ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL

Manual Order No.: 9800478A

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

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

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

Printed in U.S.A. A56/479/3K TP

PREFACE

This manual describes the operation of the PL/M-86 Compiler, Version 1.0. The compiler accepts PL/M-86 source as input and produces relocatable 8086 object code as output. The compiler runs under the ISIS-II operating system which supports relocation and linkage of object code programs. The manual is one of a series of documents describing this system and its operation.

This manual assumes that the reader is conversant with PL/M-86, is familiar with the ISIS-II operating system, and knows how to operate the Intellec Microcomputer Development System hardware. The reader is referred to the following Intel publications to gain such familiarity:

- *PL/M-86 Programming Manual* 98-466
- *ISIS-II System User's Guide* 98-306
- *Intellec Microcomputer Development System Operator's Manual* 98-129

The compiler requires the following software and hardware environments for proper execution:

Software

For the Compiler

- ISIS-II Operating System

For Object Programs

- QRL86 or LINK86 and LOC86
- PL/M-86 library file, PLM86.LIB

For Intermodule Cross-Reference Listings

- IXREF Program (if intermodule cross-reference listing is desired)

Hardware

- 8080 Intellec Microcomputer Development System
- 64K bytes of RAM memory (includes space required for ISIS-II)
- An ISIS-supported direct access device and controller (such as diskette drive)
- Console device (TTY or CRT)

CONTENTS

ILLUSTRATIONS & TABLES

CHAPTER 1
HOW TO USE THE PL/M-86 COMPILER

This chapter presents all of the information necessary to begin using the PL/M-86 Compiler. It is not necessary to be familiar with all the features described in the rest of this manual in order to make effective use of the compiler. If you are a beginning user you are particularly encouraged to start using the compiler and to gain experience with PL/M-86 before concerning yourself with special features. The example included in this chapter can be entered exactly as shown to get a feel for the procedures involved in using the compiler.

The compiler is supplied on a diskette which does not contain an operating system or relocation software. It may be desirable to copy the compiler to another diskette (such as a system diskette). Section 2.2.4 lists the files that contain the code of the compiler.

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 diskette into Drive 0. In this example, the system diskette contains the compiler.
3. Insert a nonsystem diskette into Drive 1. In this example, this diskette contains a PL/M-86 source file to be compiled.
4. Bootstrap the ISIS-II Operating System.
5. Compile the program with the PL/M-86 Compiler. After compilation, the program may be linked and relocated.

Refer to the *ISIS-II System Users Guide* for detailed instructions for all of these steps with the exception of compiling your program. This manual describes program compilation.

In the interactive sequence shown in Figure 1-1, underlined text is output by the system, all other text is typed by the user. 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 V3.1                          ;the system identifies itself

-PLM86 :F1:MYPROG.SRC                 ;the compiler is invoked

ISIS-II PL/M-86 COMPILER V1.0
PL/M-86 COMPILATION COMPLETE.      0 PROGRAM ERRORS

                    ;the program may now be linked and relocated
```

Figure 1-1. Interactive Compilation Sequence

In the normal usage of the PL/M-86 Compiler the compilation listing is written by default to a diskette file on the same diskette as the source file. This file has the same name as the 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 diskette 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.

A detailed explanation of all of the steps used in the example, with the exception of the command that invokes the PL/M-86 Compiler, may be found in the *ISIS-II System Users Guide*. See also Appendix A of this manual for an explanation of when the PLM86.LIB file should be linked in.

The normal method of invoking the compiler, when no special actions are needed, is simply to give its name (PLM86) and the name of your source file. The source file must be on a diskette and must contain a PL/M-86 source module. This command has the form

```
PLM86 source-file
```

if the compiler is in Drive 0.

In this chapter everything necessary to use the compiler in its simplest mode of operation has been shown, and you need not read the remainder of this manual unless you need additional features. The remaining chapters of the manual provide a detailed description of all available compiler features.

CHAPTER 2
COMPILER INVOCATION AND FILE USAGE

Throughout this manual, the following conventions are used in describing the commands and controls associated with the compiler:

- Upper-case letters (and numerals) represent text that must be entered as shown in the description (however, you may enter these items in lower-case).
- Lower-case letters are used to represent variable parts of the command or control.
- Square brackets [ ] are used to enclose parts of the command or control that may be omitted (the brackets themselves are not part of the command or control).

The following discussions assume that the ISIS-II system has been bootstrapped. A diskette containing the PL/M-86 Compiler must be mounted in one of the diskette drives. (Note that a system diskette must be mounted in Drive 0.)

2.1 Compiler Invocation

The PL/M-86 Compiler is invoked from the ISIS-II console using the standard command format described in the *ISIS-II System User's Guide*. Continuation line can be specified by using the amperand (&) as a continuation character. The amperand can be used any place there is a space or other delimiter.

The invocation command has the general form

```
[:device:]PLM86 source-file [controls]
```

where

- *device* identifies which drive contains the compiler diskette. This may be omitted if the compiler diskette 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. The use of these controls is described in Chapter 3.

Examples:

1. PLM86 :F1:PROG1.SRC

   The compiler is directed to compile the source module on :F1:PROG1.SRC. This file resides on the diskette in Drive 1 and has the name PROG1.SRC.

2. :F1:PLM86 :F1:MYPROG.SRC PRINT(:LP:) TITLE('TEST PROGRAM #4')

   In this example, the compiler diskette is in Drive 1. 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.

2.2 File Usage

2.2.1 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 (see Section 3.7). These files must be standard ISIS-II diskette files. The source input should contain a PL/M-86 source module.

2.2.2 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 ISIS-II 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 diskette 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

```
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.

2.2.3 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 diskette drive 1 unless the WORKFILES control (see Section 3.6) 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.

2.2.4 Compiler Code Files

The compiler's object code resides in eight diskette files. These files must be present for proper execution of the compiler:

```
PLM86
PLM86.OV0
PLM86.OV1
PLM86.OV2
PLM86.OV3
PLM86.OV4
PLM86.OV5
PLM86.OV6
```

The diskette containing these files may be mounted in any diskette drive—not necessarily Drive 0.

CHAPTER 3
COMPILER CONTROL LANGUAGE

3.1 Introduction to Compiler Controls

The exact operation of the compiler may be controlled by a number of *controls* which 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 ISIS-II 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 a 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 on 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 few will be needed for most compilations as a set of defaults is built into the compiler. The controls are summarized in Table 3-1.

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.OBJ)
```

Table 3-1: Compiler Controls

Primary Control Names	Default
PRINT / NOPRINT	PRINT(source-file.LST)
OBJECT / NOOBJECT	OBJECT(source-file.OBJ)
SYMBOLS / NOSYMBOLS	NOSYMBOLS
XREF / NOXREF	NOXREF
IXREF / NOIXREF	NOIXREF
PAGING / NOPAGING	PAGING
DEBUG / NODEBUG	NODEBUG
OPTIMIZE	OPTIMIZE(1)
DATE	no date
TITLE	module name
PAGEWIDTH	PAGEWIDTH(120)
PAGELENGTH	PAGELENGTH(60)
INTVECTOR / NOINTVECTOR	INTVECTOR
WORKFILES	WORKFILES(:F1:,:F1:)
SMALL / MEDIUM / LARGE	SMALL
General Control Names	**Default**
LIST / NOLIST	LIST
CODE / NOCODE	NOCODE
EJECT	–
INCLUDE	–
LEFTMARGIN	LEFTMARGIN(1)
OVERFLOW / NOOVERFLOW	NOOVERFLOW
SET / RESET	–
IF / ELSEIF / ELSE / ENDIF	–
SAVE / RESTORE	–
COND / NOCOND	COND
SUBTITLE	no subtitle

3.2 Listing Selection Controls

These controls determine what types of listings are to be produced and on which device they are to appear. The controls are:

PRINT / NOPRINT
LIST / NOLIST
CODE / NOCODE
XREF / NOXREF
IXREF / NOIXREF
SYMBOLS / NOSYMBOLS

3.2.1 PRINT / NOPRINT

These are primary controls. They have the form:

PRINT[(pathname)]

NOPRINT

Default: PRINT(source-file.LST)

The PRINT control specifies that printed output is to be produced. *Pathname* is a standard ISIS-II pathname which specifies the file or device to receive the printed output. Any output-type device, including a disk file, may be given. If the control is absent, or if a PRINT control appears without a pathname, printed output is directed to the same device used for source input and the output file has the same name as the source file but with the extension LST.

Example: PRINT(:LP:)

This causes printed output to be directed to the line printer.

The NOPRINT control specifies that no printed output is to be produced, even if implied by other listing controls such as LIST and CODE.

3.2.2 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.

3.2.3 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 interleaved with 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.

3.2.4 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.

3.2.5 IXREF / NOIXREF

These are primary controls. They have the form:

IXREF[(pathname)]

NOIXREF

Default: NOIXREF

The IXREF control causes an "intermediate intermodule cross-reference file" to be produced and written out to the file specified by the pathname. If no pathname is supplied, the file will be written on the same device used for source input and will have the same name as the source file but with the extension IXI.

The intermediate file contains all PUBLIC and EXTERNAL identifiers declared in the module being compiled, together with their types, dimensions, and attributes.

After compilation, the IXREF *program* (which is independent of the compiler) can be used to merge two or more of these intermediate files to produce an intermodule cross-reference listing, as explained in Appendix C.

The NOIXREF control suppresses the production of the intermediate file.

3.2.6 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.

3.3 Listing Format Controls

These controls determine the format of the listing output of the compiler. The controls are:

PAGING / NOPAGING
PAGELENGTH
PAGEWIDTH
DATE
TITLE
SUBTITLE
EJECT

3.3.1 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 and/or date.

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.

3.3.2 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 a page.

The minimum value for *length* is 5.

3.3.3 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.

3.3.4 DATE

This is a primary control. It has the form:

DATE(date)

Default: no date

where *date* is any sequence of nine or fewer characters not containing parentheses.

The *date* appears in the heading of all pages of listing output exactly as given in the DATE control.

Example: DATE(25 NOV 78)

3.3.5 TITLE

This is a primary control. It has the form:

```
TITLE('title')

Default: module name
```

where *title* is a sequence of printable ASCII characters which 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.

Example: `TITLE('TEST PROGRAM 4')`

3.3.6 SUBTITLE

This is a general control. It has the form:

```
SUBTITLE('subtitle')

Default: no subtitle
```

where *subtitle* is a sequence of printable ASCII characters which 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.

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.

3.3.7 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 line printed (following the page heading) on the new page.

If the NOPRINT, NOLIST or NOPAGING controls are in effect, the EJECT control is ignored.

3.4 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.

3.5 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:

INTVECTOR / NOINTVECTOR
OVERFLOW / NOOVERFLOW
OPTIMIZE
OBJECT / NOOBJECT
DEBUG / NODEBUG

3.5.1 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 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 Chapter 10.

3.5.2 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.

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

3.5.3 OPTIMIZE

This is a primary control. It has the form:

```
OPTIMIZE (n)

Default: OPTIMIZE (1)
```

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

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

OPTIMIZE(0) specifies that no optimization is to be performed.

OPTIMIZE(1) specifies "folding" of constant expressions, strength reduction, and elimination of common subexpressions.

OPTIMIZE(2) specifies all of the optimizations performed under OPTIMIZE(1), plus another class of optimizations including short jump optimization and static and dynamic peephole optimizations. An example program showing the code produced under each of these controls is shown in Figure 3-1 for OPTIMIZE(0), Figure 3-2 for OPTIMIZE(1), and Figure 3-3 for OPTIMIZE(2).

3.5.4 OBJECT / NOOBJECT

These are primary controls. They have the form:

```
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 ISIS-II pathname which 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 device and file name as used for source input, but with the extension OBJ.

Example: OBJECT(:F1:OTHER.OBJ)

This would cause the object code to be written to the file :F1:OTHER.OBJ.

The NOOBJECT control specifies that an object module is not to be produced.

```
PL/M-86 COMPILER    EXAMPLE

ISIS-II PL/M-86 DEBUG X012 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:EX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:EX.P86 CODE OPTIMIZE(0)


  1          EXAMPLE: DO;
  2   1        DECLARE (A,B,C) WORD, D(100) WORD, L LITERALLY '5';
  3   1        DO WHILE D(A+B) < D(A+B+1);
                                                  ; STATEMENT # 3
          0002  FA                CLI
          0003  2E8E160000        MOV     SS,CS:@@STACK$FRAME
          0008  BC0000            MOV     SP,@@STACK$OFFSET
          000B  8BEC              MOV     BP,SP
          000D  16                PUSH    SS
          000E  1F                POP     DS
          000F  FB                STI
                          @3:
          0010  8B1E0200          MOV     BX,B
          0014  031E0000          ADD     BX,A
          0018  D1E3              SHL     BX,1
          001A  8B360200          MOV     SI,B
          001E  03360000          ADD     SI,A
          0022  D1E6              SHL     SI,1
          0024  8B870600          MOV     AX,D[BX]
          0028  3B840800          CMP     AX,D[SI+2H]
          002C  7203              JB      $+5H
          002E  E91B00            JMP     @4
  4   2          D(C) = D(C) + 1;
                                                  ; STATEMENT # 4
          0031  8B1E0400          MOV     BX,C
          0035  D1E3              SHL     BX,1
          0037  8B870600          MOV     AX,D[BX]
          003B  81C00100          ADD     AX,1H
          003F  8B1E0400          MOV     BX,C
          0043  D1E3              SHL     BX,1
          0045  89870600          MOV     D[BX],AX
  5   2        END;
                                                  ; STATEMENT # 5
          0049  E9C4FF            JMP     @3
                          @4:
  6   1        IF A < B + (L - 1)
                                                  ; STATEMENT # 6
          004C  8B060200          MOV     AX,B
          0050  81C00400          ADD     AX,4H
          0054  39060000          CMP     A,AX
          0058  7203              JB      $+5H
          005A  E90D00            JMP     @1
                 THEN A = A * 2;
                                                  ; STATEMENT # 7
          005D  8B060000          MOV     AX,A
          0061  D1E0              SHL     AX,1
          0063  89060000          MOV     A,AX
          0067  E90C00            JMP     @2
                          @1:
  8   1          ELSE A = A + 1;
                                                  ; STATEMENT # 8
          006A  8B060000          MOV     AX,A
          006E  81C00100          ADD     AX,1H
          0072  89060000          MOV     A,AX
                          @2:
  9   1      END EXAMPLE;
                                                  ; STATEMENT # 9
          0076  FB                STI
          0077  F4                HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 0078H    120D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00CEH    206D
     MAXIMUM STACK SIZE = 0000H      0D
     9 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-1. Sample Program Showing the OPTIMIZE (0) Control

```
PL/M-86 COMPILER    EXAMPLE

ISIS-II PL/M-86 DEBUG X012 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:EX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:EX.P86 CODE OPTIMIZE(1)


   1          EXAMPLE: DO;
   2   1        DECLARE (A,B,C) WORD, D(100) WORD, L LITERALLY '5';
   3   1        DO WHILE D(A+B) < D(A+B+1);
                                                    ; STATEMENT # 3
            0002  FA              CLI
            0003  2E8E160000      MOV     SS,CS:@@STACK$FRAME
            0008  BC0000          MOV     SP,@@STACK$OFFSET
            000B  8BEC            MOV     BP,SP
            000D  16              PUSH    SS
            000E  1F              POP     DS
            000F  FB              STI
                            @3:
            0010  8B1E0200        MOV     BX,B
            0014  031E0000        ADD     BX,A
            0018  D1E3            SHL     BX,1
            001A  8B870600        MOV     AX,D[BX]
            001E  3B870800        CMP     AX,D[BX+2H]
            0022  7203            JB      $+5H
            0024  E90F00          JMP     @4
   4   2          D(C) = D(C) + 1;
                                                    ; STATEMENT # 4
            0027  8B1E0400        MOV     BX,C
            002B  D1E3            SHL     BX,1
            002D  818706000100    ADD     D[BX],1H
   5   2        END;
                                                    ; STATEMENT # 5
            0033  E9DAFF          JMP     @3
                            @4:
   6   1        IF A < B + (L - 1)
                                                    ; STATEMENT # 6
            0036  8B060200        MOV     AX,B
            003A  81C00400        ADD     AX,4H
            003E  39060000        CMP     A,AX
            0042  7203            JB      $+5H
            0044  E90700          JMP     @1
                   THEN A = A * 2;
                                                    ; STATEMENT # 7
            0047  D1260000        SHL     A,1
            004B  E90600          JMP     @2
                            @1:
   8   1          ELSE A = A + 1;
                                                    ; STATEMENT # 8
            004E  810600000100    ADD     A,1H
                            @2:
   9   1      END EXAMPLE;
                                                    ; STATEMENT # 9
            0054  FB              STI
            0055  F4              HLT

MODULE INFORMATION:

     CODE AREA SIZE     = 0056H     86D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00CEH    206D
     MAXIMUM STACK SIZE = 0000H      0D
     9 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-2. Sample Program Showing the OPTIMIZE (1) Control

```
PL/M-86 COMPILER    EXAMPLE

ISIS-II PL/M-86 DEBUG X012 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:EX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:FX.P86 CODE OPTIMIZE(2)


   1          EXAMPLE: DO;
   2    1       DECLARE (A,B,C) WORD, D(100) WORD, L LITERALLY '5';
   3    1       DO WHILE D(A+B) < D(A+B+1);
                                                    ; STATEMENT # 3
            0002  FA              CLI
            0003  2E8E160000      MOV     SS,CS:@@STACK$FRAME
            0008  BC0000          MOV     SP,@@STACK$OFFSET
            000B  8BEC            MOV     BP,SP
            000D  16              PUSH    SS
            000E  1F              POP     DS
            000F  FB              STI
                        @3:
            0010  8B1E0200        MOV     BX,B
            0014  031E0000        ADD     BX,A
            0018  D1E3            SHL     BX,1
            001A  8B870600        MOV     AX,D[BX]
            001E  3B870800        CMP     AX,D[BX+2H]
            0022  730D            JNB     @4
   4    2         D(C) = D(C) + 1;
                                                    ; STATEMENT # 4
            0024  8B1E0400        MOV     BX,C
            0028  D1E3            SHL     BX,1
            002A  8387060001      ADD     D[BX],1H
   5    2       END;
                                                    ; STATEMENT # 5
            002F  EBDF            JMP     @3
                        @4:
   6    1       IF A < B + (L - 1)
                                                    ; STATEMENT # 6
            0031  A10200          MOV     AX,B
            0034  83C004          ADD     AX,4H
            0037  39060000        CMP     A,AX
            003B  7306            JNB     @1
                   THEN A = A * 2;
                                                    ; STATEMENT # 7
            003D  D1260000        SHL     A,1
            0041  FB              STI
            0042  F4              HLT
                        @1:
   8    1         ELSE A = A + 1;
                                                    ; STATEMENT # 8
            0043  8306000001      ADD     A,1H
   9    1     END EXAMPLE;
                                                    ; STATEMENT # 9
            0048  FB              STI
            0049  F4              HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 004AH     74D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00CEH    206D
     MAXIMUM STACK SIZE = 0000H      0D
     9 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-3. Sample Program Showing the OPTIMIZE (2) Control

3.5.5 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 name and relative address of each symbol whose address is known at compile-time, and the statement number and relative address of each source program statement.

The NODEBUG control specifies that this information is not to be placed in the object module.

3.6 The WORKFILES Control

The WORKFILES control is a primary control, with the form

```
WORKFILES (:device:,:device:)

Default: WORKFILES(:F1:,:F1:)
```

Each *device* is the name of a direct access device such as a diskette drive.

During compilation, the compiler creates work files which are deleted at the end of compilation (see Section 2.2.3). If the WORKFILES control is not used, these files will be on :F1:. The WORKFILES control allows you to specify any two devices for storage of these files. For example, to specify storage of work files on Drives 1 and 0, use

```
WORKFILES (:F0:,:F1:)
```

Note that *two* device names are required. To specify only one device, specify it twice—for example, to put all work files on Drive 0, use

```
WORKFILES (:F0:,:F0:)
```

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 3.7 below). If only one device is used for work files, it should have twice this amount of space available.

3.7 Source Inclusion Controls

These controls allow the input source to be changed to a different file. The controls are:

```
INCLUDE
SAVE / RESTORE
```

3.7.1 INCLUDE

INCLUDE is a general control, with the form:

```
INCLUDE (pathname)
```

where *pathname* is a standard ISIS-II pathname specifying a disk file.

Example: `INCLUDE(:F1:SYSLIB.SRC)`

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*.

3.7.2 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.

3.8 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 5.

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

MEDIUM

LARGE

Default: SMALL

3.8.1 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 Chapters 4 and 5 for details.

Note that the SMALL size should always be used to compile modules originally written in PL/M-80.

3.8.2 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 Chapters 4 and 5 for details.

3.8.3 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 Chapters 4 and 5 for details.

3.9 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 3-4 shows an example program using the conditional compilation controls, while Figure 3-5 shows the same example with NOCOND being used.

The controls are

```
SET / RESET
IF / ELSEIF / ELSE / ENDIF
COND / NOCOND
```

```
PL/M-86 COMPILER    EXAMPLE


ISIS-II PL/M-86 DEBUG X012 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:CEX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:CEX.P86 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   1      DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8   1      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)
              $ELSEIF DEBUG = 3
  10   1          CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
              $   SET (TRAP)
              $ENDIF

              $IF TRAP
  11   1          CALL DISPLAY$PROMPT;
  12   1          CALL AWAIT$CR;
              $ENDIF

  13   1      END EXAMPLE;


MODULE INFORMATION:

     CODE AREA SIZE     = 001FH     31D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0008H      8D
     29 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-4. Sample Program Showing the SET(DEBUG=) Control

```
PL/M-86 COMPILER    EXAMPLE


ISIS-II PL/M-86 DEBUG X012 COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN :F1:CEX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:CEX.P86 SET(DEBUG=3) NOCOND


   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   1      DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8   1      AWAIT$CR: PROCEDURE EXTERNAL; END AWAIT$CR;

              $IF DEBUG = 1
              $ELSEIF DEBUG = 3
  10   1          CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
              $   SET (TRAP)
              $ENDIF

              $IF TRAP
  11   1          CALL DISPLAY$PROMPT;
  12   1          CALL AWAIT$CR;
              $ENDIF

  13   1      END EXAMPLE;


MODULE INFORMATION:

     CODE AREA SIZE     = 001FH     31D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0008H      8D
     29 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-5. Sample Program Showing the NOCOND Control

3.9.1 SET / RESET

These are general controls. The SET control has the general form

```
SET (switch assignment list)
```

where the *switch assignment list* consists of one or more switch assignments separated by commas. A switch assignment has the form

```
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.

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).

3.9.2 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 which have already appeared in SET controls and whole-number constants in the range 0 to 255. Parenthesized subexpressions are not allowed. Within these restrictions, the condition is evaluated according to the PL/M-86 rules for expression evaluation. Note that the *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.

3.9.3 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 which is skipped.

CHAPTER 4
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 either QRL86 alone or LINK86 and LOC86 together. A knowledge of the makeup of an object module is not necessary for PL/M-86 programming, but for those desiring to study this subject in detail, this chapter is included.

The object module output by the compiler contains five *sections*:

- Code Section
- Constant Section (Absent in LARGE case—see below)
- Data Section
- Stack Section
- Memory Section

As explained in the next chapter, these sections can be combined in various ways into "memory segments" for execution, depending on the size of the program (SMALL, MEDIUM, or LARGE).

4.1 Code Section

This section contains the object code generated by the source program. If the LARGE 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.

4.2 Constant Section

This section contains all variables initialized with the DATA initialization, as well as all REAL constants and all constant lists. If the LARGE control is used, this information is placed in the code section and no constant section is produced.

4.3 Data Section

All variables which are not parameters, based, located with an AT attribute, initialized with the DATA attribute, or local to a REENTRANT procedure are allocated space in this section.

In addition, when a nested procedure contains a reference to any parameter of an enclosing procedure, all parameters of the enclosing procedure are placed in the data section upon entry to the enclosing procedure during program execution. During compilation, space is reserved in the data section for this purpose.

4.4 Stack Section

The stack section is used in executing procedures, as explained in Chapters 9 and 10. 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 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 procedures.

4.5 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, MEDIUM, or LARGE) as explained in Chapter 5.

The compiler generates a memory section of length zero, and it is the user's responsibility to specify the actual (run-time) space required during the linkage and relocation process.

CHAPTER 5
PROGRAM SIZE

The allocation (via relocation and linkage) of runtime memory for a program depends on the size control (SMALL, MEDIUM, or LARGE) specified in compiling the modules of the program. All modules of a program must be compiled with the same size control.

The size also influences the way in which locations are referenced in the compiled program, and this in turn leads to certain programming restrictions for each size control.

A PL/M-86 programmer need not be concerned about memory addressing concepts on the 8086, as the size controls transparently handle the mechanics of program segmentation. The simple rule is:

- For programs with less than 64K bytes of code and with less than 64K bytes of data (for a maximum program size of 128K bytes) use the default (SMALL control) and observe the restrictions given in section 5.2.1.
- If you just can't squeeze your code into 64K bytes, but all your data fits in 64K bytes, use the MEDIUM control and observe the restrictions in 5.3.1.
- If you also need more than 64K bytes of data, use the LARGE control and observe the restrictions in 5.4.1.

Otherwise, the material in this chapter needn't be studied.

5.1 8086 Memory Concepts

8086 memory space has an extent of one megabyte, but a 16-bit value can only address 64K locations. A complete physical address requires 20 bits. Therefore, a 16-bit quantity is used as an *offset*, and references one of 64K possible locations within a *segment* of 8086 memory.

A segment is defined as up to 64K contiguous memory locations, beginning at a 16-byte boundary.

Any location in 8086 memory can be specified by specifying a particular segment and using a 16-bit value as the offset to specify where the location lies within that segment.

Since a segment always starts at a 16-byte boundary, the 20-bit physical address of the first location in the segment always ends with four zero bits. Therefore, it can be shifted to the right four bits without loss of information. This yields a 16-bit quantity called a *segment address*. Four CPU registers (CS, DS, SS, and ES) are used by default to hold segment addresses.

To form a 20-bit physical address, a segment address is shifted left four bits and an offset is added to it.

5.2 The SMALL Case

The SMALL case is the default case, and should be used whenever possible for greatest efficiency. As explained below in Section 5.2.2, the SMALL case *must* be used to compile PL/M-80 programs.

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 Chapter 10). 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.

5.2.1 Programming Restrictions in the SMALL Case

The following restrictions must be observed:

1. A whole-number constant may not be assigned to a POINTER variable. For example:

   ```
   DECLARE P POINTER;
   P=100;
   ```

   is not allowed.

2. A whole-number constant may not be used to initialize a POINTER variable. For example:

   ```
   DECLARE P POINTER INITIAL(100);
   ```

   is not allowed.

3. A variable that is absolutely located (by using the AT attribute with a numeric constant) may not be used with the @ operator. For example:

   ```
   DECLARE B BYTE AT(100), P POINTER; P=@B;
   ```

   is not allowed.

4. 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 AT(100) PUBLIC;
   ```

   is not allowed.

Restrictions 1, 2, and 3 have the net effect of ensuring that all POINTER values used by the program in the SMALL case are within one of the two segments. Absolutely located variables can be referenced, but not via POINTER values.

Note that Restrictions 3 and 4 do not apply when the "location" within the AT attribute is formed with the @ operator.

Restriction 4 arises because EXTERNAL variables are assumed to be in the data segment.

5.2.2 PL/M-80 Compatibility

The SMALL control should always be used 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 (INTEGER, REAL, and POINTER) as programmer-defined identifiers. Otherwise, complete upwards compatibility is provided by PL/M-86.

5.3 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 Chapter 10).

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.

5.3.1 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. This is normal practice in PL/M-86. 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 8086 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 AT(100) PUBLIC;
```

   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 4 in the SMALL case, and arises for the same reason.

5.4 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.

5.4.1 Programming Restrictions in the LARGE 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. This is normal practice in PL/M-86. 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 8086 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*/
```

These are the same as Restrictions 1 and 2 in the MEDIUM case, and arise for the same reason.

CHAPTER 6
FLOATING-POINT ARITHMETIC

Programming using the PL/M-86 REAL data type is explained in the *PL/M-86 Programming Manual*, 98-466. That manual contains all the information necessary for REAL programming, with the exception of REAL error control and the use of REALs by interrupting programs. This chapter covers these two topics, and in addition describes the general aspects of the design of the REAL math facility used to support REAL arithmetic in PL/M-86.

6.1 Representation of REAL Values

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

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

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

where

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

The following examples illustrate these concepts.

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

1010.01B

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

$1.01001B * 2^3$

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

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

The complete 32-bit representation would be

```
0 10000010 01001000000000000000000
```

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

```
highest address:  01000001
                  00100100
                  00000000
lowest address:   00000000
```

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

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

```
0.0001B
```

In "binary scientific" we would have

$1.0000B * 2^{-4}$

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

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

$1.11111111111111111111111B * 2^{127}$

or approximately $3.37*10^{38}$

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

$1.0B * 2^{-126}$

or approximately 1.10^{-38}

6.2 The REAL Math Facility

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

- The REAL stack, used to hold operands and results during REAL operations.
- The *REAL error byte*, consisting of 8 bits. The bits in this byte correspond to the possible errors that can arise during REAL operations (see Section 6.3 below). To indicate an error, the corresponding bit is set to 1. There is a builtin procedure (described below) that allows the program to read and clear the REAL error byte.

  The REAL error byte is initialized to all 0's.

- The *REAL mode word*, consisting of 16 bits.

  Bits 0-7 are a program-specified mask for error conditions. If an error occurs and the corresponding bit in the REAL mode word is 0, the REAL math facility interrupts the CPU with Interrupt 16 so that the error can be reported. If the bit is 1, no interrupt occurs (however, the error is still recorded by setting the corresponding bit in the REAL error byte). These bits are initialized to all 1's (i.e., all errors are masked). In most cases the programmer will want to change this mask. A builtin procedure (described below) allows the program to alter the comtents of the REAL mode word.

  The remaining bits (8-15) of the REAL mode word are reserved and must be set to zero.

6.3 Error Categories

WARNING: If an error occurs that is masked, the result is undefined. The undefined result can cause other errors to occur later. This is true of all errors except the INEXACT error which, if masked, will give the correctly rounded result. Therefore, the suggested value for the REAL mode word is one that masks the INEXACT error and unmasks all other errors. This value is 0040H. (See the SET$REAL$MODE procedure below for directions on setting the REAL mode word.)

The errors that can occur in floating-point operations fall into the following eight categories:

1. *Stack error*: This is caused by REAL stack underflow or overflow.

   REAL stack underflow can be caused by failing to restore the REAL status after returning from an interrupt procedure that saved the status (see below).

   REAL stack overflow can be caused by too many intermediate REAL results accumulating in the REAL stack. This might occur if REAL procedure calls are nested too deeply. No rule can be given for avoiding this situation, as it depends on the number of intermediate results involved in specific computations. However, the compiler ensures that no *single* procedure will cause REAL stack overflow when the stack is initially empty. If a stack error occurs within a particular procedure, modify the code for that procedure to call the builtin procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS (described below). Save the stack contents before performing REAL arithmetic in the procedure and restore the stack contents before returning.

   This error causes Bit 0 of the REAL error byte to be set to 1, and if Bit 0 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

2. *Invalid error*: This occurs whenever an operand is not a valid REAL value.

   This error is most likely to be caused by referencing an uninitialized REAL variable or by referencing a location that does not contain a REAL value (as might occur with an out-of-range subscript for a REAL array).

   This error causes Bit 1 of the REAL error byte to be set to 1, and if Bit 1 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

3. *Zero divide error*: This results from an attempt to divide by zero.

   This error causes Bit 2 of the REAL error byte to be set to 1, and if Bit 2 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

4. *Overflow error.* This occurs whenever a result is too large in absolute value to store.

   This error can arise in the following operations: assignment, addition, subtraction, multiplication, division, and conversion of a REAL value to an INTEGER value.

   This error causes Bit 3 of the REAL error byte to be set to 1, and if Bit 3 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

5. *Underflow error.* This occurs whenever a nonzero result is too small in absolute value to store.

   This error can arise in the following operations: assignment, addition, subtraction, multiplication, and division.

   This error causes Bit 4 of the REAL error byte to be set to 1, and if Bit 4 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

6. *Domain error.* This occurs whenever an operand is not within the domain of the operation being performed.

   This error is most likely to be caused by referencing an uninitialized REAL variable or by referencing a location that does not contain a REAL value (as might occur with an out-of-range subscript for a REAL array).

   This error causes Bit 5 of the REAL error byte to be set to 1, and if Bit 5 of the REAL mode word is 0, it will cause Interrupt 16 to occur.

7. *Inexact error.* This occurs whenever the result of a floating-point operation is rounded up or down.

   This error causes Bit 6 of the REAL error byte to be set to 1, and if Bit 6 of the REAL mode word is 0, it will cause Interrupt 16 to occur. Normally this error should always be masked.

8. This bit is not used.

6.4 THE SET$REAL$MODE PROCEDURE

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

```
CALL SET$REAL$MODE (modeword) ;
```

where

- *modeword* is an expression with a WORD value.

The value of *modeword* becomes the new contents of the REAL mode word. If bits 8-15 are not zero in *modeword* then the results are undefined. The suggested value for *modeword* is 0040H. This value will cause Interrupt 16 to occur for all but the inexact condtions. As for all interrupts, it is the user's responsibility to supply an Interrupt 16 procedure.

6.5 The GET$REAL$ERROR Procedure

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

```
GET$REAL$ERROR
```

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

6.6 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 builtin procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS make this possible.

6.6.1 The SAVE$REAL$STATUS Procedure

SAVE$REAL$STATUS is a builtin untyped procedure activated by a CALL statement with the form

```
CALL SAVE$REAL$STATUS (location) ;
```

where

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

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

WARNING

In an Interrupt 16 procedure that uses the REAL math facility, the GET$REAL$ERROR builtin must be invoked *before* the SAVE$REAL$STATUS procedure. Otherwise the cause of the error cannot be determined and an infinite loop may result.

6.6.2 The RESTORE$REAL$STATUS Procedure

RESTORE$REAL$STATUS is an untyped builtin procedure activated by a CALL statement of the form

```
CALL RESTORE$REAL$STATUS (location) ;
```

where

- *location* is a pointer to the area where REAL status was previously saved by the SAVE$REAL$STATUS procedure.

The REAL status is restored, using the saved information at the specified location.

6.7 Linkage to the Facility

The PLM86.LIB library file (supplied with the compiler) contains the REAL math facility. This will be automatically linked into an object program that requires floating-point operations, assuming PLM86.LIB is named in the link list.

CHAPTER 7
LISTING FORMATS

7.1 Program Listing

During the compilation process a listing of the source input is produced. (See Chapters 2 and 3 for details of the file conventions for this listing.) Each page of the listing carries a numbered page-header which identifies the compiler, and optionally gives a title, a subtitle, and/or a date. The first part of the listing contains a summary of the compilation beginning with the compiler identification and the name of the PL/M-86 source module being compiled. The next line names the file receiving the object code. Finally, the command line used to invoke the compiler is reproduced. The listing of the program itself follows. A sample program listing is shown in Figure 7-1.

```
PL/M-86 COMPILER    STACK MODULE                                10 NOV 77  PAGE   1


ISIS-II PL/M-86 DEBUG X001 COMPILATION OF MODULE STACK
OBJECT MODULE PLACED IN :F1:STACK.OBJ
COMPILER INVOKED BY:  PLM86 :F1:STACK.SRC PAGEWIDTH(85) CODE XREF TITLE('STACK MODULE
                -') DATE(10 NOV 77)



  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 stack*/
                                              ; STATEMENT # 3
                      PUSH      PROC NEAR
        0000  55              PUSH    BP
        0001  89EC            MOV     BP,SP
  4   2            DECLARE B BYTE;
  5   2            S(T:=T+1) = B; /*Increment T and store B*/
                                              ; STATEMENT # 5
        0003  8A066400        MOV     AL,T
        0007  80C001          ADD     AL,1H
        000A  88066400        MOV     T,AL
  6   2         END PUSH;
                                              ; STATEMENT # 6
                      PUSH      ENDP

  7   1         POP: PROCEDURE BYTE PUBLIC; /*Returns value popped from stack*/
                                              ; STATEMENT # 7
                      POP       PROC NEAR
        000E  55              PUSH    BP
        000F  89EC            MOV     BP,SP
  8   2            RETURN S((T:=T-1)+1); /*Decrement T, then return S(T+1)*/
                                              ; STATEMENT # 8
        0011  8A066400        MOV     AL,T
        0015  80E801          SUB     AL,1H
        0018  88066400        MOV     T,AL
        001C  FA              CLI
        001D  2E8E16FFFF      MOV     SS,CS:@@STACK$FRAME
        0022  BC0000          MOV     SP,@@STACK$OFFSET
        0025  89EC            MOV     BP,SP
        0027  16              PUSH    SS
        0028  1F              POP     DS
        0029  FB              STI
  9   2         END POP;
                                              ; STATEMENT # 9
        002A  5D              POP     BP
        002B  C3              RET


                                        2


                      POP       ENDP

 10   1       END STACK; /*module ends here*/
                                              ; STATEMENT # 10
```

Figure 7-1. Program Listing

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 8086 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 conforming to standard 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.

7.2 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, five or six 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.
6. Statement numbers where identifier was referenced (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 (see Chapter 8).

Figure 7-2 is an example of the cross-reference listing.

```
PL/M-86 COMPILER    STACK MODULE                                10 NOV 77  PAGE   2

CROSS-REFERENCE LISTING

  DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES

     3  0004H     1  B             BYTE PARAMETER AUTOMATIC
                                      4    5

     7  000EH    30  POP           PROCEDURE BYTE PUBLIC STACK=0000H

     3  0000H    14  PUSH          PROCEDURE PUBLIC STACK=0000H

     2  0000H   100  S             BYTE ARRAY(100)
                                      5    8

     1  0000H        STACK         PROCEDURE STACK=0000H

     2  0064H     1  T             BYTE PUBLIC INITIAL
                                      5    8
```

Figure 7-2. Cross-Reference Listing

7.3 Compilation Summary

Following the listing (or appearing alone if NOLIST is in effect) is a compilation summary. Six 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.
- *Program errors* gives the number of error messages issued during compilation.

Figure 7-3 is an example of the compilation summary. Refer to Chapter 4 for an explanation of the various object module sections.

```
PL/M-86 COMPILER    EXAMPLE

    CODE AREA SIZE      = 004AH     74D
    CONSTANT AREA SIZE  = 0000H      0D
    VARIABLE AREA SIZE  = 00CEH    206D
    MAXIMUM STACK SIZE  = 0000H      0D
    9 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 7-3. Compilation Summary

CHAPTER 8
RUN-TIME DATA REPRESENTATIONS

8.1 Byte Values

A BYTE value occupies a single byte of memory, except when it is a BYTE parameter stored on the stack.

A BYTE parameter on the stack occupies two contiguous memory bytes. The BYTE value is in the first byte (lower address), and the contents of the second byte (higher address) are undefined.

8.2 Word Values

A WORD value occupies two contiguous memory bytes. The least significant 8 bits of the value are in the first byte (lower address), and the most significant 8 bits are in the second byte (higher address).

8.3 Integer Values

An INTEGER value occupies two contiguous memory bytes. The least significant 8 bits of the value are in the first byte (lower address), and the most significant 8 bits are in the second byte (higher address).

8.4 Real Values

A REAL value occupies four contiguous memory bytes, as described in Chapter 6.

8.5 Pointer Values

The representation of a POINTER value depends on the size control used in compilation. In the SMALL case, a POINTER value is a 16-bit offset and is represented in the same manner as a WORD value.

In the MEDIUM and LARGE cases, a POINTER value consists of a segment address and an offset and occupies four contiguous memory bytes. The 16-bit offset occupies the first two bytes (lower addresses) with the least significant 8 bits in the first byte and the most significant 8 bits in the second byte. The 16-bit segment address occupies the third and fourth bytes, with the least significant 8 bits in the third byte and the most significant 8 bits in the fourth byte.

CHAPTER 9
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 8086 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 Chapter 10.

9.1 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. See Chapter 8 for details on data representations.

After the parameters are passed, the CALL instruction places the return address on the stack. In the SMALL case, 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 return address is a POINTER value consisting of a segment address and offset. The 16-bit segment address (contents of the CS register) is pushed first, and then the offset (IP register contents) is pushed.

Control is then passed to the code of the procedure, by updating the IP register and (in the MEDIUM and LARGE cases) the CS register.

At the point where the procedure gains control, then, the stack layout is as shown in Figure 9-1.

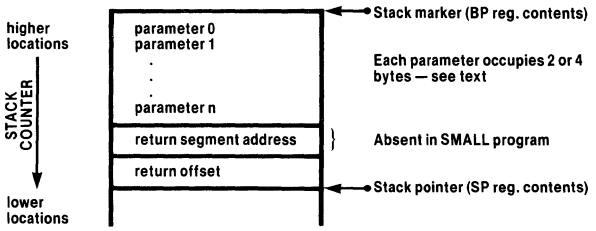

Figure 9-1. Stack Layout at Point Where a Non-Interrupt Procedure Is Activated

9.2 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, the contents of the DS register are placed on the stack. Then the DS register is updated with a value which 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 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. 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).

4. 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.

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 9-2.

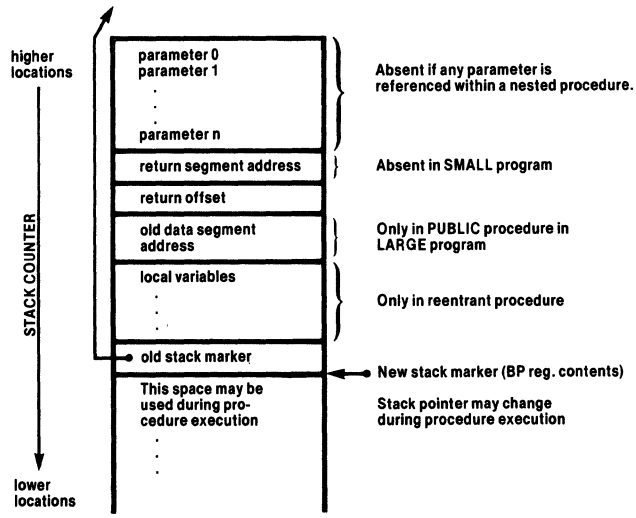

Figure 9-2. Stack Layout During Execution of Non-Interrupt Procedure Body

During execution of the procedure, further stack space may be used for temporary storage generated by the compiler.

9.3 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 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 space was reserved on the stack for variables declared within a reentrant procedure, this space is discarded by adjusting the stack pointer (SP register).
4. If the procedure has the PUBLIC attribute *and* the program size is LARGE, the old data segment address is restored by popping the stored value from the stack into the DS register.
5. 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, 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.

9.4 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
INTEGER	AX Register
POINTER (SMALL size)	BX Register
POINTER (MEDIUM size)	ES and BX Registers
POINTER (LARGE size)	ES and BX Registers
REAL	Top of RMU stack

CHAPTER 10
RUN-TIME INTERRUPT PROCESSING

10.1 General

An interrupt is initiated when the CPU receives a signal on its "maskable interrupt" pin from some peripheral device.

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.

If interrupts are enabled, the following actions take place:

1. The CPU issues an "acknowledge interrupt" signal and waits for the interrupting device to send an interrupt number.
2. The CPU flag registers are placed on the stack (occupying two bytes of stack storage).
3. Interrupts are disabled by clearing the IF flag.
4. Single stepping is disabled by clearing the TF flag.
5. The CPU activates the interrupt procedure corresponding to the interrupt number sent by the interrupting device. The mechanism for this activation is described below.

10.2 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 10.4.

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 10-1.

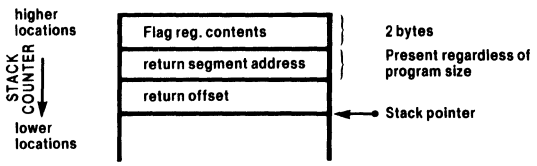

Figure 10-1. Stack Layout at Point Where an Interrupt Procedure Gains Control

10.3 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* which 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 Chapter 9).

At the point where the procedure prologue gains control, the stack layout is as shown in Figure 10-2.

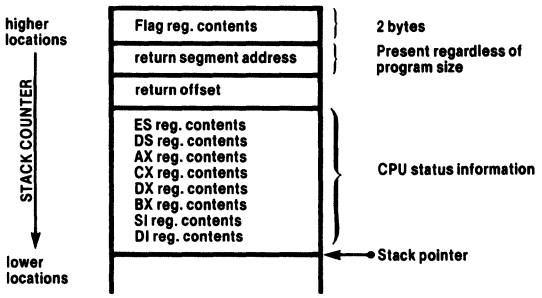

Figure 10-2. Stack Layout After Interrupt Procedure Preface and Before Procedure Prologue

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 10-3.

The return from the procedure body transfers control back into the interrupt procedure preface. At this point the procedure epilogue (see Chapter 9) has restored the stack to the layout of Figure 10-2. The interrupt procedure preface continues with the following steps.

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.

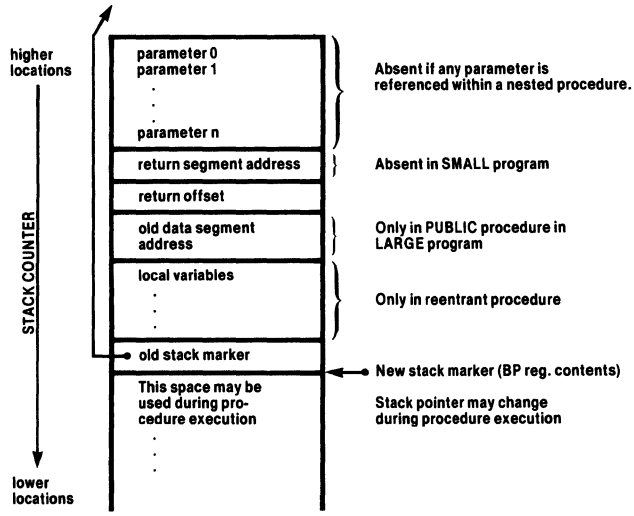

Figure 10-3. Stack Layout During Execution of Interrupt Procedure Body

18. Pop the stack into the ES register.
19. Enable interrupts.
20. 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.

10.4 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, since 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. Thus, the CODE control should be used to determine the displacement of the interrupt entry point from the normal entry point where this feature is desired.

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.

APPENDIX A
THE IXREF PROGRAM

A.1 General

The IXREF program is supplied on the same diskette as the ISIS-II PL/M-86 Compiler. It uses intermediate files produced by the compiler under the IXREF control (see Section 3.2.5) to produce an intermodule cross-reference file.

To use this facility, first compile all modules that are to be cross-referenced, using the IXREF *control* in each case. Then run the IXREF *program* as explained below.

A.2 Invoking the IXREF Program

The IXREF program invocation command has the following general form:

```
[:device:]IXREF input-list [controls]
```

where

- *device* identifies which drive contains the diskette with the IXREF program. This may be omitted if the diskette is in Drive 0.
- *input-list* is a list of pathnames of intermediate files produced by the compiler under the IXREF control. The pathnames must be separated by commas (spaces may also be inserted between pathnames). The pathnames may be in any order and may use the "wild card" construction (see *ISIS-II System User's Guide*, Intel document number 98-306). If any of the specified files is not a valid intermediate file, IXREF will type the pathname and the message BAD RECORD TYPE and will skip the file.
- *controls* is an optional sequence of one or more controls separated by spaces. Controls are described below.

If the invocation command is too long to be typed on one line, you can break it by typing an & character followed by a carriage return. The & must not be within a pathname or control. IXREF responds to the & with a ** prompt to show that it is waiting for a continuation line.

A.3 Controls

The control sequence in the IXREF program invocation is optional. If no controls are used, the output file will have the following characteristics:

- The output pathname will be the same as the first pathname in the input-list, but with the extension IXO.
- No title will be placed at the top of each page.
- All identifiers declared PUBLIC or EXTERNAL will be listed.

Four controls are provided to modify the characteristics of the output file.

A.3.1 THE PRINT CONTROL

This control has the form

```
PRINT (pathname)
```

where *pathname* is a standard ISIS-II pathname to specify the name of the output file.

A.3.2 THE TITLE CONTROL

This control has the form

```
TITLE ('string')
```

where *string* is a sequence of up to 60 characters to be placed at the top of each page of output. If the 60-character limit is exceeded, the string will be truncated on the right.

A.3.3 THE PUBLICS CONTROL

This control has the form

```
PUBLICS
```

and specifies that only PUBLIC identifiers are to be represented in the output file.

A.3.4 THE EXTERNALS CONTROL

This control has the form

```
EXTERNALS
```

and specifies that only EXTERNAL identifiers are to be represented in the output file.

A.4 The IXREF Output File

Figure A-1 shows a typical intermodule cross-reference file produced by IXREF. Note that a "wild card" construction was used in the input-list to input all files on Drive 1 with the extension IXI. Controls were used to specify a title and a pathname for the output file.

The file contains two listings, the "intermodule cross-reference listing" and the "module directory." Both are sorted alphabetically. Note that in the illustration, portions of the intermodule cross-reference listing have been omitted.

Each entry in the intermodule cross-reference listing begins with an identifier in the left column. In the right column, we have the attributes of the identifier, then a semicolon followed by the names of all modules in which it is declared PUBLIC or EXTERNAL.

The first entry after the semicolon is the name of the module in which the identifier is declared PUBLIC. If no PUBLIC declaration is found, the notation ** UNRESOLVED ** appears. Thus we can see that ACTUALBASEPTR is a WORD variable declared PUBLIC in module MACRO and EXTERNAL in modules SYMSCN and STACK.

In the next entry, we see that ACTUALBLOCKENDMARKER is an array of two BYTE elements, declared PUBLIC in module MACRO.

In the module directory, each entry begins with a module name. In the second column, we find the name of the PL/M-86 source file from which the module was compiled, and in the third column we find the name of the diskette where the source file resides. (A diskette is named when it is formatted with the ISIS-II FORMAT command.)

```
ISIS-II IXREF      INTER-MODULE CROSS-REFERENCE                    PAGE 1


ISIS-II IXREF, V1.1
INVOKED BY:
-IXREF :F1:*.IXI TITLE('INTER-MODULE CROSS-REFERENCE') &
               PRINT(:F1:ASSEMB.OUT)


INTER-MODULE CROSS-REFERENCE LISTING


NAME.......................ATTRIBUTES: MODULE NAMES

ACTUALBASEPTR..............WORD;  MACRO SYMSCN STACK
ACTUALBLOCKENDMARKER.......BYTE(2);  MACRO
ACTUALDELIMITER............PROCEDURE BYTE;  MACRO SCNFSM
ACTUALPROTECTED............BYTE;  MACRO SCNFSM
  .                          .
  .                          .
  .                          .
BLKSTK.....................BYTE(17);  PUBLICDCL DRIVE STACK MACRO
BLOCKNO....................BYTE;  ** UNRESOLVED ** ENDLIN
  .                          .
  .                          .
  .                          .
XREFSYMBUF.................BYTE(6);  PUBLICDCL ENDLIN DRIVE RELOBJ SCNFSM
XREFSYMBUFPREVIOUS.........BYTE(6);  PUBLICDCL SCNFSM
XREFUTILITYNAME............BYTE(11);  PUBLICDCL DRIVE INIT
ZERO.......................WORD;  MACRO STACK SCNFSM
ZEROADDRESS................WORD;  ASSEMB RELOBJ


MODULE DIRECTORY

MODULE NAME:...............FILE NAME...DISKETTE NAME

ASSEMB.....................ASSEMB.SRC  SOURCE
DRIVE......................DRIVE.SRC   DRIVE
ENDLIN.....................ENDLIN.SRC  END
INIT.......................INIT.SRC    INIT.OVL
MACRO......................MACRO.SRC   MACRO.SRC
PUBLICDCL..................PUBLIC.SRC  DRIVE
RELOBJ.....................RELOBJ.SRC  RELOBJ
SCNFSM.....................SCNFSM.SRC  SOURCE
STACK......................STACK.SRC   SYMSCN.STK
SYMSCN.....................SYMSCN.SRC  SYMSCN.STK
```

Figure A-1. Intermodule Cross-Reference Listing

A.5 Error Conditions

IXREF detects the following error conditions in the invocation command:

- Incorrect file specifications in input-list or PRINT control (IXREF terminates and produces no output).
- Nonexistent file in input-list (if possible, IXREF skips to next pathname and continues; otherwise it terminates and produces no output).
- Missing parenthesis in PRINT or TITLE control (IXREF terminates and produces no output).
- Misspelled or unknown controls (IXREF terminates and produces no output).
- PUBLICS and EXTERNALS controls used in same invocation of IXREF (IXREF terminates and produces no output).
- Repetition of a control (IXREF terminates and produces no output).

A.6 Temporary Files Used by IXREF

While running, IXREF uses the following temporary files:

```
:device:IXIN.TMP
:device:IXOUT.TMP
:device:MODNM.TMP
```

where *device* is the same device specified for the first file in the input-list. These files are deleted when IXREF terminates. Therefore, if you have any files with these names on the same device as the first file in the input-list, you must rename them before running IXREF.

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:

Number of elements in a factored declare list	32
Number of members in a structure	32
Number of labels on a statement	9
Number of procedures in a module	255
Number of DO blocks in a procedure	255
Nesting of blocks	18
Length of an input source line (including CR and LF)	122
Length of a string constant	255
Nesting of INCLUDE controls	5

APPENDIX C
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.

C.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
```

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

Source error message list:

1. INVALID PL/M-86 CHARACTER
2. UNPRINTABLE ASCII CHARACTER
3. IDENTIFIER, STRING, OR NUMBER TOO LONG, TRUNCATED
4. ILLEGAL NUMERIC CONSTANT TYPE
5. INVALID CHARACTER IN NUMERIC CONSTANT
6. ILLEGAL MACRO REFERENCE, RECURSIVE EXPANSION
7. LIMIT EXCEEDED: MACROS NESTED TOO DEEPLY
8. INVALID CONTROL FORMAT
9. INVALID CONTROL
10. ILLEGAL USE OF PRIMARY CONTROL AFTER NON-CONTROL LINE
11. MISSING CONTROL PARAMETER
12. INVALID CONTROL PARAMETER
13. LIMIT EXCEEDED: INCLUDE NESTING
14. INVALID CONTROL FORMAT, INCLUDE NOT LAST CONTROL
15. MISSING INCLUDE CONTROL PARAMETER
16. ILLEGAL PRINT CONTROL
17. INVALID PATH-NAME
18. INVALID MULTIPLE LABELS AS MODULE NAMES
19. INVALID LABEL IN MODULE WITHOUT MAIN PROGRAM

20.MISMATCHED IDENTIFIER AT END OF BLOCK
21.MISSING PROCEDURE NAME
22.INVALID MULTIPLE LABELS AS PROCEDURE NAMES
23.INVALID LABELLED END IN EXTERNAL PROCEDURE
24.INVALID STATEMENT IN EXTERNAL PROCEDURE
25.UNDECLARED PARAMETER
26.INVALID DECLARATION, STATEMENT OUT OF PLACE
27.LIMIT EXCEEDED: NUMBER OF DO BLOCKS *(terminal error)*
28.MISSING 'THEN'
29.ILLEGAL STATEMENT
30.LIMIT EXCEEDED: NUMBER OF LABELS ON STATEMENT
31.LIMIT EXCEEDED: PROGRAM TOO COMPLEX *(terminal error)*
32.INVALID SYNTAX, TEXT IGNORED UNTIL ';'
33.DUPLICATE LABEL DECLARATION
34.DUPLICATE PROCEDURE DECLARATION
35.LIMIT EXCEEDED: NUMBER OF PROCEDURES *(terminal error)*
36.MISSING PARAMETER
37.MISSING ')' AT END OF PARAMETER LIST
38.DUPLICATE PARAMETER NAME
39.INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL
40.DUPLICATE ATTRIBUTE
41.CONFLICTING ATTRIBUTE
42.INVALID INTERRUPT VALUE
43.MISSING INTERRUPT VALUE
44.ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH PARAMETERS
45.ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH TYPED PROCEDURE
46.ILLEGAL USE OF LABEL
47.MISSING ')' AT END OF FACTORED DECLARATION
48.ILLEGAL DECLARATION STATEMENT SYNTAX
49.LIMIT EXCEEDED: NUMBER OF ITEMS IN FACTORED DECLARE
50.INVALID ATTRIBUTES FOR BASE
51.INVALID BASE, MEMBER OF BASED STRUCTURE
52.INVALID BASE, MEMBER OF ARRAY OF STRUCTURES
53.INVALID STRUCTURE MEMBER IN BASE
54.UNDECLARED BASE
55.UNDECLARED STRUCTURE MEMBER IN BASE
56.INVALID MACRO TEXT, NOT A STRING CONSTANT
57.INVALID DIMENSION, ZERO ILLEGAL
58.INVALID STAR DIMENSION IN FACTORED DECLARATION
59.ILLEGAL DIMENSION ATTRIBUTE
60.MISSING ')' AT END OF DIMENSION
61.MISSING TYPE
62.INVALID STAR DIMENSION WITH 'STRUCTURE' OR 'EXTERNAL'

63 INVALID DIMENSION WITH THIS ATTRIBUTE
64.MISSING STRUCTURE MEMBERS
65.MISSING ')' AT END OF STRUCTURE MEMBER LIST
66.INVALID STRUCTURE MEMBER, NOT AN IDENTIFIER
67.DUPLICATE STRUCTURE MEMBER NAME
68.LIMIT EXCEEDED: NUMBER OF STRUCTURE MEMBERS
69.INVALID STAR DIMENSION WITH STRUCTURE MEMBER
70.INVALID MEMBER TYPE, 'STRUCTURE' ILLEGAL
71.INVALID MEMBER TYPE, 'LABEL' ILLEGAL
72.MISSING TYPE FOR STRUCTURE MEMBER
73.INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL
74.'DATA' OR 'INITIAL'
75.MISSING ARGUMENT OF 'AT', 'DATA', OR 'INITIAL'
76.CONFLICTING ATTRIBUTE WITH PARAMETER
77.INVALID PARAMETER DECLARATION, BASE ILLEGAL
78.DUPLICATE DECLARATION
79.ILLEGAL PARAMETER TYPE
80.INVALID DECLARATION, LABEL MAY NOT BE BASED
81.CONFLICTING ATTRIBUTE WITH 'BASE'
82.INVALID SYNTAX, MISMATCHED '('
83.LIMIT EXCEEDED: DYNAMIC STORAGE *(terminal error)*
84.LIMIT EXCEEDED: BLOCK NESTING *(terminal error)*
85.LONG STRING ASSUMED CLOSED AT NEXT SEMICOLON OR QUOTE
86.LIMIT EXCEEDED: SOURCE LINE LENGTH
87.MISSING 'END', END-OF-FILE ENCOUNTERED
88.INVALID PROCEDURE NESTING, ILLEGAL IN REENTRANT PROCEDURE
89.MISSING 'DO' FOR MODULE
90.MISSING NAME FOR MODULE
91.ILLEGAL PAGELENGTH CONTROL VALUE
92.ILLEGAL PAGEWIDTH CONTROL VALUE
93.MISSING 'DO' FOR 'END', 'END' IGNORED
94.ILLEGAL CONSTANT, TOO LARGE FOR CONTEXTUALLY DETERMINED TYPE
95.ILLEGAL RESPECIFICATION OF PRIMARY CONTROL IGNORED
96.COMPILER ERROR: SCOPE STACK UNDERFLOW
97.COMPILER ERROR: PARSE STACK UNDERFLOW
98.INCLUDE FILE IS NOT A DIRECT ACCESS FILE *(terminal error)*
99.INVALID REAL CONSTANT
100.INVALID STRING CONSTANT IN EXPRESSION
101.INVALID ITEM FOLLOWS DOT OR AT SIGN OPERATOR
102.MISSING PRIMARY OPERAND
103.MISSING ')' AT END OF SUBEXPRESSION
104.ILLEGAL PROCEDURE INVOCATION WITH DOT OR AT SIGN OPERATOR

105.UNDECLARED IDENTIFIER
106.ILLEGAL PAGELENGTH(4) AND SUBTITLE COMBINATION
107.INVALID USE OF '@' WITH LOCAL PROCEDURE
108.INVALID USE OF '.' WITH PUBLIC OR EXTERNAL PROCEDURE
110.INVALID LEFT OPERAND OF QUALIFICATION, NOT A STRUCTURE
111.INVALID RIGHT OPERAND OF QUALIFICATION, NOT IDENTIFIER
112. UNDECLARED STRUCTURE MEMBER
113.MISSING ')' AT END OF ARGUMENT LIST
114.INVALID SUBSCRIPT, MULTIPLE SUBSCRIPTS ILLEGAL
115.MISSING ')' AT END OF SUBSCRIPT
116.MISSING '=' IN ASSIGNMENT STATEMENT
117.MISSING PROCEDURE NAME IN CALL STATEMENT
118.INVALID INDIRECT CALL, IDENTIFIER NOT A WORD OR POINTER SCALAR
119.LIMIT EXCEEDED: PROGRAM TOO COMPLEX *(terminal error)*
120.LIMIT EXCEEDED: EXPRESSION TOO COMPLEX *(terminal error)*
121.LIMIT EXCEEDED: EXPRESSION TOO COMPLEX *(terminal error)*
122.LIMIT EXCEEDED: PROGRAM TOO COMPLEX *(terminal error)*
123.INVALID DOT OR AT SIGN OPERAND, BUILT-IN PROCEDURE ILLEGAL
124.MISSING ARGUMENTS FOR BUILT-IN PROCEDURE
125.ILLEGAL ARGUMENT FOR BUILT-IN PROCEDURE
126.MISSING ')' AFTER BUILT-IN PROCEDURE ARGUMENT LIST
127.INVALID SUBSCRIPT ON NON-ARRAY
128.INVALID LEFT-HAND OPERAND OF ASSIGNMENT
129.ILLEGAL 'CALL' WITH TYPED PROCEDURE
130.ILLEGAL REFERENCE TO OUTPUT OR OUTWORD FUNCTION
131.ILLEGAL REFERENCE TO UNTYPED PROCEDURE
132.ILLEGAL USE OF LABEL
133.ILLEGAL REFERENCE TO UNSUBSCRIPTED ARRAY
134.ILLEGAL REFERENCE TO UNSUBSCRIPTED MEMBER ARRAY
135.ILLEGAL REFERENCE TO AN UNQUALIFIED STRUCTURE
136.INVALID RETURN FOR UNTYPED PROCEDURE, VALUE ILLEGAL
137.MISSING VALUE IN RETURN FOR TYPED PROCEDURE
138.MISSING INDEX VARIABLE
139.INVALID INDEX VARIABLE TYPE
140.MISSING '=' FOLLOWING INDEX VARIABLE
141.MISSING 'TO' CLAUSE
142.MISSING IDENTIFIER FOLLOWING GOTO
143.INVALID REFERENCE FOLLOWING GOTO, NOT A LABEL
144.INVALID GOTO LABEL, NOT AT LOCAL OR MODULE LEVEL
145.MISSING 'TO' FOLLOWING 'GO'
146.MISSING ')' AFTER 'AT' RESTRICTED EXPRESSION
147.MISSING IDENTIFIER FOLLOWING DOT OR AT SIGN OPERATOR

148.INVALID QUALIFICATION IN RESTRICTED REFERENCE

149.INVALID SUBSCRIPTING IN RESTRICTED REFERENCE

150.MISSING ')' AT END OF RESTRICTED SUBSCRIPT

151.INVALID OPERAND IN RESTRICTED EXPRESSION

152.MISSING ')' AFTER CONSTANT LIST

153.INVALID NUMBER OF ARGUMENTS IN CALL, TOO MANY

154.INVALID NUMBER OF ARGUMENTS IN CALL, TOO FEW

155.INVALID RETURN IN MAIN PROGRAM

156.MISSING RETURN STATEMENT IN TYPED PROCEDURE

157.INVALID ARGUMENT, ARRAY REQUIRED FOR LENGTH OR LAST

158.INVALID DOT OR AT SIGN OPERAND, LABEL ILLEGAL

159.COMPILER ERROR: PARSE STACK UNDERFLOW

160.COMPILER ERROR: OPERAND STACK UNDERFLOW

161.COMPILER ERROR: ILLEGAL OPERAND STACK EXCHANGE

162.COMPILER ERROR: OPERATOR STACK UNDERFLOW

163.COMPILER ERROR: GENERATION FAILURE

164.COMPILER ERROR: SCOPE STACK OVERFLOW

165.COMPILER ERROR: SCOPE STACK UNDERFLOW

166.COMPILER ERROR: CONTROL STACK OVERFLOW

167.COMPILER ERROR: CONTROL STACK UNDERFLOW

168.COMPILER ERROR: BRANCH MISSING IN 'IF' STATEMENT

169.ILLEGAL FORWARD CALL

170.ILLEGAL RECURSIVE CALL

171.INVALID USE OF DELIMITER OR RESERVED WORD IN EXPRESSION

172.INVALID LABEL: UNDEFINED

173.INVALID LEFT SIDE OF ASSIGNMENT: VARIABLE DECLARED WITH DATA ATTRIBUTE

174.INVALID NULL PROCEDURE

175.ILLEGAL POINTER ARITHMETIC IN RESTRICTED EXPRESSION

176.INVALID ABSOLUTE ADDRESS, TOO LARGE

178.ILLEGAL REAL ARITHMETIC IN RESTRICTED EXPRESSION

179.ILLEGAL REAL CONSTANT IN 'AT' CLAUSE RESTRICTED EXPRESSION

180.INVALID OPERATOR OR OPERAND, TYPE CONFLICTS WITH EXPECTED TYPE

181.LIMIT EXCEEDED: CONSTANT OR CODE SEGMENT SIZE

182.ILLEGAL REFERENCE TO ABSOLUTE ADDRESS WITH SMALL OPTION SPECIFIED

183.INVALID 'AT' RESTRICTED REFERENCE, EXTERNAL ATTRIBUTE CONFLICTS WITH PUBLIC

184.INVALID EXPRESSION, TWO SUCCESSIVE RELATIONAL OPERATORS

185.LIMIT EXCEEDED: NUMBER OF EXTERNAL ITEMS

186.INVALID RESTRICTED EXPRESSION, TYPE CONFLICTS WITH TARGET

187.ILLEGAL INITIALIZATION TO A BASED OR AUTOMATIC ADDRESS

188.MISSING ENDIF OPTION

189.MISSING OR INVALID CONDITIONAL COMPILATION PARAMETER

190.MISSING OR INVALID CONDITIONAL COMPILATION CONSTANT

191.MISPLACED ELSE OR ENDIF OPTION

192.MISPLACED ENDIF OPTION

193.CONDITIONAL COMPILATION PARAMETER NAME TOO LONG, TRUNCATED

194.MISSING OPERATOR IN CONDITIONAL COMPILATION EXPRESSION

195.INVALID CONDITIONAL COMPILATION CONSTANT TOO LARGE

196.INVALID UNDEFINED CONDITIONAL COMPILATION PARAMETER

197.LIMIT EXCEEDED: SAVE NESTING

198.MISPLACED RESTORE OPTION

199.LIMIT EXCEEDED: PROCEDURE COMPLEXITY FOR OPTIMIZE(2) *(terminal error)*

200.LIMIT EXCEEDED: STATEMENT SIZE

201.INVALID DO CASE BLOCK, AT LEAST ONE CASE REQUIRED

202.LIMIT EXCEEDED: NUMBER OF ACTIVE CASES

203.LIMIT EXCEEDED: NESTING OF TYPED PROCEDURE CALLS

204.LIMIT EXCEEDED: NUMBER OF ACTIVE PROCEDURES OR DO CASE GROUPS

205.ILLEGAL NESTING OF BLOCKS, ENDS NOT BALANCED

206.LIMIT EXCEEDED: CODE SEGMENT SIZE

207.LIMIT EXCEEDED: SEGMENT SIZE

208.LIMIT EXCEEDED: STRUCTURE SIZE

209.ILLEGAL INITIALIZATION OF MORE SPACE THAN DECLARED

210.INVALID RESTRICTED EXPRESSION, VALUE TOO LARGE FOR TARGET

211.INVALID IDENTIFIER IN 'AT' RESTRICTED REFERENCE

212.INVALID RESTRICTED REFERENCE IN 'AT', BASE ILLEGAL

213.UNDEFINED RESTRICTED REFERENCE IN 'AT'

214.COMPILER ERROR: INVALID OPERATION

215.COMPILER ERROR: EOF READ IN FINAL ASSEMBLY

216.COMPILER ERROR: BAD LABEL ADDRESS

217.ILLEGAL INITIALIZATION OF AN EXTERNAL VARIABLE

218.LIMIT EXCEEDED: REAL EXPRESSION COMPLEXITY

219.COMPILER ERROR: REAL STACK OVERFLOW

220.LIMIT EXCEEDED: BASIC BLOCK COMPLEXITY

221.LIMIT EXCEEDED: STATEMENT SIZE

222.INVALID ABSOLUTE LOCATION FOR PUBLIC WITHOUT LARGE OPTION

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.

C.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 which may occur here are as follows:

```
ILLEGAL COMMAND TAIL SYNTAX OR VALUE
UNRECOGNIZED CONTROL IN COMMAND TAIL
INCLUDE FILE IS NOT A DISKETTE FILE
INVOCATION COMMAND DOES NOT END WITH <CR><LF>
INCORRECT DEVICE SPECIFICATION
SOURCE FILE NOT A DISKETTE FILE
SOURCE FILE NAME INCORRECT
SOURCE FILE EXTENSION INCORRECT
ILLEGAL COMMAND TAIL SYNTAX
MISPLACED CONTROL: WORKFILES ALREADY OPENED
```

C.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 I/O ERROR –
  FILE:
  NAME:
  ERROR:
COMPILATION TERMINATED
```

The errors that may occur here are as follows:

```
ILLEGAL FILENAME SPECIFICATION
ILLEGAL OR UNRECOGNIZED DEVICE SPECIFICATION IN FILENAME
ATTEMPT TO OPEN AN ALREADY OPEN FILE
NO SUCH FILE
FILE IS WRITE PROTECTED
FILE IS NOT ON A DISKETTE
DEVICE TYPE NOT COMPATIBLE WITH INTENDED USE
FILENAME REQUIRED ON DISKETTE FILE
NULL FILE EXTENSION
ATTEMPT TO READ PAST EOF
```

C.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
```

C.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:

SYNC FAILURE READING GLOBALS
UNKNOWN FATAL ERROR
96. COMPILER ERROR: SCOPE STACK UNDERFLOW
97. COMPILER ERROR: PARSE STACK UNDERFLOW
159. COMPILER ERROR: PARSE STACK UNDERFLOW
160. COMPILER ERROR: OPERAND STACK UNDERFLOW
161. COMPILER ERROR: ILLEGAL OPERAND STACK EXCHANGE
162. COMPILER ERROR: OPERATOR STACK UNDERFLOW
163. COMPILER ERROR: GENERATION FAILURE
164. COMPILER ERROR: SCOPE STACK OVERFLOW
165. COMPILER ERROR: SCOPE STACK UNDERFLOW
166. COMPILER ERROR: CONTROL STACK OVERFLOW
167. COMPILER ERROR: CONTROL STACK UNDERFLOW
168. COMPILER ERROR: BRANCH MISSING IN 'IF' STATEMENT
214. COMPILER ERROR: INVALID OPERATION
215. COMPILER ERROR: EOF READ IN FINAL ASSEMBLY
216. COMPILER ERROR: BAD LABEL ADDRESS
219. COMPILER ERROR: REAL STACK OVERFLOW

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____
TITLE _____
COMPANY NAME/DEPARTMENT _____
ADDRESS _____
CITY _____ STATE _____ ZIP CODE _____

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.

First Class
Permit No. 1040
Santa Clara, CA

BUSINESS REPLY MAIL

No Postage Stamp Necessary if Mailed in U.S.A.

Postage will be paid by:

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Attention: Technical Publications