ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS
for 8086-Based Systems

Order Number: 121628-003

Copyright © 1980, 1982 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 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.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

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

BXP	int$_e$l	iSBC	MULTICHANNEL
CREDIT	Intelevision	iSBX	MULTIMODULE
i	int$_e$ligent Identifier	iSXM	Plug-A-Bubble
I²ICE	int$_e$ligent Programming	Library Manager	PROMPT
ICE	Intellec	MCS	RMX/80
iCS	Intellink	Megachassis	RUPI
i$_m$	iOSP	MICROMAINFRAME	System 2000
iMMX	iPDS	MULTIBUS	UPI
Insite	iRMX		

A734/982/ 5K DD

REV.	REVISION HISTORY	DATE
-001	Original issue.	9/80
-002	Added information concerning invocation under iRMX 86. Title change to ASM86.	5/82
-003	Added information concerning new controls for iAPX186 assembler. Revised description of assembler controls. Revised list of error messages for iAPX186 assembler.	9/82

PREFACE

This manual is directed to those of you have read the *ASM86 Language Reference Manual*, have coded your program, and are ready to run the ASM86 Macro Assembler on an 8086-based system.

This manual instructs you in the use of the ASM86 Macro Assembler through the use of the assembler controls. It is according to these controls, or assembler commands, that the assembler creates an object file and a listing file.

Included in these pages are the error messages and how to recover from the conditions that caused them. Also included are instructions for linking ASM86 programs to programs written in higher level languages (PL/M-86, FORTRAN-86, and PASCAL-86).

How to Use This Manual

The majority of this manual is a generic, operating system-independent document. That is, the material you read there is true for all the operating systems that support the ASM86 Macro Assembler. Naturally, there are certain differences between the various operating systems. Material specific to invoking the assembler under a specific operating system is given in Chapter 2.

Related Publications

For more information on the ASM86 language, the higher level languages with which it can be linked, and the operating system, refer to the following manuals:

- *An Introduction to ASM86,* 121689
- *ASM86 Language Reference Manual,* 121703
- *PL/M-86 User's Guide,* 121636
- *FORTRAN-86 User's Guide,* 121570
- *PASCAL-86 User's Guide,* 121539
- *Intellec Series III Microcomputer Development System Product Overview,* 121575
- *Intellec Series III Microcomputer Development System Console Operating Instructions,* 121609
- *ALTER Text Editor User's Guide,* 121756
- *iRMX 86 System Programmers Reference Manual,* 142721
- *iRMX 86 Human Interface Manual,* 9803202

Notational Conventions

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.

italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename.*
filename	Is a valid name for the part of a *pathname* that names a file.
pathname	Is a valid designation for a file; in its entirety, it consists of a *directory-name* and a *filename.*
pathname1, pathname2, ...	Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ } . . .	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
input lines	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
<cr>	Indicates a carriage return.

CONTENTS

CHAPTER 1 — PAGE
BEFORE USING THE ASM86 MACRO ASSEMBLER

CHAPTER 2
INVOKING THE ASM86 MACRO ASSEMBLER

Series III Invocation (Standalone System) 2-1
  Examples 2-1
Series III Invocation on NDS Network 2-2
  Examples 2-2
iRMX 86 Invocation 2-3
  Examples 2-3

CHAPTER 3
DEFINING ASSEMBLY CONDITIONS (ASSEMBLER CONTROLS)

Specifying Controls 3-1
Types of Controls 3-1
Description of Controls 3-3
  DATE 3-3
  DEBUG/NODEBUG 3-3
  EJECT 3-4
  ERRORPRINT/NOERRORPRINT 3-4
  GEN/GENONLY/NOGEN 3-4
  INCLUDE 3-6
  LIST/NOLIST 3-6
  MACRO/NOMACRO 3-7
  MOD186 3-7
  OBJECT/NOOBJECT 3-8
  PAGELENGTH 3-8
  PAGEWIDTH 3-8
  PAGING/NOPAGING 3-9
  PRINT/NOPRINT 3-9
  SAVE/RESTORE 3-9
  SYMBOLS/NOSYMBOLS 3-10
  TITLE 3-11
  TYPE/NOTYPE 3-11
  WORKFILES 3-11
  XREF/NOXREF 3-12
Macro Calls and Control Recognition 3-12

CHAPTER 4
LISTING FILE AND ERRORPRINT FILE

The Listing File 4-1
  Header 4-1
  Body 4-1
    LOC 4-4
    OBJ 4-4
    EQUATE 4-6
    INCLUDE Nesting Indicator 4-7
    LINE 4-7
    Macro Nesting Indicator 4-7
    Source Text 4-7

PAGE
  Symbol Table 4-8
    Name 4-8
    Type 4-8
    Value 4-8
    Attributes 4-10
    XREFS 4-11
The Errorprint File 4-12

APPENDIX A
ERROR MESSAGES AND RECOVERY

Console Error Messages A-1
  Control Errors A-1
  I/O Errors A-1
  Others A-2
Source File Error Messages A-2
Macro Error Messages A-25
Control Error Messages A-27

APPENDIX B
LINKING ASSEMBLY LANGUAGE AND HIGHER LEVEL LANGUAGES

The Procedural Interface B-1
  Passing Parameters on the 8086 ..... B-1
  Retrieving Parameters from the Stack B-1
  Choosing a Method to Access Parameters B-2
  Returning Values from Functions B-2
  Register Conventions B-2
Models of Segmentation B-3
  CGROUP and DGROUP B-3
  The SMALL Model B-3
  The COMPACT Model B-3
  The MEDIUM Model B-4
  The LARGE Model B-4
Subsystems B-4
Templates B-4
  Using the Templates B-5
The SMALL Model of Segmentation B-6
  Notes on the SMALL Model B-6
The COMPACT Model of Segmentation B-8
  Notes on the COMPACT Model B-8
The MEDIUM Model of Segmentation B-10
  Notes on the MEDIUM Model B-10
The LARGE Model of Segmentation B-12
  Notes on the LARGE Model B-12

APPENDIX C
RULES FOR SHORTENING CONTROLS

APPENDIX D
USING THE 8087 NUMERIC DATA PROCESSOR AND THE 8087 EMULATOR PROGRAMS

TABLES

TABLE	TITLE	PAGE
2-1	ASM86 Macro Assembler Parameters (Rules of Thumb)	2-5
3-1	ASM86 Macro Assembler Controls Summary	3-3
4-1	Symbol Table Information	4-12
B-1	Registers Used to Return Simple Values	B-2

ILLUSTRATIONS

FIGURE	TITLE	PAGE
1-1	ASM86 Macro Assembler Logical Files	1-2
4-1	The List File	4-2
4-2	Fields of Information in the List File	4-5
4-3	Fields of Information in the Symbol Table	4-9

CHAPTER 1 BEFORE USING THE ASM86 MACRO ASSEMBLER

If this is the first time that you have used the ASM86 Macro Assembler, be sure your system includes these items, as they are required for assembler operation:

- A Series III Development System or an iRMX 86 Application System with Human Interface
- A console device, such as a CRT or TTY
- Appropriate operating system software

You may want to add a lineprinter and/or more disk drives to this configuration as it represents the minimum configuration with which the assembler can be used. For iRMX 86, you must have at least one mass storage device or directory to run ASM86. Consult the console operating instructions for your system for further information.

Next, check that the ASM86 Macro Assembler is on a disk.

Have your *ASM86 Language Reference Manual* nearby, as that document and this one are interdependent. This manual assumes that you are familiar with the ASM86 language.

This manual instructs you in the use of the ASM86 Macro Assembler through the use of its controls. The assembler creates an object file, listing file, and errorprint file in accordance with these controls. See figure 1-1 for the logical files.

The listing file contains the source file, the expanded macro source code, the assembler object code, a summary of assembly errors, if any, and a list of the symbols that you have defined in your source program.

The ASM86 Macro Assembler is a multi-overlay assembler. The overlays are all contained in one file: ASM86.86.

The assembler can reside on and be invoked from any disk.

During assembly, the ASM86 Macro Assembler creates six temporary files for its own use. These files are given temporary names by the operating system and are deleted at the end of assembly.

The files may be placed on any drive using the WORKFILES control. The default condition places the files on the system's workfile drive.

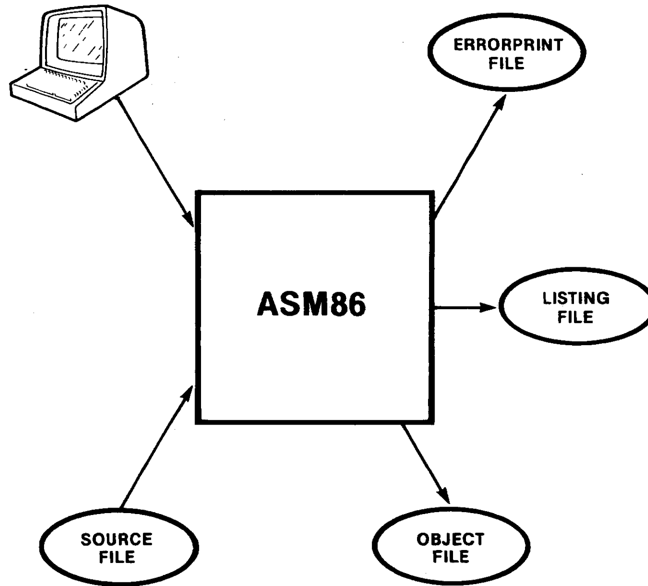


Figure 1-1. ASM86 Macro Assembler Logical Files 121624-1

CHAPTER 2
INVOKING THE ASM86 MACRO ASSEMBLER

To invoke the ASM86 Macro Assembler, enter the appropriate invocation line for your operating system as described below. For a detailed discussion of assembly controls and their place in the assembler invocation command, see Chapter 3, "Defining Assembly Conditions." Chapter 3 also explains how the assembler invocation command can be continued on another line.

Series III Invocation (Standalone System)

If the system is at the ISIS level, the general form of the invocation line is:

```
-[:Fn:]RUN [:Fn:]ASM86 [:Fn:]sourcefile [controls]
```

If the system is already at the RUN level, the general form is:

```
>[:Fn:]ASM86 [:Fn:]sourcefile [controls]
```

where

:F*n*:	represents disk drive device or directory number *n*. This may be omitted if the file is on drive 0.
sourcefile	is the name of the file containing the ASM86 source module.
controls	is an optional sequence of assembler controls (see Chapter 3).

One or more blanks or tabs must separate the items of information in the invocation line. A command line may be continued with an ampersand (&) that appears outside of a quoted string. Anything following the ampersand on that line is ignored. The invocation may end with a comment by preceding the comment with a semicolon (;).

Examples

1. Assume that the operating system and the assembler are on drive 0, and the source file named GONZO.SRC is on drive 1. In its simplest form, the invocation command can be:

   ```
   -RUN ASM86 :F1:GONZO.SRC
   ```

   The assembler will use the default values of the control settings to write the object module to the file :F1:GONZO.OBJ, and to write the print file to :F1:GONZO.LST. The default writes the object and print files to the source file drive, using the source file name with extensions OBJ and LST respectively.

2. Now assume that the assembler is on drive 7 and the source file FOOBAR.A86 is on drive 5. Furthermore, it is desired to write the listing to the file TMP on drive 6, and write the object to drive 4. The listing should not be paged, should have a width of 78 columns, and should include a cross-reference symbol table listing. Debug information is desired in the object module. Additionally, all temporary workfiles should be placed on drive 1.

The invocation line for this case can be:

```
-RUN :F7:ASM86 :F5:FOOBAR.A86 & ignore this
>>PR(:F6:TMP)                & print file
>>OJ(:F4:FOOBAR.OBJ)         & object file
>>NOPI PW(78)                & nopaging, pagewidth 78
>>XR DB                      & cross reference symbol table, debug info
>>WF(:F1:)                   ; all work files on drive 1
```

Series III Invocation on NDS Network

Examples

1. If the Series III Development System is functioning as a workstation of an NDS network, you must use ASSIGNments to associate directories with directory specifiers. With the assembler in a file ASM86 and the source code in a file GONZO.SRC in directory SOURCES.ASM, you would make the assignments:

```
ASSIGN :F0: TO /ASM86
ASSIGN :F1: TO /ROOT/SOURCES.ASM/GONZO.SRC
```

Then the invocation would be:

```
-RUN ASM86 :F1:GONZO.SRC
```

2. Now assume that the assembler is in the directory /A/CMPLRS.ASMBLR and that the source file, termed FOOBAR.A86 is in the directory /A/SRC.CODE. You want to write the listing to a file TMP in directory /DATA/LSTNGS and write the object to a filename FOOBAR.OBJ in directory /DATA/OBJ. You want the listing file to be not paged, to be 78 columns wide, and to include a cross-reference symbol table listing. You want debug information for the object module and want temporary workfiles to be created in directory /DATA/WK.FLE.

You must use ASSIGNments to associate directories with directory specifiers as shown below:

```
ASSIGN :F1: TO /A/CMPLRS.ASMBLR
ASSIGN :F2: TO /A/SRC.CODE
ASSIGN :F3: TO /DATA/OBJ
ASSIGN :F4: TO /DATA/WK.FLE
ASSIGN :F5: TO /DATA/LSTNGS
```

You would invoke the assembler with the controls indicated by entering:

```
- RUN :F1:ASM86 :F2:FOOBAR.A86
>>PR(:F5:TMP)
>>OJ(:F3:FOOBAR.OBJ)
>>NOPI PW(78)
>>WF(:F4:)
```

iRMX 86 Invocation

```
-[directory]ASM86 sourcepath [controls]
```

where

directory — is the portion of the pathname that identifies the device and directories containing the file ASM86. If you omit *directory*, the operating system automatically searches several directories for the file ASM86. The directories searched and the order of search are iRMX 86 configuration parameters.

sourcepath — is the pathname of the file containing the ASM source module. The beginning portion of this pathname may consist of a logical name enclosed in colons (such as :F1:). This indicates the place where the operating system starts its search for the file. If you omit the logical name, the operating system assumes that the file resides in the default directory (:$:).

controls — is an optional sequence of controls as defined in Chapter 3.

You can use slashes (/) and up-arrows (∧) as separators between individual components of the pathname (except immediately after the logical name). The slash separator tells the operating system to search down one level in the directory tree for the next component. The up-arrow tells the operating system to search up one level.

One or more blanks or tabs must separate the items of information in the invocation line. You can continue a command line by entering an ampersand (&) outside of a quoted string. The assembler ignores anything that follows the ampersand on that line. You can end the invocation with a comment by placing a semicolon (;) before the comment.

Examples

1. Assume that the assembler resides in a directory with logical name :LANG:; furthermore, assume that :LANG: is one of the directories that the operating system automatically searches. Also assume that a source file named GONZO.SOURCE resides on a device or directory with logical name :FDO:. In its simplest form, the assembler invocation can be:

   ```
   -ASM :FDO:GONZO.SOURCE
   ```

   The assembler uses the default values of the control settings to write the object module to the file :FDO:GONZO.OBJ and to write the print file to :FDO:GONZO.LST. In the default case, the assembler writes the object and print files to the source file device or directory using the source file name with extensions OBJ and LST respectively.

2. Now assume that the assembler resides on a device or directory with logical name :FDO: and is in a subdirectory named UTILITIES. Also assume that the source file FOOBAR.A86 resides on a device or directory with logical name :FDI: and is two levels down in the directory tree, residing in PROGRAMS/ASSEMBLY. Furthermore, it is desired to write the listing to the file TMP on device or directory :WD1: and the object to device or directory :WFX2:. The listing should not

be paged, should have a width of 78 columns, and should include a cross-reference symbol table listing. Debug information is wanted in the object module. Additionally, all temporary workfiles should be placed in a directory with logical name :JUNK:. The invocation line for this case can be:

```
-:FD0:UTILITIES ASM86 :FD1:PROGRAMS/ASSEMBLY/FOOBAR.A86 &
**PR(:WD1:TMP)             & print file
**OJ(:WF2:FOOBAR.OBJ)      & object file
**NOPI PW(78)              & nopaging, pagewidth 78
**XR DB                    & cross-reference symbol table, debug info
**WF(:JUNK:)               ; all work files in directory junk
```

Immediately after you enter the command line, the assembler sends its sign-on message to the console:

system id `8086/87/88/186 MACRO ASSEMBLER, V`*x.y*

where

x.y — is a number 2.0 or greater (the current version number of the assembler).

When the assembly of your program is complete, the sign-off message and error summary are sent to the console in this format:

```
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

If the assembler detected errors, an error summary, such as this one, appears:

```
ASSEMBLY COMPLETE, 2 ERRORS FOUND
```

When fatal errors are detected by the assembler, source file processing is aborted and an informative error message is sent to the console. Fatal errors and their accompanying console messages are described in Appendix A.

It is likely that you will want, initially, to use the assembler in the default or automatic mode of operation; that is, without specifying controls. As you gain experience in using the assembler, the assembler controls described in Chapter 3 will enable you to gain greater efficiency in developing your ASM86 Macro Assembly Language programs.

There are some restrictions that you need to know about. Since these restrictions are mostly quantitative, they are noted in table form (see table 2-1).

Table 2-1. ASM86 Macro Assembler Parameters (Rules of Thumb)

Source File	
Item	**Number**
Characters/source line	255 (including CR/LF); if more are entered, they are processed but not listed and an error message noted.
Characters/ID	31; if more are entered, they are ignored.
Symbols/module	1500 (approximately); relative to the length of the name and type of symbol
Source lines/program	No assembler-imposed limit
Cont. lines/statement	No assembler-imposed limit
Characters/string	255 (including enclosing quotes)
Characters/classname	40
Dup nesting	Up to 8 levels of nested parentheses
PROC/SEG nesting	16 (up to 16 total open at one time)
Items/PUBLIC, EXTRN, PURGE	No assembler-imposed limits
Items/GROUP	36 segments per GROUP
Codemacro size	60 bytes (approximately) of assembler generated code
Items/storage initialization-list	Items cannot exceed 16; limited to 8 levels of nested parentheses.
SEGMENT size	65,536 bytes (64K)
Record limit	16 fields.
Record size	16 bits
Structure fields	Up to 40 fields per structure
Internal	
Item	**Number**
Memory required	96K bytes (plus memory required for operating system)*
Intermediate file size	
I File	1X source file
S File	About 30 bytes per symbol
X File	4 bytes per symbol reference
M File	Varies according to GEN setting, number of macros; about 2X source
L File	About 1X source
T File	1.5X source

*iRMX 86 requires an additional 19.1K of dynamic memory.

CHAPTER 3
DEFINING ASSEMBLY CONDITIONS (ASSEMBLER CONTROLS)

Specifying Controls

The assembler controls can be specified in the (invocation) command line and in the source file. Controls in the invocation line follow the source file name, for example:

```
-ASM86 MYFILE.A86 PRINT(:LP:) XREF DEBUG
```

Controls in the source file are specified using control lines. A control line is any source line with a dollar sign ($) in the first, or leftmost, column. There can be more than one control on the line. The first control on the line may immediately follow the dollar sign.

A control line is always terminated by the end-of-line character(s). Control lines cannot be continued. Control lines may contain comments, which begin with an unquoted semicolon (;) and continue for the remainder of the line.

Example

```
$   PAGEWIDTH (80) PAGELENGTH (72)
$TITLE ('SECTION TWO') EJECT ; Section Two follows
```

Parameters to controls are specified in parentheses following the control name. If the parameter is itself a list of items, the items are delimited by commas.

Example

```
$ WORKFILES (device1, device2)
```

Blanks (or tabs) must separate controls and may be inserted adjacent to the other delimiters. For example, the following three lines are equivalent.

```
$  PRINT(:CO:)  NOOBJECT  PAGEWIDTH(78)  NOPAGING  ; comment
$ PRINT ( :CO: )    NOOBJECT    PAGEWIDTH    (78  )   NOPAGING ; comment
$PRINT(:CO:)NOOBJECT PAGEWIDTH(78)NOPAGING; comment
```

Types of Controls

Controls are classified as either PRIMARY or GENERAL. Primary controls are set at the beginning of the assembly process and cannot be changed during the assembly. Primary controls can only be specified in the invocation line and in the primary control lines. Primary control lines are the source control lines that appear before the first non-control source line. Blank lines and comment lines are considered non-control lines.

If a primary control is specified in the source file and in the invocation line, the control condition specified in the invocation line is the one that takes effect. Within the invocation line or the primary control lines, the last specification of a primary control is used. For example, if the source contains:

```
$XREF DEBUG NOPAGING
$PRINT(:LP:) PAGEWIDTH(132)
$PAGING
```

and the assembler is invoked by:

```
-ASM86 MYFILE.A86 PRINT NOOBJECT NODEBUG
```

then the control settings are:

PRINT (to the default file MYFILE.LST)
NOOBJECT
NODEBUG
XREF
PAGING
PAGEWIDTH(132)

General controls may be specified in the invocation line and on control lines anywhere in the source file. A general control either causes an immediate action (e.g., EJECT, INCLUDE) or an immediate change of conditions (e.g., LIST, GEN). In the latter case, the condition specified by the general control remains in effect until another general control causes it to change. In either case, the immediacy of the result is constrained to mean a general control takes effect at the end of the control line. General controls specified in the invocation line take effect before the first source line is read.

Some controls specify conditions that are either on or off (yes or no). The no condition is specified by adding the word NO to the front of the control name (e.g., XREF/NOXREF).

All control names have two-letter abbreviations, with the exception of the negative forms, which consist of NO plus the two-letter abbreviation of the positive command (see table 3-1). For example, OBJECT/NOOBJECT are abbreviated OJ/NOOJ.

All primary and some general controls have default settings. These defaults are built into the assembler and are used unless alternate settings are specified. Thus it is necessary to use controls only if assembly conditions or actions different from the defaults are required.

Controls and control parameters, in the invocation line or in source control lines, may be typed as upper or lower case letters.

Table 3-1 lists all the controls and their abbreviations. The default settings are shown where applicable. Following the table is a detailed discussion of each control and its parameters.

Table 3-1. ASM86 Macro Assembler Controls Summary

Control Name	Abbreviation	Default
PRIMARY CONTROLS		
DATE(*d*)	DA	*System date*
DEBUG/NODEBUG	DB/NODB	NODEBUG
ERRORPRINT[(*file*)]/NOERRORPRINT	EP/NOEP	NOERRORPRINT
MACRO[(*p*)]/NOMACRO	MR/NOMR	MACRO
MOD186	M1	8086 mode
OBJECT[(*file*)]/NOOBJECT	OJ/NOOJ	OBJECT(*sourcefile*.OBJ)
PAGELENGTH(*n*)	PL	PAGELENGTH(60)
PAGEWIDTH(*n*)	PW	PAGEWIDTH(120)
PAGING/NOPAGING	PI/NOPI	PAGING
PRINT[(*file*)]/NOPRINT	PR/NOPR	PRINT(*sourcefile*.LST)
SYMBOLS/NOSYMBOLS	SB/NOSB	NOSYMBOLS
TYPE/NOTYPE	TY/NOTY	NOTYPE
WORKFILES(*d*1[,*d*2])	WF	WORKFILES(:WORK:,:WORK:)
XREF/NOXREF	XR/NOXR	NOXREF
GENERAL CONTROLS		
EJECT	EJ	—
GEN/GENONLY/NOGEN	GE/GO/NOGE	GENONLY
INCLUDE(*file*)	IC	—
LIST/NOLIST	LI/NOLI	LIST
SAVE/RESTORE	SA/RS	—
TITLE(*ttt*)	TT	*module name*

Description of Controls

DATE

Type: Primary

Form: DATE(*date*)

Abbreviation: DA(*date*)

Default: System date

The DATE control is supplied only for compatibility with the 8080-based assemblers. This control is processed; however, the date parameter is ignored. The date that appears in the print file is set at the operating system level.

DEBUG/NODEBUG

Type: Primary

Form: DEBUG
NODEBUG

Abbreviation: DB/NODB

Default: NODEBUG

DEBUG specifies that local symbol information is to be put into the object file for use in symbolic debugging.

NODEBUG specifies that no local symbol information is to be put into the object file.

EJECT

Type: General

Form: EJECT

Abbreviation: EJ

Default: none

EJECT specifies that the next line of the source listing is to begin on a new page. Multiple ejects on a single control line are ignored. If either NOPAGING or NOLIST is in effect, EJECT controls are ignored. EJECT is not allowed in the invocation line.

ERRORPRINT/NOERRORPRINT

Type: Primary

Form: ERRORPRINT
ERRORPRINT(*filename*)
NOERRORPRINT

Abbreviation: EP/EP(*filename*)/NOEP

Default: NOERRORPRINT

ERRORPRINT specifies that a file containing a listing of all the source lines with errors is to be created. Each line and its associated error message lines appear exactly as in the print file. The header lines from the first page of the print file also appear, unless the errorprint file is :CO:. Paging is not applied to the errorprint file. If ERRORPRINT is specified without a filename parameter, the errorprint file is written to :CO:.

NOERRORPRINT specifies that this error summary file not be created.

Note that it is not necessary to generate a print file in order to create an errorprnt file; that is, ERRORPRINT and NOPRINT may be specified for the same assembly.

GEN/GENONLY/NOGEN

Type: General

Form: GEN
GENONLY
NOGEN

Abbreviation: GE/GO/NOGE

Default: GENONLY

Since the macro scanner is character oriented, macro calls can occupy a portion of a line, a whole line, or several lines. The expansions of macros may also occupy more than one line, a whole line, or a part of a line. GEN, GENONLY, and NOGEN specify the mode of listing assembly source text, macro calls, and macro expansion text in the print file. One and only one of these modes is in effect at any point in the source listing.

Specifying the GEN control produces a listing that includes all source text, all macro calls, and the expansion of every macro, i.e. the macro text. Expansions are indented to the same column as the macro call and are printed on the line below the call. Since GEN provides a complete trace of the macro call and expansion process, it is useful for debugging macros and obtaining the most complete and continuous listing of a source file. In programs containing many macro calls, however, GEN may produce an inconveniently large amount of output. Note that horizontal tabs in macro call or macro expansion lines are not expanded in GEN mode.

Specifying GENONLY produces a listing that includes only source file non-macro text, and the final resultant text of all macros called. GENONLY omits the listing of all macro calls. All object code generated inside any macro calls is listed.

Specifying NOGEN yields a listing that includes only the source file text. In other words, NOGEN produces a listing that shows only the input to the macro processor. Expansion lines resulting from macro calls contained in source lines are not listed unless they contain errors. Object code (if any) from only the first expansion line is listed with the line containing the call.

Line numbers are identical in GENONLY and NOGEN. In the NOGEN mode, line numbers will seem to skip where macros are found in the source file text. The number of lines skipped is exactly equal to the number of lines that would have resulted had GENONLY been specified. Thus, the numbered lines in a print file generated by NOGEN correspond to the same numbered lines in the print file generated by GENONLY.

Consider the example shown below, in which the macro FOO is called in each of the three listing modes. The definition of the macro FOO contains three lines. Thus, a call to FOO expands to four lines: the call line itself, and the three lines of the macro expansion. Note that the call %FOO(4,5,6), in GENONLY, takes place on line 14. Also, since the first line of the definition is on the same line as the call (i.e., no intervening EOL), and this first line generates object code, this object code is listed on the call line in NOGEN (line 19).

```
                 2 +1 $NOGEN
                 3    %*DEFINE(FOO(A,B,C))(DW %A
                      DW %B
                      DW %C
                      )
                 4    $GEN
                 5    %FOO(1,2,3)
                 6 +1 DW %A
0000 0100        7 +2    1
                 8 +1 DW %B
0002 0200        9 +2    2
                10 +1 DW %C
0004 0300       11 +2    3
                12 +1
                13 +1 $GENONLY
0006 0400       14 +2 DW 4
0008 0500       15 +2 DW 5
```

```
000A 0600    16 +2 DW 6
             17 +1
             18 +1 $NOGEN
000C 0700    19    %FOO(7,8,9)
             23    ;
```

INCLUDE

Type: General

Form: INCLUDE(*filename*)

Abbreviation: IC(*filename*)

Default: none

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 that was being processed when the INCLUDE control was encountered. If INCLUDE appears in the invocation line, then the included file is inserted before the main source file. An INCLUDE control need not be the rightmost command in a control line; however, the INCLUDE does not take effect until the end of the line. Thus, only one INCLUDE control is allowed per line. No more than 64 combinations of macro calls and INCLUDE controls may be in effect at the same time.

Note that if a file containing only control lines is INCLUDEd from the invocation line or from a primary control line, then each line of this INCLUDEd file is a primary control line, and the resumption of input from the main source file may continue the primary control lines.

LIST/NOLIST

Type: General

Form: LIST
NOLIST

Abbreviation: LI/NOLI

Default: LIST

LIST specifies that the listing of the source program in the print file is to resume with the next source line read.

NOLIST specifies that the listing of the source program in the print file, beginning with the next source line read, is to be suppressed until the next occurrence, if any, of a LIST control. However, all source lines containing errors, and the associated error message lines, do appear under NOLIST.

Note that the LIST control cannot override the NOPRINT control.

MACRO/NOMACRO

Type: Primary

Form: MACRO
MACRO(*mempercent*)
NOMACRO

Abbreviation: MR/MR (*mempercent*)/NOMR

Default: MACRO

MACRO specifies that the macro processor language is to be recognized in the source files and processed. Macros may appear anywhere in the source, including comments and control lines. Macros may also appear in the invocation line with the following two restrictions: the macro text must be at the end of the line after all controls; the macro text is limited to a maximum of 212 characters. In effect, any occurrence of the macro metacharacter in the source is considered a macro call. (Consult the *ASM86 Language Reference Manual* for a description of the macro language.)

NOMACRO specifies that the macro processor language not be recognized if it occurs in the source. If macro calls occur, they will be scanned as normal assembly language text, which will usually cause assembler errors.

The optional *mempercent* parameter for the MACRO control allows specification of the amount of memory available to the macro processor. This parameter must be a decimal number from zero to one hundred that signifies the percentage of memory allocated to the macro processor. The rest of memory is allocated to the assembler name table. THIS PARAMETER IS ONLY EFFECTIVE IN THE INVOCATION LINE. It is ignored in the primary control lines.

If an assembler source program containing macros causes either error #313 (macro space overflow) or error #906 (name table overflow), then using the memory percent parameter may remedy the problem. In the case of macro space overflow, specify a higher percentage for macros, such as MACRO(70). Conversely, in the case of name table overflow, specify a lower percentage for macros, and thus a higher percentage for the name table, such as MACRO(30). The default is about 40, depending on the version of the assembler. If a particular source module causes both errors, then there is no alternative but to divide it into smaller modules.

MOD186

Type: Primary

Form: MOD186

Abbreviation: M1

Default: 8086 mode (i.e., not MOD186)

MOD186 specifies that the iAPX186 instruction set be recognized. The eleven names: BOUND, ENTER, INS, INSB, INSW, LEAVE, OUTS, OUTSB, OUTSW, POPA, and PUSHA, become predefined symbols.

The default state is 8086 instructions only. The above eleven names are then available for user definition.

OBJECT/NOOBJECT

Type: Primary

Form: OBJECT
OBJECT(*filename*)
NOOBJECT

Abbreviation: OJ/OJ(*filename*)/NOOJ

Default: OBJECT(*sourcefile*.OBJ)

OBJECT specifies that an object module is to be created during assembly and written to the file specified. The default setting writes the object module to a file with the same name and device as the source file, but with extension .OBJ. Specifying OBJECT without a filename parameter has the same effect as the default setting.

NOOBJECT specifies that an object module not be created.

PAGELENGTH

Type: Primary

Form: PAGELENGTH(*length*)

Abbreviation: PL(*length*)

Default: PAGELENGTH(60)

PAGELENGTH specifies the number of printed lines per page in the print file. This number includes any header lines on the page. The length parameter must be a non-zero, unsigned decimal integer. The minimum pagelength is 20. PAGELENGTH is meaningless if NOPAGING is in effect.

PAGEWIDTH

Type: Primary

Form: PAGEWIDTH(*width*)

Abbreviation: PW(*width*)

Default: PAGEWIDTH(120)

PAGEWIDTH specifies the number of characters or columns per line in the print (and errorprint) file. The width parameter must be a non-zero, unsigned decimal integer. The minimum pagewidth is 60; maximum is 255. If the specified width is greater than 255, width modulo 255 is used. Print lines longer than the specified pagewidth are wrapped around to the next line in the print file.

PAGING/NOPAGING

Type: Primary

Form: PAGING
NOPAGING

Abbreviation: PI/NOPI

Default: PAGING

PAGING specifies that the print file is to be formatted into pages. A header consisting of the assembler name, the title, the date, and the page number begins each page. The symbol table listing, if present, begins on a new page. Every page is initiated with a formfeed character.

NOPAGING specifies that the print file not be formatted into pages. A single header is printed at the beginning of the file. Four blank lines separate the symbol table listing from the source listing.

PRINT/NOPRINT

Type: Primary

Form: PRINT
PRINT(*filename*)
NOPRINT

Abbreviation: PR/PR(*filename*)/NOPR

Default: PRINT(*sourcefile*.LST)

PRINT specifies that a source listing is to be created during assembly and written to the file or device specified. The default setting writes the source listing to a file with the same name and device as the source file, but with extension .LST. Specifying PRINT without a filename parameter has the same effect as the default setting.

NOPRINT specifies that a source listing not be created.

SAVE/RESTORE

Type: General

Form: SAVE
RESTORE

Abbreviation: SA/RS

Default: none

SAVE specifies that the current setting of certain general controls is to be saved on a stack. The current setting is that in effect at the beginning of the SAVE control line.

RESTORE specifies that the most recently saved settings on the stack become the current setting of the general controls.

The maximum nesting level of SAVEs is eight.

SAVE and RESTORE are not allowed in the invocation line.

The controls whose settings are saved and restored are:

LIST/NOLIST
GEN/GENONLY/NOGEN

Typically SAVE and RESTORE are used with include files, where the control settings are saved before an INCLUDE control switches the input source to another file and then restored after the end of the INCLUDEd file. Alternatively, the SAVE and RESTORE can be done in the INCLUDEd file itself, as the first and last lines respectively.

In a similar manner, SAVE and RESTORE can be used to control the listing of macros. For example, it may be desirable to establish a listing mode such that the macro call and its result are listed (i.e., a combination of NOGEN and GENONLY). In other words, the call line is listed in NOGEN mode, whereas the expansion is listed in GENONLY mode. The following example demonstrates one possible implementation. The macro BAZ SAVEs the control settings in effect at the call level, sets the mode to GENONLY for its expansion listing, and RESTOREs the call level settings upon completion. The call is made on line 26, and the expansion is listed on lines 27-30.

```
                 23        ;
                 24        $NOGEN
                 25        %*DEFINE(BAZ(D,E,F))($SAVE GENONLY
                           DW %D
                           DW %E
                           DW %F
                           $RESTORE
                           )
                 26        %BAZ(20,30,40)
0012 1400        27   +2   DW 20
0014 1E00        28   +2   DW 30
0016 2800        29   +2   DW 40
                 30   +2   $RESTORE
                 32        ;
```

SYMBOLS/NOSYMBOLS

Type: Primary

Form: SYMBOLS
NOSYMBOLS

Abbreviation: SB/NOSB

Default: NOSYMBOLS

SYMBOLS specifies that a symbol table listing is to be appended to the source listing in the print file. The symbol table is an alphabetical list of all source-defined assembler identifiers and their attributes. Macro processor identifiers are not included.

NOSYMBOLS specifies that a symbol table listing not be created.

Note that the SYMBOLS control cannot override the NOPRINT control.

TITLE

Type: General

Form: TITLE(*title*)

Abbreviation: TT(*title*)

Default: module name

TITLE specifies a character string to appear in a page header. The title parameter is a sequence of printable ASCII characters. Unquoted parentheses in the string must be balanced. In the invocation line, ampersands (&) and semicolons (;) must be enclosed in quotes. The maximum length of a title string is 60 characters; however a narrow pagewidth may restrict this further. Title strings are truncated on the right, without error, to fit the pagewidth requirement.

In the primary controls area, the TITLE control functions as a primary control. That is, a title specification in either the invocation line or primary control lines appears on the first page of the print file. Any titles in the primary control lines are ignored if a title is also specified in the invocation line.

After the primary control lines, the TITLE control functions as a general control. A title specification appears in the page header of the next page after the title control line. Note, however, that TITLE itself does not cause a new page to start. This must be done with the EJECT control or via normal paging.

Once a title is specified, it appears on all subsequent pages until changed by another TITLE control. In the absence of any TITLE controls, the module name specified in the assembler NAME directive is used for the title string in page headers.

TYPE/NOTYPE

Type: Primary

Form: TYPE
NOTYPE

Abbreviation: TY/NOTY

Default: NOTYPE

TYPE specifies that information about the types of variables output in symbols records is to be put in the object module. This information may be used later for type checking by LINK86, CREF86, or a symbolic debugger.

NOTYPE specifies that no type information is to be put in the object module.

WORKFILES

Type: Primary

Form: WORKFILES(*devicename* [,*devicename*])

Abbreviation: WF(*devicename*[,*devicename*])

Default: WORKFILES(:WORK:,:WORK:)

WORKFILES specifies devices or logical names for devices or directories for storage of assembler-created temporary work files. These intermediate files are deleted at the end of assembly. (See table 2-1 for the size of these files.) The M, X, and S files are placed on the first name in the parameter list; the T, I, and L files are placed on the second device or directory listed. A single name may be specified as the parameter; this is equivalent to specifying that name twice.

The definition of the :WORK: name used as the default is done at the operating system level. The actual filenames assigned to these intermediate files are also operating system dependent.

XREF/NOXREF

Type: Primary

Form: XREF
NOXREF

Abbreviation: XR/NOXR

Default: NOXREF

XREF specifies that a symbol table listing including cross-reference line numbers is to be appended to the source listing in the print file. The line numbers of lines where a symbol is defined, referenced, or purged follow the symbol's attributes in the listing.

NOXREF specifies that cross-reference line numbers not be included in the symbol table listing.

NOTE

The XREF control overrides the NOSYMBOLS control.
The NOXREF control does not override the SYMBOLS control.
The XREF control cannot override the NOPRINT control.

Macro Calls and Control Recognition

This section discusses the interaction of controls and macro processing.

Control lines are usually recognized and processed immediately when they appear in the source file. Several situations arise in macro processing, however, that require extension of this basic notion of control processing. It should be possible to generate entire control lines or parts of control lines as the result of macro calls. It is particularly important to provide for the conditional generation of control lines, especially INCLUDEs. This requires that it be possible to enter a control line into a macro definition (or in the body of an IF, WHILE, or REPEAT) and to delay the recognition and subsequent execution of the control line until the macro is called (or the IF, WHILE, or REPEAT is expanded).

Such a mechanism is provided by linking the scanning of control lines to the two scanning modes of the macro processor, the "normal" and "literal" scanning modes. In "normal" scanning mode all macro calls are recognized and expanded. In "literal" scanning mode all macro calls are not recognized, but are passed through as ordinary strings of text. Some examples of the "literal" scanning mode are: the body of a %*DEFINE function, the expansion of a user macro invoked by %*, inside the %(...) function, etc. More examples follow.

Control lines scanned in "normal" mode are recognized and processed. Control lines scanned in "literal" mode are not recognized. Since different portions of a line could be scanned in different modes, the exact control recognition mode is determined by the scanning mode when the control indicator, the $, is scanned. If the $ is scanned in normal mode, the rest of the line is treated as a control line and processed as such. If the $ is encountered when the macro processor is in "literal" mode, the $, as well as the rest of the line, will be treated as ordinary text. Examples of the controls encountered in each scanning mode follow.

It is important to note several items resulting from the way control lines are scanned. First, the line feed (LF) at the end of a control line must be at the same nesting level as the opening $ (i.e., no "ascending" calls are allowed). Second, a control line in a macro adds one to the macro nesting level. Finally, if a macro error occurs inside a control line, the traceback of macro nesting information includes an item for the control, as a "call" to the $.

Examples

1. Defining a macro whose definition is INCLUDEd from a side file:

```
%DEFINE(FOO)(
$INCLUDE(file1)
)
```

   Since DEFINE is called normally (i.e., with % and not %*), the body of the definition is scanned in "normal" mode. Consequently, the $INCLUDE control line is recognized immediately, and FOO is defined as the contents of the INCLUDE file (the contents of file1 are read in and stored as the value of FOO).

2. Defining a macro that INCLUDEs a file when it is called:

```
%*DEFINE(FOO)(
$INCLUDE(file2)
)
```

   Here, FOO is defined literally, using %*DEFINE, so the $INCLUDE control line itself becomes the definition of FOO. The result of calling FOO is the creation of the INCLUDE control line, at which point the file is read.

3. Conditionally INCLUDE-ing one of two files:

```
%IF(condition)THEN(
$INCLUDE(file3)
)ELSE(
$INCLUDE(file4)
)FI
```

   Both the THEN and ELSE clauses are scanned literally, and only one is expanded (in this case the selected clause is expanded normally since %IF is used rather than %*IF). As a result, only one of the two files will be INCLUDEd. In this situation, %*IF would not be useful.

4. Defining a macro that generates a control:

```
%*DEFINE(PRINT(X))(
$%X%()LIST
)
```

   Because %*DEFINE is used, the control line is not processed at the point of definition, but is delayed until the macro is actually called.

The macro call:

```
%PRINT()
```

produces the control line:

```
$LIST
```

while:

```
%PRINT(NO)
```

produces the control line:

```
$NOLIST
```

CHAPTER 4
LISTING FILE AND ERRORPRINT FILE

The Listing File

The listing file, often called the list file or print file, provides you with information on the assembly of your program. As a programming tool, it presents both assembler-generated information and user-generated information.

The example in this chapter contains some of the most used features of the ASM86 assembly language; however, it does not cover all of them. Use this example to identify where and how you might find information in the list file. As you use this chapter, it is important to note that the primary purpose of this example is to illustrate the list file; it is not intended as an example of excellent programming techniques.

Generally speaking, the listing file contains your program and object code information along with any errors produced by the assembler.

Header

Header information is at the top of the page (see figure 4-1). It identifies the assembler, the program title, the date, and the page number. The title is specified by the TITLE control. If left unspecified, the default is the module name. The date is specified by the operating system. The console operating system instructions for your system explains exactly how. The width of the page of the list file is set by the PAGEWIDTH control; the length of the page (if PAGING has been specified) is set by the PAGELENGTH control.

Additional headerlines display this information.

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER X156 ASSEMBLY OF MODULE
                        MYPROG
OBJECT MODULE PLACED IN :F5:MYPROG.OBJ
ASSEMBLER INVOKED BY:   ASM86X.86 :F5:MYPROG.A86 PAGEWIDTH(65) XRE
                        F EP
```

Beneath the headerlines is another line that prints out the names of the fields of information. Strictly speaking, these are known as fields of information; in visual terms it is easier to see them as columns. Because there is so much information, it is helpful to think of it in these broad terms:

- Any information to the left of the line number is assembler generated.
- Any information to the right of the line number is user generated.

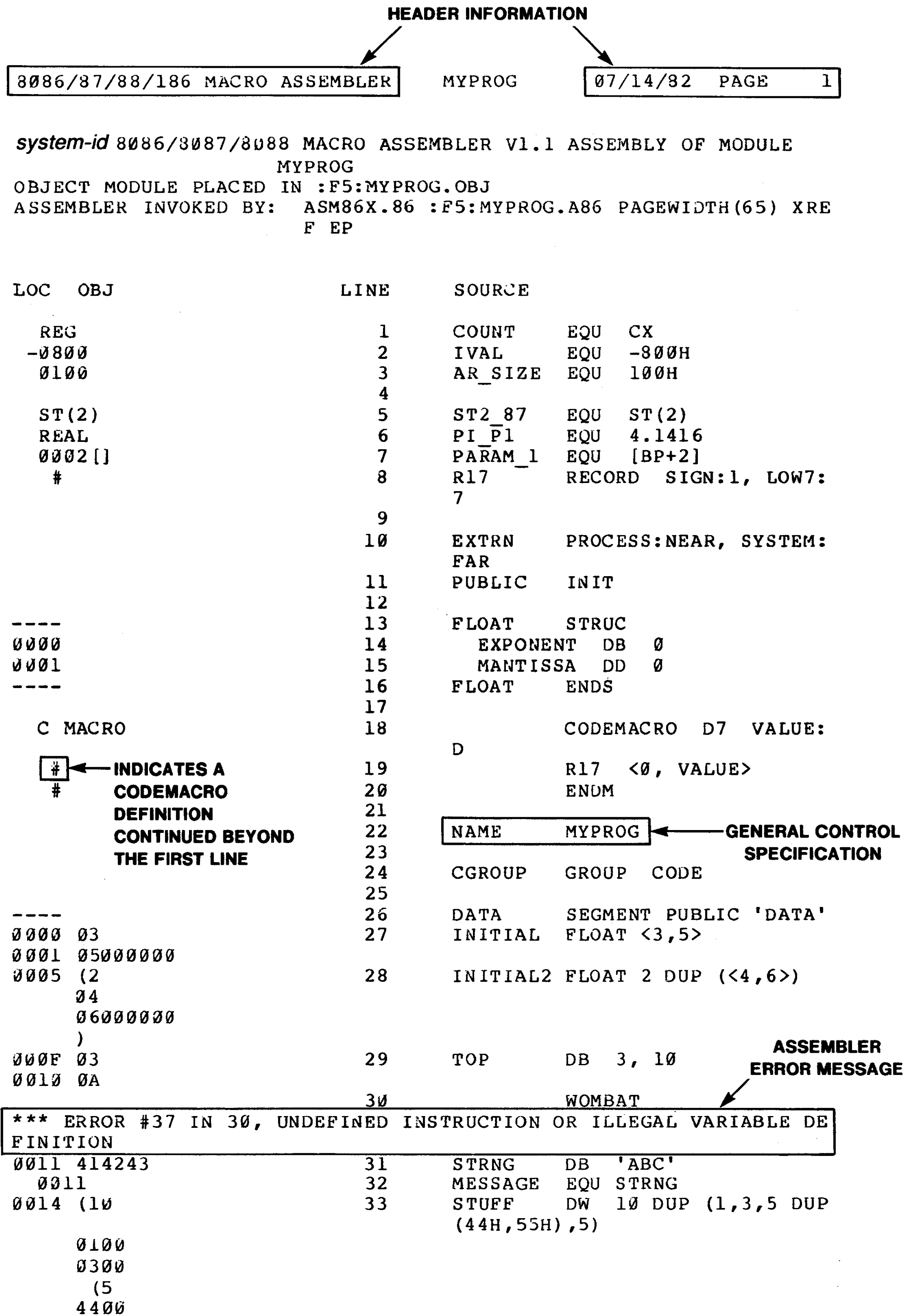
Figure 4-2 notes the fields of information in the list file.

Body

The body consists of columns of information typically organized as previously described. A discussion of the specifics of the information displayed follows.

These names identify the fields of information: LOC, the location counter; OBJ, the object code; LINE, the line number; and SOURCE, the line of source code. They appear in the following format:

```
LOC  OBJ                    LINE      SOURCE
```



```
8Ø86/87/88/186 MACRO ASSEMBLER    MYPROG    07/14/82   PAGE     1

system-id 8Ø86/8Ø87/8Ø88 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE
                    MYPROG
OBJECT MODULE PLACED IN :F5:MYPROG.OBJ
ASSEMBLER INVOKED BY:   ASM86X.86 :F5:MYPROG.A86 PAGEWIDTH(65) XRE
                        F EP

LOC  OBJ                  LINE     SOURCE

  REG                        1     COUNT     EQU   CX
 -Ø8ØØ                       2     IVAL      EQU   -8ØØH
  Ø1ØØ                       3     AR_SIZE   EQU   1ØØH
                             4
  ST(2)                      5     ST2_87    EQU   ST(2)
  REAL                       6     PI_P1     EQU   4.1416
  ØØØ2[]                     7     PARAM_1   EQU   [BP+2]
   #                         8     R17       RECORD  SIGN:1, LOW7:
                                   7
                             9
                            1Ø     EXTRN     PROCESS:NEAR, SYSTEM:
                                   FAR
                            11     PUBLIC    INIT
                            12
----                        13     FLOAT     STRUC
ØØØØ                        14       EXPONENT  DB   Ø
ØØØ1                        15       MANTISSA  DD   Ø
----                        16     FLOAT     ENDS
                            17
  C MACRO                   18               CODEMACRO  D7  VALUE:
                                   D
   #                        19               R17  <Ø, VALUE>
   #                        2Ø               ENDM
                            21
                            22     NAME      MYPROG
                            23
                            24     CGROUP    GROUP  CODE
                            25
----                        26     DATA      SEGMENT PUBLIC 'DATA'
ØØØØ  Ø3                    27     INITIAL   FLOAT <3,5>
ØØØ1  Ø5ØØØØØØ
ØØØ5  (2                    28     INITIAL2 FLOAT  2 DUP  (<4,6>)
       Ø4
       Ø6ØØØØØØ
       )
ØØØF  Ø3                    29     TOP       DB   3, 1Ø
ØØ1Ø  ØA
                            3Ø               WOMBAT
*** ERROR #37 IN 3Ø, UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DE
FINITION
ØØ11  414243                31     STRNG     DB   'ABC'
  ØØ11                      32     MESSAGE   EQU  STRNG
ØØ14  (1Ø                   33     STUFF     DW   1Ø DUP (1,3,5 DUP
                                   (44H,55H),5)

       Ø1ØØ
       Ø3ØØ
        (5
       44ØØ
```


Figure 4-1. The List File

```
LOC   OBJ                         LINE     SOURCE

      5500                  RELOCATABLE
       )                  SYMBOL INDICATOR
      0500
       )
0118 0F00            [R]       34      ITOP      DW   TOP
011A 1801----         R        35      IITOP     DD   ITOP
011E 07                        36                D7   87H
011F ----             R        37      ES_BASE   DW   EXTRA
----                           38      DATA      ENDS
                               39
----                           40      EXTRA     SEGMENT
0000 (256                      41      ARRAY1    DW   AR_SIZE DUP (?)
      ????
      )
----                           42      EXTRA     ENDS
                               43
  000A:[]                      44      AR1BX     EQU  ES:ARRAY1[BX+10]
                               45
----                           46      CODE      SEGMENT PUBLIC 'CODE'
                               47                ASSUME  DS:DATA, CS:C
                                       ODE
                               48
0000 ----             R        49      DS_BASE   DW   DATA
                               50
0002                           51      INIT      PROC   FAR
0002 B9F600                    52                MOV  COUNT, AR_SIZE -
                                        10
0005 8BD9                      53                MOV  BX, COUNT
0007 26C7470A00F8              54      INITLOOP:MOV  AR1BX, IVAL
000D E2F8                      55                LOOP  INITLOOP
000F CB                        56                RET
                               57      INIT      ENDP
                               58
0010 2E8E1E0000       R        59      START:    MOV DS, DS_BASE
0015 8E061F01         R        60                MOV ES, ES_BASE
0019 9A0200----       R        61                CALL INIT
001E E80000           E        62                CALL PROCESS
0021 9A0000----      [E]       63                CALL SYSTEM
                               64 +1   $NOGEN
   EXTERNAL SYMBOL             65     [$SAVE]  <--NOGEN IS SAVED
      INDICATOR                66     [$INCLUDE (:F5:PYG)] <--INCLUDE CONTROL
0026 00              =1        67      DB 100H
*** ERROR #39 IN 67 (:F5:PYG, LINE 1), (PASS 2) VALUE WILL NOT FI
T IN STORAGE FIELD SPECIFIED
                     =1        68
                     =1        69     [%*DEFINE (INC1(NOUN, ADJ))   (] <-- MACRO
                     =1               ;THIS %NOUN IS %ADJ)              DEFINITION
                     =1        70
                     =1        71     [$GEN]  <--GEN IS SPECIFIED
                     =1        72     [%INC1(EXAMPLE,SIMPLE)] <--MACRO CALL
                     =1        73 [+1]                          WITH PARAMETERS
                     =1        74 +1   ;THIS %NOUN
                     =1        75 +2         EXAMPLE IS %ADJ
                     =1        76 +2                    SIMPLE
INDICATES INCLUDE --> [=1]     77      INDICATES A MACRO
  EXPANSION AND                        EXTENSION LINE AND
  NESTING LEVEL                        THE NESTING LEVEL
```

Figure 4-1. The List File (Cont'd.)

```
LOC   OBJ                    LINE     SOURCE

                              78
                              79 +1  $RESTORE  <------ NOGEN IS RESTORED
                              80
----                          81     CODE      ENDS
                              82
                              83     MELLON    EUQ   AR1BX
***                                            ^
*** ERROR #1 IN 83 (LINE 72), SYNTAX ERROR
                              84
                              85     %INC1(MACRO,NOTEXPANDED)
                              87
                              88               END   START
```

Figure 4-1. The List File (Cont'd.)

LOC

The location counter is the hexadecimal number that represents the offset from the beginning of the SEGMENT or STRUCTURE being assembled. In lines that generate object code, and for LABEL or PROC, the value is the one at the beginning of the line. For ORG lines, the value shown is the new value.

```
0002                          51     INIT      PROC   FAR
```

For any other line (such as the second or third line in a Dup construction), there is no display, as shown in the following example:

```
0014 (10                      33     STUFF     DW   10 DUP (1,3,5 DUP
                                     (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
```

If there is a '----' in the LOC field, then an open or close SEGMENT or STRUCTURE statement has been coded.

```
----                          40     EXTRA     SEGMENT
```

If the LOC area is blank, either a directive or a comment has been encountered by ASM86.

```
                              47               ASSUME   DS:DATA, CS:C
```

OBJ

The object code is the hexadecimal number that displays the object bytes generated in the assembly. If there is ---- in this column, it indicates that segment base values were assembled. To the right of the OBJ field of information can be found either an R or an E or a blank area. R indicates that relocatable code has been generated; E

```
system-id 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE MYPROG
OBJECT MODULE PLACED IN 86EX.OBJ
ASSEMBLER INVOKED BY:  ASM86.86 86EX XREF PW(80)

LOC   OBJ                 LINE     SOURCE

  REG                       1      COUNT     EQU   CX
 -0800                      2      IVAL      EQU   -800H
 0100                       3      AR_SIZE   EQU   100H
  #                         4      R17       RECORD  SIGN:1, LOW7:7
                            5
                            6      EXTRN     PROCESS:NEAR, SYSTEM:FAR
                            7      PUBLIC    INIT
                            8
----                        9      FLOAT     STRUC
0000                       10        EXPONENT  DB   0
0001                       11        MANTISSA  DD   0
----                       12      FLOAT     ENDS
                           13
  C MACRO                  14                CODEMACRO  D7  VALUE:D
  #                        15                R17  <0, VALUE>
  #                        16                ENDM
                           17
                           18      NAME      MYPROG
                           19
                           20      CGROUP    GROUP  CODE
                           21
----                       22      DATA      SEGMENT PUBLIC 'DATA'
0000 03                    23      INITIAL   FLOAT <3,5>
0001 05000000
0005 03                    24      TOP       DB   3, 10
0006 0A
                           25                WOMBAT
*** ERROR #37, LINE #25, UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DEFINITION
0007 414243                26      STRNG     DB   'ABC'
 0007                      27      MESSAGE   EQU STRNG
000A (10                   28                DW   10 DUP (1,3,5 DUP(44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
010E 0500          R       29      ITOP      DW   TOP
0110 0E01----      R       30      IITOP     DD   ITOP
0114 07                    31                D7   87H
0115 ----          R       32      ES_BASE   DW   EXTRA
----                       33      DATA      ENDS
                           34
----                       35      EXTRA     SEGMENT
0000 (256                  36      ARRAY1    DW   AR_SIZE DUP (?)
     ????
     )
----                       37      EXTRA     ENDS
```

ADDITIONAL HEADER LINES

NAMES OF FIELDS OF INFORMATION

ASSEMBLER GENERATED

USER GENERATED

Figure 4-2. Fields of Information in the List File

that external code has been generated. An E takes precedence over an R on lines with both kinds of code. The following figure illustrates the location of the dashes and E or R.

```
0015 8E061F01      R     60        MOV ES, ES_BASE
0019 9A0200----    R     61        CALL INIT
001E E80000        E     62        CALL PROCESS
0021 9A0000----    E     63        CALL SYSTEM
```

Object code generated by Dups constructs has a special format. Whenever a DUP field begins, a left parenthesis appears in the left column of the object field, followed by the count in decimal numbers. The content bytes are presented left-justified on the following lines, concluded with a right parenthesis in the leftmost column. These bytes appear reversed here, since the listing has the low-order byte leftmost. For nested DUPs, the left parenthesis, number, and right parenthesis are indented one column for each nesting level, but the content bytes are never indented.

```
0014 (10                   33     STUFF     DW   10 DUP (1,3,5 DUP
                                  (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
```

EQUATE

This field is not named in the listing file but is composed of one half of the LOC field and one half of the OBJ field. The information begins in column 3.

If you equate to a variable, label, or structure field, the equate field will contain the hexadecimal offset of the symbol.

```
  0011                     32     MESSAGE   EQU STRNG
```

Variable or label equates can have segment override and indexing attributes here. A colon after the offset signals an override; square brackets signal an indexing attribute.

```
  000A:[]                  44     AR1BX     EQU   ES:ARRAY1[BX+10]
```

If you equate to a number, the field will contain the value of the number.

```
 -0800                     2      IVAL      EQU   -800H
```

If you equate to a register, segment, group, external symbol, codemacro, or record field, the equate field will contain REG, SEGMENT, GROUP, EXTRN, C MACRO, or RFIELD.

```
  REG                      1      COUNT     EQU   CX
```

The field will also contain C MACRO to signal a codemacro definition.

```
  C MACRO                  18               CODEMACRO   D7   VALUE:
                                  D
   #                       19               R17   <0, VALUE>
   #                       20               ENDM
```

Column 4 will contain a number sign (#) to indicate a record definition or the continuation of a codemacro definition.

```
   #                                      8      R17       RECORD  SIGN:1, LOW7:
```

INCLUDE Nesting Indicator

The symbol = appears in column 23 when the line is part of an INCLUDEd file. Column 24 contains the INCLUDE nesting level indicator. When the nesting level exceeds 9, a * appears in this column.

```
                              =1      69      %*DEFINE (INC1(NOUN, ADJ))  (
```

LINE

The line number is the decimal number indicating each input line, starting from 1 and incrementing with every source line. If listing of the line is suppressed (i.e., by NOLIST or NOGEN), the number increases by one anyway.

```
                                      79 +1   $RESTORE
                                      8Ø
 ----                                 81      CODE      ENDS
```

Macro Nesting Indicator

The symbol + appears in column 32 when the line is part of a macro expansion. Columns 33 and 34 contain the nesting level indicator of the macro, as shown in the following example:

```
                              =1      72      %INC1(EXAMPLE,SIMPLE)
                              =1      73 +1
                              =1      74 +1   ;THIS %NOUN
                              =1      75 +2           EXAMPLE IS %ADJ
                              =1      76 +2                     SIMPLE
```

Source Text

The source text is a copy of the source line or macro-generated text, as selected by the setting of the GEN/NOGEN/GENONLY control. For ease of reading in this list file, tabs are expanded with sufficient numbers of blank spaces to place the character (that you entered) immediately after the tab to column 1 modulo 8; this means columns 9, 17, 25, etc. If the GEN listing mode is in effect, tabs are not expanded on lines that contain macro calls or macro expansion lines. The source code information remains within the column noted as SOURCE.

Errors are included in the list file following the line in which they occurred. They are documented by error number, line number, (pass number if other than the first pass), and error message. Appendix A details recovery from error conditions.

```
                                      3Ø              WOMBAT
*** ERROR #37 IN 3Ø, UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DE
FINITION
ØØ11 414243                           31      STRNG    DB   'ABC'
```

Symbol Table

The symbol table follows the listing of the source and object code (see figure 4-3). If PAGING is in effect, the symbol table begins on a new page; otherwise, it is preceded by four blank lines. Header information identifies the assembler, the title from the last TITLE control, the date, and the page number. The listing itself is documented as the SYMBOL TABLE LISTING. Beneath that title are the columns of information:

```
NAME   TYPE   VALUE   ATTRIBUTES
```

If the XREF control has been invoked, the symbol listing is headed by the title XREF SYMBOL TABLE LISTING, and the columns of information are:

```
NAME   TYPE   VALUE   ATTRIBUTES,XREFS
```

The list of symbols is organized in alphabetic order, using the ASCII ordering of characters except for underscore, which comes first. Reserved names are not included unless they were redefined in some way.

Name

The name of the symbol appears as it was entered: periods and spaces are added to fill out the field if the name is too short. A name may be up to 31 characters long.

```
R17 . . . RECORD              SIZE=1 WIDTH=8 DEFAULT=ØØØØH   8# 19
```

Type

This is the kind of symbol that you have defined and it may be any of these:

BYTE, WORD, DWORD, QWORD, TBYTE, ABS, STRUC for variables (V); NEAR, FAR for labels (L) and procedures (P); NUMBER for numbers; REG for registers; C MACRO for codemacros, ------- for an undefined symbol; --PURGED-- for a symbol that has been purged and not redefined; and SEGMENT, STRUC, RECORD, GROUP, or RFIELD for other fields or blocks of memory.

External symbols have the type that appears in the EXTRN statement. This area of information may be shifted to accomodate the length of the name.

```
EXPONENT. V BYTE    ØØØØH  S FIELD  14#
EXTRA . . SEGMENT          SIZE=Ø2ØØH PARA    37 4Ø# 42
FLOAT . . STRUC            SIZE=ØØØ5H #FIELDS=2   13 16# 27 28
IITOP . . V DWORD   Ø11AH  DATA  35#
INIT. . . P FAR     ØØØ2H  SIZE=ØØØEH CODE PUBLIC   11 51# 57 61
INITIAL . V STRUC   ØØØØH  FLOAT DATA  27#
INITIAL2. V STRUC   ØØØ5H  FLOAT (2) DATA  28#
INITLOOP. L NEAR    ØØØ7H  CODE  54# 55
ITOP. . . V WORD    Ø118H  DATA  34# 35
IVAL. . . NUMBER   -Ø8ØØH    2# 54
LOW7. . . R FIELD     ØØH  R17 WIDTH=7  8#
```

Value

Variables and labels have their offset written as a hexadecimal number.

```
START . . L NEAR    ØØ1ØH  CODE  59# 88
STRNG . . V BYTE    ØØ11H  (3) DATA  31# 32
```

```
8086/87/88/186 MACRO ASSEMBLER     MYPROG        07/14/82   PAGE     4

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME        TYPE      VALUE   ATTRIBUTES, XREFS

??SEG . . SEGMENT             SIZE=0000H PARA PUBLIC
AR_SIZE . NUMBER     0100H      3# 41 52
AR1BX . . V WORD     000AH    ES:[BX]  44# 54
ARRAY1. . V WORD     0000H    (256) EXTRA  41# 44
CGROUP. . GROUP               CODE  24#
CODE. . . SEGMENT             SIZE=0027H PARA PUBLIC 'CODE'   24# 46
                                 47 81
COUNT . . REG        CX         1# 52 53
D7. . . . C MACRO             #DEFS=1  18 20# 36
DATA. . . SEGMENT             SIZE=0121H PARA PUBLIC 'DATA'   26# 38
                                 47 49
DS_BASE . V WORD     0000H    CODE  49# 59
ES_BASE . V WORD     011FH    DATA  37# 60
EXPONENT. V BYTE     0000H    S FIELD  14#
EXTRA . . SEGMENT             SIZE=0200H PARA    37 40# 42
FLOAT . . STRUC               SIZE=0005H #FIELDS=2  13 16# 27 28
IITOP . . V DWORD    011AH    DATA  35#
INIT. . . P FAR      0002H    SIZE=000EH CODE PUBLIC  11 51# 57 61
INITIAL . V STRUC    0000H    FLOAT DATA  27#
INITIAL2. V STRUC    0005H    FLOAT (2) DATA  28#
INITLOOP. L NEAR     0007H    CODE  54# 55
ITOP. . . V WORD     0118H    DATA  34# 35
IVAL. . . NUMBER    -0800H      2# 54
LOW7. . . R FIELD      00H    R17 WIDTH=7  8#
MANTISSA. V DWORD    0001H    S FIELD  15#
MELLON. . -------             --UNDEFINED--  83
MESSAGE . V BYTE     0011H    (3) DATA  32#
PARAM_1 . -------    0002H    [BP]  7#
PI_P1 . . NUMBER              REAL  6#
PROCESS . L NEAR     0000H    EXTRN  10# 62
R17 . . . RECORD              SIZE=1 WIDTH=8 DEFAULT=0000H  8# 19
SIGN. . . R FIELD      07H    R17 WIDTH=1  8#
ST2_87. . F STACK    ST(2)      5#
START . . L NEAR     0010H    CODE  59# 88
STRNG . . V BYTE     0011H    (3) DATA  31# 32
STUFF . . V WORD     0014H    (130) DATA  33#
SYSTEM. . L FAR      0000H    EXTRN  10# 63
TOP . . . V BYTE     000FH    (2) DATA  29# 34
WOMBAT. . -------             --UNDEFINED--  30

END OF SYMBOL TABLE LISTING

ASSEMBLY COMPLETE, 3 ERRORS FOUND
```

HEADER INFORMATION

FIELDS OF INFORMATION

LIST OF SYMBOLS

Figure 4-3. Fields of Information in the Symbol Table

External symbols always have the value of 0000H as shown in the following example.

```
SYSTEM. . L FAR     0000H  EXTRN  10# 63
```

Numbers have the value of the number, not the offset, written as a hexadecimal number (the value can be negative).

```
IVAL. . . NUMBER  -0800H     2# 54
```

Structure fields have the offset, from the structure in which defined, written as a hexadecimal number.

```
MANTISSA. V DWORD  0001H  S FIELD  15#
```

Record fields have the shift count for the record field as shown in the following example.

```
SIGN. . . R FIELD     07H  R17 WIDTH=1  8#
```

If the value is blank, you have coded one of these items: SEGMENT, GROUP, STRUC, CMACRO, RECORD, or an undefined symbol.

```
D7. . . . C MACRO             #DEFS=1  18 20# 36
DATA. . . SEGMENT             SIZE=0121H PARA PUBLIC 'DATA'   26# 38
                                 47 49
```

Symbols equated to registers have the register to which they are equated.

```
COUNT . . REG       CX        1# 52 53
```

Attributes

If the symbol is a variable defined as an array, the item count appears as a parenthesized decimal number.

```
TOP . . . V BYTE    000FH  (2) DATA  29# 34
```

If the symbol is a variable defined by a structure, the structure name is indicated.

```
INITIAL . V STRUC  0000H  FLOAT DATA  27#
```

If your symbol type is a variable or label, the Attributes field contains the name of the segment that contains the symbol definition.

If the symbol is a procedure, the size in bytes of the procedure is given.

```
INIT. . . P FAR    0002H  SIZE=000EH CODE PUBLIC  11 51# 57 61
```

External symbols always have EXTRN in this field.

```
SYSTEM. . L FAR    0000H  EXTRN  10# 63
```

For numbers, the Attributes field contains RELOC if the number is relocatable. Otherwise the field is blank.

For registers the Attributes field is blank.

If your symbol is a structure, the total size in bytes and the number of fields is given.

```
FLOAT . . STRUC              SIZE=0005H #FIELDS=2   13 16# 27 28
```

If the symbol is a record, the Attributes field indicates the number of bytes, the number of bits (width) required for that record, and the default value.

```
R17 . . . RECORD             SIZE=1 WIDTH=8 DEFAULT=0000H   8# 19
```

If the symbol is a structure field, the Attributes field contains SFIELD.

```
EXPONENT. V BYTE   0000H   S FIELD   14#
```

If the symbol is a record field, the name of the record containing the field and the number of bits required by the field are given.

```
SIGN. . . R FIELD    07H   R17 WIDTH=1   8#
```

If the symbol is a segment, this field indicates its total size in bytes, its relocatability, its align type, and its classname.

```
CODE. . . SEGMENT            SIZE=0027H PARA PUBLIC 'CODE'   24# 46
                               47 81
```

If the symbol is a group, this field lists all segments defined to be in the group.

```
CGROUP. . GROUP              CODE  24#
```

For undefined symbols, --UNDEFINED-- appears in the Attributes field.

```
WOMBAT. . --------           --UNDEFINED--  30
```

Segment overrides and indexing registers associated with a symbol appear in the Attributes field.

```
AR1BX . . V WORD   000AH   ES:[BX]   44# 54
```

For public symbols, PUBLIC appears after all other information in the Attributes field.

XREFS

If the XREF control is in effect, the Attributes field for each entry in the symbol table is followed by the line numbers where the symbol appears in the list file. A number sign (#) follows line numbers where the symbol is defined. A line number followed by a P indicates the symbol is PURGED on that line.

```
DS_BASE . V WORD   0000H   CODE   49# 59
ES_BASE . V WORD   011FH   DATA   37# 60
```

Table 4-1 summarizes the information that can be found and interpreted in the symbol table.

Table 4-1. Symbol Table Information

Type	Value	Attributes
Variable V BYTE V WORD V DWORD V QWORD V TBYTE V STRUC V *n*	offset (in hex) from segment or structure in which it was defined	1. structure name, if V STRUC 2. (item count), if any 3. a. segment name b. SFIELD if structure field 4. a. EXTRN if external b. PUBLIC or blank
V ABS	0000H	EXTRN
Label or Procedure L NEAR L FAR P NEAR P FAR	offset (in hex) from segment in which it was defined	1. Size = *nnn*H if procedure 2. segment name 3. EXTRN, PUBLIC, or blank
NUMBER	value of number (in hex)	RELOC, PUBLIC, or blank, REAL
REG	register	
SEGMENT		1. size = *nnnn*H or 64K 2. align type: PARA, PAGE, INPAGE, BYTE, WORD 3. relocatability: blank, PUBLIC, ABS MEMORY, STACK, COMMON 4. 'classname'
GROUP		segment names or SEG: external name in group
C MACRO		#DEFS = *nnn* (decimal)
STRUC		1. size = *nnnn*H 2. # FIELDS = *nn* (decimal)
RECORD		1. size = *n* (# of bytes) 2. WIDTH = *nn* (# of bits) 3. DEFAULT = *nnnn*H
R FIELD	shift count	1. record name 2. WIDTH = *nn* (# of bits)
Equate to any of the above or to address expressions with colon (:)	offset	1. segment register override *XX*: 2. indexing registers [*XX* + *YY*] 3. segment or group
F STACK	ST (i)	

The Errorprint File

If you selected ERRORPRINT as a control with assembler invocation, then all source lines containing errors and the respective messages are sent to a file or a device or directory (whichever you specified), as follows:

```
system id 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE BAD
OBJECT MODULE PLACED IN :F4:BAD.OBJ
ASSEMBLER INVOKED BY: ASM86.86 :F4:BAD.SRC ERRORPRINT(:F4:BAD.ERR)
```

```
LOC   OBJ              LINE    SOURCE

0000  909090909.0         1    MOV       AL, 100H
*** ERROR #2 IN 1, OPERANDS DO NOT MATCH THIS INSTRUCTION
```

If you selected the console device or directory as output for ERRORPRINT, or specified no device or directory, the errors will appear on :CO: followed by the standard assembler sign-off message. Here is how it looks on the console device:

```
system id 8086/87/88/186 MACRO ASSEMBLER, V2.0
0000  909090909.0         1    MOV       AL, 100H
*** ERROR #2 IN 1, OPERANDS DO NOT MATCH THIS INSTRUCTION

ASSEMBLY COMPLETE, NO WARNINGS, 1 ERROR
```

APPENDIX A ERROR MESSAGES AND RECOVERY

Console Error Messages

Upon detecting certain catastrophic conditions, ASM86 will print an informative error message to the console and abort processing. This type of fatal error handling is more severe than the fatal errors that cause error messages with numbers in the 800's and 900's, in that no listing is produced. Assembly is terminated.

These errors fall into three broad classes: control errors, I/O errors, and "others."

Control Errors

Control errors are those found in the invocation line. Such errors are reported by a message on the console. ASM86 is terminated and control is returned to the operating system. The message appears as below:

```
ASM86 CONTROL ERROR
   CONTROL:       control
   PARAMETER:     param
   DELIMITER:     char
   ERROR:         message
ASM86 TERMINATED
```

The PARAMETER and DELIMITER lines are only included when they are necessary.

The possible error messages for control errors are:

BAD COMMAND
: The control word given is not legal. Check for misspelling or see the list of legal controls in Chapter 3.

BAD DELIMITER
: Where ASM86 expected to find a valid delimiter, it has found some other character. Check to see that you use all the correct characters and that all the parameters are entered correctly.

BAD PARAMETER
: A parameter is out of bounds, or of the wrong type, or missing entirely. Check for typographical errors or consult Chapter 3.

I/O Errors

I/O errors are reported in a similar manner:

```
ASM86 I/O ERROR-
   FILE:          file type
   FILENAME:      file name
   ERROR:         operating system error number and brief description
ASM86 TERMINATED
```

The list of possible file types is:

SOURCE
PRINT
OBJECT
INCLUDE
ERRORPRINT
WORKFILE

For description of the error messages included in the I/O error indication, see the appropriate console operating instructions.

Others

The ASM86 internal errors indicate that an internal consistency check has failed. A likely cause is that one of the assembler's overlays was corrupted or that a hardware failure occurred. If the problem persists, contact Intel Corporation via the Software Problem Report in this manual.

These messages have the format:

`***ASM86 INTERNAL ERROR:` *message*

Be sure to include the exact text of the message in the problem report.

Source File Error Messages

In keeping with the high-level nature of a macro assembly language, ASM86 features an advanced error-reporting mechanism. Some messages pinpoint the symbol, character, or token at which the error was detected. Error messages are inserted into the listing after the line on which they were detected.

Non-fatal errors have the following format:

`***ERROR #`*m*, `LINE #`*n*, *message*

where

m is the error number.

n is the number of the listing line in which the error occurred. If the line is from an INCLUDE file, or if the number of the line in the source file is different from *n*, then this will be indicated by (*filename*, LINE *i*), where *filename* is the INCLUDE file, and *i* is the source file line number.

message is the English message corresponding to the error number. If the error is detected in pass 2, the clause (PASS 2) precedes the message. (MACRO) precedes the message for macro errors; (CONTROL) precedes the message for control errors.

Errors numbered less than 800 are ordinary, non-fatal errors. Assembly of the error line can usually be regarded as suspect, but subsequent lines can be assembled correctly. If an error occurs within a codemacro definition, a structure definition, or record definition, the definition does not take place.

Errors numbered in the 800's are assembler errors. They should be reported to Intel Corporation if they occur.

Errors numbered in the 900's are fatal errors. They are marked by the line FATAL ERROR preceding the message line. Assembly of the source code is halted. The remainder of the program is scanned and listed, but not acted upon.

Errors that refer to characters in a particular line of the source file do so by printing a pointer to the first character in the line that is not valid, for example:

```
***________________________________↑
```

The up-arrow or vertical bar points to the first incorrect character in the line.

A list of the error messages provided by ASM86, ordered by number, follows:

```
***ERROR #1 SYNTAX ERROR
```

This message is preceded by a pointer to the character at which the syntax error was detected.

Many times the syntax error will be at the character given in the error message. For example:

```
ASSUME DS
```

gives a syntax error after DS, meaning a line is missing things at the end — in this case, a colon followed by a segment name. More often, however, the assembler will not detect the error until one or more characters later. For example:

```
AAA DB 0
```

gives a syntax error at DB. The error is that AAA is already defined as an instruction (ASCII adjust for addition). The assembler interprets the line as an AAA instruction with DB 0 as the operand field. Since the keyword DB is not a legal parameter, the DB is flagged, even though AAA is the user's mistake.

ASM86 treats codemacro, register, and record names as unique syntactic entities; thus, when you use these kinds of names improperly you will often get a syntax error. For example:

```
SS EQU 7
```

is a syntax error since SS is a register name and thus is syntactically distinct from an undefined symbol.

Some grammatical constructs are larger than single lines, for example, SEGMENT-ENDS pairs, PROC-ENDP pairs, and CODEMACRO-ENDM pairs. You can thus get syntax errors for lines that by themselves are syntactically correct, but are misplaced within the program, for example:

```
FOO ENDS      ; with no corresponding SEGMENT statement
BAZ ENDP      ; with no corresponding PROC statement
DATA SEGMENT  ; within a codemacro
```

Note that you will get a syntax error at an END statement if you have SEGMENT or PROC statements without corresponding ENDS or ENDP statements.

ASM86 will usually discard the rest of the line when it finds a syntax error. If the error occurs within a codemacro definition, the assembler exits definition mode. This will cause your ENDM statement to produce another syntax error, which will go away when you fix the first error.

***ERROR #2 OPERANDS DO NOT MATCH THIS INSTRUCTION

This error usually indicates that the type of one of the operands is inappropriate for the instruction.

For example, the following sequence will generate error #2:

```
BAZ DW 0
MOV BL,BAZ
```

Since BAZ is a word variable, it cannot be moved into the byte register BL.

***ERROR #3 INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE

This error occurs when the instruction contains a forward reference, and the assembler guesses too optimistically about how much code the forward reference will cause the instruction to generate. There are several situations in which this happens:

1. The forward reference is a variable that requires a segment override prefix. For forward references, you must explicitly code the override if the operand is in a different segment:

   ```
   MOV CX,ES:FWD_REF
   ```

   Otherwise, the assembler will guess that it is not needed.

2. The forward reference is a FAR label. You must explicitly provide the type in this case:

   ```
   JUMP FAR PTR FWD_LABEL
   ```

   Otherwise, the assembler will guess NEAR.

3. You have promised SHORT, or you have used an instruction that takes only SHORT displacements. You must change your code not to use a SHORT jump.

***ERROR #4 INSUFFICIENT TYPE INFORMATION TO DETERMINE CORRECT INSTRUCTION

This error occurs when one of the operands to an instruction is a register expression that does not have a BYTE or WORD attribute attached to it. If one of the other operands can identify the type, then no error is issued. For example:

```
MOV AX,[BX]
MOV [BX],0FFFEH
MOV BL,[DI+500]
```

are all correct because the AX and the 0FFFEH indicate that WORD PTR[BX] is intended, and the BL indicates that BYTE PTR[DI] is intended. However:

```
INC[BX]
MOV[BX],0
```

are both flagged. The 0 does not commit [BX] to being a BYTE or a WORD memory location. You must specify BYTE PTR[BX] or WORD PTR[BX] for both instructions.

```
***ERROR #5 OPERAND NOT REACHABLE FROM SEGMENT
   REGISTERS
```

This error occurs when the ASSUME statement is used incorrectly. Every time you reference a variable, the segment in which that variable occurs must be ASSUMEd to be reachable from one of the segment registers.

For most programs, a single ASSUME statement at the top of the program for each of the four segment registers CS, DS, ES, and SS will suffice.

If you want more than one segment to be reachable from the same segment register at the same time, you must GROUP the segments together, and ASSUME the group to be reachable.

```
***ERROR #6 CANNOT JUMP NEAR TO A LABEL WITH A
   DIFFERENT CS-ASSUME
```

This error detects the following inconsistency in your program: you demand a NEAR jump to another section of code. NEAR jumps do not change the CS register. Yet the other piece of code is expecting the CS register to have a different value from the code from which you are jumping. You must either make a FAR jump, or change your CS-assume so they are consistent.

```
***ERROR #7 NO CS-ASSUME IN EFFECT--NEAR LABEL
   CANNOT BE DEFINED
```

The assembler must store the CS-ASSUME associated with each label. It needs this in order to instruct the LINK program to generate the correct displacement for NEAR jumps between different segments of the same group. For most programs, a single ASSUME statement at the top of the code will suffice.

```
***ERROR #8 NO CS-ASSUME IN EFFECT--NEAR JUMP CANNOT
   BE GENERATED
```

This is a special case of error 6: you are missing a CS-ASSUME.

```
***ERROR #9 DEFAULT SEGMENT CANNOT BE OVERRRIDDEN
```

The string imperatives that involve the DI register do not allow for an override of the default ES register; thus the assembler requires the operand to the instruction to be reachable from the ES register.

```
***ERROR #10 LABEL CANNOT BE USED AS A VARIABLE
   (NO COLON ALLOWED)
```

This error occurs when you put a colon on the label to a storage initialization line, for example:

```
FOO:DB 3
```

If your intention is to define FOO as a type label on the DB line, put the FOO: on a line by itself above the DB.

```
***ERROR #11 ILLEGAL LABEL TO THIS DIRECTIVE
   (NO COLON ALLOWED)
```

This error is reported when a label with a colon appears on a GROUP, PROC, RECORD, SEGMENT, or STRUC directive. These directives call for a label without a colon.

```
***ERROR #12 THIS DIRECTIVE REQUIRES A LABEL
   (WITHOUT A COLON)
```

This error is reported for a missing label to a GROUP, PROC, RECORD, SEGMENT, or STRUC declarative.

```
***ERROR #13 THIS DIRECTIVE DOES NOT ACCEPT A LABEL
   TO ITS LEFT
```

The ASSUME, CODEMACRO, EVEN, EXTRN, NAME, ORG, PURGE, and PUBLIC directives cannot be labeled.

```
***ERROR #14 LABEL IS NOT REACHABLE FROM CS--WILL
   NOT BE DEFINED
```

This happens when you have no ASSUME for CS, or when your CS-ASSUME is for a segment other than the one you are assembling. For example, if FOO is a segment:

```
ASSUME CS:FOO
BAZ SEGMENT
GORN PROC
```

is illegal — the assembler does not know what offset to generate for the label GORN, since GORN's segment BAZ is not ASSUMEd to be in the CS register. To correct this error, you can either provide an ASSUME CS:BAZ, or group FOO and BAZ together, and ASSUME that CS contains the group, as follows:

```
FOOBAZ GROUP FOO, BAZ
ASSUME CS:FOOBAZ
BAZ SEGMENT
GORN PROC
```

```
***ERROR #15 ALREADY DEFINED SYMBOL, THIS
   DEFINITION IGNORED
```

This message is preceded by a pointer to the previously defined symbol. This error is given when a symbol has an illegal multiple definition.

```
***ERROR #16 ALREADY EQUATED SYMBOL, THIS
   DEFINITION IGNORED
```

This message is preceded by a pointer to the previously equated symbol. This is identical to case 15, except that the name has appeared EQUated to a forward reference name that has not yet been resolved.

***ERROR #17 ARITHMETIC OVERFLOW IN EXPRESSION OR LOCATION COUNTER

This error is reported whenever a 17-bit calculation takes place whose answer is not in the bounds −65,535 to 65,535. Notable particular instances of this include:

1. User expressions with large answers or intermediate values
2. Division by zero
3. Oversize constants
4. Overflow of the location counter

***ERROR #18 ILLEGAL CHARACTER IN NUMERIC CONSTANT

Numeric constants begin with decimal digits and are delimited by the first non-token character (not alpha, numeric, /, @, or _). The set of legal characters for a constant is determined by the base:

1. Base 2: 0, 1, and the concluding B.
2. Base 8: 0-7, and the concluding O or Q.
3. Base 10: 0-9, and the optional concluding D.
4. Base 16: 0-9, A-F, and the concluding H.

***ERROR #19 ABSOLUTE, NON-FORWARD-REFERENCE, SMALL-INTEGER NUMBER REQUIRED

This error is reported in cases in which the absolute number expected cannot be completely computed at pass 1 assembly time. A small-integer number is one that can be represented in 17 bits or less (this range is from −65,535 to 65,535). Note that this excludes relocatable numbers. The situation where this is required include:

1. A SEGMENT directive with an AT
2. A DUP count
3. Widths and defaults in a RECORD definition
4. Range specifiers in a CODEMACRO definition
5. Initialization values in a CODEMACRO definition

***ERROR #20 ADDRESS EXPRESSION REQUIRED AS OPERAND TO THIS OPERATOR

Some expression operators don't make any sense if their operands are not address expressions (see *ASM86 Language Reference Manual* for a discussion of address expressions). These operators include segment override, OFFSET, bracket combination, subtraction with non-absolute minuend, SEG, TYPE, LENGTH, and SIZE (except that SIZE can be applied to a structure-name or record-name).

***ERROR #21 ILLEGAL OPERANDS TO ADDITION OR COMBINATION OPERATION

One of the operands to an addition or combination operation has to be either an absolute number or an absolute register expression. Note that this error may occur if the operation is subtraction; since if the right-hand operator is an absolute number it is negated and then added.

```
***ERROR #22 NEGATIVE NUMBER NOT ALLOWED IN THIS
    CONTEXT
```

Certain contexts disallow negative numbers. They include:

1. SEGMENT declaratives with AT
2. DUP counts

```
***ERROR #23, #24 ILLEGAL USE OF REGISTER NAME
    OUTSIDE OF BRACKETS
```

Inside of square brackets, a register can undergo arithmetic; the operations are performed on the memory address represented by the bracketed expression. Outside of the brackets, the arithmetic makes no sense and is flagged. For example:

```
JMP BX + 3
```

is illegal;

```
JMP[BX + 3]
```

is legal.

```
***ERROR #25 SHORT JUMP DISPLACEMENT DOES NOT FIT IN
    A BYTE
```

This error occurs in situations where a parameter mismatch occurs in a user prepared codemacro.

```
***ERROR #26, #27 TWO BASE OR TWO INDEX REGISTERS
    BEING COMBINED
```

At most, one base register and one indexing register are supported in an indexing expression.

```
***ERROR #28, #29, #30 BAD OPERANDS FOR RELATIONAL
    OR SUBTRACTION OPERATION
```

Subtraction and relational operations are legal only if the right side is an absolute number or if both sides match in relocation type and attributes. If neither of these conditions hold, this error is reported.

```
***ERROR #31 ILLEGAL CHARACTER
```

This message is preceded by a pointer to the illegal character.

A character that is not accepted by ASM86 was found in the input file. Either it is an unprintable ASCII character, in which case it is printed as an up arrow (↑), or it is printable but has no function in the assembly language. A likely cause of this error is the occurrence of macro functions (triggered by %) in a file that is assembled with the NOMACRO switch. Edit the file to remove the illegal character.

If an unprintable character occurs in a string or comment, the string or comment is terminated, and processing continues with the next character. If an unprintable

character occurs in a string, it will cause an error 43. Unprintable characters in strings and comments will also usually cause a syntax error.

```
***ERROR #32 INSTRUCTION OPERAND DOES NOT HAVE A
   LEGAL TYPE
```

The only case in which this error should occur is if you use a structure, structure field, record, or record field name by itself as an operand to an instruction.

```
***ERROR #33 MORE ERRORS DETECTED, NOT REPORTED
```

After the ninth error on a given source line, this message is given and no more errors are reported for the line. Normal reporting resumes on the next source line.

```
***ERROR #34 FORWARD-REFERENCE EQUATE CHAIN MAY NOT
   RESOLVE TO A REGISTER OR CODEMACRO
```

Forward references to codemacros and registers are not supported.

```
***ERROR #35 CANNOT EQUATE TO EXPRESSIONS INVOLVING
   FORWARD REFERENCES
```

You may equate to simple forward-reference names, or you may equate to expressions without forward references, but you cannot do both. For example:

```
FOO EQU BAZ + 1
BAZ EQU 5
```

is not allowed.

```
***ERROR #36 LABELS MAY NOT BE SUBSCRIPTED
```

Subscripts may not be used with labels.

```
***ERROR #37 UNDEFINED INSTRUCTION OR ILLEGAL
   VARIABLE DEFINITION
```

This error is reported when you give an undefined label, without a colon, at the beginning of a line, in a context where it cannot be taken as a variable definition. Usually this is just a misspelled instruction.

```
***ERROR #38 UNDEFINED SYMBOL, ZERO USED
```

This error is reported when an undefined symbol occurs in an expression context. The absolute number zero that is used in its place may cause other errors to occur.

```
***ERROR #39 VALUE WILL NOT FIT IN STORAGE FIELD
   SPECIFIED
```

This error is issued for DB lines in which the absolute operand is not in the range −255 to 255, for DW lines in which the absolute operand is not in the range −65,535

to 65,535, and for DD lines in which the absolute operand is not in the range −4,294,967,295 to 4,294,967,295.

```
***ERROR #40 CANNOT HAVE A VARIABLE OR A LABEL IN A
    DB, DQ, OR DT
```

This is another case where a symbol is of the wrong type for the context. Although conversion to the offset number automatically occurs for DW, it does not occur for DB, DQ, or DT — you must explicitly provide the OFFSET operator, and you must be sure that the resulting number is absolute and, in the case of DB, small enough.

```
***ERROR #41 RELOCATABLE VALUE DOES NOT FIT IN ONE
    BYTE
```

The only relocatable numbers acceptable as operands to DB (alone or within codemacros) are numbers to which HIGH or LOW have been applied.

```
***ERROR #42 STORAGE INITIALIZATION EXPRESSION IS OF
    THE WRONG TYPE
```

The only kinds of expressions allowed in initialization lists (i.e., as operands to DB, DW, DD, DQ, DT, record names, or structure names) are variables, labels, strings, formals, and numbers. Other types will produce this error.

```
***ERROR #43 STRING TERMINATED BY END-OF-LINE OR
    ILLEGAL CHARACTER
```

All strings must be completely contained on one line. The ampersand continuation feature does not work in the middle of a string. The assembler will treat the string as if you had inserted a quotation mark as the last character of your line. If a string contains an illegal character (see error 31), the string will terminate at the illegal character. An error 31 will appear also.

```
***ERROR #44 STRING LONGER THAN 2 CHARACTERS ALLOWED
    ONLY IN DB
```

Outside of the DB context, all strings are treated as absolute numbers; hence, strings of 3 or more characters are overflow quantities.

```
***ERROR #45 STRING CONSTANT CANNOT EXCEED 255
    CHARACTERS
```

The string is ignored, which may also generate a syntax error.

```
***ERROR #46 DUP NESTING ALLOWED ONLY TO A DEPTH
    OF 8
```

No reasonable program will ever run into this limitation. The kind of line that would cause it is:

```
DW 2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(3 DUP(1234H)))))))))
```

```
***ERROR #47 PARENTHESIS NESTING ALLOWED ONLY TO A
   DEPTH OF 8
```

An example of this error would be:

```
DW 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+2)))))))))
```

It is not likely that you will run into this limitation in any practical application.

```
***ERROR #48 ABSOLUTE, SMALL-INTEGER OPERAND
   REQUIRED IN THIS EXPRESSION
```

Most expression operators require their operands to be absolute numbers that can be represented in less than 17 bits. These operators include unary minus, divide, multiply, AND, MOD, NEG, OR, SHL, SHR, and XOR.

```
***ERROR #49 CANNOT TAKE HIGH OR LOW OF A PARAGRAPH
   NUMBER
```

The only kind of relocatable number that can undergo HIGH or LOW is the offset. The address of a segment does not accept HIGH or LOW.

```
***ERROR #50 OPERAND TO HIGH OR LOW MUST BE A
   VARIABLE, LABEL, OR NUMBER
```

Other types of operands (e.g., segment names, structure names, or record names) are disallowed.

```
***ERROR #51 ILLEGAL USE OF A GROUP AS A SEGMENT
   OVERRIDE
```

This error should occur only if you attempt to provide a segment override that is a group name to an expression that already has a segment override that is a group name.

```
***ERROR #52 SEGMENT OVERRIDE MAY BE APPLIED ONLY TO
   AN ADDRESS EXPRESSION
```

For example, the expression DS:0 is illegal. You must convert the number 0 into an address expression. This can be accomplished via the PTR operator, e.g., DS:BYTE PTR 0.

```
***ERROR #53 LEFT OPERAND TO SEGMENT OVERRIDE HAS AN
   ILLEGAL TYPE
```

The left operand to the segment override (colon) operator must be either a segment register, a segment name, a group name, or SEG of a variable or label.

```
***ERROR #54 LABEL MAY NOT HAVE INDEXING REGISTER
```

If the left operand to PTR is NEAR or FAR, then the right operand may not have any indexing registers. Labels with indexing registers are not supported.

```
***ERROR #55 INVALID EXPRESSION IN SQUARE BRACKETS
```

The only kind of expression allowed in square brackets is an expression involving registers and/or numbers. Address expressions and other constructs (e.g., record names) are not allowed.

```
***ERROR #56 VARIABLE AND SUBSCRIPT MAY NOT BOTH BE
    RELOCATABLE
```

Example: if FOO and BAZ are both relocatable numbers, the expressions [BX + FOO] and BAZ[BX] are both legal; the expression BAZ[BX + FOO] is not, since it requires the addition of two relocatable quantities.

```
***ERROR #57 OPERAND OF WIDTH MUST BE A RECORD OR
    RECORD FIELD NAME
```

WIDTH of anything else has no meaning.

```
***ERROR #58 OPERAND OF MASK MUST BE A RECORD FIELD
    NAME
```

MASK of anything else has no meaning.

```
***ERROR #59 OPERAND TO OFFSET MUST BE A VARIABLE OR
    LABEL
```

OFFSET is an operator provided to allow you to convert variables or labels to numbers. If you get this error message, you probably already have a number.

```
***ERROR #60 OPERAND TO LENGTH CANNOT BE A LABEL
```

LENGTH is intended to give the number of units initialized at a variable definition. Since labels are associated with instructions, not with storage initializations, LENGTH does not apply.

```
***ERROR #61 OPERAND TO SIZE CANNOT BE A LABEL
```

SIZE is intended to give the number of bytes initialized at a variable definition. Since labels are associated with instructions and with storage initializations, SIZE does not apply.

```
***ERROR #62 LEFT OPERAND TO PTR CANNOT BE ZERO
```

Besides the usual keywords BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, and FAR, you can also give a numeric value as a left operand to PTR; e.g., 3 PTR 0. This creates a variable whose constituent unit size (i.e., TYPE) is the left operand. However, 0 PTR 4 is illegal, since 0 as a constituent unit size makes no sense.

***ERROR #63 LEFT OPERAND TO PTR IS OF INVALID TYPE

The only valid left operands to PTR are absolute numbers and the keywords BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, and FAR (which are synonyms for 1, 2, 4, 8, 10, −1, and −2, respectively).

***ERROR #64 ILLEGAL NEGATIVE TYPE TO PTR, NEAR USED INSTEAD

The only negative numbers allowed as the left operand to PTR are −1 and −2, which are synonyms for NEAR and FAR. Other negative numbers are converted to NEAR, and this message is issued.

***ERROR #65 INVALID RIGHT OPERAND TO PTR

Only variables, labels, numbers, and address or register expressions may appear to the right of PTR.

***ERROR #66 CANNOT MAKE A SEGMENT REGISTER OVERRIDDEN VARIABLE INTO A LABEL

This error occurs when you have a variable with a segment register override as the right operand to PTR, and NEAR or FAR as the left operand. The resulting combination is illegal, since labels cannot be overridden.

***ERROR #67 CANNOT OVERRIDE A LABEL WITH A SEGMENT REGISTER

This, like error 66, is an attempt to create a label with a segment register override. In this case, the attempt is made via the override operator.

***ERROR #68 ILLEGAL OPERAND TO SEG OPERATOR

The operand to SEG as it appears in a GROUP or ASSUME statement must be a variable or a label; i.e., it must have a segment associated with it.

***ERROR #69 OPERAND TO SEG HAS NO SEGMENT

The operand to SEG as it appears in an expression must be a variable or a label. If not, it has no segment associated with it, and SEG therefore has no meaning.

***ERROR #70 RELOCATION OF LABEL TOO COMPLICATED

In practical programs, you should never see this error. An example of what it takes to produce it is:

```
JMP GROUPNAME:SEGNAME:FOO
```

where FOO is a label in a segment whose offsets require relocation.

***ERROR #71 SOURCE LINE CANNOT EXCEED 255 CHARACTERS

The only effect of this mistake is that the excess characters are not listed. The line is otherwise processed correctly.

***ERROR #72 ATTEMPT TO SHIFT A RELOCATABLE VALUE

This error results when a relocatable value is passed as an operand to an instruction that shifts the operand. It does not make sense to shift a relocatable value.

***ERROR #73 CANNOT PUT A RELOCATABLE VALUE INTO A RECORD OR MODRM FIELD

This error results when a relocatable value is passed as an operand to an instruction whose codemacro squeezes the operand into a record field or a MODRM field. It does not make sense to extract fields from relocatable values.

***ERROR #74 STARTING ADDRESS MUST BE A LABEL

The starting address of the program, given as an optional operand to the END statement, is the point to which the loader of the program will jump. As such, it must be a label (and not, for example, a variable or a number).

***ERROR #75 UNDEFINED RIGHT SIDE OF EQU

The left side will in this case remain undefined.

***ERROR #76 RIGHT SIDE OF EQU IS OF ILLEGAL TYPE

Only simple names and expressions are allowed on the right side of EQU. An example of a wrong type is:

```
FOO EQU 'STRING'
```

***ERROR #77 CANNOT EQU SYMBOL TO ITSELF

The example:

```
FOO EQU FOO
```

is illegal.

***ERROR #78 CIRCULAR CHAIN OF EQUATES

An example is:

```
FOO EQU BAZ
BAZ EQU FOO
```

***ERROR #79 LEFT SIDE OF EQU ALREADY DEFINED, THIS EQU IGNORED

Only previously undefined or purged names can appear to the left of EQU.

***ERROR #80 SYMBOL NOT DEFINED, CANNOT BE PURGED

If you get this message, the symbol was never defined or was already purged.

***ERROR #81 OPERAND TO ORG NOT IN THIS SEGMENT

The operand to ORG can be either an absolute number or a relocatable number. If it is relocatable, it must be offset-relocatable from the segment currently being assembled. Such a number is usually obtained by applying OFFSET to a variable or label in the current segment; for example:

```
ORG OFFSET $+2
```

***ERROR #82 ILLEGAL FORWARD REFERENCE OF A REGISTER

The only time this can happen is if you use EQU to give an alternate name to a register, but use the alternate name somewhere above the EQU statement. This is not allowed. You should always put EQUs to registers at the top of your program; in fact, we recommend that you put all your EQUs at the top of your program.

***ERROR #83 ALIGN-TYPE DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

Each SEGMENT-ENDS pair for the same segment in your program must have the same align-type. For example, you cannot specify one to be BYTE and the other to be PARA. Note that if you leave the align-type off the first SEGMENT declaration, that segment has align-type PARA. Therefore, all subsequent declarations of that segment must have either no align-type or align-type PARA. It is always acceptable to leave the align-type blank for subsequent SEGMENT declaratives — the align-type given in the first declarative is used.

***ERROR #84 COMBINE-TYPE DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

Each SEGMENT-ENDS pair for the same segment in your program must have the same combine-type. For example, you cannot specify the first one to be no combine-type (private) and a subsequent one to be PUBLIC. It is always acceptable to leave the combine-type blank for subsequent SEGMENT declaratives — the combine-type given in the first declarative is used.

***ERROR #85 CLASS DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

Each SEGMENT-ENDS pair for the same segment in your program must have the same classname. It is always acceptable to leave the classname blank for subsequent declaratives — the classname given in the first declarative is used.

```
***ERROR #86 MISMATCHED LABEL ON ENDS OR ENDP
```

ENDS and ENDP require a label that matches the corresponding SEGMENT, STRUCTURE and PROC declaratives. If this error occurs, one of several things could be wrong: a typographical error, or a missing ENDS or ENDP for a nested SEGMENT or PROC, or an error in the corresponding SEGMENT, STRUCTURE, or PROC line. In the latter case this error will go away when the other error is fixed.

```
***ERROR #87 CANNOT HAVE MORE THAN ONE NAME
    DECLARATIVE
```

The first NAME declarative is honored and this one is ignored.

```
***ERROR #88 TEXT FOUND BEYOND END STATEMENT-
    IGNORED
```

This is a warning — there are no ill effects. The extra text appears in the listing but is not assembled.

```
***ERROR #89 PREMATURE END OF FILE (NO END
    STATEMENT)
```

If your program is missing an ENDM, ENDS, or ENDP statement, the END statement is syntactically invalid and is thus not recognized. This error message will follow the syntax error message. This error will always occur if you get an error 312.

```
***ERROR #90 RECORD FIELD WIDTH MUST BE BETWEEN 1
    AND 16 BITS
```

Zero-width record fields are disallowed. Widths greater than 16 make no sense, since the containing record cannot exceed 16 bits.

```
***ERROR #91 RECORD WIDTH MAY NOT EXCEED 16 BITS
```

The record is not defined when this happens.

```
***ERROR #92 DEFAULT VALUE DOES NOT FIT INTO RECORD
    FIELD
```

The default value for the record field is too large; the number of bits needed to represent the number is greater than the width of the field.

```
***ERROR #93 INVALID LEFT OPERAND TO DOT OPERATOR
```

```
***ERROR #94 RIGHT OPERAND TO DOT OPERATOR MUST BE
    A RECORD FIELD
```

The dot operator is allowed in two contexts: (1) to select a structure field (normal usage), and (2) as a shift operator inside codemacros. Error 94 applies to the second

context, where the right operand to the dot operator must be a record field, representing the shift count. Error 116 covers errors in usage of the dot operator outside of codemacros.

Error 93 happens when the left operand to the dot operator is not a formal parameter (context 2, above), and it is not an address expression (context 1).

```
***ERROR #96 CODEMACRO NAME ALREADY DEFINED AS
    SOMETHING OTHER THAN A CODEMACRO
```

It is legal to have multiple definitions of a codemacro. In that case, however, all definitions of the symbol must be codemacro definitions. If the symbol has been defined as anything else, it cannot be redefined as a codemacro, unless it is first purged.

```
***ERROR #97 TWO FORMALS WITH THE SAME NAME
```

Within a given codemacro definition, all formals must have a different name.

```
***ERROR #98 CANNOT HAVE MORE THAN 7 FORMALS TO A
    CODEMACRO
```

This limitation is imposed by the internal codemacro coding formats.

```
***ERROR #99 ILLEGAL SPECIFIER LETTER TO A CODEMACRO
    FORMAL
```

The only specifier letters allowed are A, C, D, E, M, R, S, X, F, and T.

```
***ERROR #100 ILLEGAL MODIFIER LETTER TO A CODEMACRO
    FORMAL
```

The only modifier letters allowed are B, D, W, Q, T, and nothing.

```
***ERROR #101 ILLEGAL EXTRA CHARACTERS AFTER
    SPECIFIER AND MODIFIER
```

You have either made typographical error or have mistaken the syntax of CODEMACRO lines.

```
***ERROR #102 ONLY A, D, R, S SPECIFIERS CAN TAKE
    A RANGE
```

Range checking for codemacro matching is done only for parameters that are numbers or registers.

```
***ERROR #103 FORMAL PARAMETER EXPECTED BUT NOT SEEN
```

In certain contexts in codemacros (i.e., RELB, RELW, SEGFIX, NOSEGFIX, and MODRM), the only construct allowed is a formal parameter. If it is not seen, this error is given.

```
***ERROR #104 UNDEFINED OR FORWARD REFERENCE ILLEGAL
    IN CODEMACRO
```

All numbers provided in a codemacro definition must be determined in pass 1.

```
***ERROR #105 ILLEGAL STORAGE INITIALIZATION
    CONSTRUCT FOR A CODEMACRO
```

This error occurs when an operand to a storage initialization (DB, DW, DD, DQ, DT, or record initialization) is of illegal type; for example, a record name by itself as an operand would produce this error.

```
***ERROR #106 INSTRUCTIONS NOT ALLOWED IN
    CODEMACROS, USE INITIALIZATIONS INSTEAD
```

This error results when you place an instruction (a codemacro call) within a codemacro definition. For example:

```
CODEMACRO NOP
XCHG AX,AX
ENDM
```

is an error. You must hand-expand the codemacro with the appropriate storage initialization:

```
CODEMACRO NOP
DB 90H
ENDM
```

```
***ERROR #107 NESTED ANGLE BRACKETS NOT ALLOWED
```

For example, the construct <<0,1>,2> is flagged by this message.

```
***ERROR #108 A NULL ENTRY IS LEGAL ONLY WITHIN
    ANGLE BRACKETS
```

The line RECNAME <0,,1> is legal within a record initialization — the default value is used for the second field. However, outside of a record or structure initialization context: DB 0,,1 the null entry is not permitted.

```
***ERROR #109 DEFINITION TOO BIG FOR INTERNAL BUFFER
```

The internal storage limit for groups, records, and codemacros is 128 bytes. For groups, this is a limit of 40 segments. For records, the limit cannot be reached (you will run into the width limit before this one). The limit for codemacros is not easy to define; a rough guess is that a codemacro that generates 60 bytes of object code is near the limit. Structures have a limit of 40 fields.

```
***ERROR #110 RECORD INITIALIZATION TOO COMPLICATED
    FOR CODEMACRO ENCODING
```

The internal codemacro storage formats disallow a record initialization to produce more than 15 bytes of internal code. What this means externally is complicated to

describe, but if none of your records has more than seven fields, you should never run into this limit.

***ERROR #112 TYPE IS ILLEGAL FOR PUBLIC SYMBOL

This message is preceded by a pointer to the symbol. Only variables, labels, and numbers may be declared public. No subscripting or overrides are allowed.

***ERROR #113 NO DEFINITION FOR PUBLIC SYMBOL

This message is preceded by a pointer to the symbol. A public symbol must be defined within the program.

***ERROR #114 CANNOT ASSUME AN UNDEFINED SEGMENT OR GROUP

If a symbol is ASSUMEd into a segment register and is a forward reference, the assembler always guesses that it is a segment. If the symbol is never defined, it is an undefined segment. When a group statement lists only undefined elements the group itself remains undefined. To assume an undefined group is not allowed.

***ERROR #115 DUP COUNT MUST BE GREATER THAN ZERO

The repetition count of a DUP must be greater than 0. If it is not, 1 will be used. It is not unusual for this error to immediately follow error 22.

***ERROR #116 RIGHT OPERAND TO DOT OPERATOR IS NOT A STRUCTURE FIELD

The dot operator used outside a codemacro (see error 94) is legal only if the left operand is an address expression and the right operand is a structure field.

***ERROR #117 STRUCTURE WILL NOT BE DEFINED

Because of another error, the pending structure definition is not done. This error appears at the ENDS statement for the structure.

***ERROR #118 TOO MANY OVERRIDING INITIALIZATIONS

When using a structure to allocate and initialize storage, there are more overriding expressions between angle brackets than there are fields in the structure. All extra values at the right end of the list will be ignored, for example:

```
s STRUC
   a  DB  0
   b  D   3
   c  DW  99H
s ENDS

foo   s<1,4,0AAAH>
```

This is all right.

```
baz    s<2,5,0BBBH,93>
```

This is not. There are four overriding values and only three fields.

```
abc    s<,,,88>
```

This is also bad. Even though only one value appears, the commas force it into the fourth position, and the structure has no fourth field.

```
***ERROR #119 STRUCTURE FIELD CANNOT BE OVERRIDDEN
```

Only structure fields initialized with

- A single expression
- A single question mark, or
- A single string

may be overridden.

```
***ERROR #120 OVERRIDING STRING TOO LARGE FOR FIELD
```

If a structure field is initialized with a single string, then it can be overridden with a string that is less than or equal in length. If the overriding string is too long, then it is truncated so that it will fit into the field; if it is too short, it will be padded out by the necessary last characters from the initializing string.

```
***ERROR #121 FORWARD REFERENCE NOT ALLOWED IN
    STRUCTURE DEFINITION
```

A structure field may not be initialized by an expression containing a forward reference. Zero will be used as an initial value.

```
***ERROR #122 ILLEGAL USE OF STRUCTURE NAME
```

A structure name can appear as a storage initialization operator, as an operand of the SIZE operator, or as a type in an EXTRN or LABEL statement. Any other use of a structure name is illegal.

```
***ERROR #123 TOO MANY OVERRIDING RECORD VALUES
```

This error is similar to error 118, except that it is for records. The extra values at the right end of the list (between angle brackets) will be ignored; the record will will be formed with the remaining values.

```
***ERROR #124 FIELD MUST BE OVERRIDDEN WITH A STRING
```

If a DB structure field is initialized with a string longer than one character, then it can be overridden only with a string, not an expression or question mark.

***ERROR #125 EVEN DIRECTIVE MAY NOT BE USED IN
BYTE-ALIGNED SEGMENT

In order to guarantee even address alignment, a segment containing an EVEN directive must not be BYTE aligned.

***ERROR #126 PAGEWIDTH BELOW MINIMUM, SET TO 60

The minimum pagewidth value is 60. If a pagewidth value less than 60 is given, it is increased to 60.

***ERROR #127 PAGELENGTH BELOW MINIMUM, SET TO 20

The minimum page length value is 20. If a value less than 20 is requested, it is increased to 20.

***ERROR #128 ILLEGAL OR UNDEFINED GROUP ELEMENT

An item in the list in a GROUP statement must be a segment name and must eventually be defined. Any other item, in particular a group-name, is illegal. It is possible for the SEG operator to return a group-name if the operand to SEG was defined with EQU to have a group as its segment attribute; i.e.:

```
FOO  EQU    A_GROUP:BYTE  PTR 3
G    GROUP  SEG FOO
```

will cause this error (A_GROUP is a previously defined group).

***ERROR #129 FWD-REF EQUATE CHAIN MAY NOT RESOLVE
TO F-STACK, LONG-INT, OR REAL NUMBER

Forward references to floating-point stack elements, long integers, or real numbers are illegal.

***ERROR #130 INDEX FOR FLOATING-POINT STACK ELEMENT
MUST BE AN ABSOLUTE NUMBER

The index for a floating-point stack element must be a number or the result of an expression that can be calculated or known at assembly time. The index cannot be any form of relocatable quantity.

***ERROR #131 INDEX FOR FLOATING-POINT STACK ELEMENT
OUT-OF-RANGE

The index for the floating-point stack must be in the range 0 to 7, inclusive.

***ERROR #132 ILLEGAL USE OF LONG INTEGER CONSTANT
OR DECIMAL REAL NUMBER

The use of long integers (requiring more than 17 bits to represent) and decimal real numbers in expressions is very restricted; this message appears for some cases when a long integer or decimal real number is used illegally in an expression.

***ERROR #133 ILLEGAL OPERAND FOR UNARY MINUS OR NOT

Error 133 will occur whenever unary logical or mathematical negation is not permitted. Such cases include (but are not limited to) hex-real numbers, for which negation is disallowed, group or segment names, and labels.

***ERROR #134 CANNOT USE A RELOCATABLE NUMBER FOR DD, DQ, OR DT INITIALIZATION

External absolute numbers cannot be used in these initializations because it is impossible to determine at assembly-time how to sign-extend the number into the high-order bytes. All other relocatable numbers are disallowed for the same reason. With the exception of variables and labels, which are allowed in DD's, as long as they require 4 bytes to represent (base and offset).

***ERROR #135 NUMBER IS TOO LARGE FOR CONVERSION TO PACKED-DECIMAL FORMAT

There are some 64-bit integer values that cannot be represented in packed-decimal form. The approximate range of 64-bit binary numbers is $\pm 1.8*10^{19}$, whereas the range of values that can be represented by the packed-decimal format is approximately $-10^{18}+1$ to $10^{18}-1$.

***ERROR #136 TYPE OF R-FORMAT REAL MUST MATCH STORAGE INITIALIZATION TYPE EXACTLY

The R-format Hex-real representation permits you to specify the exact bit pattern you wish to store. The assembler will not allow you to specify the bit pattern for a DWORD (4 bytes) and place this value in a QWORD (8 bytes), since the conversion from 4 to 8 bytes would defeat the purpose of the R-format. Similarly, you may not specify 8 bytes and try to force it into a DWORD.

***ERROR #137 R-FORMAT REAL NUMBER INCORRECTLY SPECIFIED

R-format numbers must conform to the following:

Single precision	4 bytes	8 or 9 digits
Double precision	8 bytes	16 or 17 digits
Temp-real values	10 bytes	20 or 21 digits

If the number of digits is odd, then the first digit must be a 0.

***ERROR #138 INTEGER CONSTANT IS TOO LARGE

Only those values that can be represented in 64 bits can be stored internally.

***ERROR #139 CANNOT USE A DECIMAL REAL NUMBER FOR DB OR DW INITIALIZATION

No floating-point representation can fit into one byte or one word, so decimal real numbers are not allowed in DB or DW statements.

```
***ERROR #140 DECIMAL REAL NUMBER CANNOT BE
   REPRESENTED IN THE INTERNAL FORM
```

This indicates an error in conversion from decimal to temp real, which implies that the number is too large or too small to be represented in the temp-real format.

```
***ERROR #141 DECIMAL REAL NUMBER CANNOT BE
   CONVERTED TO THE STORAGE INITIALIZATION TYPE
   SPECIFIED
```

The decimal real number stored internally in the temp-real format is either too large or too small for conversion to single or double precision external representation.

```
***ERROR #142 ILLEGAL OPERAND TO THIS OPERATOR
```

The THIS operator only accepts a type specifier or a small-integer absolute number as an operand.

```
***ERROR #143 CS-IP NOT INITIALIZED, REQUIRED FOR
   MAIN MODULE
```

There is no CS-IP initialization in the END statement. This initialization, which provides the starting address, is necessary for the main module.

```
***ERROR #144 IDENTIFIER NOT A VARIABLE OR LABEL
```

An identifier that is not a variable or label is used as such in an END statement initialization.

```
***ERROR #145 IDENTIFIER MUST BE LABEL FOR A CS-IP
   INITIALIZATION
```

The identifier used in the CS-IP initialization must be a label, either:

```
CS:label
```

or

```
label
```

would be legal. Check the definition of the indicated identifier.

```
***ERROR #146 IDENTIFIER MUST BE A VARIABLE FOR
   SS-SP INITIALIZATION
```

The correct form is SS:*segname:variable*, SS:*groupname:variable* or SS:*segname*. Check to see that the indicated identifier is, indeed, a variable.

***ERROR #147 VARIABLE OR LABEL NOT ALLOWED WITH DS
INTIALIZATION

The use of variables or labels is not permitted. The only legal forms for DS initialization are DS:segname and DS:groupname.

***ERROR #148 IDENTIFIER IS NOT A SEGMENT OR GROUP

The identifier in question is expected to be a segment or group name, but is not.

***ERROR #149 INITIALIZATION OF ES IS NOT ALLOWED

You cannot initialize the ES register in the END statement.

***ERROR #150 UNDEFINED SYMBOL IN INITIALIZATION

All identifiers must be defined before they are used in an initialization.

***ERROR #151 NO NAME DIRECTIVE ENCOUNTERED,
DEFAULT MODULE NAME USED

Every module must contain the NAME directive to name the object module. If the NAME directive is omitted, then the name ANONYMOUS is used.

***ERROR #152 ILLEGAL DUPLICATE INITIALIZATION FOR
A SEGMENT REGISTER

There is more than one initialization in the END statement for the same segment register.

***ERROR #153 EXTERNAL NOT ALLOWED FOR
INITIALIZATION

Because the value of the external symbol cannot be known at assembly-time, the initialization cannot be completed.

***ERROR #154 SS INITIALIZATION WITH GROUP REQUIRES
A VARIABLE

As stated in the discussion of error #146, the correct form for SS initialization is SS:segname:variable, SS:groupname: variable, or SS:segname. You have left out the variable.

***ERROR #155 DUPLICATE PUBLIC DECLARATION FOR
SYMBOL-IGNORED

A symbol previously defined as Public is being declared Public again. The assembler ignores such duplicate declarations.

***ERROR #156 CANNOT PURGE REGISTER

A register name cannot be used in a purge directive. However, a symbol equated to a register name can be purged.

***ERROR #157 iAPX186 INSTRUCTION REQUIRES $MOD186 CONTROL

The default state of the assembler is 8086 only mode. If assembling programs written for the iAPX186, use the primary control MOD186.

Macro Error Messages

Error messages with numbers in the 300's indicate macro call/expansion errors. Macro errors are followed by a trace of the macro call/expansion stack. Each error is followed by a series of lines that print out the nesting of macro calls, expansions, include files, and so forth.

***ERROR #301 UNDEFINED MACRO NAME

The text following a metacharacter (%) is not a recognized user function name or built-in macro function. The reference is ignored and processing continues with the character following the name.

***ERROR #302 ILLEGAL EXIT MACRO

The built-in macro EXIT is not valid in this context. The call is ignored. A call to EXIT must allow an exit through a user function, or through the WHILE or REPEAT built-in functions.

***ERROR #303 FATAL SYSTEM ERROR

Loss of hardware and/or software integrity was discovered by the macro processor. Contact Intel Corporation.

***ERROR #304 ILLEGAL EXPRESSION

A numeric expression was required as a parameter to one of the built-in macros EVAL, IF, WHILE, REPEAT, and SUBSTR. The built-in function call is aborted, and processing continues with the character following the illegal expression.

***ERROR #305 MISSING "FI" IN "IF"

The IF built-in function did not have a FI terminator. The macro is processed normally.

***ERROR #306 MISSING "THEN" IN "IF"

The IF built-in function did not have a THEN clause following the conditional expression clause. The call to IF is aborted and processing continues at the point in the string at which the error was discovered.

***ERROR #307 ILLEGAL ATTEMPT TO REDEFINE MACRO

It is illegal to have a built-in function name or a parameter name be redefined (with the DEFINE or MATCH built-ins). Also, a user function cannot be redefined inside an expansion of itself.

***ERROR #308 MISSING IDENTIFIER IN DEFINE PATTERN

In a DEFINE, the occurrence of @ indicated that an identifier type delimiter followed. It did not. The DEFINE is aborted and scanning continues from the point at which the error was detected.

***ERROR #309 MISSING BALANCED STRING

A balanced string, (...) in a call to a built-in function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

***ERROR #310 MISSING LIST ITEM

In a built-in function, a parenthesized parameter is missing. The macro function call is aborted and scanning continues from the point at which the error was detected.

***ERROR #311 MISSING DELIMITER

A delimiter required by the scanning of a user-defined function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

This error can occur only if a user function is defined with a call pattern containing two adjacent delimiters. If the first delimiter is scanned, but is not immediately followed by the second, this error is reported.

***ERROR #312 PREMATURE EOF

The end of the input file occurred while the call to the macro was being scanned. This usually occurs when a delimiter to a macro call is omitted, causing the macro processor to scan to the end of the file searching for the missing delimiter. Note that even if the closing delimiter of a macro call is given, if any preceding delimiters are not given, this error may occur, since the macro processor searches for delimiters one at a time.

***ERROR #313 DYNAMIC STORAGE (MACROS OR ARGUMENTS) OVERFLOW

Either a macro argument is too long (possibly because of a missing delimiter), or not enough space is available because of the number and size of macro definitions. All pending and active macros and INCLUDEs are popped and scanning continues in the primary source file. (See also the discussion of the Macro control in Chapter 3 of this manual.)

```
***ERROR #314 MACRO STACK OVERFLOW
```

The macro context stack has overflowed. This stack is 64 deep and contains an entry for each of the following items:

1. Every currently active input file (primary source plus currently nested INCLUDEs).
2. Every pending macro call, that is, all calls to macros whose arguments are still being scanned.
3. Every active macro call, that is, all macros whose values or bodies are currently being read. Included in this category are various temporary strings used during the expansion of some built-in macro functions.

The cause of this error is excessive recursion in macro calls, expansions, or INCLUDEs. All pending and active macros and INCLUDEs are popped and scanning continues in the primary source file.

```
***ERROR #315 INPUT STACK OVERFLOW
```

The input stack is used in conjunction with the macro stack to save pointers to strings under analysis. The cause and recovery is the same as for macro stack overflow.

```
***ERROR #317 PATTERN TOO LONG
```

An element of a pattern, an identifier, or a delimiter, is longer than 31 characters, or the total pattern is longer than 255 characters. The DEFINE is aborted and scanning continues from the point at which the error was detected.

```
***ERROR #318 ILLEGAL METACHARACTER: "char"
```

The METACHAR built-in function has specified a character that cannot legally be used as a metacharacter: a blank, letter, numeral, left or right parenthesis, or asterisk. The current metacharacter remains unchanged.

```
***ERROR #319 UNBALANCED ) IN ARGUMENT TO USER
    DEFINED MACRO
```

During the scan of a user-defined macro, the parenthesis count went negative, indicating an unmatched right parenthesis. The macro function call is aborted and scanning continues from the point at which the error was detected.

```
***ERROR #320 ILLEGAL ASCENDING CALL
```

Ascending calls are not permitted in the macro language. If a call is not complete when the end of a macro expansion is encountered, this message is issued and the call is aborted. A macro call beginning inside the body of a user-defined or built-in macro was incompletely contained inside that body, possibly because of a missing delimiter for the macro call.

Control Error Messages

Control errors are announced when something is wrong with a control line in the source file.

```
***ERROR #401 BAD PARAMETER TO CONTROL
```

What appears to be the parameter to a control is not correctly formed. This may be caused if the parameter has a missing right parenthesis or if parentheses are not correctly nested, or it is out of bounds, or the wrong type, etc.

```
***ERROR #402 MORE THAN ONE INCLUDE CONTROL ON A
    SINGLE LINE
```

ASM86 allows a maximum of one INCLUDE control on a single line. If more than one INCLUDE control appears on a line, only the first (leftmost) is included, the rest are ignored.

```
***ERROR #403 BAD DELIMITER IN COMMAND
```

When scanning a command line or the invocation line, ASM86 is either looking for a letter (to start a control) or a left parenthesis (to start a parameter) or a right parenthesis (to end a parameter). If some other character is encountered, then this error is issued.

```
***ERROR #407 UNRECOGNIZED CONTROL OR MISPLACED
    PRIMARY CONTROL: control-name
```

The indicated control is not recognized as an ASM86 control in this context. It may be misspelled, mistyped, or incorrectly abbreviated.

A misplaced primary control is a likely cause of this error. Primary control lines must be at the start of the source file, preceding all non-control lines (even commments and blank lines).

```
***ERROR #408 NO TITLE FOR TITLE CONTROL
```

This error is issued if the title control has no parameter. The new title will be the empty string.

```
***ERROR #409 NO PARAMETER ALLOWED WITH ABOVE
    CONTROL
```

The following controls do not have parameters:

EJECT
SAVE
RESTORE
LIST
NOLIST
GENONLY
GEN
NOGEN

If one is included, then this error will be issued, and the parameter will be ignored.

***ERROR #410 SAVE STACK OVERFLOW

The save stack has a depth of eight. If the program tries to save more than eight levels, then this error message will be printed.

***ERROR #411 SAVE STACK UNDERFLOW

A RESTORE command is encountered and there has been no corresponding SAVE command.

***ERROR #413 SAVE, RESTORE, AND EJECT ARE NOT ALLOWED IN THE COMMAND LINE

Since these controls have no effect in the ASM86 command line, they are illegal there.

***ERROR #800 UNRECOGNIZED ERROR MESSAGE NUMBER

***ERROR #802 INTERMEDIATE FILE READING UNSYNCHRONIZED

***ERROR #803 BAD OPERAND STACK RECORD

***ERROR #804 BAD OPERAND STACK READ REQUEST

***ERROR #805 BAD OPERAND STACK POP REQUEST

***ERROR #806 PARSE STACK UNDERFLOW

***ERROR #807 AUXILIARY STACK UNDERFLOW

***ERROR #808 BAD AUXILIARY STACK READ REQUEST

***ERROR #809 BAD OPERAND STACK TYPE IN EXPRESSION

***ERROR #810 BAD STORAGE INITIALIZATION RECORD

***ERRORS #812, #813 INSTRUCTION OPERAND HAS IMPOSSIBLE TYPE

***ERROR #814 LISTING INTERMEDIATE FILE READING UNSYNCHRONIZED

Error messages in the 800's should never occur. If you get one of these error messages, and all the other errors in your program have been corrected, please notify Intel Corporation via the Software Problem Report included in this manual.

***ERROR #900 USER SYMBOL TABLE SPACE EXHAUSTED

You must either eliminate some symbols from your program, or break your program into smaller modules.

***ERROR #901 PARSE STACK OVERFLOW

This error will be given only for grammatical entities far beyond the complication seen in normal programs.

***ERROR #902 OVERFLOW IN OPERAND STACK-TOO MANY ELEMENTS

This error typically occurs when a list of storage initialization elements is too long — about 20 elements, depending on the complication of the last elements. You can correct this by breaking your initialization up into several lines.

***ERROR #903 OVERFLOW IN OPERAND STACK-ELEMENTS TOO COMPLICATED

This error is similar to error 902. You should break your list of elements into several lines.

***ERROR #904 AUXILIARY STACK OVERFLOW

This error indicates that one of ASM86's minor stacks has overflowed. This can come about through excessively complicated storage initialization operands, or by excessively deep nesting of SEGMENTs and PROCs.

***ERROR #905 INTERMEDIATE FILE BUFFER OVERFLOW

This error indicates that a single source line has generated an excessive amount of information for pass 2 processing. In practice, the limit should be reached only for lines with a gigantic number of errors — correcting the other errors should make this one go away.

***ERROR #906 USER NAME TABLE SPACE EXHAUSTED

This error indicates that the sum of the number of characters used to define the set of symbols contained in a source file exceeds the assembler's capacity. Either use shorter symbol names or break your program into smaller modules. (See also the discussion of the Macro control in Chapter 3 of this manual.)

APPENDIX B LINKING ASSEMBLY LANGUAGE AND HIGHER LEVEL LANGUAGES

This appendix describes the data passing and data definition conventions used to link assembly language programs to programs written in high-level languages. In short, it explains how programs coded in ASM86 can communicate with programs coded in such languages as PL/M-86, Pascal-86, or FORTRAN-86. Some of the information provided may also be of interest to the assembly language "purist." For example, you may want to use a high-level language procedural interface even if your entire program is coded in assembly language. For more detailed information on each higher level language, consult the user's guide for that language.

Examples are provided for the simple SMALL, COMPACT, MEDIUM, and LARGE segmentation models, as well as for subsystems. Note that FORTRAN-86 supports only the LARGE model, and Pascal-86 does not support the MEDIUM model.

The Procedural Interface

When you write assembly language procedures to be called by high-level language code, and when you call high-level language procedures from assembly language, you must conform to the procedural interface conventions used by high-level languages. Simply put, the assembly language code that "talks to" high-level code must do what high-level code expects it to do.

Passing Parameters on the 8086

All functional and procedural parameters are passed on the run-time stack. Byte, word, and integer arguments (8-bit and 16-bit) are pushed onto the 8086 stack as words. In the case of a byte argument, the value passed occupies the low-order byte of the word pushed onto the stack; the high-order byte is undefined. Double-word and long integer (32-bit) arguments are passed as two words.

Pointer parameters (addresses of variables and labels) are also pushed onto the stack. Short pointers (offsets from segment register values) are passed as words on the stack, while long pointers (complete base:offset addresses) are passed as two words: the base word is pushed first, followed by the offset word.

The first seven real arguments are passed on the 8087 register stack with each argument value occupying one 80-bit register. If there are more than seven real argument values, the rest are passed on the 8086 stack.

Parameters are pushed onto the stack in left-to-right order (Pascal-86), or in the order that they are seen in the call statement (PL/M-86 and FORTRAN-86). Since the stack grows from higher locations to lower locations, the first argument occupies the highest position on the stack. Because PL/M and FORTRAN parameters are pushed onto the stack before the CALL instruction is executed, they are located above the return address, which is also stored on the stack.

Retrieving Parameters from the Stack

A program written in assembly language and called from a high-level language may access its parameters on the stack in either of two ways. One technique is to pop each of the parameters off the stack and into either a register or a local variable. Another method of accessing parameters passed on the run-time stack is to address them using

a BP-relative addressing mode. This is the technique used by high-level language code. Establish SS:BP as a pointer to the same fixed offset as the data structure on the stack containing the parameters, and then address the parameters using offsets from BP.

Since high-level language procedures make heavy use of the BP register, assembly language code used with high-level code must preserve the value of BP. When parameters are popped off the stack, BP may be preserved by simply not using this register. However, since the "BP method" requires that BP be loaded with a new value, the contents of BP must first be saved. The method used by high-level languages is to first push its value (allowing you to safely load it with a new value), then restore its old value with a pop before the procedure returns to its caller.

Choosing a Method to Access Parameters

The method you choose for accessing parameters depends on the nature of the procedure you are writing. The pop method can be an effective optimization when all parameters are popped into registers, since accessing registers is faster than accessing memory. Consequently, the pop method should be considered first for short procedures with few parameters.

If there are a number of parameters in your procedure, however, overhead for the pop method (the sequence of POP instructions) can cancel the advantages gained from register accessing. The pop method should not be used when register space is at a premium, as in a procedure that does extensive calculations on temporary values held in the registers. Another alternative is the BP method; its big advantage over the pop method is that parameter values may be left unaltered and thus may be referenced many times in the procedure.

Returning Values From Functions

A function is a procedure that returns a single value to its caller. PL/M-86 and Pascal-86 functions return values in registers. Byte values are returned in AL, word and integer values are returned in AX, and double-word and long integer values are returned in DX:AX. Short pointers (offsets) are returned in BX, and long pointers (base:offset) are returned in ES:BX.

Table B-1 summarizes the registers used to return simple variables for PL/M-86, Pascal-86, and FORTRAN-86.

Table B-1. Registers Used to Return Simple Values

Register	PL/M-86 Type	Fortran-86 Type	Pascal-86 Type
8086:			
AL	BYTE	INTEGER*1 LOGICAL*1	CHAR, BOOLEAN, unsigned subrange, or enumeration stored in eight bits.
AX	INTEGER, WORD, or SELECTOR	INTEGER*2 LOGICAL*2	INTEGER, WORD, subrange, or enumeration stored in 16 bits.
DX:AX	DWORD	INTEGER*4 LOGICAL*4	LONGINT
ES(sgmt) BX(ofst)	POINTER (all models except SMALL RAM)		Pointer (all models except SMALL(-CONST IN DATA-))
BX(offst only)	POINTER (SMALL RAM)		Pointer (SMALL (-CONST IN DATA-) model)
8087:			
ST	REAL	REAL	REAL, LONGREAL, TEMPREAL

Register Conventions

High-level languages expect procedures and functions to preserve the values of BP, SS, and DS. In an assembly language procedure to be called from high-level language code, you must ensure that the appropriate registers are preserved. Calling a high-level language procedure from assembly languages destroys the AX, BX, CX, DX, SI, DI, and ES registers.

Models of Segmentation

In PL/M-86 and Pascal-86, there are controls that specify how program segments are to be combined and addressed in memory. These compile-time controls are called models of segmentation. The model of segmentation you choose will determine what you must put in your assembly language SEGMENT and GROUP statements. The model will also affect the particulars of the procedural interface — for example, whether long (base:offset) or short (offset) pointers should be passed as parameters.

CGROUP and DGROUP

The code for a SMALL program is stored in a segment named CODE, the data is stored in the DATA segment, and the stack in the STACK segment. Two other segments, CONST and MEMORY, are also available to hold data values. The CODE segment makes up CGROUP, which has its base in the CS register. The DATA, STACK, and MEMORY segments are all members of DGROUP, which has its base in DS (with an identical copy in SS). The CONST segment is by default a member of DGROUP. PL/M-86 and Pascal-86 allow you to put it in DGROUP by specifying -CONST IN CODE-, however, though this makes all pointers in long (32-bits).

The SMALL Model

The SMALL segmentation model is easily summarized: code in one physical segment, data and stack in another. It is used for programs that require no more than 64K of code and 64K of combined data and stack. The advantage of the SMALL model is that all pointers are merely 16-bit offsets. CS is fixed, so a JMP or CALL needs only to change IP. DS and SS are fixed — to the same value — so only an offset is needed to specify the address of a variable or item on the stack. The SMALL model offers the tightest code and fastest execution time of all the models.

The COMPACT Model

The COMPACT model of segmentation differs only slightly from the SMALL model. The CODE segment still makes up CGROUP, but now DGROUP contains only the DATA and CONST segments. (As in the SMALL case, the CONST segment is put in CGROUP only if -CONST IN CODE- is specified.) The STACK and MEMORY segments stand alone, outside of any group. As a result, these segments may occupy a full 64K bytes of memory.

Because variables on the stack have a different base from those in the data region, long pointers (base:offset) are used with the COMPACT model. This means that the POINTER date type in PL/M-86 is a two-word address, and that means the @ operator refers to a long address. Long pointers passed as parameters on the stack occupy two words, with the base part pushed first, followed by the offset part. They allow high-level language code to address data anywhere in the physical memory space.

The MEDIUM Model

In this model, DGROUP is exactly the same as it is in SMALL, containing the DATA, STACK, CONST (by default), and MEMORY segments. There is no CGROUP, however; each module produces its own, non-combinable code segment. Thus, the key feature of the MEDIUM model is that it allows large amounts of program code, while limiting the total DATA, STACK, CONST, and MEMORY segments to 64K.

Because each module produces its own code segment, inter-module calls use the long form of the CALL instruction; that is, they change both CS and IP. Therefore, calls to assembly language procedures should be declared as type FAR in the ASM86 EXTRN statement.

The LARGE Model

The LARGE model, which is the only model used by FORTRAN-86, allows for large amounts of both code and data. In this model, all code and data segments are non-combinable, and no groups are used. Constants are stored not with the DATA segments but with the CODE segments, unless you specify -CONST IN DATA-. There is still only one STACK segment, with the stack combine-type.

The LARGE model requires that inter-module calls use the long form of the CALL instruction, which saves both CS and IP in the return address. Because data references across modules refer to different base locations, all address parameters for inter-module calls should be long pointers. Each module has its own local data segment; therefore, a procedure to be called from other modules must save the caller's DS value, set up DS so that its own local variables can be addressed, and then, before returning, restore the caller's DS value.

Subsystems

A subsystem as defined in PL/M-86 and Pascal-86 is a collection of tightly coupled, logically related modules that obey the same model of segmentation. (A program can be made up of one or more subsystems.) Within a subsystem, calls and data references are long or short depending on the segmentation model chosen. Between subsystems, all calls are long, and most data references require 32-bit pointers. Any object that must be accessible to modules outside its subsystem must be exported from its subsystem.

When you declare an object in high-level languages as being exported from a subsystem, it must be declared public in ASM86 using the FAR attribute. In other words, the assembly language module should be written as though it conforms to the LARGE segmentation model.

Templates

The diagram that follow are ASM86 source module templates to be used with the SMALL, COMPACT, MEDIUM, and LARGE models of segmentation. (FORTRAN-86 uses the LARGE model only; Pascal does not support MEDIUM.) These templates show the assembly language statements that make up the framework of each of the modules.

Using the Templates

The templates are designed to be used in a "fill in the blanks" fashion. The basic statements to be copied into your source are capitalized. The italicized statements are placeholders for text to be supplied by you. These statements are instructions to you — they should not be copied into your source file.

Each template contains SEGMENT statements for all the other segments used by HLL code. You may define additional segments, as when you extend the SMALL model, and you may omit segments that you will not be using. If you omit a segment belonging to a group, you must remember not to name this segment in the GROUP statement. For example, you may be using the SMALL model and have no need for the CONST and MEMORY segments. If these are omitted from your source module, then the GROUP statement for DGROUP should only mention the DATA and STACK segments:

```
DGROUP   GROUP   DATA, STACK
```

Below each template is a notes section, which briefly summarizes some of the programming considerations associated with the model. You should keep these in mind as you build your assembly language module from a particular template.

The Small Model of Segmentation

```
NAME module-name

*CGROUP  GROUP  *CODE
 DGROUP GROUP CONSTS, DATA, STACK, MEMORY

ASSUME *CS:CGROUP, DS:DGROUP, SS-DGROUP

    CONST  SEGMENT  PUBLIC  'CONST'

   Program constants may be put here.    (optional)

    CONST  ENDS

DATA  SEGMENT  PUBLIC  'DATA'

   EXTRN external variables
   Define program data here.

DATA  ENDS

STACK  SEGMENT  STACK  'STACK'

   Use a DW statement here to add words to stack.

STACK  ENDS

MEMORY  SEGMENT  MEMORY  'MEMORY'

   This is a special data segment, above the other segments.

MEMORY  ENDS

*CODE  SEGMENT  PUBLIC  'CODE'

   EXTRN external NEAR labels, such as procedure names
   Put instruction statements here.

*CODE ENDS

END   Optional start-address, for main module only.
```

For SMALL subsystems, * = Subsystem Name (if subsystem is named) or Null String (if subsystem is not named).

Notes on the SMALL Model

- Total program code may be up to 64 bytes.
- Combined size of code, data, stack, and memory segments may be up to 64K bytes.
- The segment registers do not change: CS holds the CGROUP base; DS and SS both hold the DGROUP base.
- All procedures should be given type NEAR. (Note, however, that NEAR cannot be used with subsystems or with public and external procedures.)

- Offsets of variables are group-relative, so the group override operator (DGROUP:) must be used with the OFFSET operator and when initializing a DW to a variable's offset.
- All addresses are short pointers (offsets), except when -CONST IN CODE- is used. Thus, the PL/M-86 POINTER data type and @ operator use a short (offset) address, just like the WORD data type and dot (.) operator.
- Pascal-86 supports SMALL subsystems, PL/M-86 does not.

The Compact Model of Segmentation

```
NAME module-name

*CGROUP  GROUP  *CODE
*DGROUP  GROUP  *CONSTS, *DATA

ASSUME CS:*CGROUP, DS:*DGROUP, SS:STACK

  *CONST  SEGMENT  PUBLIC 'CONST'

 Program constants may be put here.    (Optional)

  *CONST  ENDS

*DATA  SEGMENT  PUBLIC  'DATA'

   EXTRN external variables
   Define program data here.

*DATA  ENDS

STACK  SEGMENT  STACK  'STACK'

   Use a DW statement here to add words to stack.

STACK  ENDS

MEMORY  SEGMENT  MEMORY  'MEMORY'

   This is a special data segment, above the other segments.

MEMORY  ENDS

*CODE  SEGMENT  PUBLIC  'CODE'

   EXTRN external NEAR labels, such as procedure names
   Put instruction statements here.

*CODE  ENDS

END Optional start-address, for main module only.
```

For COMPACT subsystems, *=Subsystem Name (if subsystem is named) or Null String (if subsystem is not named).

Notes on the COMPACT Model

- Total program code may be up to 64K bytes.
- Combined size of data and constant segments may be up to 64K bytes.
- The stack may be up to 64K bytes in size.
- Memory segments may be up to 64K bytes in size.

- The segment registers do not change: CS holds the base of CGROUP; DS holds the DGROUP base; and SS holds the base of the STACK segment. ES should be used to access the MEMORY segment and for indirect references using long pointers.
- All procedures should be given type NEAR. (Note that NEAR cannot be used with subsystems or with public and external procedures.)
- Offsets of variables are group-relative, so the group override operator (DGROUP:) must be used with the OFFSET operator and when initializing a DW to a variable's offset.
- The PL/M-86 POINTER data type and @ operator use a long address.
- Both Pascal-86 and PL/M-86 support COMPACT subsystems.

The Medium Model of Segmentation

```
NAME module-name

DGROUP GROUP CONSTS, DATA, STACK, MEMORY

ASSUME  CS:CGROUP, DS:DGROUP, SS--DGROUP

CONST  SEGMENT  PUBLIC  'CONST'

    Program constants may be put here.

CONST  ENDS

DATA  SEGMENT  PUBLIC  'DATA'

    EXTRN external variables
    Define program data here.

DATA  ENDS

STACK  SEGMENT  STACK  'STACK'

    Use a DW statement here to add words to stack.

STACK ENDS

MEMORY  SEGMENT  MEMORY  'MEMORY'

    This is a special data segment, above the other segments.

MEMORY  ENDS

EXTRN external FAR labels, such as procedure names

*CODE  SEGMENT  'CODE'

    Put instructions here.

*CODE  ENDS

END Optional start-address, for main module only
```

Notes on the MEDIUM Model

- Program code may exceed 64K bytes.
- Combined size of data, constant, stack, and memory segments must be less than 64K bytes.
- The DS and SS segment registers hold the base of DGROUP and do not change. ES should be used for indirect references using long pointers.
- Local procedures may have type NEAR, but all pubic and external procedures must have type FAR.

- Offsets of variables are group-relative, so the group override operator (DGROUP:) must be used with the OFFSET operator and when initializing a DW to a variable's offset.
- The PL/M-86 POINTER data type and @ operator use a long address.
- Pascal-86 and Fortran-86 do not support this model. PL/M-86 does not support MEDIUM subsystems.

The Large Model of Segmentation

```
NAME module-name

ASSUME  CS:CODE, DS:DATA, SS:STACK

EXTRN external variables

DATA SEGMENT  'DATA'

    Use a DW statement here to add words to stack.

STACK  ENDS

MEMORY  SEGMENT  MEMORY  'MEMORY'

    This is a special data segment,above the other segment.

MEMORY  ENDS

    EXTRN external FAR labels, such as procedure names

CODE  SEGMENT  'CODE'

    Put instruction statements here.

CODE  ENDS

*CODE  ENDS

END Optional start-address, for main module only.
```

Notes on the LARGE Model

- Program code may exceed 64K bytes.
- Program data may exceed 64K bytes.
- Stack may be up to 64K bytes in size.
- Memory segment may be up to 64K bytes in size.
- The SS segment register holds the base of the STACK segment and does not change.
- FORTRAN-86 supports only this module, but does not support LARGE subsystems. (PL/M-86 and Pascal-86 do support LARGE subsystems.)
- The DS segment register holds the base of the local data segment; thus, its value is different for each module. The previous value of DS should always be saved when DS is reloaded, and later restored.
- Local procedures may have type NEAR, but all public and external procedures must have type FAR.
- All pointers passed between modules must be long (base:offset) addresses. The PL/M-86 POINTER data type and @ operator use a long address.
- External variables use a different base than local variables. Thus, you must load DS or ES with the appropriate segment base before addressing an external variable.
- Large subsystem maps directly to the LARGE model.

APPENDIX C
RULES FOR SHORTENING CONTROLS

Any of the controls mentioned in this book have a legal short form. This appendix contains rules that can be used to shorten most of the controls found in Intel languages. Here are the rules:

- If the control is a one-syllable word, use the first two characters.
- If the control is a polysyllabic word, but not a compound word, use the first character from the first two syllables.
- If the control is a compound word, use the first character from each of the compounding words; however,
- If the control begins with NO, NO cannot be shortened.

APPENDIX D
USING THE 8087 NUMERIC DATA PROCESSOR AND THE 8087 EMULATOR PROGRAMS

This appendix is directed to the programmer who has an ASM86 Macro Assembly Language program that makes use of numeric instructions. The program must meet both of the following requirements:

1. It must include a declaration of the EXTERNAL FAR procedure INIT87.
2. It must execute a call to INIT87 before any numeric instruction is executed.

Assemble the program as usual with ASM86. Next, perform one of these LINK86 commands:

```
-RUN LINK86<progrm.obj>,E8087,E8087.LIB[TO<progrm.lnk>]
```

or

```
-RUN LINK86<progrm.obj>,8087.LIB[TO<progrm.lnk>]
```

If your program uses floating-point instructions, but your system does not include an 8087 Numeric Data Processor (NDP), then you must use the 8087 Emulator. The first of the preceding LINK86 commands will connect your program to E8087.LIB. LINK86 will alter your code so that the numeric instructions will access the Emulator, E8087, rather than the 8087 NDP. The Emulator library provides the following services:

- The library satisfies the call to INIT87, which initializes interrupts 20 - 31 for the Emulator. (You must reserve interrupt 16 as well if your program includes an exception handler to process numerical errors.) INIT87 contains a FINIT instruction that initializes the Emulator when it is executed.
- The object code will be altered so that the escape opcodes used by the 8087 NDP will be replaced by the interrupt opcodes used by the Emulator.

When disassembling your Emulator-linked program, you may notice the change from escape instructions to interrupt instructions. This is because a call to the Emulator interrupts execution of the calling program, while the 8087 executes those instructions as your program runs. When using the 8087 NDP, your program does not always have to wait for numeric results before it can continue. You may also notice that the list files show 8087 escape opcodes, even though you are using the Emulator, because the list files are written at assembly-time, while the code changes are made later, at link-time.

If your program makes use of numeric instructions and your system incorporates an 8087 NDP, then you will link your programs to 8087.LIB. This library contains a call to INIT87 that performs a FINIT instruction that initializes the 8087 NDP.

You may link your program's segments within the same classnames as the Emulator's segments. To do so, use the following classnames for your segments:

SEGMENT	CLASSNAME
CODE	AQMCODE
DATA	AQMDATA
STACK	STACK

There are some restrictions upon linking PL/M-86 programs and ASM86 Macro Assembly Language programs with the 8087 Emulator. A version of the 8087 Emulator is available that satisfies the numeric requirements of PL/M-86 programs.

It is referred to as the partial 8087 Emulator, PE8087. Since the partial Emulator is a subset of the full Emulator, PL/M-86 numeric instructions can be satisfied by either Emulator. Assembly language programs, on the other hand, require the full Emulator. Since you may not link both versions of the Emulator into the same program, you must use the full Emulator if you intend to link PL/M-86 and ASM86 Macro Assembly Language programs.

INDEX

8087 Emulator Programs, using, D-1
8087 Numeric Data Processor, using, D-1

ASM86, *see also* Assembler, ASM86 Macro Assembler
Assembler, ASM86 Macro Assembler
  before using, 1-1
  calling, B-1
  controls, 3-1
    shortened form, C-1
    summary of, 3-3
  defaults, 3-2
  errors, A-1
  invoking, 1-1
  parameters, 2-5
assembly language, ASM86, 1-1

body, 4-1

CGROUP, B-3

DATE (DA), 3-3
DEBUG (DB), 3-3
DGROUP, B-3

EJECT (EJ), 3-4
EQUATE, 4-6
  (CMACRO) codemacro, 4-6
  with external symbol, 4-6
  with group, 4-6
  with label, 4-6
  with number, 4-6
  with record field, 4-6
  with register, 4-6
  with segment, 4-6
  with structure field, 4-6
  with variable, 4-6
error messages and recovery, A-1
  console error messages, A-1
  control error messages, A-1
  I/O error messages, A-1
  Macro error messages, A-2
  other error messages, A-2
  source file error messages, A-2
ERRORPRINT (EP), 3-4

FORTRAN-86, linking to ASM, B-1

GEN (GE), 3-4
GENONLY (GO), 3-4

higher level languages, linking ASM to, B-1

INCLUDE (IC), 3-6
  nesting indicator (+), 4-7
iRMX86 Operating System, 2-3

LINE, 4-7
LIST (LI), 3-6
LOC field, list file, 4-4
  with STRUCTURE, 4-4

MACRO (MR), 3-7
mempercent, 3-7
MOD186 (M1), 3-7
models of segmentation, B-3
  COMPACT, B-8
  LARGE, B-12
  MEDIUM, B-10
  SMALL, B-6

OBJECT (OJ), 3-8
operating systems
  invoking the various, 2-1

PAGELENGTH (PL), 3-8
PAGEWIDTH (PW), 3-8
PAGING (PI), 3-9
parameters
  accessing, B-2
  passing, B-1
  retrieving from stack, B-1
Pascal-86, linking to ASM, B-1
PL/M-86, linking to ASM, B-1
PRINT (PR), 3-9
procedural interface, for linking higher level languages
    to ASM, B-1

register conventions, B-3
  values returned to, B-2
RESTORE (RS), 3-9

SAVE (SA), 3-9
Series III Development System, Standalone, 2-1
Series III Development System, Workstation, 2-2
source text, 4-7
SYMBOLS (SB), 3-10

templates
    for linking ASM to higher level languages, B-4
TYPE (TY), 3-11

WORKFILES (WF), 3-11

XREF (XR), 3-12

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

____
____
____
____
____
____

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

____
____
____
____
____

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

____
____
____
____
____
____

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

____
____
____
____

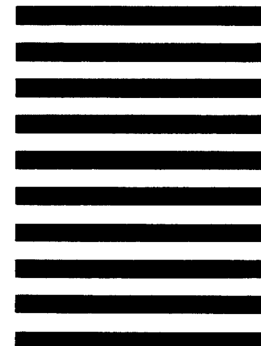
5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ______________

NAME ____ DATE ______________
TITLE ____
COMPANY NAME/DEPARTMENT ____
ADDRESS ____
CITY ________________________ STATE ________________ ZIP CODE ____________
(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

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

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

Printed in U.S.A.