DUPLICATE PROCEDURE NAME

Procedure names must be unique. See section 10.1.

47. LIMIT EXCEEDED: PROCEDURES (253)

Too many procedures in this module. Break it into smaller modules.

48. DUPLICATE PARAMETER NAME

A parameter must be declared exactly once. This message indicates that the flagged parameter already has a definition at this block level, as in:

```
YAR: PROCEDURE (YAR77, YAR78);
DECLARE YAR77 BYTE;
DECLARE YAR77 BYTE;
```

Perhaps a different spelling was intended.

49. NOT AT MODULE LEVEL

The flagged attribute or initialization can only be valid at the module level, not in a procedure. See Chapters 1, 3, 9, and 10.

50. DUPLICATE ATTRIBUTE

Attributes should be specified at most once. This message means the compiler has found a declaration like:

```
DECLARE B BYTE EXTERNAL EXTERNAL;
```

51. ILLEGAL INTERRUPT VALUE

Interrupt numbers must be whole-number constants between 0 and 255. Thus -7 or 272 would be invalid. See section 10.5.

52. INTERRUPT WITH PARAMETERS

No parameters are allowed in interrupt procedures. See section 10.5.

53. INTERRUPT WITH TYPED PROCEDURE

Interrupt procedures must be untyped. See section 10.5.

54. INVALID DIMENSION

See section 6.1.

55. STAR DIMENSION WITH STRUCTURE

Star dimensions(*) are for initialization. See section 3.2.

56. STAR DIMENSION WITH STRUCTURE MEMBER

Star dimension(*) must not be used with structures. See section 3.2.

57. CONFLICT WITH PARAMETER

Attributes may not be used with parameters.

58. DUPLICATE DECLARATION

The flagged item already has a definition declared at this block level.

59. ILLEGAL PARAMETER TYPE

Parameters may not be declared of type structure or array. See section 10.1.

60. DUPLICATE LABEL

Each label must be unique within its block or scope. Otherwise GOTOs and CALLs would have ambiguous targets. See sections 7.3 and 10.2.

61. DUPLICATE MEMBER NAME

For example, in the case:

```
DECLARE AIR STRUCTURE (F4 BYTE, F4 BYTE);
```

subsequent references to AIR.F4 would be ambiguous. See sections 6.2, 6.3.

62. UNDECLARED PARAMETER

A parameter named in the procedure statement did not get defined in the body of the procedure. See sections 10.1 and 10.2.

63. CONFLICTING ATTRIBUTES

Certain attributes are not allowed when declaring a parameter, e.g. PUBLIC, EXTERNAL, DATA, INITIAL, AT, or BASED.

64. LIMIT EXCEEDED: DO BLOCKS

See Appendix B for correct limit.

65. ILLEGAL PARAMETER ATTRIBUTE

Certain attributes are not allowed when declaring a parameter, e.g. PUBLIC, EXTERNAL, DATA, INITIAL, AT, or BASED.

66. UNDEFINED BASE

A variable was declared BASED using an undeclared identifier.

67. INVALID ATTRIBUTE FOR BASE

A base must be a non-subscripted scalar of type POINTER, WORD, or SELECTOR. It is not permitted to have the attribute BASED. See section 4.5.

68. MISPLACED DECLARATION

You can intersperse declarations and procedures, but not declarations and executable statements. See Chapter 1.

69. INVALID BASE WITH LABEL OR MACRO

BASED may not be used with LABEL or LITERALLY types.

70. INVALID DIMENSION WITH LABEL OR MACRO

LABEL or LITERALLY may not be dimensioned.

71. INITIALIZATION LIST REQUIRED

A list of initial values is required if the INITIAL attribute is used, if the non-external * dimension form is used, or if the non-external DATA attribute is used.

72. BASED CONFLICTS WITH ATTRIBUTES

Examples of attributes conflicting with base include AT, DATA, INITIAL, PUBLIC, and EXTERNAL. See section 4.6.

73. EXECUTABLE STATEMENTS IN EXTERNAL

An EXTERNAL procedure, being defined elsewhere, may not contain executable statements. See Chapter 9 and section 10.5.

74. MISSING RETURN FOR TYPED PROCEDURE

A typed procedure must return a value, so its RETURN statement must specify one.

75. INVALID NESTED REENTRANT PROCEDURE

Reentrant procedures may not contain nested procedures. See section 10.5.

76. LIMIT EXCEEDED: FACTORED LIST (64)

Too many variables were named in a factored declaration. Break it into several declarations.

77. LIMIT EXCEEDED: STRUCTURE ELEMENTS (64)

Too many elements were declared in a structure.

78. MISSING PROCEDURE NAME

Every procedure must have a name. See section 10.1.

79. MULTIPLE PROCEDURE LABELS

Procedures must have exactly one name; no more, no less. See section 10.1.

80. DECLARATIONS MAY NOT BE LABELED

Labels may not be used on declaration statements.

81. STAR DIM WITH FACTORED LIST NOT ALLOWED

Star dimensions (*) are for initializations. See section 3.2.

82. SIZE EXCEEDS 64K BYTES

Storage for the declared item exceeds 64K bytes.

(W) 83. PROCEDURE CONTAINS NO EXECUTABLE STATEMENTS

88. LIMIT EXCEEDED: PROGRAM TOO COMPLEX

The program has too many complex expressions, cases, etc. Break it into smaller procedures.

89. COMPILER ERROR: BAD ERROR RECOVERY

An unrecoverable error occurred. Trying a different copy of the compiler on a different drive might reveal that the first copy had been damaged or gone bad. Contact Intel.

91. PROGRAM TOO COMPLEX

The program has too many complex expressions, cases, etc. Break it into smaller procedures.

93. LIMIT EXCEEDED: PROGRAM TOO COMPLEX

The program has too many complex expressions, cases, and procedures. Break it into smaller modules.

95. COMPILER ERROR: PARSE BUFFER OVERFLOW

See 89.

96. LIMIT EXCEEDED: BLOCK NESTING (18)

The program has too many nested DO blocks. Break it into smaller procedures.

97. COMPILER ERROR: STACK UNDERFLOW

See 89.

98. LIMIT EXCEEDED: STATEMENT TOO COMPLEX

The statement is too large for the compiler. Break it into several smaller statements.

99. COMPILER ERROR: SEMANTIC UNDERFLOW

See 89.

100. STRING CONSTANT TOO LONG

String constants may have a maximum of 4 characters.

101. UNSUBSCRIPTED ARRAY

In the context of the flagged statement, the array reference requires a subscript. See sections 4.4, 10.2, and Chapter 6.

102. UNQUALIFIED STRUCTURE

This statement was ambiguous as to which structure or member was intended. See sections 4.4, 10.2, and Chapter 6.

103. NOT AN ARRAY

Subscripts are permitted only on identifiers declared as arrays. Check spelling consistency. See Chapter 6.

104. MULTIPLE SUBSCRIPTS

For any array TING, references of the form TING(2,4) or TING(3,7,9,6) are invalid because of multiple subscripts. Only references of one subscript are valid, e.g., TING(5). See section 6.1.

105. NOT A STRUCTURE

For example, a reference of the form GNU.F1 where GNU was not declared a structure. See Chapter 6.

106. UNDEFINED IDENTIFIER

Every identifer must be declared. See Chapter 3.

107. UNDEFINED MEMBER

For example, KAPI.HORN where KAPI is a valid, declared structure but HORN was never declared. See Chapter 6.

108. ILLEGAL INDEX TYPE

Only BYTE, WORD, and INTEGER can be used. See section 6.1.

109. NOT A LABEL

The identifier following GOTO must be a label; the flagged item was declared otherwise.

110. MISSING RETURN VALUE

A typed procedure must return a value, so its RETURN statement must specify one. See section 10.3.

111. INVALID RETURN WITH UNTYPED PROCEDURE

An untyped procedure does not return a value, so its RETURN statement may not specify one. See section 10.3.

112. INVALID INDIRECT TYPE

Only WORD or POINTER scalars can be used for indirect calls. This excludes WORD or POINTER expressions, BYTE or REAL scalars, all structures, and all arrays. See section 10.2.

113. INVALID PARAMETER COUNT

The number of actual parameters supplied in a CALL must be equal to the number of formal parameters declared in the procedure. See section 10.2. (See also Chapter 11 for the requirements of built-ins.)

114. QUALIFIED PROCEDURE NAME

Procedure names may not be qualified.

115. INVALID FUNCTION REFERENCE

Typed procedures are validly invoked only by use in an expression, not by a CALL. See sections 10.1 and 10.2.

116. INVALID CASE EXPRESSION TYPE

Case expressions must be of type BYTE, WORD, or INTEGER.

117. LIMIT EXCEEDED: CASES

See Appendix B for correct limit.

118. TYPE CONFLICT

See section 5.6.

119. INVALID BUILT-IN REFERENCE

Built-in reference was qualified with a member name, or OUTPUT/ OUTWORD did not appear on the left side of an assignment.

120. INVALID PROCEDURE REFERENCE

Untyped procedures must be invoked by a CALL statement; references to such procedures are not permitted in expressions. See sections 10.1 and 10.2.

121. INVALID LEFT-HAND SIDE OF ASSIGNMENT

An example is PROCEDURE=4 or INWORD(7)=9. See sections 3.2 and 5.7.

122. INVALID REFERENCE

Invalid label reference

123. Not used

124. PROCEDURE NAME REQUIRED

Procedure name is required for SET$INTERRUPT and INTERRUPT$PTR built-ins.

125. PROCEDURE NAME ONLY

Parameters are not allowed on the procedure name in SET$INTERRUPT and INTERRUPT$PTR.

126. BAD INTERRUPT NUMBER

Interrupt numbers must be whole-number constants between 0 and 255. Thus, −7 and 272 would be invalid. See section 10.5.

127. CONSTANT ONLY

In this instance, a constant is required.

128. ARRAY REQUIRED

Some built-ins need an array name. See section 11.1.

129. INTERRUPT PROCEDURE REQUIRED

Procedure names for SET$INTERRUPT and INTERRUPT$PTR must have the INTERRUPT attribute.

130. INVALID RESTRICTED OPERAND

For example, PL/M-86 reserved words and predeclared identifiers would be invalid. See Appendixes C, D, and section 11.1.

131. INVALID RESTRICTED OPERATOR

Only + and − may be used in restricted expressions.

132. REAL CONSTANT EXPRESSION

A constant expression with REALs is not allowed.

133. REFERENCE REQUIRED

A variable reference is required for LENGTH, LAST, and SIZE.

134. VARIABLE REQUIRED

The operand to LENGTH, LAST, and SIZE must be a variable.

135. VALUE TOO LARGE

A value is too large for its contextually determined type.

136. ABSOLUTE POINTER WITH SHORT POINTERS

An absolute pointer was used in the SMALL case.

137. INVALID RESTRICTED EXPRESSION

Only addresses or constant types are allowed in restricted expressions.

138. PUBLIC AT EXTERNAL

See sections 3.2, 9.2, for example:

```
DECLARE DARTH BYTE EXTERNAL;
DECLARE VADER BYTE PUBLIC AT (.DARTH);
```

139. PUBLIC AT ABSOLUTE

Absolute locations for PUBLICs are supported only under the LARGE option. See Chapter 17.

140. PUBLIC AT MEMORY

PUBLIC AT (@MEMORY) is not supported by SMALL.

141. AT BASED VARIABLE

Based variables cannot be used in AT clauses. See section 4.8.

142. ILLEGAL FORWARD REFERENCE

An AT expression cannot have a forward reference.

143. VARIABLE TYPE REQUIRED IN AN AT EXPRESSION

The expression must be AT a variable.

144. LIMIT EXCEEDED: DATA OR STACK SEGMENT TOO LARGE

See Appendix B for correct limit.

145. LIMIT EXCEEDED: CODE OR CONST SEGMENT TOO LARGE

See Appendix B for correct limit.

146. LIMIT EXCEEDED: TOO MANY EXTERNALS

See Appendix B for correct limit.

147. LABEL NOT AT LOCAL OR MODULE LEVEL

See sections 7.3 and 9.3.

148. ABSOLUTE ADDRESS REFERENCE IN SMALL

See section 15.4.

149. ILLEGAL MODULE NAME REFERENCE

Module names cannot be referenced.

(W) 150. USE OF "." WITH FAR PROCEDURE

A subsequent indirect call made through the respective address/pointer generates the wrong type of call.

(W) 151. USE OF "@" WITH SHORT PROCEDURE

See 150.

152. INVALID "." OR "@" OPERAND

Must be used with a variable, procedure, or constant list.

153. INVALID RETURN IN MAIN PROGRAM

A main program must have no returns. See section 10.3.

154. STAR DIMENSIONED EXTERNAL WITH LENGTH, LAST, or SIZE

The LENGTH, LAST, and SIZE built-in functions cannot be used with variables declared with * and the EXTERNAL attribute.

155. PUBLIC SYMBOL EXPORTED FROM ANOTHER SUBSYSTEM

A PUBLIC symbol in this module is also exported by another subsystem. See Appendix G.

156. LONG POINTER REQUIRED FOR THIS CONSTRUCT

A model with long pointers is required.

157. MEMORY MAY NOT BE USED WITH LENGTH, LAST, OR SIZE

Memory has no compile-time determinable dimension.

162. LIMIT EXCEEDED: PROGRAM COMPLEXITY

Too many complex expressions, cases, etc. Break it into smaller procedures.

163. COMPILER ERROR: SEMANTIC UNDERFLOW

See 89.

164. COMPILER ERROR: INVALID NODE

See 89.

165. COMPILER ERROR: INVALID OPERATOR

See 89.

166. COMPILER ERROR: INVALID TREE

See 89.

167. COMPILER ERROR: SCOPE STACK UNDERFLOW

See 89.

168. LIMIT EXCEEDED: PROGRAM COMPLEXITY

Too many complex expressions, cases, etc. Break it into smaller procedures.

169. COMPILER ERROR: INVALID RECORD

See 89.

170. INVALID DO CASE BLOCK, AT LEAST ONE CASE REQUIRED

See section 7.1.

171. LIMIT EXCEEDED: NUMBER OF ACTIVE CASES

See Appendix B for correct limit.

172. LIMIT EXCEEDED: NESTING OF TYPED PROCEDURE CALLS

See Appendix B for correct limit.

173. LIMIT EXCEEDED: NUMBER OF ACTIVE PROCEDURES OR DO CASE GROUPS

See Appendix B for correct limit.

174. ILLEGAL NESTING OF BLOCKS, ENDS NOT BALANCED

For every DO, an END is needed. See section 7.1.

175. COMPILER ERROR: INVALID OPERATION

See 89.

176. LIMIT EXCEEDED: REAL EXPRESSION COMPLEXITY

The REAL stack has eight registers. Heavily nested use of REAL functions with REAL expressions as parameters could get excessively complex. See Chapter 13.

177. COMPILER ERROR: REAL STACK UNDERFLOW

See 89 and 176.

178. LIMIT EXCEEDED: BASIC BLOCK COMPLEXITY

You have a very long list of statements without labels, CASEs, IFs, GOTOs, RETURNs, etc. Either break the procedure into several smaller procedures, or add labels to some of your statements.

179. LIMIT EXCEEDED: STATEMENT SIZE

The statement is too large for the compiler. Break it into several smaller statements.

199. LIMIT EXCEEDED: PROCEDURE COMPLEXITY OF OPTIMIZE(2) (terminal error)

The combined complexity of expressions, user labels, and compiler-generated labels is too great. Simplify as much as possible, perhaps breaking the procedure into several smaller procedures.

200. ILLEGAL INITIALIZATION OF MORE SPACE THAN DECLARED

The number of intitialization values exceeds the number of declared elements. See section 3.2.

201. INVALID LABEL: UNDEFINED

No definition for this label was found. See sections 3.4 and 7.3.

202. LIMIT EXCEEDED: NUMBER OF EXTERNAL ITEMS

See Appendix B for correct limit.

203. COMPILER ERROR: BAD LABEL ADDRESS

See 89.

204. LIMIT EXCEEDED: CODE SEGMENT SIZE

See Appendix B for correct limit.

205. COMPILER ERROR: BAD CODE GENERATED

See 89.

251. COMPILER ERROR: INVALID OBJECT

See 89.

252. COMPILER ERROR: SELF NAME LINK

See 89.

253. COMPILER ERROR: SELF ATTR LINK

See 89.

254. LIMIT EXCEEDED: PROGRAM COMPLEXITY

The program has too many complex expressions, cases, and procedures. Break it into smaller modules.

255. LIMIT EXCEEDED: SYMBOLS

See Appendix B for correct limit.

NOTE

If a terminal error is encountered, program text beyond the point of error is not compiled. A terminal error message will appear at the beginning of the program listing and at the point of error in the program listing.

## 18.2 Fatal Command Tail and Control Errors

Fatal command tail errors are caused by an improperly specified compiler invocation command or an improper control. The errors that may occur here are as follows:

```
ILLEGAL COMMAND TAIL SYNTAX OR VALUE
UNRECOGNIZED CONTROL IN COMMAND TAIL
INCLUDE FILE IS NOT A DIRECT ACCESS FILE
INVOCATION COMMAND DOES NOT END WITH <cr><LF>
INCORRECT DEVICE SPECIFICATION
SOURCE FILE NOT A DIRECT ACCESS FILE
SOURCE FILE NAME INCORRECT
SOURCE FILE EXTENSION INCORRECT
ILLEGAL COMMAND TAIL SYNTAX
MISPLACED CONTROL: WORKFILES ALREADY OPENED
```

## 18.3 Fatal Input/Output Errors

Fatal input/output errors occur when the user incorrectly specifies a pathname for compiler input or output. These error messages are of the form:

```
PL/M-86 ERROR -
FILE:
NAME:
ERROR:
COMPILATION TERMINATED
```

## 18.4 Fatal Insufficient Memory Errors

The fatal insufficient memory errors are caused by a system configuration with not enough RAM memory to support the compiler.

The errors that may occur due to insufficient memory are as follows:

```
NOT ENOUGH MEMORY FOR COMPILATION
DYNAMIC STORAGE OVERFLOW
NOT ENOUGH MEMORY
```

## 18.5 Fatal Compiler Failure Errors

The fatal compiler failure errors are internal errors that should never occur. If you encounter such an error, please report it to Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, Attn: Software Marketing Department. The errors falling into this class are as follows:

```
89. COMPILER ERROR: BAD ERROR RECOVERY
95. COMPILER ERROR: PARSE BUFFER OVERFLOW
97. COMPILER ERROR: STACK UNDERFLOW
```

```
 99. COMPILER ERROR: SEMANTIC UNDERFLOW
163. COMPILER ERROR: SEMANTIC UNDERFLOW
164. COMPILER ERROR: INVALID NODE
165. COMPILER ERROR: INVALID OPERATOR
166. COMPILER ERROR: INVALID TREE
167. COMPILER ERROR: SCOPE STACK UNDERFLOW
169. COMPILER ERROR: INVALID RECORD
175. COMPILER ERROR: INVALID OPERATION
177. COMPILER ERROR: REAL STACK UNDERFLOW
203. COMPILER ERROR: BAD LABEL ADDRESS
205. COMPILER ERROR: BAD CODE GENERATED
251. COMPILER ERROR: INVALID OBJECT
252. COMPILER ERROR: SELF NAME LINK
253. COMPILER ERROR: SELF ATTR LINK
```

# APPENDIX A
# GRAMMAR OF THE PL/M-86 LANGUAGE

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. Its appearance differs substantially from 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. When items are stacked vertically between square brackets, only one of them may be used.

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.

## A.1 Lexical Elements

### A.1.1 Character Sets

```
<character>::= <apostrophe>
             | <non-quote character>

<non-quote character>::= <letter>
                       | <decimal digit>
                       | $
                       | <special character>
                       | blank

<letter>::= <upper case letter>
          | <lower case letter>

<upper case letter>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R| S|T|U|V|W|X| Y|Z
<lower case letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<decimal digit>::= 0|1|2|3|4|5|6|7|8|9
<special character>::= +|-|*|/|<|>|=|:|;|.|,|(|)|@|_
<apostrophe>::= '
```

### A.1.2 Tokens

```
<token>::= <delimiter>
         | <identifier>
         | <reserved word>
         | <numeric constant>
         | <string>
```

## A.1.3 Delimiters

```
<delimiter>::= <simple delimiter>
             | <compound delimiter>

<simple delimiter>::= +|-|*|/|<|>|=|:|;|.|,|(|)|@
<compound delimiter>::= <>
                      | <=
                      | >=
                      | :=
```

## A.1.4 Identifiers and Reserved Words

```
<identifier>::= <letter> [ <letter>        ] ...
                         | <decimal digit> |
                         | $               |
                         | _               |
```

<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 [ + | - ] <digit string>
<digit string>::= <decimal digit> [ <decimal digit> ] ...
                                  |       $        |

<binary digit>::= 0|1
<octal digit>::= <binary digit> |2|3|4|5|6|7
<decimal digit>::= <octal digit> |8|9
<hexadecimal digit>::= <decimal digit>|A|B|C|D|E|F
```

## A.1.6 Strings

```
<string>::= ' <string body element>  ...  '
<string body element>::= <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>]
                       | [EXTERNAL][<constant attribute>]


<locator>::= AT (<expression>)
```

```
<constant attribute>::= DATA

<array specifier>::= <explicit dimension>
                   | <implicit dimension>

<explicit dimension>::= ( <numeric constant> )
<implicit dimension>::= ( * )

<variable type>::= <basic type>
                 | <structure type>

<basic type>::= INTEGER
              | REAL
              | POINTER
              | SELECTOR
              | BYTE
              | WORD
              | DWORD
```

### A.3.3 Label Element

```
<label element>::= <identifier> LABEL [ PUBLIC ]
                                      [ EXTERNAL ]
```

### A.3.4 Literal Elements

```
<literal element>::= <identifier> LITERALLY <string>
```

### A.3.5 Factored Variable Element

```
<factored variable element>::= ( <variable name specifier>
    [,<variable name specifier>]...)
    [<explicit dimension>] <variable type>
    [<variable attributes>]
```

### A.3.6 Factored Label Elements

```
<factored label element>::= (<identifier> [,<identifier>]...) LABEL [ PUBLIC ]
                                                                   [ EXTERNAL ]
```

### A.3.7 The Structure Type

```
<structure type>::= STRUCTURE (<member element> [,<member element>]...)
<member element>::= <member name> [<explicit dimension>] <basic type>
<member name>::= <identifier>
                |(<identifier>[,<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
| REAL
| POINTER
| SELECTOR
| BYTE
| WORD
| DWORD

<formal parameter list>::= ( <formal parameter> [,<formal parameter>]... )
<formal parameter>::= <identifier>

<procedure attributes>::= { { {<interrupt> / EXTERNAL} ... } { {<interrupt> / PUBLIC / REENTRANT} ... } }

## 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 / DATA} ( <initial value> [,<initial value>] ... )

<initial value>::= <expression>
| <string>

# A.4 Units

<unit>::= <conditional clause>
| <do block>
| <basic statement>
| <label definition> <unit>

<basic statement>::= <assignment statement>
| <call statement>
| <goto statement>
| <null statement>
| <return statement>
| <iAPX 86 dependent statement>

<scoping statement>::= <simple do statement>
| <do-case statement>
| <do-while statement>
| <iterative do statement>
| <end statement>
| <procedure statement>

<label definition> ::= <identifier> :

## A.4.1 Basic Statements

### A.4.1.1 Assignment Statement

<assignment statement>::= <left part> = <expression> ;
<left part>::= <variable reference> [, <variable reference>] ...

### A.4.1.2 CALL Statement

<call statement>::= CALL <simple variable> [<parameter list>] ;
<parameter list>::= (<expression> [, <expression>] ... )
<simple variable>::= <identifier>
| <identifier> . <identifier>

### A.4.1.3 GOTO Statement

<goto statement>::= {GOTO | GO TO} <identifier> ;

### A.4.1.4 Null Statement

<null statement>::= ;

### A.4.1.5 RETURN Statement

<return statement>::= <typed return>
| <untyped return>

<typed return>::= RETURN <expression> ;
<untyped return>::= RETURN ;

### A.4.1.6 iAPX 86 Dependent Statements

<iAPX 86 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>
| <if condition> <true element> ELSE <false element>

<if condition>::= IF <expression> THEN
<true element>::= [<label definition> ...] <do block>
| [<label definition> ...] <basic statement>

<false element>::= <unit>
<true unit>::= <unit>

## A.4.4 DO Blocks

<do block>::= <simple do block>
| <do-case block>
| <do-while block>
| <iterative do block>

### A.4.4.1 Simple DO Blocks

<simple do block>::= <simple do statement>[<declaration>...][<unit>...]<ending>
<ending>::=[<label definition>...]<end statement>

### A.4.4.2 DO-CASE Blocks

<do-case block>::= <do-case statement> {<unit>...} <ending>

### A.4.4.3 DO-WHILE Blocks

<do-while block>::= <do-while statement> [<unit>...] <ending>

### A.4.4.4 Iterative DO Blocks

<iterative do block>::=<iterative do statement> [<unit>...] <ending>

# A.5 Expressions

## A.5.1 Primaries

```
<primary>::= <constant>
           | <variable reference>
           | <location reference>
           | <subexpression>

<subexpression>::= ( <expression> )
```

### A.5.1.1 Constants

```
<constant>::= <numeric constant>
            | <string>
```

### A.5.1.2 Variable References

```
<variable reference>::= <data reference>
                      | <function reference>

<data reference>::= <name>[<subscript>][<member specifier>]
<subscript>::= ( <expression> )
<member specifier>::= .<member name>[<subscript>]
<function reference>::= <name>[<actual parameters>]
<actual parameters>::= ( <expression>[,<expression>]...)
<member name>::= <identifier>
<name>::= <identifier>
```

### A.5.1.3 Location References

```
<location reference>::= {@|.}  <constant list>

                      | {@|.} <variable reference>

<constant list>::= (<constant>[,<constant>]...)
```

## A.5.2 Operators

```
<operator>::= <logical operator>
            | <relational operator>
            | <arithmetic operator>

<logical operator>::= AND
                    | OR
                    | NOT
                    | XOR

<relational operator>::= < | > | <= | >= | <> | =
<arithmetic operator>::= + | - | PLUS | MINUS | * | / | MOD
```

## A.5.3 Structure of Expressions

```
<expression>::= <logical expression>
              | <embedded assignment>
```

```
<embedded assignment>::= <variable reference> := <logical expression>
<logical expression>::= <logical factor>
                      | <logical expression> <or operator> <logical factor>

<or operator>::= OR
               | XOR

<logical factor>::= <logical secondary>
                  | <logical factor> <and operator> <logical secondary>

<and operator>::= AND
<logical secondary>::= [<not operator>] <logical primary>
<not operator>::= NOT
<logical primary>::= <arithmetic expression>
     [<relational operator> <arithmetic expression>]

<relational operator>::= < | > | <= | >= | <> | =

<arithmetic expression>::= <term>
                         | <arithmetic expression> <adding operator> <term>

<adding operator>::= + | - | PLUS | MINUS
<term>::= <secondary>
        | <term> <multiplying operator> <secondary>

<multiplying operator>::= * | / | MOD
<secondary>::=[<unary minus>] <primary>
              [<unary plus> ]

<unary minus>::= -
<unary plus>::= +
```

| NONTERMINALS | SECTION |
|---|---|
| 

# APPENDIX B
# PROGRAM CONSTRAINTS

Certain fixed size tables within the compiler constrain various features of a user program to certain maximums. These limits are summarized below:

*MAXIMUM:*

| | |
|---|---|
| Nesting of LITERALLY invocations | 5 |
| Nesting of INCLUDE controls | 5 |
| Number of nested procedures and DO cases | 7 |
| Number of labels on a statement | unlimited |
| Nesting of blocks | 18 |
| Number of nested typed procedures | 20 |
| Number of elements in a factored list | 32 |
| Number of members in a structure | 64 |
| Structure size | 64K |
| | |
| Numbers of characters in a line | 128 |
| Length of a string constant | 255 |
| Number of DO blocks in a procedure | 255 |
| Number of cases in a DO CASE block | 255 |
| Number of active cases | 255 |
| Number of EXTERNAL items | 255 |
| Number of procedures in a module | 253 |
| Segment Size | 64K |

**NOTE**

The PL/M-86 compiler has a symbol capacity of approximately 5000 symbols. Of these, 800 are held in memory when the compiler has a partition size of 96K. Any symbols over this amount will spill onto the workfiles disk, causing performance degradation. If another 64K of memory is added to the compiler's partition (either by adding more memory to the system or by increasing its share of available memory), a total of 2300 symbols will then be held in memory. For large programs or programs with a lot of symbols, providing the compiler with more memory to work in will improve its performance.

# APPENDIX C
# PL/M-86 RESERVED WORDS

These are the reserved words of PL/M-86. They may not be used as identifiers.

ADDRESS
AND
AT
BASED
BY
BYTE
CALL
CASE
CAUSEINTERRUPT
DATA
DECLARE
DISABLE
DO
DWORD
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
SELECTOR
STRUCTURE
THEN
TO
WHILE
WORD
XOR

# APPENDIX D
# PL/M-86 PREDECLARED IDENTIFIERS

These are the identifiers for the builtin procedures and predeclared variables. If one of these identifiers is declared in a DECLARE statement, the corresponding built-in procedure or predeclared variable becomes unavailable within the scope of the declaration.

ABS
BUILDPTR
CARRY
CMPB
CMPW
DEC
DOUBLE
FINDB
FINDRB
FINDRW
FINDW
FIX
FLOAT
HIGH
IABS
INPUT
INT
INTERRUPTPTR
INWORD
LAST
LOCKSET
LENGTH
LOW
MEMORY
MOVB
MOVE
MOVRB
MOVRW
MOVW
OFFSETOF
OUTPUT
OUTWORD
PARITY
ROL
ROR
SAL
SAR
SCL
SCR
SELECTOROF
SETB
SETINTERRUPT
SETW
SHL
SHR
SIGNED
SIZE
SKIPB
SKIPRB
SKIPRW
SKIPW
STACKBASE
STACKPTR
TIME
UNSIGN
XLAT
ZERO

# APPENDIX E
# PL/M-80 AND PL/M-86

## 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 iAPX 86 is supported by two new data types, POINTER and SELECTOR, for storage of iAPX 86 locations, and a new location reference operator, @.
- The set of built-in 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 seven: BYTE, WORD, DWORD, INTEGER, REAL, POINTER, and SELECTOR.

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 that operate correctly on an 8080 can be recompiled with the PL/M-86 compiler to produce code that will run on an iAPX 86. 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 that might occur as identifiers in a PL/M-80 source program are WORD, DWORD, INTEGER, REAL, POINTER, SELECTOR, and CAUSEINTERRUPT (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, affecting the types of expressions allowed in the AT attribute, the INITIAL and DATA initializations, and location references. See also the discussions of size controls and the dot and @ operators in this manual.

(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.)

# APPENDIX F
# ASCII CODES

| 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 |

# APPENDIX G
# PL/M-86 ADVANCED SEGMENTATION

## G.1 Basic Controls

The simplest way to compile PL/M-86 code is to use the segmentation controls outlined in Chapter 17. However, these controls may severely restrict some applications, since MEDIUM and LARGE programs may produce non-optimal code, and COMPACT and SMALL program modules are unable to call procedures or reference variables outside their 64K code and data areas.

This appendix describes segmentation control extensions you can use to gain more optimal code. Optimization is obtained by breaking large applications (greater than 64K of code, data, or both) into loosely-coupled subsystems (less than 64K each of code and data) whose variable and procedure references are largely self-contained. Each subsystem is a collection of tightly coupled, logically related modules that obey one of the specified models of segmentation. (Subsystems within a single program can use different segmentation models if appropriate.)

Segmentation control extensions are best used to break LARGE and MEDIUM programs into separate subsystems that have few references between them. In SMALL and COMPACT programs, these extensions are used to specify that a procedure or variable is outside the current module, requiring a "far" reference. (Use of these extensions also allows easier access to the one megabyte address space.)

## G.2 Long Calls and Far References

Occasionally programs compiled under SMALL or COMPACT must access data or procedures outside their limited address space. They do so using an EXPORTS list, which indicates that a "far" reference must be made to this data or procedure.

A symbol included in a subsystem's EXPORTS list must be a public symbol defined in one of the modules belonging to that subsystem. It is called an exported symbol and may be referenced by modules in other subsystems. (A public symbol defined within a subsystem, but not listed in its EXPORTS list, is called a domestic symbol. It may be referenced only by modules within the same subsystem.

Using the EXPORTS list, you can interract with operating systems or call shared routines that were loaded outside your program's 64K code space. It provides a means for generating long calls to procedures and simple far references to variables outside your 64K code or data address space. However, data references are assumed to be in separate segments, so segment register loading is not optimized. (If such optimization is required, see section G.3.)

This ability to generate long calls also means that operating systems and libraries need only supply one interface model—LARGE. To interface with these routines, you must provide the compiler with the information it needs to generate long calls.

The far extension takes the form:

**$LARGE(***subsystem name* **EXPORTS** *interface list***)**

where

*subsystem name* is the name of the library or other subsystem being referenced.

*interface list* is the list of code entry points or data items, separated by commas, which are accessible to other modules via far references.

This control may be used with any of the simple segmentation schemes described in Chapter 17 and has the following characteristics:

- All references to variables in the interface list are long.
- All references to procedures in the interface list are long calls.
- Because the named subsystem is LARGE, optimization of segment register loading cannot take place, since two given variables may not be contained in the same segment. (If this optimization is required, see section G.3.)
- The EXPORTS list can be kept small by tailoring it to the requirements of each referencing module. It can also be constructed as an INCLUDE file and made available to all modules.
- Only those items that are referenced within this module need to be named in the EXPORTS list. Those that are referenced must be declared and have the same syntax as an EXTERNAL declaration (see section 9.2). Those items that are named in the EXPORTS list but are not referenced need not be declared.

### Example

```
$LARGE(dq$work EXPORTS dq$attach,dq$open,dq$read,dq$close)
```

If this were included in a compilation that had used the SMALL control, all calls would be short except those to the exported routines (dq$attach, dq$open, etc.), which would be long.

Note that the segmentation control extensions may be continued over more than one control line. To do this, simply begin each line with a $ in the first column. (The control lines must be contiguous.)

The follow restrictions apply when using far references in the SMALL case:

1. POINTER functions and POINTER variables may not be exported to a SMALL module.
2. The location addresses of variables or procedures exported to a SMALL module may not be obtained using the @ or dot operators.
3. POINTER actual parameters passed to far procedures will always use the current contents of the DS register as the base portion of the long pointer.

## G.3 Subsystems

By carefully constructing subsystems, you can minimize references to outside resources (code and data) and, in return, receive highly optimized code for all internal resource references. This section describes how to create these subsystems.

A subsystem control takes the form:

$*model* (*subsystem name* [*submodel*] [HAS *id-list*;] [*exports list*])

where

*model* specifies the model of segmentation that the subsystem will follow. (COMPACT and LARGE are allowed, but only COMPACT subsystems provide optimized code.) This may be modified by the submodel specification.

*subsystem name* specifies a unique name for each subsystem and is any valid PL/M-86 identifier.

*submodel* specifies the placement of constants. It can be either:

-CONST IN CODE- provides for burning code and constants into ROM)
-CONST IN DATA- (the default case)

HAS *id-list* describes the mapping of module names to subsystems. This is optional for all subsystems except the one that includes the module currently being compiled. It takes the form:

HAS *module name*[,*module name*] . . .

*exports list* is the list of procedures and variables exported by this subsystem. Any procedure or variable not named in this list will be local to its subsystem. This takes the form:

EXPORTS *id*[,*id*]...

## Examples

1. Consider the following system definition:

```
$COMPACT(Par -CONST IN CODE- HAS A1, A2, A3)
$COMPACT(Quad HAS Q1, Q2, Q3, Q4; EXPORTS Middle)
$LARGE(Name EXPORTS Number, Store)
```

The sample program consists of three subsystems named Par, Quad, and Name. Par and Quad use the COMPACT model of segmentation, while Name uses the LARGE model. Constants are stored with the code in the Par and Name subsystems. The Par subsystem, consisting of the modules A1, A2, and A3, is apparently the main program since it exports no procedures.

2. A SMALL program wishing to communicate with the TEMPREAL CEL library might have the following controls:

```
$SMALL
$LARGE(CEL EXPORTS MQERSGN, MQERDIM, MQERANT, MQERNIN, MQERMOD;
$    EXPORTS MQEREXP, MQERLGE, MQERLGD, MQERSNH, MQERCSH, MQERTNH;
$    EXPORTS MQERSIN, MQERCOS, MQERTAN, MQERASN, MQERACS, MQERATN;
$    EXPORTS MQERY2X, MQERAT2, MQERRNT, MQERINT, MQERRMD)
```

The LARGE model is used to generate long calls to the named procedures. It also forces the compiler to make no assumptions about the sharing of code or data within the library.

3. Four subsystems (main_sub, read_sub, write_sub, sort_sub) have less than 64K each of code and data. The modules in main_sub contain the following controls:

```
$COMPACT(main_sub  HAS main1,main2)  EXPORTS buf,file,error$exit)
$COMPACT(read_sub  EXPORTS read)
$COMPACT(write_sub  EXPORTS write,consol)
$COMPACT(sort_sub  EXPORTS sort$buf,sorter)
```

Therefore, we know that the subsystems share the following resources:

| SUBSYSTEM NAME | EXPORTED ID | EXPORTED TYPE |
|---|---|---|
| main__sub: | buf | (variable) |
| | file | (variable) |
| | error$exit | (procedure) |
| read__sub: | read | (procedure) |
| write__sub: | write | (procedure) |
| | consol | (procedure) |
| sort__sub: | sort$buf | (variable) |
| | sorter | (procedure) |

Note from this example that:

- Because each subsystem has only one segment for data (a characteristic of the COMPACT model), references to data exported by another subsystem can be optimized.
- Each of the four subsystems follow the COMPACT model of segmentation. This will provide optimal code for references to data and procedures local to each subsystem.
- The actual declarations for the procedures and variables in the EXPORTS list take the form of normal PUBLICs and EXTERNALs. The EXTERNALs need not be declared unless they are referenced within the module.

Breaking an application that is currently using the LARGE model into loosely coupled COMPACT subsystems may reduce code size by 20-30%.

# APPENDIX H
# RUN-TIME PROCEDURE AND ASSEMBLY LANGUAGE LINKAGE

This chapter describes the handling at run time of non-interrupt procedures. Assembly-language subroutines that are to be linked with PL/M-86 programs or procedures must be compatible with these conventions. The easiest way to ensure compatibility is simply to write a dummy procedure in PL/M-86 with the same argument list as the desired assembly language subroutine and with the same attributes. Then compile the dummy procedure with the correct size control and with the CODE control specified. This will produce a pseudoassembly listing of the generated iAPX 86 code, which may then be simply copied as the prologue and epilogue of the assembly language subroutine. This having been done, an understanding of the material in this chapter is not needed.

For the handling of interrupt procedures, see Appendix I.

## H.1 Calling Sequence

For each procedure activation (CALL statement or function reference) in the source, the object code uses a *calling sequence*. The calling sequence places the procedure's actual parameters (if any) on the stack and then activates the procedure with a CALL instruction.

The parameters are placed on the stack in left-to-right order. Since the direction of stack growth is from higher locations to lower locations, this means that the first parameter occupies the highest position on the stack and the last parameter occupies the lowest position. Note that a BYTE parameter value occupies two bytes on the stack, with the value in the lower byte. The contents of the higher byte are undefined. A POINTER parameter value in the COMPACT, MEDIUM, and LARGE cases consists of a segment address and an offset. The 16-bit segment address is pushed first, and then the 16-bit offset is pushed.See Chapter 4 for details on data representations.

After the parameters are passed, the CALL instruction places the return address on the stack. In the SMALL and COMPACT cases, this is a 16-bit offset (the contents of the IP register) and occupies two contiguous bytes on the stack.

In the MEDIUM and LARGE cases, the type of the return address depends on whether the procedure is local or public. The return address for a local procedure, like any return address for the SMALL case, is a 16-bit offset and occupies two contiguous bytes on the stack. For a public procedure in the MEDIUM or LARGE case, and for procedures exported from COMPACT, the return address is a POINTER value consisting of a segment address and an offset and is passed in the same way as a POINTER parameter. The 16-bit segment address (contents of the CS register) is pushed first, and then the 16-bit offset (IP register contents) is pushed.

Control is then passed to the code of the procedure by updating the IP register. In MEDIUM and LARGE cases, and for procedures exported from COMPACT, the CS register is also updated.

At the point where the procedure gains control, then, the stack layout is as shown in figure H-1.

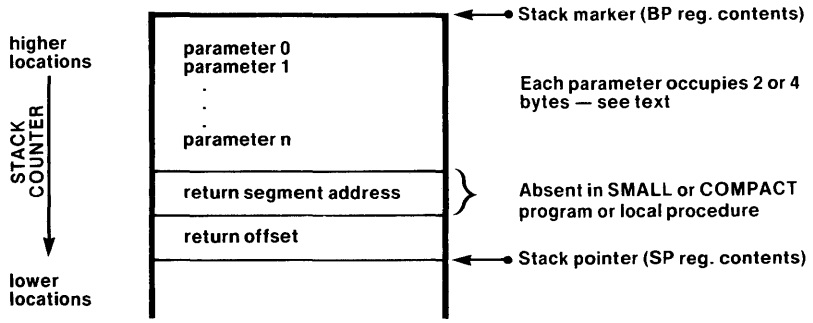



Figure H-1. Stack Layout at Point Where a Non-Interrupt Procedure is Activated

121636-6

## H.2 Procedure Prologue

In compiling the procedure itself, the compiler inserts at the beginning a sequence of code called the *prologue*. This code accomplishes the following steps:

1. If the procedure has the PUBLIC attribute and the program size is LARGE, or if it is exported from a COMPACT subsystem, the contents of the DS register are placed on the stack. Then the DS register is updated with a value that is found in the current code segment (i.e., the segment containing the procedure). (The DS register contains the segment address for the current data segment; thus this step implements the pairing of code and data segments in the LARGE case and is not needed in the SMALL, COMPACT, and MEDIUM cases because the data segment does not change.)

2. If any parameter of the procedure is referenced by a nested procedure, all parameters are removed from the stack and placed in space reserved for them in the data segment.

3. The stack marker offset (BP register contents) is placed on the stack, and the current stack pointer (SP register contents) is used to update the BP register.

4. If the procedure has the REENTRANT attribute, space is reserved on the stack for any variables declared within the procedure (this does not include based variables, variables with the DATA attribute, or variables with the AT attribute).

Control then passes to the code compiled from the executable statements in the procedure body. At this point, the stack layout is as shown in Figure H-2.

During execution of the procedure, further stack space may be used for temporary storage generated by the compiler.

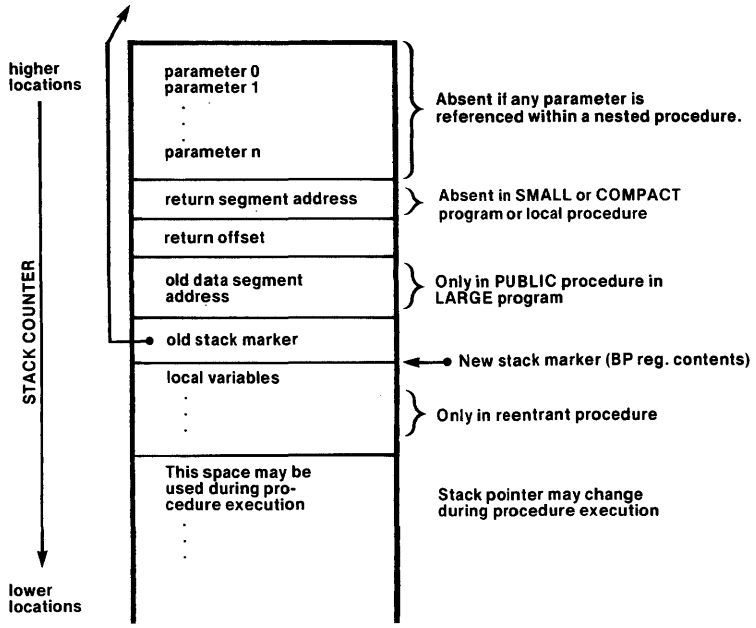



Figure H-2. Stack Layout During Execution of Non-Interrupt Procedure Body 121636-7

## H.3 Procedure Epilogue

To return from the procedure, the compiler inserts a code sequence called the *epilogue*. This accomplishes the following steps:

1. If the compiler has used stack locations for temporary storage or local variables during procedure execution, the stack pointer (SP register) is loaded with the stack marker (BP register contents). This has the effect of discarding the temporary storage.

2. The old stack marker is restored by popping the stored value from the stack into the BP register.

3. If the procedure has the PUBLIC attribute and the program size is LARGE or it is exported from a COMPACT subsystem, the old data segment address is restored by popping the stored value from the stack into the DS register.

4. A RET instruction is used to return from the procedure. If the program size is SMALL, the RET pops the stored return address (a 16-bit offset) into the IP register. It also discards any parameters stored on the stack.

If the program size is MEDIUM or LARGE and the procedure is local, the RET performs the same actions described above for a return in the SMALL or COMPACT case. If the program size is MEDIUM or LARGE and the procedure is public, the RET pops the stored return-address offset from the stack into the IP register and then pops the return-address segment address into the CS register. It also discards any parameters stored on the stack.

## H.4 Value Returned from Typed Procedure

The result of a typed procedure is returned as follows:

| Procedure Type | Result Returned in: |
|---|---|
| BYTE | AL Register |
| WORD | AX Register |
| DWORD | DX and AX Registers |
| INTEGER | AX Register |
| POINTER (SMALL size)* | BX Register |
| POINTER (COMPACT size) | ES and BX Registers |
| POINTER (MEDIUM size) | ES and BX Registers |
| POINTER (LARGE size) | ES and BX Registers |
| SELECTOR | AX Register |
| REAL | Top of RMU stack |

*Under the ROM option, the result is returned in ES and BX registers.

# APPENDIX I
# RUN-TIME INTERRUPT PROCESSING

## I.1 General

An interrupt is initiated when the CPU receives a signal on its "maskable interrupt" pin from some peripheral device or control is transferred to an interrupt vector by the CAUSE$INTERRUPT statement. (If your program runs under an operating system that traps interrupts, you do not need the information in this appendix.)

Note that the CPU does not respond to this signal unless interrupts are enabled. The "main program prologue" (code inserted by the compiler at the beginning of the main program) enables interrupts.

**NOTE**

If you require your program to begin with interrupts disabled, simply start with the instruction DISABLE;. Since the iAPX 86 processor does not actually allow an interrupt to occur until the first machine instruction following the enabling instruction has been processed, the resulting code sequence will not allow any maskable interrupts to occur.

If interrupts are enabled, the following actions take place:

1. The CPU completes any instruction currently in execution.
2. The CPU issues an "acknowledge interrupt" signal and waits for the interrupting device to send an interrupt number.
3. The CPU flag registers are placed on the stack (occupying two bytes of stack storage).
4. Interrupts are disabled by clearing the IF flag.
5. Single stepping is disabled by clearing the TF flag.
6. The CPU activates the interrupt procedure corresponding to the interrupt number sent by the interrupting device.
7. When that procedure terminates, the stack is automatically restored to its state when the interrupt was received, and control returns to the point where it was interrupted.

The mechanism for this activation and restoration are described below.

## I.2 The Interrupt Vector

If the NOINTVECTOR control is not used, an interrupt vector entry is automatically generated by the compiler for each interrupt procedure. Collectively, the interrupt vector entries form the *interrupt vector*. If NOINTVECTOR is used, the programmer must supply the interrupt vector as explained below in section I.4.

The interrupt vector is an absolutely located array of POINTER values beginning at location 0. Thus the *n*th entry is at location 4**n* and contains the location of a procedure declared with the INTERRUPT *n* attribute.

Note that the first and second bytes of each entry contain an offset, while the second two bytes contain a segment address. The entries are always four-byte pointers, and the segment address is always used in transferring to the interrupt procedure, even if the program size is SMALL.

The CPU uses the interrupt vector entry to make a long indirect call to activate the appropriate procedure. At this point, the current code segment address (CS register contents) and instruction offset (IP register contents) are placed on the stack.

At the point where the procedure is activated, the stack layout is as shown in figure I-1.

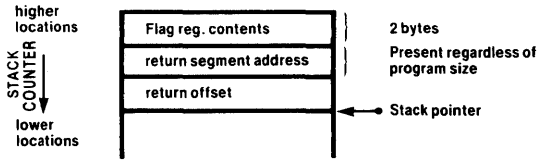



Figure I-1. Stack Layout at Point Where an Interrupt Procedure Gains Control 121636-8

## I.3 Interrupt Procedure Preface

At the beginning of each interrupt procedure, before the prologue described in the preceding chapter, the compiler inserts an *interrupt procedure preface* that accomplishes the following steps:

1. Push the ES register contents onto the stack.
2. Push the DS register contents onto the stack.
3. Load the DS register with a new data segment address taken from the current code segment (i.e., the segment containing the interrupt procedure).
4. Push the AX register contents onto the stack.
5. Push the CX register contents onto the stack.
6. Push the DX register contents onto the stack.
7. Push the BX register contents onto the stack.
8. Push the SI register contents onto the stack.
9. Push the DI register contents onto the stack.
10. At this point, a CALL instruction transfers control to the procedure prologue (described in Appendix H).

NOTE

The compiler may make temporary use of the DS register in some cases (e.g., string built-ins), but always restores it. Care must be taken when writing your own interrupt procedure in assembly language to note this possibility.

At the point where the procedure prologue gains control, the stack layout is as shown in figure I-2.

After the procedure prologue is executed, at the point where the code compiled from the procedure body gains control, the stack layout is as shown in figure I-3.

The return from the procedure body transfers control back into the interrupt procedure preface. At this point the procedure epilogue (see Appendix H) has restored the stack to the layout of figure I-2. The interrupt procedure preface continues with the following steps:

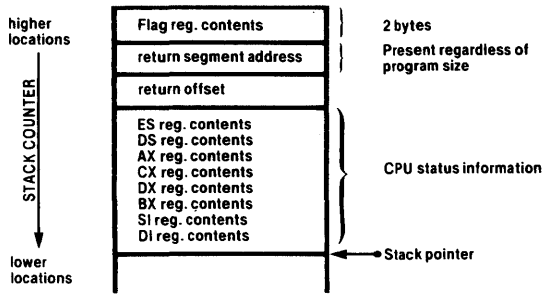



Figure I-2. Stack Layout After Interrupt Procedure Preface and Before Procedure Prologue

121636-9

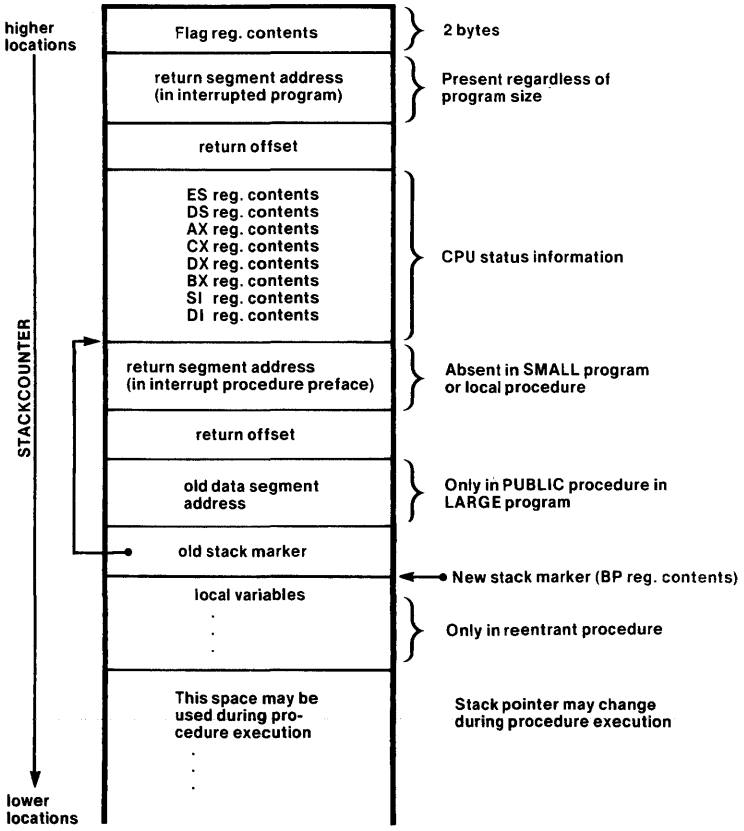



Figure I-3. Stack Layout During Execution of Interrupt Procedure Body

121636-10

11. Pop the stack into the DI register.
12. Pop the stack into the SI register.
13. Pop the stack into the BX register.
14. Pop the stack into the DX register.
15. Pop the stack into the CX register.
16. Pop the stack into the AX register.
17. Pop the stack into the DS register.
18. Pop the stack into the ES register.
19. Execute an IRET instruction to return from the interrupt procedure. This restores the IP, CS, and flag register contents from the stack.

At this point the stack has been restored to the state it was in before the interrupt occurred, and processing continues normally.

## I.4 Writing Interrupt Vectors Separately

In some cases it may be desirable to write the interrupt vector separately (in PL/M-86 or assembly language). This can be done by using NOINTVECTOR to prevent generation of an interrupt vector by the compiler. The separately created interrupt vector can then be linked into the program.

Creation of a separate explicit interrupt vector requires some care. The @ operator in PL/M-86 provides access to a procedure's normal (i.e., called) entry point, not to its interrupt entry point. The interrupt entry point first saves the status of the interrupted program before invoking the interrupt procedure through its normal entry point. The exact length of these operations depends on the compilation options chosen, the attributes of the interrupt procedure, and the version of the compiler being used. The built-in function INTERRUPT$PTR can be used during execution to return the actual interrupt entry point. Discussion of this function appears in Chapter 11.

The usefulness of a separately created interrupt vector can be seen by considering an example.

Suppose that two modules for a multimodule program are developed separately. Both use interrupt procedures, but at the time when the modules are written the assignment of interrupt numbers to the various interrupt procedures has not been determined.

The two modules are therefore compiled with the NOINTVECTOR control. When this is done, the *n* in an INTERRUPT *n* attribute is ignored—since normally it would only be used to put the procedure's entry in the proper location within the interrupt vector.

Later, when the program is linked together, a separately created interrupt vector can be linked in. Within this interrupt vector, the placement of the entry for a given interrupt procedure determines which interrupt number will activate that procedure.

Similarly, you could have a library of interrupt procedures, all compiled with NOINTVECTOR. Any program could then have any of these procedures linked in, with a separately created interrupt vector.

The built-in procedure SET$INTERRUPT can be used during execution to create the correct interrupt vector for each interrupt routine. This procedure is discussed in Chapter 11.

# APPENDIX J
# COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR SERIES III USERS

This appendix contains information that is specific to the Intellec Series III Microcomputer Development System. It covers the following areas:

- Compiler invocation and file usage
- Examples of system-dependent floating-point library linkage
- Examples of system-dependent compiler controls
- Related publications

## J.1 Compiler Invocation

The compiler is supplied on a diskette that does not contain an operating system or relocation software. It may be desirable to copy the compiler to another disk (such as a system disk). The Compiler consists of a single file: PLM86.86.

The following example illustrates the normal sequence of operations used to compile a PL/M-86 program from system bootstrap to the end of compilation. The steps involved are as follows:

1. Power up the Intellec hardware.
2. Insert a system disk into drive 0. In this example, the system disk contains the compiler.
3. Insert a nonsystem disk into drive 1. In this example, this disk contains a PL/M-86 source file to be compiled.
4. Bootstrap the Operating System (full instructions appear in the *Series III Console Operating Instructions*).
5. Compile the program with the PL/M-86 Compiler. After compilation, the program may be linked and relocated.

The PL/M-86 Compiler is invoked from the system console using the standard command format described in the *Series III Console Operating Instructions*. Continuation lines can be specified by using the ampersand (&) as a continuation character. The ampersand can be used any place there is a space or other delimiter.

The invocation command has the general form:

**RUN** *directory-name* **PLM86** *source-file* [*control*]

where

*directory-name* identifies which device contains the compiler disk. In the Series III operating environment, this takes the form :F*n*:, where *n* is the disk drive number. Directory-name may be omitted if the compiler is in Drive 0.

*source-file* is the name of the file containing the PL/M-86 source module.

*controls* is an optional sequence of compiler controls. (See Chapter 15.)

In the interactive sequence shown in figure J-1, comments appearing to the right of semicolons are for clarification, not material entered by the user. This example shows how to compile a complete program that does not require more than 64K bytes of storage for the code or more than 64K bytes for data.

```
ISIS-II V4.1                          ; the system identifies itself

-RUN PLM86 :F1:MYPROG.SRC             ; the compiler is invoked

SERIES-III PL/M-86 COMPILER V2.0
PL/M-86 COMPILATION COMPLETE.     0 WARNINGS,    0 ERRORS

                        ; the program may now be linked and relocated
```

Figure J-1. Interactive Compilation Sequence

In the normal usage of the PL/M-86 Compiler the compilation listing is written by default to a disk file on the same disk as the source file. This file has the same name as a source file, but has the extension LST. Thus, in the example above, the listing is found in :F1:MYPROG.LST. Similarly, the object code file is on the same disk and has the same file name, but has the extension OBJ. In the example :F1:MYPROG.OBJ contains the object code produced by compiling :F1:MYPROG.SRC.

Examples

1. `RUN PLM86 :F1:PROG1.SRC`

   In this example, the operating system and the compiler are in drive 0. The compiler is directed to compile the source module on :F1:PROG1.SRC. This file resides on the disk in drive 1 and has the name PROG1.SRC.

2. `RUN :F1:PLM86 :F1:MYPROG.SRC PRINT(:LP:)TITLE('TEST PROGRAM 4')`

   In this example, the compiler disk is in drive 1 and the operating system is in drive 0. The compiler is directed to compile the source module on :F1:MYPROG.SRC, directing all printed output to :LP:, and placing 'TEST PROGRAM 4' in the header on each page of the listing.

## J.2 File Usage

### Input Files

The compiler reads the PL/M-86 source from the source-file specified on the command line (see previous section) and also from any files specified with INCLUDE controls (as described Chapter 15). These files must be standard Series III disk files. The source input should contain a PL/M-86 source module.

### Output Files

Two output files are produced during each compilation unless specific controls are used to suppress them. These are the listing and object code files. Each of these may be explicitly directed to some standard Series III pathname (device or file) by using the PRINT and OBJECT controls respectively. If the user does not control these outputs explicitly, the compiler writes them to disk files on the disk containing the input file. These files have the same file name as the input file, but have the extensions LST for the listing and OBJ for the object code. For example, if the compiler is invoked by:

```
RUN PLM86 :F1:MYPROG.SRC
```

the listing and all other printed output is written to :F1:MYPROG.LST and the object code to :F1:MYPROG.OBJ. If these files already exist they are overwritten. If they do not exist the compiler creates them.

The object code file may be used as input to the ISIS-II relocation and linkage facilities. (See the *iAPX 86,88 Family Utilities User's Guide*.)

### Compiler Work Files

The compiler uses work files during its operation which are deleted at the completion of compilation. All of these files are on the device :WORK: unless the WORKFILES control is used to specify another device.

All of the work files have names with the extension TMP. Therefore, you should avoid naming files with the extension TMP on any device used by the compiler for work files, as there is a possibility that they will be destroyed by the operation of the compiler.

## J.3 Linking to Floating-Point Libraries with the Series III

Suppose you write a PL/M-86 program called EASY that, at first, uses no REAL math at all. No interface library is needed. As modules are added during the development process, you supply a PL/M-86 REAL math routine called ACURAT, and you revise EASY to call it.

If you have no 8087 chip installed in your system, the correct linking statement for the above conditions would be:

```
LINK86 ACURAT.OBJ, EASY.OBJ, E8087.LIB, PE8087
```

However, if ACURAT were written in some other language such as FORTRAN86 or ASM86, the following command should be used instead:

```
LINK86 ACURAT.OBJ, EASY.OBJ, E8087.LIB, E8087
```

If you DO have an actual 8087 chip installed in your system, then the two examples above should become:

```
LINK86 ACURAT.OBJ, EASY.OBJ, 8087.LIB
```

More detailed and advanced discussions of the features and functions of the iAPX 86 utilities appear in the manual titled *iAPX 86,88 Family Utilities User's Guide.*

## J.4 Series III-Specific Compiler Controls

This appendix includes a fold-out page for the system-specific examples of several compiler controls. This page is designed to be opened out and used in conjunction with the corresponding text in Chapter 15.

## J.5 Related Publications

Below is a list of other Intel publications you might need along with this manual. The manual order number for each publication is given immediately following the title. The paragraph below each title describes the book.

*Intellec Series III Product Overview,* 121575

A summary description of the set of manuals that describe the Intellec Series III development system and its supporting hardware and software. This short manual includes a description of each manual related to the Series III, plus a glossary of terms used in the manuals.

*Intellec Series III Console Operating Instructions,* 121609
*Intellec Series III Pocket Reference,* 121610

Instructions for using the console features of the Series III, including the resident monitor. The *Console Operating Instructions* provides complete instructions, and the *Pocket Reference* gives a summary of this information.

*Intellec Series III Programmer Reference Manual,* 121618

Instructions for calling system routines from user programs for both microprocessor environments (8080/8085 and 8086) in the Series III.

*ISIS-II CREDIT (CRT-Based Text Editor) User's Guide,* 9800902
*ISIS-II CREDIT (CRT-Based Text Editor) Pocket Reference,* 9800903

Instructions for using CREDIT, the CRT-based text editor supplied with the Series III. The *User's Guide* provides complete operating instructions, and the *Pocket Reference* summarizes this information for quick reference.

*iAPX 86,88 Family Utilities User's Guide,* 121616

Instructions for using the utility programs LINK86, LIB86, LOC86, CREF86, and OH86 in iAPX 86-based environments to prepare compiled or assembled programs for execution.

*8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems,* 121627
*8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems,* 121628
*8086/8087/8088 Macro Assembly Language Pocket Reference,* 121674

Instructions for using the 8086/8087/8088 macro assembly language and its assembler in iAPX 86-based environments. The *Language Reference Manual* gives a complete description of the assembly language the *Operating Instructions* gives complete instructions for operating the assembler and the *Pocket Reference* provides summary information for quick reference. You need these publications if you are coding some of your routines in assembly language.

*ICE-86 In-Circuit Emulator Operating Instructions for ISIS-II Users,* 9800714
*ICE-86 Pocket Reference,* 9800838
*ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users,* 9800949
*ICE-88 Pocket Reference,* 9800950

Instructions for using the ICE-86 and ICE-88 In-Circuit Emulators for hardware and software development. The *Operating Instructions* manuals give complete user descriptions of the In-Circuit Emulators, and the *Pocket Reference* guides provide summary information for quick reference. You need the corresponding publications if you are using the ICE-86 or ICE-88 emulator.

| Control | Examples |
| --- | --- |
| **WORKFILES** | `WORKFILES(:F0:,:F1:)` |
| | `WORKFILES(:F0:,:F0:)` |
| **OBJECT** | `OBJECT(:F1:OTHER.OBJ)` |
| **PRINT** | `PRINT(:LP:)` |
| **INCLUDE** | `INCLUDE(:F1:SYSLIB.SRC)` |

# INDEX

# REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

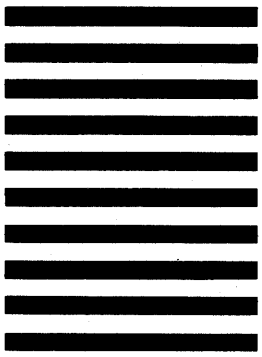
CITY _____ STATE _____ ZIP CODE _____
(COUNTRY)

Please check here if you require a written reply. ☐

**WE'D LIKE YOUR COMMENTS ...**

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.