PL/M-86 COMPILER OPERATING INSTRUCTIONS

for 8080/8085-Based Development Systems

Manual Order Number 9800478-04 Rev. D

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

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

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

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

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

BXP	Intellec	Multibus
CREDIT	iSBC	Multimodule
i	iSBX	PROMPT
ICE	Library Manager	Promware
iCS	MCS	RMX
Insite	Megachassis	UPI
Intel	Micromap	μScope
Intelevision		

and the combination of ICE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix.

A230/880/5K DD

PREFACE

This manual describes the operation of the PL/M-86 Compiler, Version 2.1. 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 for 8080/8085-Based Development Systems* 9800466
- *8086 Family Utilities User's Guide for 8080/8085-Based Development Systems* 9800639
- *ISIS-II User's Guide* 9800306
- *Intellec Microcomputer Development System Operator's Manual* 9800129

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

Software

For the Compiler

- ISIS-II Operating System

For Object Programs

- LINK86 and LOC86

For Intermodule Cross-Reference Listings

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

For Debugging

- An ICE-86 or ICE-88 emulator (if 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)
- For hardware execution of floating-point operations, an INTEL 8087 chip (described in the *8087 Supplement to the 8086 Family User's Guide*).

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 disk (such as a system disk). 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 disk into Drive 0. In this example, the system disk contains the compiler.
3. Insert a nonsystem disk into Drive 1. In this example, this disk contains a PL/M-86 source file to be compiled.
4. Bootstrap the 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 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 V3.4                          ;the system identifies itself

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

ISIS-II PL/M-86 COMPILER, V1.2
PL/M-86 COMPILATION COMPLETE      0 PROGRAM ERROR(S)

                    ;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 disk file on the same disk 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 disk and has the same file name, but has the extension OBJ. In the example :F1:MYPROG.OBJ contains the object code produced by compiling :F1:MYPROG.SRC.

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 Users Guide*. See Chapter 6 of this manual (PL/M-86 Compiler Operating Instructions) for a discussion of libraries available for linking with your program, e.g., for performing floating-point arithmetic using software or hardware.

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 disk and must contain a PL/M-86 source module. This command has the form

```
PLM86 source-file
```

if the compiler is in Drive 0.

The remaining chapters of this manual provide a detailed description of operating the compiler, including discussions 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 disk containing the PL/M-86 Compiler must be mounted in one of the disk drives. (Note that a system disk 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 User's Guide*. Continuation lines can be specified by using the ampersand (&) as a continuation character. The ampersand can be used any place there is a space or other delimiter.

The invocation command has the general form

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

where

- *device* identifies which drive contains the compiler disk. This may be omitted if the compiler disk 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 disk 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 disk 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 disk 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 disk containing the input file. These files have the same file name as the input file, but have the extensions LST for the listing and OBJ for the object code. For example, if the compiler is invoked by

```
PLM86 :F1:MYPROG.SRC
```

the listing and all other printed output is written to :F1:MYPROG.LST and the object code to :F1:MYPROG.OBJ. If these files already exist they are overwritten. If they do not exist the compiler creates them.

The object code file may be used as input to the ISIS-II relocation and linkage facilities. (See the *8086 Family Utilities User's Guide for 8080/8085-Based Development Systems*.)

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 disk 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 disk 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 disk containing these files may be mounted in any disk 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
TYPE/NOTYPE	TYPE
OPTIMIZE	OPTIMIZE(1)
DATE	no date
TITLE	module name
PAGEWIDTH	PAGEWIDTH(120)
PAGELENGTH	PAGELENGTH(60)
INTVECTOR/NOINTVECTOR	INTVECTOR
WORKFILES	WORKFILES(:F1:,:F1:)
RAM/ROM	RAM
SMALL/COMPACT/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 A.

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 characters not containing parentheses. The *date* appears in the heading of all pages of listing output exactly as given in the DATE control, except that if more than nine characters are specified, only the first nine characters are printed.

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
TYPE / NOTYPE

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, 2, or 3.

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

OPTIMIZE(0)

OPTIMIZE(0) specifies no extensive optimization beyond "folding" of constant expressions and short-circuit evaluation of Boolean expressions.

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

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

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

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

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

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

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

is evaluated in full.

OPTIMIZE(1)

OPTIMIZE(1) specifies strength reduction plus elimination of common subexpressions, in addition to the above optimizations of level (0).

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

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

Example:
```
A = B + C*D/3
C = E + D*C/3
```

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

OPTIMIZE(2)

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

- machine code optimizations (e.g., short jumps, moves)
- elimination of superfluous branches
- re-use of duplicate code
- removal of unreachable code and reversal of branch-condition

Optimizing machine code means using shorter forms for identical machine instructions, to save space. This is possible because the 8086 has multiple forms for some of its instructions. For example:

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

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

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

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

will be transformed into

```
        JNZ     LAB2    ;Jump on non-zero to LAB2
LAB1:   .....
        ...
        ...
LAB2:   .....
```

Similarly, multiple branches like the following are eliminated:

```
LAB0:   JMP     LAB1
        ...
        ...
LAB1:   JMP     LAB2
        ...
        ...
LAB2:   .....
```

is transformed into

```
LAB0:   JMP     LAB2
        ...
        ...
LAB1:   JMP     LAB2
        ...
        ...
LAB2:   .....
```

Reuse of duplicate code can refer to identical code at the end of two converging paths. In such a case the code is inserted in only one path, and a jump to that path is inserted in the other path.

Example:

```
DECLARE A BYTE, SPOT POINTER ;
DECLARE S BASED SPOT STRUCTURE (B BYTE, C BYTE) ;
IF A = 1 THEN
      S.C = INPUT (0F7H) AND 07FH ;
ELSE
      S.C = INPUT (0F9H) and 07FH ;
```

Before

```
        CMP     A,1H
        JZ      $+5H
        JMP     @1
        IN      0F7H
        AND     AL, 7FH
        MOV     BX, SPOT
        MOV     S [BX+1H], AL
        JMP     @2
@1:     IN      0F9H
        AND     AL, 7FH
        MOV     BX, SPOT
        MOV     S [BX+1H], AL
@2:
```

After

```
        CMP     A,1H
        JNZ     @1

        IN      0F7H
        JMP     @2


@1:     IN      0F9H
@2:     AND     AL, 7FH
        MOV     BX, SPOT
        MOV     S [BX+1H], AL
```

Reuse of duplicate code can also refer to machine instructions immediately preceding a loop being identical to those ending the loop. A branch can be generated to re-use the code generated at the beginning of the loop. For example:

Before

```
        ADD   AX, BX
        MOV   ANS, AX
LAB0    MOV   AL, DUM1
        CMP   AL, DUM2
        JNZ   LAB1
        ...
        ...
        ADD   AX, BX
        MOV   ANS, AX
        JMP   LAB0
LAB1:   .....
```

After

```
LAB0:   ADD   AX, BX
        MOV   ANS, AX
        MOV   AL, DUM1
        CMP   AL, DUM2
        JNZ   LAB1
        ...
        ...
        JMP   LAB0
LAB1:   ....
```

This is safe so long as LAB0 is not the target of a jump instruction. The compiler normally handles a whole procedure at a time, and is thus aware of such a condition. The optimization cannot be safely applied to labels in the outer level of the main program module. Under these restrictions, then, this optimization does not alter the resulting operation and results in saving space.

The optimization which removes unreachable code takes a second look at the generated object code, finding areas which can never be reached due to the control structures created earlier.

Example: If the following code were generated before optimization

```
        MOV   AX, A
        RCR   AC, 1
        JB    @1
        JMP   @2

@1:     MOV   AX, 0FFFFH
        OUTW  0F6H
        JMP   @2
        MOV   AX, B
        ADD   A, AX
        JMP   @3
@2:     ....
        ....
        ....
@3:     ......
        ......
```

then the removal of unreachable code would produce

```
        MOV   AX, A
        RCR   AC, 1
        JB    @1
        JMP   @2

@1:     MOV   AX, 0FFFFH
        OUTW  0F6H
        JMP   @2
@2:     ...
        ...
@3:     .....
```

This can be further optimized by reversing the branch condition in the third instruction and removing the unnecessary JMP @2:

```
        MOV     AX, A
        RCR     AL, 1
        JNB     @2

@1:     MOV     AX, 0FFFFH
        OUTW    0F6H
@2:     ...
        ...
@3:     ......
```

OPTIMIZE(3)

OPTIMIZE(3) includes all of the above optimizations. It also optimizes indeterminate storage operations (e.g., those using based variables) and pointer comparisons. The two assumptions validating these new optimizations are that based variables are not overlaying user-declared variables, and that segments are not overlapped.

The benefits of this optimization level include more efficient use of code space because the user guarantees he has not caused an overlay of needed values. Faster execution of pointer comparisons is a consequence of the user guaranteeing there are no overlapped segments.

The first guarantee is a consequence of user caution in variable-declaration and usage. For example the sequence

```
DECLARE (I,J) WORD;
DECLARE THETA (19) AT (@I) ;
DECLARE A BASED J (10)
            STRUCTURE (F1 BYTE,F2 WORD);
..
J=.I
....
..
A(I).F1 = 7;
A(I).F2 = 99;
THETA(I) = 31;
..
..
```

violates this caution and guarantee because it causes the values being used as pointers and subscripts to be overlaid. The compiler normally takes steps to avoid the difficulties implied here, but in OPTIMIZE(3) these steps are omitted due to the implicit user guarantee that such situations have been avoided.

OPTIMIZE(3) also changes the way POINTER values are compared. The normal case in comparing PTR_1 and PTR_2 is this: for each pointer, the segment word is effectively multiplied by 16 and the offset word is added, giving a full 20-bit address. The two 20-bit addresses are compared and the correct result is returned.

These manipulations are not needed under this level of optimization due to the implicit guarantee that no segments overlap. Thus it is sufficient to compare the segment parts bit for bit in order to determine which is a lower number. Only if the segment parts are equal is it necessary to compare the offset parts. Pointer comparisons are therefore faster under this level of optimization.

The second guarantee mentioned above required no special action unless the AT attribute and the segment-locating controls of LINK86 and LOC86 are invoked. Users exercising these controls must consider carefully their full effects. If segments are overlapped and pointer comparison is used in the program, this optimization level must not be used.

Figures 3-1 through 3-4 illustrate the levels of optimization described above.

```
ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F7:EXMPLE.OBJ
COMPILER INVOKED BY:  PLM86 :F7:EXMPLE.SRC NOPAGING COMPACT CODE OPTIMIZE(0)

   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2   1        DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1,PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
   3   1        DO WHILE D(A+B) < D(A+B+1);
                                                    ; STATEMENT # 3
            0004  FA                CLI
            0005  2E8E160000        MOV     SS,CS:@@STACK$FRAME
            000A  BC0200            MOV     SP,@@STACK$OFFSET
            000D  8BEC              MOV     BP,SP
            000F  2E8E1E0200        MOV     DS,CS:@@DATA$FRAME
            0014  FB                STI
                        @3:
            0015  8B1E0200          MOV     BX,B
            0019  031E0000          ADD     BX,A
            001D  D1E3              SHL     BX,1
            001F  8B360200          MOV     SI,B
            0023  03360000          ADD     SI,A
            0027  D1E6              SHL     SI,1
            0029  8B870600          MOV     AX,D[BX]
            002D  3B840800          CMP     AX,D[SI+2H]
            0031  7203              JB      $+5H
            0033  E97700            JMP     @4
   4   2          IF PTR_1 = PTR_2 THEN DO;
                                                    ; STATEMENT # 4
            0036  C406CE00          LES     AX,PTR_1
            003A  06                PUSH    ES      ; 1
            003B  C416D200          LES     DX,PTR_2
            003F  8CC7              MOV     DI,ES
            0041  5E                POP     SI      ; 1
            0042  B104              MOV     CL,4H
            0044  8BD8              MOV     BX,AX
            0046  D3EB              SHR     BX,CL
            0048  03F3              ADD     SI,BX
            004A  8BDA              MOV     BX,DX
            004C  D3EB              SHR     BX,CL
            004E  03FB              ADD     DI,BX
            0050  3BF7              CMP     SI,DI
            0052  7507              JNE     $+9H
            0054  240F              AND     AL,0FH
            0056  80E20F            AND     DL,0FH
            0059  3AC2              CMP     AL,DL
            005B  7403              JZ      $+5H
            005D  E94100            JMP     @1
   6   3                  A = A*2;
                                                    ; STATEMENT # 6
            0060  8B060000          MOV     AX,A
            0064  D1E0              SHL     AX,1
            0066  89060000          MOV     A,AX
   7   3                ABASED(A) = ABASED(B);
                                                    ; STATEMENT # 7
            006A  8B360200          MOV     SI,B
            006E  D1E6              SHL     SI,1
            0070  8B3E0000          MOV     DI,A
            0074  D1E7              SHL     DI,1
            0076  C41ECE00          LES     BX,PTR_1
            007A  268B00            MOV     AX,ES:[BX].ABASED[SI]
            007D  C41ECE00          LES     BX,PTR_1
            0081  268901            MOV     ES:[BX].ABASED[DI],AX
   8   3                ABASED(B) = ABASED(C);
                                                    ; STATEMENT # 8
            0084  8B360400          MOV     SI,C
            0088  D1E6              SHL     SI,1
            008A  8B3E0200          MOV     DI,B
            008E  D1E7              SHL     DI,1
            0090  C41ECE00          LES     BX,PTR_1
            0094  268B00            MOV     AX,ES:[BX].ABASED[SI]
            0097  C41ECE00          LES     BX,PTR_1
            009B  268901            MOV     ES:[BX].ABASED[DI],AX
   9   3                END;
                                                    ; STATEMENT # 9
            009E  E90900            JMP     @2
                        @1:
  10   2             ELSE A = A+1;
                                                    ; STATEMENT # 10
            00A1  8B060000          MOV     AX,A
            00A5  40                INC     AX
            00A6  89060000          MOV     A,AX
                        @2:
  11   2           END;
                                                    ; STATEMENT # 11
            00AA  E968FF            JMP     @3
                        @4:
  12   1       END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 12
            00AD  FB                STI
            00AE  F4                HLT

MODULE INFORMATION:

     CODE AREA SIZE     = 00AFH    175D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

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

```
ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F7:EXMPLE.OBJ
COMPILER INVOKED BY:  PLM86 :F7:EXMPLE.SRC NOPAGING COMPACT CODE OPTIMIZE(1)


  1          EXAMPLES_OF_OPTIMIZATIONS: DO;
  2   1         DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1,PTR_2) POINTER,
                       ABASED BASED PTR_1 (10) WORD;
  3   1         DO WHILE D(A+B) < D(A+B+1);
                                                  ; STATEMENT # 3
            0004  FA              CLI
            0005  2E8E160000      MOV     SS,CS:@@STACK$FRAME
            000A  BC0200          MOV     SP,@@STACK$OFFSET
            000D  8BEC            MOV     BP,SP
            000F  2E8E1E0200      MOV     DS,CS:@@DATA$FRAME
            0014  FB              STI
                        @3:
            0015  8B1E0200        MOV     BX,B
            0019  031E0000        ADD     BX,A
            001D  D1E3            SHL     BX,1
            001F  8B870600        MOV     AX,D[BX]
            0023  3B870800        CMP     AX,D[BX+2H]
            0027  7203            JB      $+5H
            0029  E96800          JMP     @4
  4   2           IF PTR_1 = PTR_2 THEN DO;
                                                  ; STATEMENT # 4
            002C  C406CE00        LES     AX,PTR_1
            0030  06              PUSH    ES      ; 1
            0031  C416D200        LES     DX,PTR_2
            0035  8CC7            MOV     DI,ES
            0037  5E              POP     SI      ; 1
            0038  B104            MOV     CL,4H
            003A  8BD8            MOV     BX,AX
            003C  D3EB            SHR     BX,CL
            003E  03F3            ADD     SI,BX
            0040  8BDA            MOV     BX,DX
            0042  D3EB            SHR     BX,CL
            0044  03FB            ADD     DI,BX
            0046  3BF7            CMP     SI,DI
            0048  7507            JNE     $+9H
            004A  240F            AND     AL,0FH
            004C  80E20F          AND     DL,0FH
            004F  3AC2            CMP     AL,DL
            0051  7403            JZ      $+5H
            0053  E93700          JMP     @1
  6   3                 A = A*2;
                                                  ; STATEMENT # 6
            0056  8B060000        MOV     AX,A
            005A  D1E0            SHL     AX,1
            005C  89060000        MOV     A,AX
  7   3               ABASED(A) = ABASED(B);
                                                  ; STATEMENT # 7
            0060  8B360200        MOV     SI,B
            0064  D1E6            SHL     SI,1
            0066  D1E0            SHL     AX,1
            0068  C41ECE00        LES     BX,PTR_1
            006C  268B08          MOV     CX,ES:[BX].ABASED[SI]
            006F  89C6            MOV     SI,AX
            0071  268908          MOV     ES:[BX].ABASED[SI],CX
  8   3               ABASED(B) = ABASED(C);
                                                  ; STATEMENT # 8
            0074  8B360400        MOV     SI,C
            0078  D1E6            SHL     SI,1
            007A  8B3E0200        MOV     DI,B
            007E  D1E7            SHL     DI,1
            0080  C41ECE00        LES     BX,PTR_1
            0084  268B00          MOV     AX,ES:[BX].ABASED[SI]
            0087  268901          MOV     ES:[BX].ABASED[DI],AX
  9   3               END;
                                                  ; STATEMENT # 9
            008A  E90400          JMP     @2
                        @1:
 10   2             ELSE A = A+1;
                                                  ; STATEMENT # 10
            008D  FF060000        INC     A
                        @2:
 11   2           END;
                                                  ; STATEMENT # 11
            0091  E981FF          JMP     @3
                        @4:
 12   1       END EXAMPLES_OF_OPTIMIZATIONS;
                                                  ; STATEMENT # 12
            0094  FB              STI
            0095  F4              HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 0096H    150D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

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

```
ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F7:EXMPLE.OBJ
COMPILER INVOKED BY:  PLM86 :F7:EXMPLE.SRC NOPAGING COMPACT CODE OPTIMIZE(2)


   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2   1         DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1,PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
   3   1         DO WHILE D(A+B) < D(A+B+1);
                                                    ; STATEMENT # 3
          0004  FA                CLI
          0005  2E8E160000        MOV     SS,CS:@@STACK$FRAME
          000A  BC0200            MOV     SP,@@STACK$OFFSET
          000D  8BEC              MOV     BP,SP
          000F  2E8E1E0200        MOV     DS,CS:@@DATA$FRAME
          0014  FB                STI
                          @3:
          0015  8B1E0200          MOV     BX,B
          0019  031E0000          ADD     BX,A
          001D  D1E3              SHL     BX,1
          001F  8B870600          MOV     AX,D[BX]
          0023  3B870800          CMP     AX,D[BX+2H]
          0027  7361              JNB     @4
   4   2            IF PTR_1 = PTR_2 THEN DO;
                                                    ; STATEMENT # 4
          0029  C406CE00          LES     AX,PTR_1
          002D  06                PUSH    ES      ; 1
          002E  C416D200          LES     DX,PTR_2
          0032  8CC7              MOV     DI,ES
          0034  5E                POP     SI      ; 1
          0035  B104              MOV     CL,4H
          0037  8BD8              MOV     BX,AX
          0039  D3EB              SHR     BX,CL
          003B  03F3              ADD     SI,BX
          003D  8BDA              MOV     BX,DX
          003F  D3EB              SHR     BX,CL
          0041  03FB              ADD     DI,BX
          0043  3BF7              CMP     SI,DI
          0045  7507              JNE     $+9H
          0047  240F              AND     AL,0FH
          0049  80E20F            AND     DL,0FH
          004C  3AC2              CMP     AL,DL
          004E  7534              JNZ     @1
   6   3                  A = A*2;
                                                    ; STATEMENT # 6
          0050  A10000            MOV     AX,A
          0053  D1E0              SHL     AX,1
          0055  A30000            MOV     A,AX
   7   3                ABASED(A) = ABASED(B);
                                                    ; STATEMENT # 7
          0058  8B360200          MOV     SI,B
          005C  D1E6              SHL     SI,1
          005E  D1E0              SHL     AX,1
          0060  C41ECE00          LES     BX,PTR_1
          0064  268B08            MOV     CX,ES:[BX].ABASED[SI]
          0067  89C6              MOV     SI,AX
          0069  268908            MOV     ES:[BX].ABASED[SI],CX
   8   3                ABASED(B) = ABASED(C);
                                                    ; STATEMENT # 8
          006C  8B360400          MOV     SI,C
          0070  D1E6              SHL     SI,1
          0072  8B3E0200          MOV     DI,B
          0076  D1E7              SHL     DI,1
          0078  C41ECE00          LES     BX,PTR_1
          007C  268B00            MOV     AX,ES:[BX].ABASED[SI]
          007F  268901            MOV     ES:[BX].ABASED[DI],AX
   9   3                END;
                                                    ; STATEMENT # 9
          0082  EB91              JMP     @3
                          @1:
  10   2            ELSE A = A+1;
                                                    ; STATEMENT # 10
          0084  FF060000          INC     A
  11   2           END;
                                                    ; STATEMENT # 11
          0088  EB8B              JMP     @3
                          @4:
  12   1       END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 12
          008A  FB                STI
          008B  F4                HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 008CH    140D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

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

```
ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN :F7:EXMPLE.OBJ
COMPILER INVOKED BY:  PLM86 :F7:EXMPLE.SRC NOPAGING COMPACT CODE OPTIMIZE(3)


   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2   1         DECLARE (A,B,C) WORD, D(100) WORD, (PTR_1,PTR_2) POINTER,
                        ABASED BASED PTR_1 (10) WORD;
   3   1         DO WHILE D(A+B) < D(A+B+1);
                                                    ; STATEMENT # 3
            0004  FA              CLI
            0005  2E8E160000      MOV     SS,CS:@@STACK$FRAME
            000A  BC0200          MOV     SP,@@STACK$OFFSET
            000D  8BEC            MOV     BP,SP
            000F  2E8E1E0200      MOV     DS,CS:@@DATA$FRAME
            0014  FB              STI
                        @3:
            0015  8B1E0200        MOV     BX,B
            0019  031E0000        ADD     BX,A
            001D  D1E3            SHL     BX,1
            001F  8B870600        MOV     AX,D[BX]
            0023  3B870800        CMP     AX,D[BX+2H]
            0027  7346            JNB     @4
   4   2            IF PTR_1 = PTR_2 THEN DO;
                                                    ; STATEMENT # 4
            0029  C406CE00        LES     AX,PTR_1
            002D  06              PUSH    ES      ; 1
            002E  C416D200        LES     DX,PTR_2
            0032  8CC7            MOV     DI,ES
            0034  5E              POP     SI      ; 1
            0035  3BF7            CMP     SI,DI
            0037  7502            JNE     $+4H
            0039  3BC2            CMP     AX,DX
            003B  752C            JNZ     @1
   6   3                A = A*2;
                                                    ; STATEMENT # 6
            003D  A10000          MOV     AX,A
            0040  D1E0            SHL     AX,1
            0042  A30000          MOV     A,AX
   7   3                ABASED(A) = ABASED(B);
                                                    ; STATEMENT # 7
            0045  8B360200        MOV     SI,B
            0049  D1E6            SHL     SI,1
            004B  D1E0            SHL     AX,1
            004D  C41ECE00        LES     BX,PTR_1
            0051  268B08          MOV     CX,ES:[BX].ABASED[SI]
            0054  56              PUSH    SI      ; 1
            0055  8BC6            MOV     SI,AX
            0057  268908          MOV     ES:[BX].ABASED[SI],CX
   8   3                ABASED(B) = ABASED(C);
                                                    ; STATEMENT # 8
            005A  8B360400        MOV     SI,C
            005E  D1E6            SHL     SI,1
            0060  268B00          MOV     AX,ES:[BX].ABASED[SI]
            0063  5E              POP     SI      ; 1
            0064  268900          MOV     ES:[BX].ABASED[SI],AX
   9   3                END;
                                                    ; STATEMENT # 9
            0067  EBAC            JMP     @3
                        @1:
  10   2             ELSE A = A+1;
                                                    ; STATEMENT # 10
            0069  FF060000        INC     A
  11   2           END;
                                                    ; STATEMENT # 11
            006D  EBA6            JMP     @3
                        @4:
  12   1       END EXAMPLES_OF_OPTIMIZATIONS;
                                                    ; STATEMENT # 12
            006F  FB              STI
            0070  F4              HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 0071H    113D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     12 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-4. Sample Program Showing the OPTIMIZE (3) Control

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.

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 statement number and relative address of each source program statement, information about each local symbol including based symbols and procedure parameters, and block information for each procedure. This information may be used later for symbolic debugging by an ICE-86 or ICE-88 emulator.

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

3.5.6 TYPE/NOTYPE

These are primary controls. They have the form:

TYPE

NOTYPE

Default: TYPE

The TYPE control specifies that the object module is to contain information on the types of the variables output in symbols records. This information may be used later for type checking by LINK86, or an ICE-86 and ICE-88 emulator.

The NOTYPE control specifies that such type definitions are not to be placed in the object module.

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 disk 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 RAM/ROM Control

This primary control directs the object-module placement of all constants, both user-defined and compiler-generated. Its form is

```
RAM
ROM

Default: RAM
```

The default setting places the CONSTANT section within the DATA segment in all segmentation models (sizes) except LARGE, in which constants are placed in the CODE segment instead.

The ROM setting places constants in the CODE segment. Under this setting, the INITIAL attribute on a variable produces a warning message. The dot operator is not advised for use under the ROM option. If SMALL is also specified, then pointers will be four bytes instead of two. (See also section 8.5.)

3.9 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

COMPACT

MEDIUM

LARGE

Default: SMALL

See Chapters 4 and 5 for discussions of the output of the compiler and of programming restrictions under each size control.

3.9.1 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.

3.9.2 COMPACT

The COMPACT control provides for programs with the following space requirements:

- Not more than 64K bytes total for code sections from all modules
- Not more than 64K bytes total for data and constant sections for all modules
- Not more than 64K bytes total for stack sections from all modules
- Not more than 64K bytes total for memory sections from all modules

See Chapters 4 and 5 for details.

3.9.3 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.

3.9.4 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.

3.10 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 3-5 shows an example program using the conditional compilation controls, while Figure 3-6 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 V1.2 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 = 0006H       6D
     29 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-5. Sample Program Showing the SET(DEBUG=) Control

```
PL/M-86 COMPILER    EXAMPLE

ISIS-II PL/M-86 V1.2 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 = 0006H      6D
     29 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 3-6. Sample Program Showing the NOCOND Control

3.10.1 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).

3.10.2 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.

3.10.3 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 LINK86 and LOC86. 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 and in ROM—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, COMPACT, MEDIUM, or LARGE).

4.1 Code Section

This section contains the object code generated by the source program. If either the LARGE control or the ROM control is used, this section also contains the information that would otherwise be in the constant section.

In addition, the code section for the main program module contains a "main program prologue" generated by the compiler. This code precedes the code compiled from the source program, and sets the CPU up for program execution by initializing various registers and enabling interrupts.

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 or ROM controls are 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, COMPACT, 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.

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 PUBLIC AT(100);
   ```

   is not allowed.

5. If the ROM option is used with SMALL, then pointers will be four bytes instead of two. (See also section 8.5.) Use of the INITIAL attribute under the ROM option produces a warning message.

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. The dot operator is provided for this purpose, for use under the SMALL and RAM options only. Its results may not be appropriate if this restriction is not observed, i.e., if used under other options.

5.3 The COMPACT Case

A program compiled with the COMPACT control has four segments: code, data, stack, and memory. Each of these is the result of combining the same-type sections from all modules, and each has a maximum size of 64K bytes. The constant sections from all modules are merged into the data segment unless the ROM control is used, which causes all constant sections to be merged into the CODE segment instead. Since the code, data, and stack segments are fully defined by the time the program is loaded, the segment base addresses in CS, DS, and SS registers are never changed. All code and a few prologue constants are addressed relative to CS. All data except absolute data (declared with the AT (constant) attribute) are addressed relative to DS. The stack is addressed relative to SS. ES is not initialized and can change during execution. References to any location require only a 16-bit offset address against these segment base addresses.

The sole programming restriction in the COMPACT case is that PUBLIC variables may not be declared AT an absolute location, e.g.,

```
DECLARE ANVIL BYTE PUBLIC AT (100)
```

is not allowed. This restriction does not apply when the "location" within the AT attribute is formed with the @ operator, i.e., DECLARE ANVIL BYTE PUBLIC AT (@HAMR); is valid. However, the phrases "@ MEMORY" and ".MEMORY" are undefined and not allowed.

Use of the INITIAL attribute under the ROM option produces a warning message.

5.4 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.

5.4.1 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 PUBLIC AT(100);
```

   is not allowed.

Restrictions 1 and 2 arise from the fact that the code segment address may change during program execution. Restriction 3 is the same as Restriction 4 in the SMALL case, and arises for the same reason.

4. Use of the INITIAL attribute under the ROM option produces a warning message.

5.5 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.

5.5.1 Programming Restrictions in the LARGE Case

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

1. When a PUBLIC or EXTERNAL procedure is indirectly activated, a POINTER variable must be used in the CALL statement. 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*/
```

3. Use of the INITIAL attribute under the ROM option produces a warning message.

CHAPTER 6
FLOATING-POINT LINKAGE

Programming considerations for the use of REAL arithmetic in PL/M-86 are explained in the *PL/M-86 Programming Manual for 8080/8085-Based Development Systems*.

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

These linkage specifications make available to your program the libraries of floating-point functions. The circumstances determining which library is appropriate are given in Table 6-1. The libraries themselves are discussed briefly below the table.

Table 6-1. Linkage Choices For REAL-Math Usage

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

The Interface libraries do the following:

- 8087.LIB resolves external references inserted by the translator of an 8086 program so that floating-point instructions will correctly invoke the 8087 chip. 8087.LIB is the library of floating-point functions written for the chip itself rather than for emulation.
- E8087LIB resolves such references to invoke the Emulator software instead of the actual 8087 chip.

Emulation is performed by E8087 or PE8087.

- E8087 is the actual library of emulation routines, which provide every function and feature of an actual 8087 chip except speed. Emulation is invoked automatically as needed, using interrupts 20 through 31.
- PE8087 is a subset of E8087. The REAL arithmetic performed in PL/M-86 programs does not require the complete set of routines in the full Emulator. Use of the subset saves substantial space.
- WARNING: The 8087 Emulator processes exceptions exactly as the 8087 does. However, if your 8086/8087 implementation includes some external interrupt masking device such as an 8259A, this external device cannot be simulated by the 8087 Emulator. An Interrupt 16 will occur after the execution of any instruction when the 8087 interrupt is active and the 8086 interrupt is enabled.

Examples:

Suppose you write a PL/M-86 program called EASY that, at first, uses no REAL math at all. No interface library is needed. As modules are added during the development process, a PL/M-86 calculation routine called ACURAT is supplied, and you revise EASY to call it.

If you have no 8087 chip installed in your system, the correct linking statement for the above conditions would be

```
LINK86 ACURAT.OBJ, EASY.OBJ, E8087.LIB, PE8087
```

However, if ACURAT were written in some other language such as FORTRAN86 or ASM86, the following command should be used instead:

```
LINK86 ACURAT.OBJ, EASY.OBJ, E8087.LIB, E8087
```

If you DO have an actual 8087 chip installed in your system, then the two examples above should become

```
LINK86 ACURAT.OBJ, EASY.OBJ, 8087.LIB
```

More detailed and advanced discussions of the features and functions of the 8086 utilities appear in the manual titled *8086 Family Utilities User's Guide For 8080/8085-Based Development Systems*, order number 9800639.

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                              28 JUN 79  PAGE   1

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE STACK
OBJECT MODULE PLACED IN :F1:STACK.OBJ
COMPILER INVOKED BY:  PLM86 :F1:STACK.SRC PAGEWIDTH(80) CODE XREF TITLE('STACK M
                      -ODULE') DATE(28 JUN 79)


   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  8BEC                  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  FEC0                  INC     AL
        0009  88066400              MOV     T,AL
        000D  B400                  MOV     AH,0H
        000F  89C3                  MOV     BX,AX
        0011  8A4E04                MOV     CL,[BP].B
        0014  888F0000              MOV     S[BX],CL
   6    2        END PUSH;
                                                ; STATEMENT # 6
        0018  5D                    POP     BP
        0019  C20200                RET     2H
                           PUSH      ENDP

   7    1        POP: PROCEDURE BYTE PUBLIC; /*Returns value popped from stack*/
                                                ; STATEMENT # 7
                           POP       PROC NEAR
        001C  55                    PUSH    BP
        001D  8BEC                  MOV     BP,SP
   8    2           RETURN S((T:=T-1)+1); /*Decrement T, then return S(T+1)*/
                                                ; STATEMENT # 8
        001F  8A066400              MOV     AL,T
        0023  FEC8                  DEC     AL
        0025  88066400              MOV     T,AL
        0029  B400                  MOV     AH,0H
        002B  89C3                  MOV     BX,AX
        002D  8A870100              MOV     AL,S[BX+1H]
        0031  5D                    POP     BP
        0032  C3                    RET
   9    2        END POP;
                                                ; STATEMENT # 9
                           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 in a pseudo-assembly language format:

1. Location counter (hexadecimal notation)
2. Resultant binary code (hexadecimal notation)
3. Label field
4. Opcode mnemonic
5. Symbolic arguments
6. Comment field

Not all six of these columns will appear on any one line of the code listing. Compiler generated labels (e.g. those which mark the beginning and ending of a DO WHILE loop) are preceded by "@". The comments appearing on PUSH and POP instructions indicate the stack depth associated with the stack instruction.

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).

The AUTOMATIC attribute indicates that the identifier was declared as a parameter or as a local variable in a REENTRANT procedure, and therefore is allocated dynamically on the stack.

Figure 7-2 is an example of the 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    STACK MODULE                          28 JUN 79  PAGE   2

CROSS-REFERENCE LISTING
-----------------------

  DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
 ----- ------ -----  ------------------------------

     3  0004H     1  B                     BYTE PARAMETER AUTOMATIC
                                              4     5

.    7  001CH    23  POP                   PROCEDURE BYTE PUBLIC STACK=0002H

     3  0000H    28  PUSH                  PROCEDURE PUBLIC STACK=0004H

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

     1  0000H        STACK                 PROCEDURE STACK=0000H

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

Figure 7-2. Cross-Reference Listing

```
MODULE INFORMATION:

     CODE AREA SIZE     = 0033H     51D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0065H    101D
     MAXIMUM STACK SIZE = 0004H      4D
     16 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 14 of the *PL/M-86 Programming Manual for 8080/8085-Based Development Systems*.

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. Under the ROM option, however, it follows the rules below.

In the COMPACT, 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.

8.6 Structures

The maximum number of elements in a structure is 64. As described in the *PL/M-86 Programming Manual for 8080/8085-Based Development Systems*, each of these elements may be an array of arbitrary size (though the 64K byte limit on segment size applies to the total storage allocation). The structure may be a member of an array.

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. A POINTER parameter value in the COMPACT, MEDIUM, and LARGE cases consists of a segment address and an offset. The 16-bit segment address is pushed first, and then the 16-bit offset is pushed.See Chapter 8 for details on data representations.

After the parameters are passed, the CALL instruction places the return address on the stack. In the SMALL and COMPACT cases, this is a 16-bit offset (the contents of the IP register) and occupies two contiguous bytes on the stack.

In the MEDIUM and LARGE cases, the type of the return address depends on whether the procedure is local or public. The return address for a local procedure, like any return address for the SMALL case, is a 16-bit offset and occupies two contiguous bytes on the stack. For a public procedure in the MEDIUM or LARGE case, the return address is a POINTER value consisting of a segment address and an offset, and is passed in the same way as a POINTER parameter. The 16-bit segment address (contents of the CS register) is pushed first, and then the 16-bit offset (IP register contents) is pushed.

Control is then passed to the code of the procedure, by updating the IP register 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.

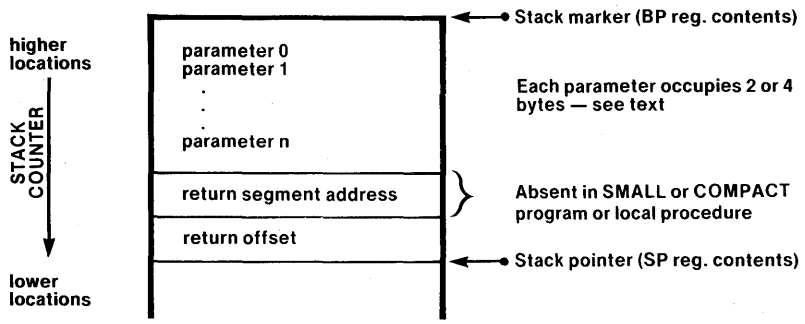

Figure 9-1. Stack Layout at Point Where a Non-Interrupt Procedure Is Activated 478-1

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, COMPACT, and MEDIUM cases because the data segment does not change.)

2. If any parameter of the procedure is referenced by a nested procedure, all parameters are removed from the stack and placed in space reserved for them in the data segment.

3. The stack marker offset (BP register contents) is placed on the stack, and the current stack pointer (SP register contents) is used to update the BP register.

4. If the procedure has the REENTRANT attribute, space is reserved on the stack for any variables declared within the procedure (this does not include based variables, variables with the DATA attribute, or variables with the AT attribute).

Control then passes to the code compiled from the executable statements in the procedure body. At this point, the stack layout is as shown in Figure 9-2.

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

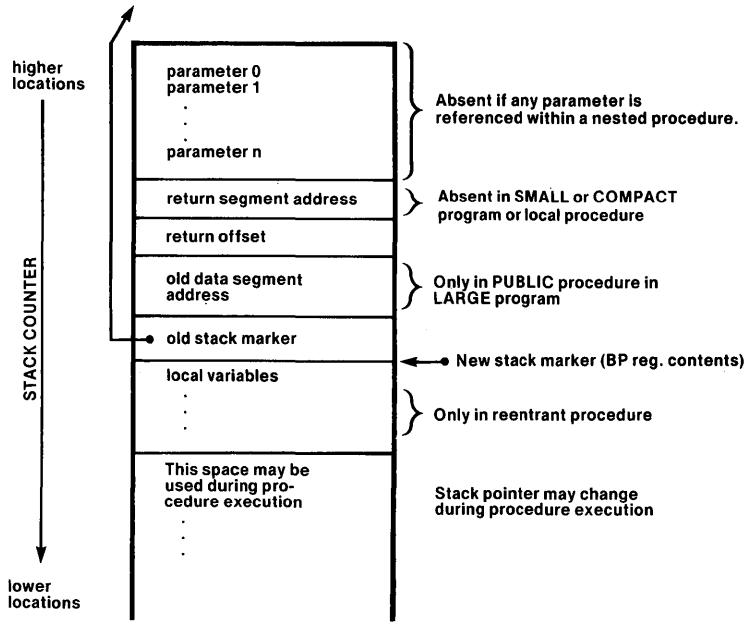

Figure 9-2. Stack Layout During Execution of Non-Interrupt Procedure Body 478-2

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 or local variables during procedure execution, the stack pointer (SP register) is loaded with the stack marker (BP register contents). This has the effect of discarding the temporary storage.

2. The old stack marker is restored by popping the stored value from the stack into the BP register.

3. If the procedure has the PUBLIC attribute *and* the program size is LARGE, the old data segment address is restored by popping the stored value from the stack into the DS register.

4. A RET instruction is used to return from the procedure. If the program size is SMALL, the RET pops the stored return address (a 16-bit offset) into the IP register. It also discards any parameters stored on the stack.

If the program size is MEDIUM or LARGE and the procedure is local, the RET performs the same actions described above for a return in the SMALL or COMPACT case. If the program size is MEDIUM or LARGE and the procedure is public, the RET pops the stored return-address offset from the stack into the IP register and then pops the return-address segment address into the CS register. It also discards any parameters stored on the stack.

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 (COMPACT size)	ES and BX Registers
POINTER (MEDIUM size)	ES and BX Registers
POINTER (LARGE size)	ES and BX Registers
REAL	Top of RMU stack

*Under the ROM option, the result is returned in ES and BX registers.

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 or control is transferred to an interrupt vector by the CAUSE$INTERRUPT statement (see Section 9.2.6 of the *PL/M-86 Programming Manual for 8080/8085-Based Development Systems*).

Note that the CPU does not respond to this signal unless interrupts are enabled. The "main program prologue" (code inserted by the compiler at the beginning of the main program) enables interrupts.

NOTE

If you require your program to begin with interrupts disabled, simply start with the instruction DISABLE;. Since the 8086 processor does not actually allow an interrupt to occur until the first machine instruction following the enabling instruction has been processed, the resulting code sequence will not allow any maskable interrupts to occur.

If interrupts are enabled, the following actions take place:

1. The CPU 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.

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.

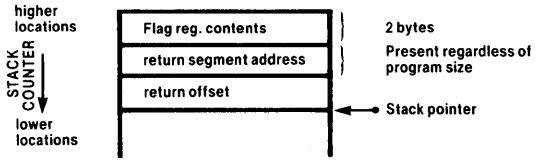

Figure 10-1. Stack Layout at Point Where an Interrupt Procedure Gains Control 478-3

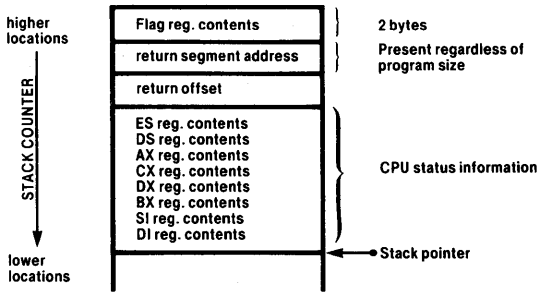

Figure 10-2. Stack Layout After Interrupt Procedure Preface and Before Procedure Prologue 478-4

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.

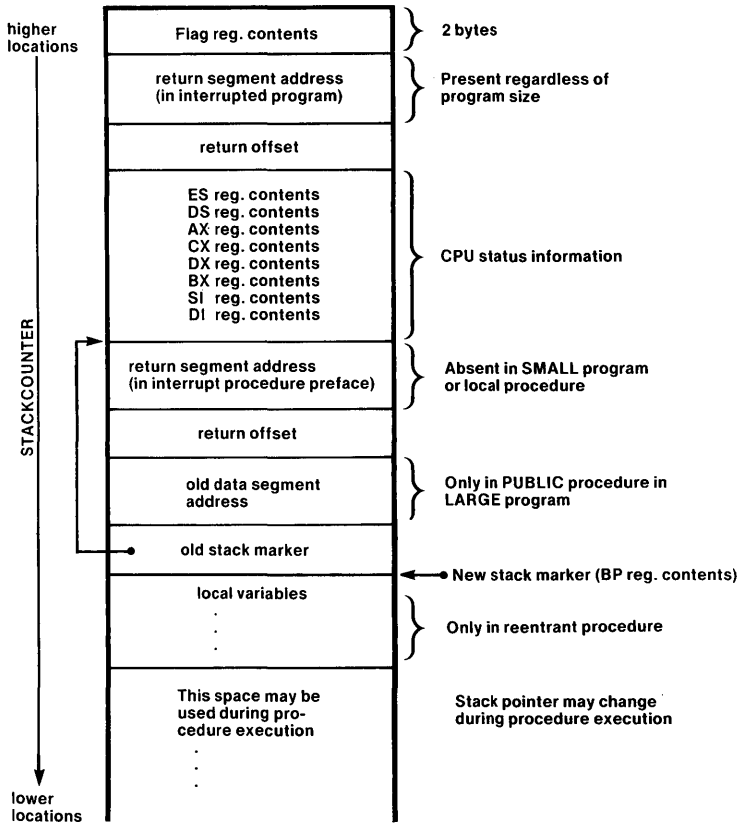

Figure 10-3. Stack Layout During Execution of Interrupt Procedure Body 478-5

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.
18. Pop the stack into the ES register.
19. Execute an IRET instruction to return from the interrupt procedure. This restores the IP, CS, and flag register contents from the stack.

At this point the stack has been restored to the state it was in before the interrupt occurred, and processing continues normally.

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. The @ operator in PL/M-86 provides access to a procedure's normal (i.e., called) entry point, not to its interrupt entry point. The interrupt entry point first saves the status of the interrupted program before invoking the interrupt procedure through its normal entry point. The exact length of these operations depends on the compilation options chosen, the attributes of the interrupt procedure, and the version of the compiler being used. The builtin function INTERRUPT$PTR can be used during execution to return the actual interrupt entry point. Discussion of this function appears in Chapter 12 of the *PL/M-86 Programming Manual.*

The usefulness of a separately created interrupt vector can be seen by considering an example.

Suppose that two modules for a multimodule program are developed separately. Both use interrupt procedures, but at the time when the modules are written the assignment of interrupt numbers to the various interrupt procedures has not been determined.

The two modules are therefore compiled with the NOINTVECTOR control. When this is done, the *n* in an INTERRUPT *n* attribute is ignored—since normally it would only be used to put the procedure's entry in the proper location within the interrupt vector.

Later, when the program is linked together, a separately created interrupt vector can be linked in. Within this interrupt vector, the placement of the entry for a given interrupt procedure determines which interrupt number will activate that procedure.

Similarly, you could have a library of interrupt procedures, all compiled with NOINTVECTOR. Any program could then have any of these procedures linked in, with a separately created interrupt vector.

The builtin procedure SET$INTERRUPT can be used during execution to create the correct interrupt vector for each interrupt routine. This procedure is discussed in Chapter 12 of the *PL/M-86 Programming Manual.*

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 disk with the IXREF program. This may be omitted if the disk 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.
- A page width of 120 will be used.

Five 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.3.5 The PAGEWIDTH Control

This control has the form

```
PAGEWIDTH(width)
```

where *width* is an unsigned positive 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.

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.

```
PL/M-86 COMPILER    STACK MODULE                                15 MAY 80  PAGE   1

ISIS-II PL/M-86 V2.1 COMPILATION OF MODULE STACK
OBJECT MODULE PLACED IN :F1:STACK.OBJ
COMPILER INVOKED BY:  PLM86 :F1:STACK.SRC PAGEWIDTH(80) CODE XREF &
                      TITLE('STACK MODULE') DATE(15 MAY 80)

   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  8BEC                  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  FEC0                  INC     AL
          0009  88066400              MOV     T.AL
          000D  B400                  MOV     AH,0H
          000F  89C3                  MOV     BX,AX
          0011  8A4F04                MOV     CL,[BP].B
          0014  888F0000              MOV     S[BX],CL
   6  2          END PUSH;
                                              ; STATEMENT # 6
          0018  5D                    POP     BP
          0019  C20200                RET     2H
                           PUSH       ENDP

   7  1          POP: PROCEDURE BYTE PUBLIC; /*Returns value popped from stack*/
                                              ; STATEMENT # 7
                           POP        PROC NEAR
          001C  55                    PUSH    BP
          001D  8BEC                  MOV     BP,SP
   8  2             RETURN S((T:=T-1)+1); /*Decrement T, then return S(T+1)*/
                                              ; STATEMENT # 8
          001F  8A066400              MOV     AL,T
          0023  FEC8                  DEC     AL
          0025  88066400              MOV     T.AL
          0029  B400                  MOV     AH,0H
          002B  89C3                  MOV     BX,AX
          002D  8A870100              MOV     AL,S[BX+1H]
          0031  5D                    POP     BP
          0032  C3                    RET
   9  2          end POP;
                                              ; STATEMENT # 9
                           POP        ENDP

  10  1       END STACK; /*Module ends here*/
                                              ; STATEMENT # 10


PL/M-86 COMPILER    STACK MODULE                                15 MAY 80  PAGE   2

CROSS-REFERENCE LISTING
-----------------------

   DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
  ------ ------ ----- ------------------------------------------

      3  0004H     1  B. . . . . .    BYTE PARAMETER AUTOMATIC
                                          4    5
      7  001CH    23  POP. . . . . .  PROCEDURE BYTE PUBLIC STACK=0002H
      3  0000H    28  PUSH . . . . .  PROCEDURE PUBLIC STACK=0004H
      2  0000H   100  S. . . . . .    BYTE ARRAY(100)        5    8
      1  0000H        STACK. . . . .  PROCEDURE STACK=0000H
      2  0064H     1  T. . . . . .    BYTE PUBLIC INITIAL        5
                                          8

MODULE INFORMATION:

     CODE AREA SIZE     = 0033H     51D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0065H    101D
     MAXIMUM STACK SIZE = 0004H      4D
     16 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure A-1. Intermodule Cross-Reference Listing

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 disk where the source file resides. (A disk is named when it is formatted with the ISIS-II FORMAT command.)

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:

Nesting of MACRO invocations	5
Nesting of INCLUDE controls	5
Number of nested procedures and DO cases	7
Number of labels on a statement	9
Nesting of blocks	18
Number of nested typed procedures	20
Number of elements in a factored list	32
Number of members in a structure	64
Structure size	64K

Numbers of characters in a line	122
Length of a string constant	255
Number of DO blocks in a procedure	255
Number of cases in a DO CASE block	255
Number of active cases	255
Number of EXTERNAL items	255
Number of procedures in a module	255
Segment Size	64K

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; if the explanation of an error makes a reference to a chapter or section without an asterisk, it means this book. If an asterisk follows the reference, it means in the *PL/M-86 Programming Manual for 8080/8085-Based Development Systems*.

Source error message list:

1. INVALID PL/M-86 CHARACTER

   Look near the text flagged for an invalid character, or one that is inappropriate in context. Edit it out or possibly retype the entire statement.

2. UNPRINTABLE ASCII CHARACTER

   Retype the line in question using valid characters.

3. IDENTIFIER, STRING, OR NUMBER TOO LONG, TRUNCATED

   Match your intended variable type with the length of the flagged item. For the correct maximum lengths, see Sections 2.4*, 8.7*, and 14.1*.

4. ILLEGAL NUMERIC CONSTANT TYPE

   This might reflect missing operators, e.g., A=4T instead of 4+T. For the list of valid types, see Section 2.4*.

5. INVALID CHARACTER IN NUMERIC CONSTANT

   For example, 107B and 0ABCD must cause this error because neither can be valid in any PL/M-86 interpretation: 7 is not a binary numeral, B may not occur in decimal or octal, and neither string ends in H. See Section 2.4*.

6. ILLEGAL MACRO REFERENCE, RECURSIVE EXPANSION

   Here is an example causing this error:

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'A';
.
.
.
B=4 ; error discovered here
```

   The error is that no type can be assigned to variables declared circularly, i.e., solely in terms of each other.

7. LIMIT EXCEEDED: MACROS NESTED TOO DEEPLY

   For maximum nesting of DECLAREs, see Appendix B. This error occurs when too many DECLARE statements refer back through each other to the one that actually supplies a type. For example,

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'C';
DECLARE C LITERALLY 'D';
   .     .     .       . ;
   .     .     .       . ;
DECLARE Y LITERALLY 'Z';
DECLARE Z BYTE INITIAL (77);
.
.
.
A=7      ; error discovered here
```

8. INVALID CONTROL FORMAT

   See Chapter 3 for correct formatting of control lines. An example that could cause this error is

```
$LIST (MYPROG.LST) ;
```

   because no pathname is expected on this control.

9. INVALID CONTROL

   See Chapter 3. Example:

```
$NXCODE ; probably intended NOCODE
```

10. ILLEGAL USE OF PRIMARY CONTROL AFTER NON-CONTROL LINE

    Primary controls may appear as control lines in your source program, but they must come first. No other statements may precede them. See Chapter 3.

11. MISSING CONTROL PARAMETER

    Certain controls, e.g., DATE, require you to specify a parameter. See Chapter 3.

12. INVALID CONTROL PARAMETER

    For example, an illegal pathname for a control like OBJECT. See Chapter 3.

13. LIMIT EXCEEDED: INCLUDE NESTING

    See Appendix B for limit. For example, if you INCLUDE a file named A, which INCLUDEs a file named B, and so on, this error will arise when the limit is exceeded.

14. INVALID CONTROL FORMAT, INCLUDE NOT LAST CONTROL

    An INCLUDE may not be followed by another control on the same line, including the compiler invocation statement. As a control line in your source program, it may not be followed by a primary control line. See Chapter 3.

15. MISSING INCLUDE CONTROL PARAMETER

    The requisite pathname is missing or wrongly specified. See Chapter 3.

16. ILLEGAL PRINT CONTROL

    PRINT (:C1:) would be an example, since you cannot print to the console input device. See Chapter 3.

17. INVALID PATH-NAME

    See the *ISIS-II User's Guide*. (underline)

18. INVALID MULTIPLE LABELS AS MODULE NAMES

    The outermost DO-block may not have multiple labels. See Section 9.2* and Chapter 11*.

19. INVALID LABEL IN MODULE WITHOUT MAIN PROGRAM

    A label was found outside any procedure block, without executable statements that would constitute a main program. Perhaps other intended statements are missing, or this extra label was coded by mistake.

20. MISMATCHED IDENTIFIER AT END OF BLOCK

    See Section 6.1*. If a label is supplied in an END statement, the label must match that of a prior DO statement, in fact the first unmatched DO above the END. If multiple labels appear on a DO, the rightmost must match the END. Sometimes the error involves a confusion of module name with procedure name: see also Chapters 10* and 11*.

21. MISSING PROCEDURE NAME

    Every procedure must have a name. See Section 9.2*.

22. INVALID MULTIPLE LABELS AS PROCEDURE NAMES

    Procedures must have exactly one name; no more, no less. See Section 9.2*.

23 INVALID LABELLED END IN EXTERNAL PROCEDURE

    An EXTERNAL procedure, by definition, is declared PUBLIC elsewhere. The END of an EXTERNAL procedure must not be labeled. See Section 9.2.5*.

24. INVALID STATEMENT IN EXTERNAL PROCEDURE

    Such a procedure, being defined elsewhere, may not contain executable statements. See Section 9.2.5*.

25. UNDECLARED PARAMETER

    A parameter named in the procedure statement did not get defined in the body of the procedure. See Section 9.2.1*.

26. INVALID DECLARATION, STATEMENT OUT OF PLACE

    You can intersperse declarations and procedures, but not declarations and executable statements. See Section 8.1.3*.

27. LIMIT EXCEEDED: NUMBER OF DO BLOCKS (terminal error)

    See Appendix B for correct limit.

28. MISSING 'THEN'

    In an IF statement, the THEN clause is required. See Section 6.2*.

29. ILLEGAL STATEMENT

    This may be due to misspelling or missing parts of an otherwise valid statement. Look up your intended statement in the index of this book or the *PL/M-86 Programming Manual*, and reread the sections listed.

30. LIMIT EXCEEDED: NUMBER OF LABELS ON STATEMENT

    See Appendix B for correct limit.

31. LIMIT EXCEEDED: PROGRAM TOO COMPLEX (terminal error)

    See 199.

32. INVALID SYNTAX, TEXT IGNORED UNTIL ';'

    Despite repeated trials, the compiler failed to find a reasonable interpretation of this line. Perhaps keywords were mistyped, or punctuation omitted.

    The problem may lie earlier. For example, when an embedded quote mark inadvertently ends an earlier string, the remainder of the string may be uninterpretable. Or, if a closing quote is missing, subsequent statements may be seen as part of the unclosed string; when the next quote is encountered, it closes that prior string, leaving inappropriate text as compiler input.

33. DUPLICATE LABEL

    Each label must be unique within its block or scope. Otherwise GOTOs and CALLs would have ambiguous targets. See Sections 8.6* and 10.3*.

34. DUPLICATE PROCEDURE DECLARATION

    Procedure names must be unique. See Sections 8.6*, 9.1*, 9.2*, and 10.3*.

35. LIMIT EXCEEDED: NUMBER OF PROCEDURES (terminal error)

    See Appendix B for correct limit.

36. MISSING PARAMETER

    Fewer parameters were supplied in a CALL than were declared in the procedure. See Section 9.3*.

37. MISSING ')' AT END OF PARAMETER LIST

    A parameter list must be enclosed in a pair of parentheses. See Sections 9.1* and 9.2*.

38. DUPLICATE PARAMETER NAME

    A parameter must be declared exactly once. This message indicates that the flagged parameter already has a definition at this block level, as in

```
YAR: PROCEDURE (YAR77,YAR78);
DECLARE YAR77 BYTE ;
DECLARE YAR77 BYTE ;
```

    Perhaps a different spelling was intended.

39. INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL

    The flagged attribute or initialization can only be valid at the module level, not in a procedure. See Sections 8.2*, 8.4*, and Chapter 11*.

40. DUPLICATE ATTRIBUTE

    Attributes should be specified at most once. This message means the compiler has found a declaration like

```
DECLARE B BYTE EXTERNAL EXTERNAL ;
```

41. CONFLICTING ATTRIBUTE

    The attributes declared are contradictory, as in

```
DECLARE PAK BYTE WORD ;
```

    perhaps resulting from an editing error.

42. INVALID INTERRUPT VALUE

    Interrupt numbers must be whole-number constants between 0 and 255. Thus −7 or 272 would be invalid. See Section 9.2.6*.

43. MISSING INTERRUPT VALUE

    The interrupt attribute requires a number as above in 42.

44. ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH PARAMETERS

    No parameters are allowed in interrupt procedures. See Section 9.2.6*.

45. ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH TYPED PROCEDURE

    Interrupt procedures must be untyped. See Section 9.2.6*.

46. ILLEGAL USE OF LABEL

    Check the flagged statement against the rules* (see the index*). Here are two types of statement that might cause this message:

```
L1: DECLARE DORN BYTE ;/* cannot label declares */
L2: DORN = DORN + L2; /* labels can't be variables */
```

47. MISSING ')' AT END OF FACTORED DECLARATION

    See Section 3.1*. The variable list in a factored declaration must be enclosed in a pair of parentheses.

48. ILLEGAL DECLARATION STATEMENT SYNTAX

    See index* for "declare". Possible misspelling or order.

49. LIMIT EXCEEDED: NUMBER OF ITEMS IN FACTORED DECLARE

    See Appendix B for correct limit.

50. INVALID ATTRIBUTES FOR BASE

    A base must be a non-subscripted scalar of type POINTER or WORD. It is not permitted to have the attribute BASED. See Section 5.4*.

51. INVALID BASE, MEMBER OF BASED STRUCTURE

    See 50 above.

52. INVALID BASE, MEMBER OF ARRAY OF STRUCTURES

    See 50 above.

53. INVALID STRUCTURE MEMBER IN BASE

    Perhaps an insufficiently qualified reference, or a form like

    ```
    DECLARE LATTICE BASED FRAME.
    ```

    i.e., no member stated. See Sections 5.2* and 5.3*.

54. UNDECLARED BASE

    A variable was declared BASED using an undeclared identifier. See 50 above.

55. UNDECLARED STRUCTURE MEMBER IN BASE

    The named structure does not contain the member given as the base. See 50 and 53 above.

56. INVALID MACRO TEXT, NOT A STRING CONSTANT

    Possibly a missing apostrophe. See Sections 8.7* and 2.5*.

57. INVALID DIMENSION, ZERO ILLEGAL

    Declaring an array to contain zero scalars is invalid. See Section 5.1*.

58. INVALID STAR DIMENSION IN FACTORED DECLARATION

    See Section 8.4*. Star dimensions are for initializations.

59. ILLEGAL DIMENSION ATTRIBUTE

    Perhaps negative, or larger than 64K bytes. See Section 5.1*.

60. MISSING ')' AT END OF DIMENSION

    Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Section 5.1*.

61. MISSING TYPE

    A type is required in declaring a variable. See Chapter 3*.

62. INVALID STAR DIMENSION WITH 'STRUCTURE' OR 'EXTERNAL'

    See Sections 8.2* and 8.4*. Star dimensions are for initialization.

63. INVALID DIMENSION WITH THIS ATTRIBUTE

    Dimension is not allowed in declaring a label. See Section 8.6.1*.

64. MISSING STRUCTURE MEMBERS

    Perhaps no structure members were named, or a reference was insufficiently qualified. See Sections 5.2* and 5.3*.

65. MISSING ')' AT END OF STRUCTURE MEMBER LIST

    Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Section 5.2* and 5.3*.

66. INVALID STRUCTURE MEMBER, NOT AN IDENTIFIER

    Possibly a misspelling, e.g., coding

    ```
    DECLARE AIR STRUCTURE (F4 BYTE, 5 BYTE);
    ```

    The 5 probably should have been F5. See Sections 5.2*, 5.3*.

67. DUPLICATE STRUCTURE MEMBER NAME

    In the example of 66 above, saying F4 BYTE, F4 BYTE would make subsequent references to AIR.F4 ambiguous. See Sections 5.2*, 5.3*.

68. LIMIT EXCEEDED: NUMBER OF STRUCTURE MEMBERS

    See Appendix B for correct limit.

69. INVALID STAR DIMENSION WITH STRUCTURE MEMBER

    Star dimension must not be used with structures. See Section 8.4*.

70. INVALID MEMBER TYPE, 'STRUCTURE' ILLEGAL

    Structures must not contain structures. See Chapter 5*.

71. INVALID MEMBER TYPE, 'LABEL' ILLEGAL

    Labels may not be structure members. See Chapter 5*.

72. MISSING TYPE FOR STRUCTURE MEMBERS

    A type must accompany the declaration of each member of a structure. See Section 5.2*.

73. INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL

    The flagged attribute or initialization can only be valid at the module level, not in a procedure. See Sections 8.2*, 8.4*, and Chapter 11*.

74. INVALID STAR DIMENSION, NOT WITH 'DATA' OR 'INITIAL'

    Array declarations using the asterisk must use either DATA or INITIAL also. See Section 8.4*.

75. MISSING ARGUMENT OF 'AT', 'DATA', OR 'INITIAL'

    These attributes require arguments to be effective. See Sections 8.3*, 8.4*, 8.5*, and 12.6.2*.

76. CONFLICTING ATTRIBUTE WITH PARAMETER

    Certain attributes are not allowed in declaring a parameter, e.g. PUBLIC, EXTERNAL, DATA, INITIAL, AT, BASED.

77. INVALID PARAMETER DECLARATION, BASE ILLEGAL

    A procedure parameter cannot be declared BASED. See Section 9.2.1*.

78. DUPLICATE DECLARATION

    The flagged item already has a definition declared at this block level.

79. ILLEGAL PARAMETER TYPE

    Parameters may not be declared of type structure or array. See Section 9.2.1*.

80. INVALID DECLARATION, LABEL MAY NOT BE BASED.

    See Section 8.6.2*.

81. CONFLICTING ATTRIBUTE WITH 'BASE'

    Examples of attributes conflicting with base include AT, DATA, INITIAL, PUBLIC, and EXTERNAL. See "based variable" references in the index*.

82. INVALID SYNTAX, MISMATCED '('

    Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See section 5.1*.

83. LIMIT EXCEEDED: DYNAMIC STORAGE (terminal error)

    Too many symbols were defined. Eliminate unused symbols or break this module into several modules.

84. LIMIT EXCEEDED: BLOCK NESTING

    See Appendix B for correct limit.

85. LONG STRING ASSUMED CLOSED AT NEXT SEMICOLON OR QUOTE

    Perhaps an intended closing apostrophe is missing. See Appendix B for string length limit. Unbalanced quotes often cause multiple apparent errors due to "swallowing" later required words or punctuation as if part of a string. See also 32 above.

86. LIMIT EXCEEDED: SOURCE LINE LENGTH

    See Appendix B for correct limit.

87. MISSING 'END', END-OF-FILE ENCOUNTERED

    The source file ended before a needed 'END' statement (for a prior DO) was encountered. This might indicate an editing problem or a string not closed off. See Sections 1.2.3*, 1.2.4*, and 6.1*.

88. INVALID PROCEDURE NESTING, ILLEGAL IN REENTRANT PROCEDURE

    Reentrant procedures may not contain nested procedures. See Section 9.2.7*.

89. MISSING 'DO' FOR MODULE

    A module must be a labeled simple DO-block. See Section 11.1*.

90. MISSING NAME FOR MODULE

    Every module must have exactly one name, i.e., the label on the outermost DO-block. See Chapter 11*.

91. ILLEGAL PAGELENGTH CONTROL VALUE

    See Section 3.3.2.

92. ILLEGAL PAGEWIDTH CONTROL VALUE

    See Section 3.3.3.

93. MISSING 'DO' FOR 'END', 'END' IGNORED

    More ENDs were found than prior DOs. See references in 87.

94. ILLEGAL CONSTANT, TOO LARGE FOR CONTEXTUALLY DETERMINED TYPE

    See Section 4.5.2*.

95. ILLEGAL RESPECIFICATION OF PRIMARY CONTROL IGNORED

    See Section 3.1.

96. COMPILER ERROR: SCOPE STACK UNDERFLOW

    Unrecoverable error. Trying a different copy of the compiler on a different drive might reveal the first copy had somehow gotten clobbered or gone bad. Contact INTEL.

97. COMPILER ERROR: PARSE STACK UNDERFLOW

    See 96.

98. INCLUDE FILE IS NOT A DIRECT ACCESS FILE (terminal error)

    See 3.7.1. See also the *ISIS-II User's Guide*.

99. INVALID REAL CONSTANT

    Examples: 1.7F or 1.7. See Chapter 14*.

100. INVALID STRING CONSTANT IN EXPRESSION

    See Section 4.1.1*.

101. INVALID ITEM FOLLOWS DOT OR AT SIGN OPERATOR

    Examples: @5, @.S1, @'THERE'. See Sections 3.5.1*, 3.5.2*, and Chapter 5*.

102. MISSING PRIMARY OPERAND

    An identifier, number, string, address, or other primary was expected. Example: A = A + ; when you meant A = A + B; See Sections 4.1.4* and 4.1.5*.

103. MISSING ')' AT END OF SUBEXPRESSION

    Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Sections 4.1.4*, 4.1.5*.

104. ILLEGAL PROCEDURE INVOCATION WITH DOT OR AT SIGN OPERATOR

    "CALL @MYPROC" is an example. See Sections 9.3* and 3.5.1*.

105. UNDECLARED IDENTIFIER

    Every identifier must be declared. See Section 3.1.1*.

106. ILLEGAL PAGELENGTH (4) AND SUBTITLE COMBINATION

    See Sections 3.3.2 and 3.3.6.

107. INVALID USE OF '@' WITH LOCAL PROCEDURE

See Sections 5.4.1 and 5.5.1. Dot operator should be used instead.

108. INVALID USE OF '.' WITH PUBLIC OR EXTERNAL PROCEDURE

@ operator should be used instead. See Sections 5.4.1, 5.5.1.

109. Not used.

110. INVALID LEFT OPERAND OF QUALIFICATION, NOT A STRUCTURE

For example, a reference of the form GNU.F1 where GNU was not declared a structure. See Chapter 5*.

111. INVALID RIGHT OPERAND OF QUALIFICATION, NOT IDENTIFIER

For example, GNU.6 where GNU is a valid, declared structure; 6, however, is not an identifier. See Section 2.2* and Chapter 5*.

112. UNDECLARED STRUCTURE MEMBER

For example, KAPI.HORN where KAPI is a valid, declared structure but HORN was never declared. See Chapter 5*.

113. MISSING ')' AT END OF ARGUMENT LIST

Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See also Chapter 12* for requirements of built-in procedures.

114. INVALID SUBSCRIPT, MULTIPLE SUBSCRIPTS ILLEGAL

For any array TING, references of the form TING (2,4) or TING (3,7,9,6) are invalid because of multiple subscripts. Only references of one subscript are valid, e.g., TING (5). See Section 5.1*.

115. MISSING ')' AT END OF SUBSCRIPT.

Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Section 5.1*.

116. MISSING '=' IN ASSIGNMENT STATEMENT

The flagged statement looked like an assignment statement but the equal sign was missing. See Section 4.6*. Examples:

```
SUM SUM + TING(I)
R1,R2,R3 CIRC/TWOPI
```

117. MISSING PROCEDURE NAME IN CALL STATEMENT

See Section 4.3*. Perhaps the left parenthesis for the parameter list was inadvertently placed before the procedure name, as in

```
CALL (MOVE 3, ORIG, FINAL);
```

instead of

```
CALL MOVE (3, ORIG, FINAL);
```

118. INVALID INDIRECT CALL, IDENTIFIER NOT A WORD OR POINTER SCALAR

See Section 9.3.1*. Only word or pointer scalars can be used for indirect calls. This excludes word or pointer expressions, byte or real scalars, all structures, and all arrays.

119. LIMIT EXCEEDED: PROGRAM TOO COMPLEX (terminal error)

Too many complex expressions, cases, and procedures. Break it up into smaller modules.

120. LIMIT EXCEEDED: EXPRESSION TOO COMPLEX (terminal error)

Too many subexpressions and typed procedure calls. Break it up.

121. LIMIT EXCEEDED: EXPRESSION TOO COMPLEX (terminal error)

See 120.

122. LIMIT EXCEEDED: PROGRAM TOO COMPLEX (terminal error)

See 119.

123. INVALID DOT OR AT SIGN OPERAND, BUILT-IN PROCEDURE ILLEGAL

References to built-in procedures may not use the dot or @ operators. See Chapter 12*.

124. MISSING ARGUMENTS FOR BUILT-IN PROCEDURE

See Chapter 12* for required arguments.

125. ILLEGAL ARGUMENT FOR BUILT-IN PROCEDURE

See 124.

126. MISSING ')' AFTER BUILT-IN PROCEDURE ARGUMENT LIST.

Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Chapter 12*.

127. INVALID SUBSCRIPT ON NON-ARRAY

Subscripts are permitted only on identifiers declared as arrays. Check spelling consistency. See Chapter 5*.

128. INVALID LEFT-HAND OPERAND OF ASSIGNMENT

For example, PROCEDURE = 4 or INWORD(7) = 9. See Section 4.6*.

129. ILLEGAL 'CALL' WITH TYPED PROCEDURE

Typed procedures are validly invoked only by use in an expression, not by a CALL. See Sections 9.2.2*, 9.2.4*, and 9.3*.

130. ILLEGAL REFERENCE TO OUTPUT OR OUTWORD FUNCTION

These may be used only to the left of an equal sign, i.e., as the left part of an assignment statement. See Section 12.4.2*.

131. ILLEGAL REFERENCE TO UNTYPED PROCEDURE

Untyped procedures must be invoked by a CALL statement; references to such procedures are not permitted in expressions. See references in 129 above.

132. ILLEGAL USE OF LABEL

See references under "label" in index*.

133. ILLEGAL REFERENCE TO UNSUBSCRIPTED ARRAY

In the context of the flagged statement, the array reference requires a subscript. See Sections 3.5.1*, 3.8.4*, 9.3.1*, and Chapter 5*.

134. ILLEGAL REFERENCE TO UNSUBSCRIPTED MEMBER ARRAY

See references in 133. Here a structure member is an array and the reference requires a subscript.

135. ILLEGAL REFERENCE TO AN UNQUALIFIED STRUCTURE

See references in 133. This statement was ambiguous as to which structure or member was intended.

136. INVALID RETURN FOR UNTYPED PROCEDURE, VALUE ILLEGAL

An untyped procedure does not return a value, so its RETURN statement may not specify one. See Section 9.2.3*.

137. MISSING VALUE IN RETURN FOR TYPED PROCEDURE

A typed procedure must return a value, so its RETURN statement must specify one. See reference in 136.

138. MISSING INDEX VARIABLE

An iterative DO block requires an index variable. See Section 6.1.4*.

139. INVALID INDEX VARIABLE TYPE

Only BYTE, WORD, or INTEGER are valid. See Section 6.1.4*.

140. MISSING '=' FOLLOWING INDEX VARIABLE

Something like DO I 17 TO 34, when it should say DO I =17 TO 34. See Section 6.1.4*.

141. MISSING 'TO' CLAUSE

A statement like DO I = 17 either doesn't need the DO, or it does need a TO clause. See Section 6.1.4*.

142. MISSING IDENTIFIER FOLLOWING GOTO

See Sections 6.3.2* and 10.3*. The target destination was absent.

143. INVALID REFERENCE FOLLOWING GOTO, NOT A LABEL

The identifier following GOTO must be a label; the flagged item was declared otherwise. See references in 142.

144. INVALID GOTO LABEL, NOT AT LOCAL OR MODULE LEVEL

See references in 142.

145. MISSING 'TO' FOLLOWING 'GO'

GO cannot appear alone. See references in 142.

146. MISSING ')' AFTER 'AT' RESTRICTED EXPRESSION

The expression following an AT must be enclosed in parentheses. Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Section 8.3*.

147. MISSING IDENTIFIER FOLLOWING DOT OR AT SIGN OPERATOR

See Section 3.5*, 3.6*, and Section 5.4.1, 5.4.2 in this manual.

148. INVALID QUALIFICATION IN RESTRICTED REFERENCE

See Sections 3.5.1* and 8.3*.

149. INVALID SUBSCRIPTING IN RESTRICTED REFERENCE

See 148.

150. MISSING ')' AT END OF RESTRICTED SUBSCRIPT

Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Sections 4.1.4*, 4.1.5*, and 5.1*.

151. INVALID OPERAND IN RESTRICTED EXPRESSION

For example, PL/M-86 reserved words and predeclared identifiers would be invalid. See Appendices C*, D*, and page 12-1*.

152. MISSING ')' AFTER CONSTANT LIST

Parentheses must balance in any statement using them, i.e., same number of '(' as of ')'. See Sections 4.1.4*, 4.1.5*, 8.4*, 8.5*.

153. INVALID NUMBER OF ARGUMENTS IN CALL, TOO MANY

See Section 9.3*. The number of actual parameters supplied in a CALL must equal the number of formal parameters declared in the procedure. See also Chapter 12* for the requirements of built-ins.

154. INVALID NUMBER OF ARGUMENTS IN CALL, TOO FEW

See 153.

155. INVALID RETURN IN MAIN PROGRAM

A main program must have no returns. See Section 9.2.3* and Chapter 11*.

156. MISSING RETURN STATEMENT IN TYPED PROCEDURE

A typed procedure must return a value, so it must have a RETURN statement, with a value of the declared type. See Section 9.2.3*.

157. INVALID ARGUMENT, ARRAY REQUIRED FOR LENGTH OR LAST

See Section 12.1*. These built-ins need an array name.

158. INVALID DOT OR AT SIGN OPERAND, LABEL ILLEGAL

A variable-reference is required, not a label. See Section 3.5.1*.

159. COMPILER ERROR: PARSE STACK UNDERFLOW.

See 96.

160. COMPILER ERROR: OPERAND STACK UNDERFLOW.
See 96.

161. COMPILER ERROR: ILLEGAL OPERAND STACK EXCHANGE
See 96.

162. COMPILER ERROR: OPERATOR STACK UNDERFLOW
See 96.

163. COMPILER ERROR: GENERATION FAILURE
See 96.

164. COMPILER ERROR: SCOPE STACK OVERFLOW
See 96.

165. COMPILER ERROR: SCOPE STACK UNDERFLOW
See 96.

166. COMPILER ERROR: CONTROL STACK OVERFLOW
See 96.

167. COMPILER ERROR: CONTROL STACK UNDERFLOW
See 96.

168. COMPILER ERROR: BRANCH MISSING IN 'IF' STATEMENT
See 96.

169. ILLEGAL FORWARD CALL
A procedure (other than reentrant) must be declared before it is referenced in a CALL or expression. See Section 9.2*.

170. ILLEGAL RECURSIVE CALL
A non-REENTRANT procedure is not allowed to call itself. See Section 9.2.7*.

171. INVALID USE OF DELIMITER OR RESERVED WORD IN EXPRESSION
Examples: I =)+9 or I = DO + 9. See Section 2.2* and Appendix C*.

172. INVALID LABEL: UNDEFINED
No definition for this label was found. See Sections 6.3* and 8.6*.

173. INVALID LEFT SIDE OF ASSIGNMENT: VARIABLE DECLARED WITH DATA ATTRIBUTE
See Section 8.5*. The item is really a constant; unchangeable.

174. INVALID NULL PROCEDURE
A procedure must contain at least a semicolon.

175. ILLEGAL POINTER ARITHMETIC IN RESTRICTED EXPRESSION
Pointer arithmetic is not allowed. See Section 4.5.2*.

176. INVALID ABSOLUTE ADDRESS, TOO LARGE
Absolute addresses must not exceed 1048575. See Section 8.3*.

177. Not used

178. ILLEGAL REAL ARITHMETIC IN RESTRICTED EXPRESSION
Real arithmetic is not allowed in restricted expressions. See Section 8.4*.

179. ILLEGAL REAL CONSTANT IN 'AT' CLAUSE RESTRICTED EXPRESSION
The restricted expression in an AT clause may not contain a real constant. See Sections 3.5.1* and 8.3*.

180. INVALID OPERATOR OR OPERAND, TYPE CONFLICTS WITH EXPECTED TYPE
See Section 4.5.2*.

181. LIMIT EXCEEDED: CONSTANT OR CODE SEGMENT SIZE
See Appendix B for correct limit.

182. ILLEGAL REFERENCE TO ABSOLUTE ADDRESS WITH SMALL OPTION SPECIFIED
See Section 5.2.1.

183. INVALID 'AT' RESTRICTED REFERENCE, EXTERNAL ATTRIBUTE CONFLICTS WITH PUBLIC
See Sections 8.2* and 8.3*. For example,

```
DECLARE DARTH BYTE EXTERNAL ;
DECLARE VADER BYTE PUBLIC AT (.DARTH) ;
```

184. INVALID EXPRESSION, TWO SUCCESSIVE RELATIONAL OPERATORS
See Section 4.3*.

185. LIMIT EXCEEDED: NUMBER OF EXTERNAL ITEMS
See Appendix B for correct limit.

186. INVALID RESTRICTED EXPRESSION, TYPE CONFLICTS WITH TARGET
See Section 8.4* for appropriate initialization values.

187. ILLEGAL INITIALIZATION TO A BASED OR AUTOMATIC ADDRESS
See Section 8.4*, 8.5*, and page 12-1*.

188. MISSING ENDIF OPTION
See Section 3.10.

189. MISSING OR INVALID CONDITIONAL COMPILATION PARAMETER
See Section 3.10.

190. MISSING OR INVALID CONDITIONAL COMPILATION CONSTANT
See Section 3.10.

191. MISPLACED ELSE OR ENDIF OPTION
See Section 3.10.

192. MISPLACED ENDIF OPTION
See Section 3.10.

193. CONDITIONAL COMPILATION PARAMETER NAME TOO LONG, TRUNCATED
See Section 3.10.

194. MISSING OPERATOR IN CONDITIONAL COMPILATION EXPRESSION
See Secton 3.10.

195. INVALID CONDITIONAL COMPILATION CONSTANT TOO LARGE
See Section 3.10.

196. INVALID UNDEFINED CONDITIONAL COMPILATION PARAMETER
See Section 3.10.

197. LIMIT EXCEEDED: SAVE NESTING
The limit is 5. See Section 3.7.2.

198. MISPLACED RESTORE OPTION
RESTORE can only work if there has been a prior SAVE. See Section 3.7.2.

199. LIMIT EXCEEDED: PROCEDURE COMPLEXITY FOR OPTIMIZE(2) (terminal error)
The combined complexity of expressions, user labels, and compiler generated labels is too great. Simplify as much as possible, perhaps breaking the procedure into several.

200. LIMIT EXCEEDED: STATEMENT SIZE
The statement is too large for the compiler. Break it up.

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

See Section 6.1.5*.

202. LIMIT EXCEEDED: NUMBER OF ACTIVE CASES

See Appendix B for correct limit.

203. LIMIT EXCEEDED: NESTING OF TYPED PROCEDURE CALLS

See Appendix B for correct limit.

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

See Appendix B for correct limit.

205. ILLEGAL NESTING OF BLOCKS, ENDS NOT BALANCED

For every DO, an END is needed. See Section 6.1*.

206. LIMIT EXCEEDED: CODE SEGMENT SIZE

See Appendix B for correct limit.

207. LIMIT EXCEEDED: SEGMENT SIZE

See Appendix B for correct limit.

208. LIMIT EXCEEDED: STRUCTURE SIZE

See Appendix B for correct limit.

209. ILLEGAL INITIALIZATION OF MORE SPACE THAN DECLARED

The number of initialization values exceeds the number of declared elements. See Sections 8.4* and 8.5*.

210. INVALID RESTRICTED EXPRESSION, VALUE TOO LARGE FOR TARGET

DECLARE BOZO BYTE INITIAL (259) would be an example, because the maximum byte constant is 255.

211. INVALID IDENTIFIER IN 'AT' RESTRICTED REFERENCE

Example: DECLARE LOGOS WORD AT (@START); START must be a valid pointer or word expression. See Section 8.3*.

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

See Section 8.3*. Based variables cannot be used in AT clauses.

213. UNDEFINED RESTRICTED REFERENCE IN 'AT'

The variable used in the AT clause was not already declared. Example: DECLARE APPLE BYTE AT (.B); with B undefined. See Section 8.3*.

214. COMPILER ERROR: INVALID OPERATION

See 96.

215. COMPILER ERROR: EOF READ IN FINAL ASSEMBLY

See 96.

216. COMPILER ERROR: BAD LABEL ADDRESS

See 96.

217. ILLEGAL INITIALIZATION OF AN EXTERNAL VARIABLE

External variables may be initialized only where they are declared PUBLIC. See Section 8.2*

218. LIMIT EXCEEDED: REAL EXPRESSION COMPLEXITY

The REAL stack has 8 registers. Heavily nested use of REAL functions with REAL expressions as parameters could get excessively complex. See Chapter 14*.

219. COMPILER ERROR: REAL STACK UNDERFLOW

See 96 and 218. See also Chapter 14*.

220. LIMIT EXCEEDED: BASIC BLOCK COMPLEXITY

This means you have a very long list of statements without labels, procedures, if's, gotos, etc. Either break the program into several modules or try adding labels to some of your statements.

221. LIMIT EXCEEDED: STATEMENT SIZE

The statement is too large for the compiler. Break it up.

222. INVALID ABSOLUTE LOCATION FOR PUBLIC WITHOUT LARGE OPTION

See Sections 5.2.1, 5.3, 5.4.1. Absolute locations for PUBLICS are supported only under the LARGE option.

223. SUBSCRIPTED VARIABLE NOT ALLOWED IN FACTORED DECLARATION

See Section 3.1*.

224. WARNING: INITIAL ATTRIBUTE USED WITH ROM OPTION

See Sections 3.8, 5.2.1, 5.3, 5.4.1, and 5.5.1.

225. ILLEGAL INDIRECT REFERENCE TO A CONSTANT WHILE USING ROM OPTION

See Section 3.8.

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 DIRECT ACCESS FILE
INVOCATION COMMAND DOES NOT END WITH <CR><LF>
INCORRECT DEVICE SPECIFICATION
SOURCE FILE NOT A DIRECT ACCESS FILE
SOURCE FILE NAME INCORRECT
SOURCE FILE EXTENSION INCORRECT
ILLEGAL COMMAND TAIL SYNTAX
MISPLACED CONTROL: WORKFILES ALREADY OPENED

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 ISIS ERROR –
FILE:
NAME:
ERROR:
COMPILATION TERMINATED

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
```

APPENDIX D
PL/M-86 MODELS OF SEGMENTATION

The segments, classes, and groups in the PL/M-86 compiler output module vary according to the size control specified to the compiler. The segment, class, and group names generated by the PL/M-86 compiler for the SMALL, COMPACT, MEDIUM, and LARGE model as shown below. Recall, however, that under the ROM option, the constant section is merged into the CODE segment in every model.

Table D-1. Models of Segmentation

Small Model

Segment Name	Class Name	Group Name
CODE	CODE	CGROUP
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	
MEMORY	MEMORY	

Compact Model

Segment Name	Class Name	Group Name
CODE	CODE	CGROUP
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	none
MEMORY	MEMORY	none

Medium Model

Segment Name	Class Name	Group Name
*modname*__CODE	CODE	none
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	
MEMORY	MEMORY	

Large Model

Segment Name	Class Name	Group Name
*modname*__CODE	CODE	none
*modname*__DATA	DATA	none
STACK	STACK	none
MEMORY	MEMORY	none

Table D-1. Models of Segmentation (Cont'd.)

Number of Segments Allowed

Size Control	Code	Constant RAM	Constant ROM	Data	Stack	Memory	Total
SMALL	one	part	*	part	part	part	2
COMPACT	one	part	*	part	one	one	4
MEDIUM	many	part	*	part	part	part	≥2
LARGE	many	*	*	many	one	one	≥4

Legend:

part – these sections are combined (with sections of other types) to produce single segment.

one – sections combined (with sections of the *same type*) to produce single segment.

many – sections not combined.

* – section has been combined with the CODE segment by the compiler.

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.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.