ASM386 Macro Assembler Operating Instructions

Order Number: 469165-003

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:
Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641
Or you can call the following toll-free number: 1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation (Intel) 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 assumes no responsibility for any errors that may appear in this document. Intel makes no commitment to update nor to keep current the information contained in this document. No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel. Intel retains the right to make changes to these specifications at any time, without notice.

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 Agreement.
U.S. GOVERNMENT RESTRICTED RIGHTS: These software products and documentation were developed at private expense and are provided with "RESTRICTED RIGHTS." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR 52.227-14 and DFAR 252.227-7013 et seq. or its successor.

The Intel logo, i960, Pentium, and iRMX are registered trademarks of Intel Corporation, registered in the United States of America and other countries. Above, i287, i386, i387, i486, Intel287, Intel386, Intel387, Intel486, Intel487 and EtherExpress are trademarks of Intel Corporation.

Adaptec is a registered trademark of Adaptec, Inc. AT, IBM and PS/2 are registered trademarks and PC/XT is a trademark of International Business Machines Corporation. All Borland products are trademarks or registered trademarks of Borland International, Inc. CodeView, Microsoft, MS, MS-DOS and XENIX are registered trademarks of Microsoft Corporation. Comtrol is a registered trademark and HOSTESS is a trademark of Comtrol Corporation. DT2806 is a trademark of Data Translation, Inc. Ethernet is a registered trademark of Xerox Corporation. Hayes is a registered trademark of Hayes Microcomputer Products. Hazeltine and Executive 80 are trademarks of Hazeltine Corporation. Hewlett-Packard is a registered trademark of Hewlett-Packard Co. MIX® is a registered trademark of MIX Software, Incorporated. MIX is an acronym for Modular Interface eXtension. MPI is a trademark of Centralp Automatismes (S.A.). NetWare and Novell are registered trademarks of Novell Corp. NFS is a trademark of Sun Microsystems, Inc. Phar Lap is a trademark of Phar Lap Software, Inc. Soft-Scope is a registered trademark of Concurrent Sciences, inc. TeleVideo is a trademark of TeleVideo Systems, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. VAX is a registered trademark and VMS is a trademark of Digital Equipment Corporation. Visual Basic and Visual C++ are trademarks of Microsoft Corporation. All Watcom products are trademarks or registered trademarks of Watcom International Corp. Windows, Windows 95 and Windows for Workgroups are registered trademarks and Windows NT is a trademark of Microsoft in the U.S. and other countries. Wyse is a registered trademark of Wyse Technology. Zentec is a trademark of Zentec Corporation. Other trademarks and brands are the property of their respective owners.

Copyright © 1991 - 1995 Intel Corporation, All Rights Reserved

REVISION HISTORY		**DATE**
-001	Original Issue	12/91
-002	Update for Release 2.0 of the OS	08/92
-003	Update for Release 2.2 of the OS	11/95

Quick Contents

Chapter 1. Introduction

Introduces the manual, and the assembler and its related utilities.

Chapter 2. Using the Assembler

An overview of the two methods by which the assembler's actions can be controlled in the host environment: the command-line syntax and the standard assembler controls that may be embedded in program sources.

Chapter 3. Assembler Control Reference

A complete annotated list of the assembler-control switches.

Chapter 4. The Listing (Print File)

A description of the assembly listing's contents.

Appendix A. Error Messages

A descriptive list of error and warning messages issued by the assembler.

Appendix B. System Hardware and Software Requirements

A description of the hardware and software requirements, and the procedure for making required modifications to the operating system.

Index

Service Information

Notational Conventions

The notational conventions described below are used throughout this manual:

italics	Indicate a metasymbol that must be replaced with an item that fulfills the rules for that symbol.
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.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the equal sign in the following statement must be entered as shown: `PAGEWIDTH=78`
<ENTER>	Indicates a carriage return.
file-spec	Is the device name, the filename, and the file extension, if any.

Related Publications

These manuals contain information on system utilities:

- *Intel386™ Family Utilities User's Guide*
- *Intel386™ Family System Builder User's Guide*

These additional Intel manuals may be of interest to users of the assembler and utilities in the DOS environment:

- *iC-386 Compiler User's Guide*
- *PL/M-386 Programmer's Guide*

Contents

1 Introduction

About This Manual 9
About This Chapter 9
The Macro Assembler 9
The System Utilities 11
Inter-tool Consistency 12
Inter-host Portability 12

2 Using the Assembler

Command Syntax 13
- Using Controls on the Command Line 14
- Sample Invocation Commands 15
- Interrupting the Assembler 16

Controls 16
- Primary Controls 16
- General Controls 19

Using Controls in the Source File 20
Using Controls within Macros 22
File Usage 24
- Source Program Restrictions 24
- Output Files 26
- Work Files 27

Messages 28
Automation of Program Invocation and Execution 29
- DOS Batch Files 29
  - Passing Parameters to Batch Files 29
  - Using Batch Commands 30
- DOS Command Files 30
- Redirection of Command Input to Batch Files 31

3 Assembler Control Reference

DATE 34
DEBUG.......... 35
EJECT.......... 36
ERRORPRINT 37
GEN/NOGEN/GENONLY 39
INCLUDE.......... 42
LIST.......... 43
MACRO.......... 44
MOD386/MOD376/MOD486.......... 45
N387/N287.......... 47
OBJECT.......... 48
PAGELENGTH 49
PAGEWIDTH.......... 50
PAGING 51
PRINT.......... 52
SAVE/RESTORE.......... 53
SYMBOLS.......... 55
TITLE.......... 56
TYPE 58
USE32/USE16.......... 59
WORKFILES.......... 61
XREF.......... 62

4 The Listing (Print File)

The Default Print File.......... 63
Print File Headers.......... 67
Location Counter (LOC) 68
  Equated Symbols (EQU Directive) 68
  Floating-point Stack Elements (ST).......... 69
  COMM Variables and Labels 69
Object Code (OBJ).......... 70
  Relocatable or External Code (R, E).......... 70
  Include Nesting Indicator (=).......... 70
Line Numbers (LINE) 71
  Macro Expansion Indicator (+) 71
Source Statements (SOURCE).......... 71

The Symbol Table 72
Symbol Table Fields 74
Code macros (C MACRO) 74
Public and External Symbols (PUBLIC, EXTRN) 74
Floating-point Stack Elements (F STACK) 75
Instruction 75
Keyword 75
Labels (L NEAR, L FAR) 75
Numbers (NUMBER) 75
Procedures (P NEAR, P FAR) 75
Records and Record Fields (RECORD, R FIELD) 75
Registers (REG) 76
Segments (SEGMENT) 76
Stack Segments (STACK) 76
Structures and Structure Fields (STRUC, S FIELD) 76
Undefined Symbols (--------) 76
Variables (V BIT . . . V n) 77

A Error Messages

Fatal Errors 79
Invocation Control Errors 79
I/O Errors 80
Internal Errors 80
Nonfatal Errors and Warnings 81
Syntax Errors 81
Warnings 82
Macro Errors 83
Control Errors 83
Source File Error and Warning Messages 84

B System Hardware and Software Requirements

Hardware and Software Requirements 113
Modifying the System Configuration 114

Index 115

Service Information Inside Back Cover

Tables

2-1. Assembler Primary Controls 18
2-2. Assembler General Controls 19
2-3. Assembler Program Restrictions 25

Figures

1-1. Processor Translation System 10
2-1. Macro Assembler Logical File 27
3-1. Sample Listing for GEN/NOGEN/GENONLY 41
3-2. Sample Listing for SAVE/RESTORE 54
4-1. Sample Print File Page 64
4-2. Sample Symbol Table 73

Introduction 1

About This Manual

This manual describes how to use the Macro Assembler on DOS and iRMX® host systems. The information contained in this manual supplements the manual set for the ASM386 assembler and its associated utilities.

ASM386 supports the Intel386™, Intel486™, and Pentium® microprocessors as well as floating-point coprocessors. Throughout this manual, the word "processor" refers to any of the above microprocessors and the words "floating-point coprocessor" refer to any of the related math coprocessors, as well as the Intel486 and Pentium processor's built-in floating-point functions.

Bound with this manual is the *ASM386 Assembly Language Reference*. This is the basic reference for the assembler language, and contains information that is independent of the host operating system (e.g., the complete instruction set).

About This Chapter

This chapter introduces the assembler and its related utilities. The assembler generates code for target systems based on the Intel386, Intel486, and Pentium microprocessors. The utilities are tools that prepare loadable and executable modules for execution on the target system. Figure 1-1 illustrates the development process using Intel translators and utilities.

The Macro Assembler

The assembly language translator has the following characteristics:

- Translates files written in assembly language into linkable object modules
- Produces object modules (OMF-386) that can be assembled separately and linked to programs written in ASM286 assembly language

The assembler supports the full processor and floating-point coprocessor instruction sets. The instruction set and assembler mnemonics are compatible with ASM286, the assembly language for the 286 processor.

Program modules may be debugged with Intel debuggers or in-circuit emulators such as the ICE™-386 system.

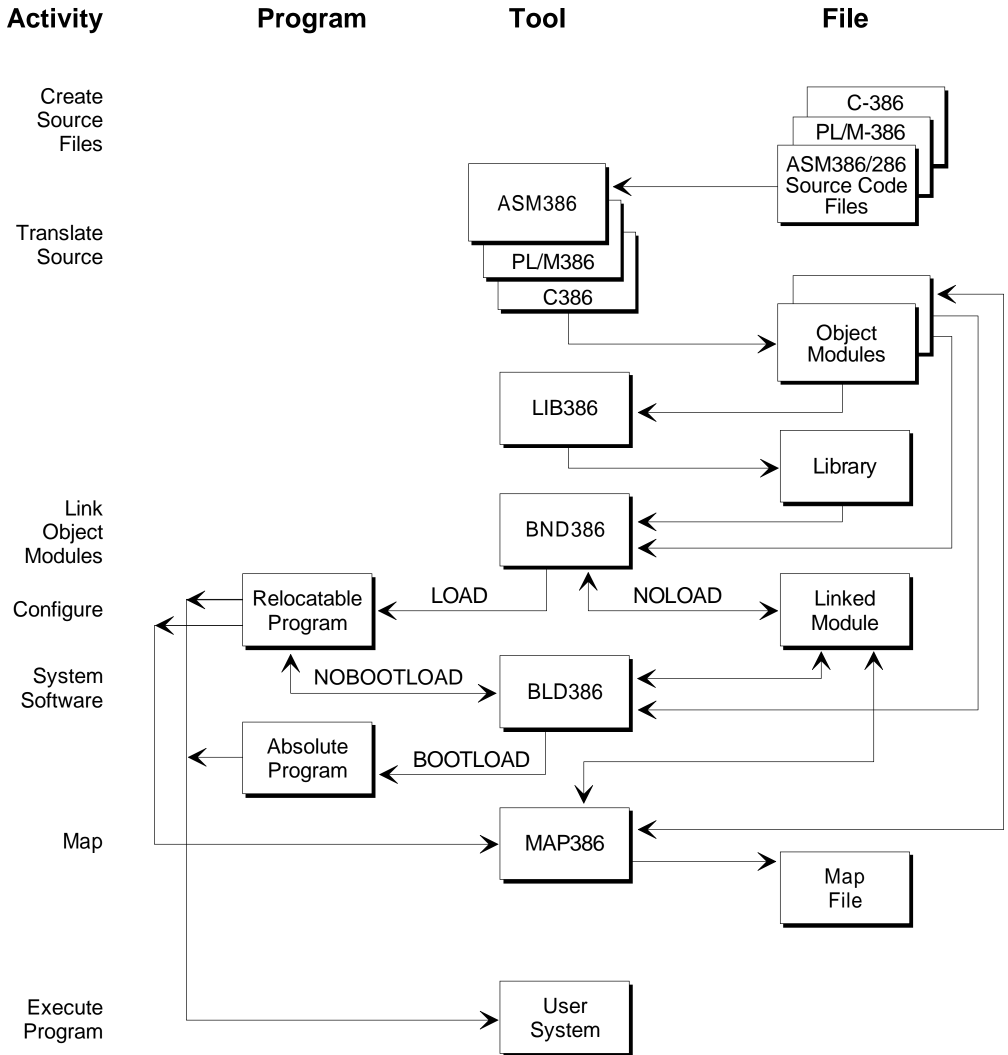


Figure 1-1. Processor Translation System

The System Utilities

The system utilities are a set of software development tools that:

- Combine modules produced by the assembler, by compilers generating OMF-386 code, and by the Librarian (LIB386) into executable programs
- Support incremental linking
- Assign addresses to code in the processor's 4-gigabyte physical address space
- Generate print files containing system cross-reference listings, error and warning messages

Object files created by the assembler must be processed by the binder (BND386) and/or the system builder (BLD386) before they can be loaded and executed. BND386 creates an executable, relocatable program from separately translated modules. The librarian (LIB386) organizes linkable modules into a library. The mapper (MAP386) produces a listing describing the features of linkable or executable object files. All three are considered linking tools.

BLD386 is not a linking tool. It configures system software for the processor operating in protected mode and using virtual addressing. BLD386 may be used to perform the following tasks:

- Creating and modifying descriptor tables, segment and system descriptors (including gates), and task state segments
- Creating page tables and directories for use in paged memory systems
- Assigning physical addresses to segments and descriptor tables
- Configuring system interface files for use in developing application programs

See also: *Intel386 Family Utilities User's Guide*
Intel386 Family System Builder User's Guide

Inter-tool Consistency

Whether or not you have previous experience using Intel software development tools, such as language translators like the assembler and related system utility programs, you will find broad consistency among the tools described in this manual. Most Intel language processors, for example, have similar invocation syntax, message formats, and features.

The various Intel assemblers and compilers for any particular target generate the same object module format. Therefore you can use the appropriate mix of assembly and higher-level language modules to develop your application system.

Inter-host Portability

The user interfaces of the various tools within a family are also consistent across host environments. This means that if you can operate an assembler or a compiler on a DOS system, for example, you already know most of what is required to operate the iRMX-hosted version of that tool.

Using the Assembler 2

This chapter explains how to anticipate and control the input and output of the assembler. It contains full textual explanations for new users and tabular summaries for those already familiar with the assembler.

Command Syntax

Assembler invocation syntax is as follows:

```
ASM386 file-spec [control...| %macro-string]
```

Where:

`ASM386` is the command.

`file-spec` represents the name and extension (optional) of the source file to be processed.

`control` represents a switch that controls the process, such as `DEBUG`, `NOOBJECT`, `PRINT` and others. You may abbreviate them as shown in Chapter 3.

`%macro-string` directs the assembler to include the specified macro. The macro string is a legal statement of up to 212 characters in the assembler macro processing language. This macro is processed before reading the source files. The macro metacharacter % must precede the macro-string, as follows:

```
ASM386 MYPROG "%set(a,1)"
```

Only one macro may be specified in each command.

See also: Using Controls with Macros, in this chapter

Using Controls on the Command Line

A set of assembler-control switches govern the format, processing, and content of both the input source and the output files. These switches are called controls and they also may be embedded in source files and included files. Controls are widely used among Intel language translators.

Most controls allow you to regulate the form and/or content of assembler output files. For example, the `USE16` control directs the assembler to generate 16-bit addresses and offsets for the current module. Some controls are in pairs that specify on/off conditions. The off condition is indicated by the word `NO` at the beginning of the name. For example, use `PRINT` to create a source listing or `NOPRINT` to suppress the listing.

Not all controls are used in commands. The `EJECT` and `SAVE/RESTORE` controls cannot be specified on the command line.

Control use is optional. If you use no controls in your invocation commands, the assembler and utilities function according to default settings described in Chapter 3 and in the other supplied publications.

For the following command example, a source module named `PROG1.SRC` is assembled using default control settings. The assembler writes the object and listing file to predetermined file specifications, using the source file name with the extensions `OBJ` and `LST`.

```
ASM386 PROG1.SRC
```

The object file is `PROG1.OBJ` and the listing is named `PROG1.LST`; both are placed in the current working directory.

Some controls take one or more parameters. Use parentheses to indicate the parameter delimiters. Separate multiple parameters by commas and enclose the entire group in parentheses.

Enclose a control's parameter in quotation characters if it contains any of the following characters:

```
, ( ) = # ! ' '' ~ + - & | < >
```

For example:

```
ASM386 MYPROG.SRC TITLE("Joe's Program")
```

If the control's parameter contains quotation characters, enclose it in apostrophes. This allows the assembler to distinguish parameter strings from strings to be parsed. For this reason, a macro or title statement in the command must also be enclosed in quotation characters.

See also: Using Controls in the Source File, in this chapter
assembler controls, Chapter 3
listing file, Chapter 4

Sample Invocation Commands

The following invocation examples show general guidelines for control usage.

1. Assume that a source file named MYPROG.SRC is in the working directory. In its simplest form, the command line is:

   ```
   ASM386 MYPROG.SRC
   ```

   The assembler uses the default values of the control settings to write the object module to the file MYPROG.OBJ and the listing to MYPROG.LST.

2. Assume that the source file is again named MYPROG.SRC and the command line is:

   ```
   ASM386 MYPROG.SRC PRINT(PROG1.LST) TITLE(PLANS)
   PAGEWIDTH(78)
   ```

   The results are:

   - The object file is named MYPROG.OBJ (the default) and the listing file is named PROG1.LST, as specified by the PRINT control.
   - TITLE places PLANS in the header of each page in the listing.
   - The pages are 78 characters wide, as specified, and 60 lines long, the default value for PAGELENGTH.

3. Assume that the source file is named MYPROG.SRC and the command is:

   ```
   ASM386 MYPROG.SRC XREF DEBUG TYPE
   ```

   This invocation results in the following:

   - By default, the object file is named MYPROG.OBJ and the listing is named MYPROG.LST.
   - The object file contains local symbol information (DEBUG) and type information (TYPE) for variables and labels. This information is useful for symbolic debugging.
   - The listing has the default format: width of 120 characters and length of 60 characters.
   - The cross-referenced symbol table listing is included at the end of the listing (XREF).

Interrupting the Assembler

Use <Ctrl-Break> to interrupt or abort the assembler. <Ctrl-C> does not work as the interrupt character like <Ctrl-Break>.

Controls

The assembler recognizes two kinds of controls, primary and general, which affect the assembly of a program as explained in Primary Controls or General Controls. Assembler control names can be abbreviated as shown in Table 2-1 and Table 2-2.

See also: Using Controls in the Source File, in this chapter

Primary Controls

Primary controls set conditions that apply throughout the entire assembly of a module. For example, the DEBUG primary control causes all local symbol information from the source module to be included in the object file. Table 2-1 lists the primary controls. The actions of the NO controls are the opposite of the descriptions of their companion control.

Place primary controls in the first line in the source file. Such lines are called primary control lines. Blank lines and comment lines are considered noncontrol lines.

If you specify the same primary control in a source file as you've entered on the command line, the command-line control's specification takes effect. If you specify a primary control in multiple primary control lines, the condition specified last takes effect.

For example, assume that the source file contains the following primary control lines:

```
$DEBUG NOPAGING
$PRINT(MYLIST)
$PAGING
```

Assume the invocation line is as follows:

```
ASM386 MYFILE.ASM PRINT NODEBUG
```

The assembly proceeds as follows:

- The source listing is sent to a file named MYFILE.LST. The default file name is the source module name with the LST extension. PRINT in a command overrides PRINT in the control line, so that the listing appears as MYFILE.LST instead of MYLIST.
- No debug information is included in the object file because NODEBUG in the command line has precedence over DEBUG in the control line.
- The listing is paged because PAGING in the third control line cancels out NOPAGING in the first control line.

Table 2-1. Assembler Primary Controls

Controls	Abbr.*	Default	Action by Assembler
DATE (*date*)	DA	System time	No effect; provided for compatibility with ASM86.
DEBUG NODEBUG	DB NODB	NODB	Places local symbol information in the object file.
ERRORPRINT[(*file-spec*)] NOERRORPRINT	EP NOEP	NOEP	Creates a file containing error or warning messages.
MACRO(*parameter*) NOMACRO	MR NOMR	MR	Specifies that macros are processed during assembly.
MOD386 MOD376 MOD486	--	MOD386	Verifies that the input file meets Intel386, 376, or Intel486 requirements, respectively.
N387 N287	--	N387	Generates code for Intel387™ or Intel287™ coprocessors.
OBJECT[(*file-spec*)] NOOBJECT	OJ NOOJ	OJ	Creates an object module.
PAGELENGTH(*length*) PAGEWIDTH(*width*)	PL PW	PL(60) PW(120)	Specifies lines/page in the listing. Specifies characters/line in the listing.
PAGING NOPAGING	PI NOPI	PI	Formats the listing in pages.
PRINT[(*file-spec*)]	PR	PR(source-file.LST)	Creates a source listing to be printed or displayed.
NOPRINT	NOPR		
SYMBOLS NOSYMBOLS	SB NOSB	NOSB	Places a symbol table in the listing.
TYPE NOTYPE	TY NOTY	NOTY	Places type information for public symbols in the object file.
USE32 USE16	U32 U16	U32	Generates 16- or 32-bit addresses and offsets for the current module.
WORKFILES(*dir*[,...])	WF	WF(:WORK:)	Names directory to contain intermediate files.
XREF NOXREF	XR NOXR	NOXR	Places a cross-referenced symbol table in the listing.

* Abbreviations may be used only in control files or in source files, not in invocation commands.

General Controls

A general control causes an immediate action and takes effect with the next source line. For example, `EJECT` places the next line of the source file listing on a new page, and `LIST` specifies that the source listing resumes with the next source line read. Table 2-2 lists the general controls.

You can specify general controls many times within a source file to set conditions during assembly. For example, you can selectively include portions of the source code in the listing by starting it with `LIST` and stopping it with `NOLIST` as desired. As another example, you can specify `INCLUDE` at selected locations in order to insert the contents of files.

The command-line equivalents of general controls take effect before the first source line is read, but have no precedence over general controls in the source file. The last setting specified is in effect.

Table 2-2. Assembler General Controls

Control	Abbr.*	Default	Assembler Function
EJECT	EJ	---	Starts a new page in the listing (print file).
GEN GENONLY NOGEN	GE GO NOGE	GENONLY	Controls the listing of macros in the listing (print file).
INCLUDE	IC	---	Inserts the specified file in the source input.
LIST NOLIST	LI NOLI	LIST	Turns the source listing on. Turns the source listing off.
SAVE	SA	---	Saves settings of affected controls on the stack.
RESTORE	RS	---	Restores settings of affected controls from the stack.
TITLE	TT		Determines the page header for the listing (print file).

* Abbreviations may be used only in control files or in source files, not in invocation commands.

Using Controls in the Source File

You can place assembler controls directly in the source file, giving you selective control over sections of the program. For example, you can suppress certain sections of the source listing with the `NOLIST` control. Placing controls within a source module can also save time because you do not need to retype them each time you invoke the assembler for a particular module.

To temporarily change a condition specified in a control line, you need not edit the source. You can simply specify the new condition using a primary control on the command line, and it takes effect because a primary control in a command has precedence over the same primary control within the source file. This technique cannot be used with general controls.

Source file lines containing controls are called control lines. They begin with the dollar sign (`$`) and contain any number of controls and their parameters, up to the host operating system's limit for characters in a source line. Control lines do not contain other types of assembly language statements. If you do not specify a control, its default value is in effect. Table 2-1 and Table 2-2 give the default value of each control.

See also: Control defaults, Chapter 3

Control lines are recognized and processed immediately when they appear in the source file except when included in macro definitions. The guidelines for specifying controls given earlier in this chapter apply to control lines with the following additions and exceptions:

- Begin the control line with a dollar sign (`$`) in column one. The first control must follow the dollar sign immediately, as follows:

  ```
  $PAGEWIDTH(132)
  ```

- Terminate control lines with a carriage return (CR).
- Separate each control with a space.
- Primary controls must be placed before any general controls or source code in a source file.
- Specify multiple controls on a single line but, unlike other assembly language statements, do not continue control lines.
- Control lines may end with comments. A comment begins with a semicolon (;) and continues for the remainder of the line. For example:

  ```
  $TITLE  (Section2)  EJECT;  next  section
  ```

- A control with a parameter uses parentheses to indicate the parameter. Multiple parameters are separated by commas and the entire group is enclosed in parentheses.
- No blanks are required between controls and parameters because the parentheses around the parameters act as separators. For example, the following two lines are equivalent:

  ```
  $PRINT  (MYPROG.PRT)  PAGEWIDTH(78)  NOPAGING
  $PRINT(MYPROG.PRT)PAGEWIDTH(78)NOPAGING
  ```

  However, you must enter blanks between controls where no parentheses act as separators. For example:

  ```
  $XREF  NOPAGING
  ```

- Enclose names of included, listing, errorprint, and object files specified within either single or double quotation marks if they contain spaces or any of the following characters:

  ```
  ' , ( ) = # ! $ % \ ~ + - & | < > [ ]
  ```

  For example:

  ```
  $INCLUDE ("Clude(s).INC")
  ```

See also: Specifying controls, in this chapter
Using Controls within Macros, in this chapter

In the following example, controls specified at invocation can override controls within the source file.

Example

Assume that the source file `MYPROG.ASM` contains the following control line:

```
$SYMBOLS  PAGEWIDTH(60)
```

The command is:

```
ASM386 MYPROG.SRC NOPRINT
```

In this case, the control lines do not produce the usual results. Normally, `SYMBOLS` adds a symbol table to the listing and `PAGEWIDTH` sets the width of lines in the listing. Placing `NOPRINT` on the command line causes `SYMBOLS` and `PAGEWIDTH` to be ignored because no listing is generated.

The following section explains some of the considerations for specifying controls within macro definitions.

Using Controls within Macros

The assembler usually recognizes and processes control lines as soon as they appear in a source file. However, the assembler can conditionally generate control lines if you place them within macro definitions or the body of a statement containing the `IF`, `WHILE`, or `REPEAT` predefined macros. The assembler then delays recognition and execution of the control line until the macro is called or the `IF`, `WHILE`, or `REPEAT` is expanded.

The assembler macro processor has two scanning modes: normal and literal. In normal scanning mode, the assembler recognizes and expands all macros. In literal scanning mode, the assembler treats nested macro calls as ordinary text strings.

Literal mode is selected by placing an asterisk (*) after the macro metacharacter, which is % by default, as in the following example:

```
%*DEFINE(AB) (%EVAL(%TOM))
```

Literal mode is also in effect by default for `THEN` and `ELSE` clauses, because these clauses are conditional in nature. The examples at the end of this section also illustrate these concepts.

See also: Macro processing language and scanning modes, *ASM386 Assembly Language Reference*

The scanning mode in effect when the control line indicator `$` is scanned determines how the assembler processes a control line. If the `$` is encountered when the macro processor is in normal mode, the assembler treats the rest of the line as a control line and processes it immediately. If the `$` is scanned in literal mode, the `$` and the rest of the line are treated as ordinary text.

The following criteria apply to the way control lines are scanned:

- The line feed (LF) at the end of a control line must be at the same nesting level as the opening `$`. Parentheses must be used in pairs.
- A control line in a macro adds one level to the macro nesting.
- If a macro error occurs inside a control line, the traceback of macro nesting information includes an item for the control, as a "call" to the `$`.

The following examples illustrate the use of controls in macros.

Examples

1. This example shows a macro whose definition is included from another file:

```
%DEFINE(MAC) (
$INCLUDE(FILE1)
)
```

   Because DEFINE is called normally, with % instead of %*, the body of the definition is scanned in normal mode. Consequently, the INCLUDE control line is recognized immediately and MAC is defined as the contents of the INCLUDE file. In other words, the contents of FILE1 are stored as the value of MAC.

2. The following example shows the definition of a macro that includes a file when it is called:

```
 %*DEFINE(MAC) (
$INCLUDE(FILE2)
)
```

   MAC is scanned in literal mode because of the %* notation preceding the DEFINE function. Consequently, the INCLUDE control line itself is the definition of MAC, not the contents of FILE2, as would have been the case in normal mode (refer to the previous example). When MAC is called, FILE2 is read.

3. The following example illustrates how to conditionally include one of two files using IF..THEN..ELSE clauses:

```
%IF(condition) THEN (
$INCLUDE(FILE3)
)ELSE(
$INCLUDE(FILE4)
)FI
```

   This example demonstrates how scanning modes are combined. Even though IF is not preceded by %*, both the THEN and ELSE clauses are scanned in literal mode because they are conditional statements. However, because %*IF was not used, the selected THEN or ELSE clause is scanned in normal mode and FILE3 or FILE4 is immediately included. In this situation, %*IF would not be useful. IF must always be closed by an FI after the last parenthesis.

4. The following example shows the definition of a macro that generates either the `LIST` or `NOLIST` control:

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

   The macro call

   ```
   %PRINT()
   ```

   would produce the control line

   ```
   $LIST
   ```

   while the call

   ```
   %PRINT(NO)
   ```

   would produce the control line

   ```
   $NOLIST
   ```

File Usage

File sharing conflicts may occur when using an Intel translator or Relocation and Linkage (R&L) tool in a DOS network environment. Before invoking an Intel translator or R & L tool (with network support from DOS), invoke the DOS V3.0 or later SHARE command. It is recommended that you invoke the SHARE command in your AUTOEXEC.BAT file.

After successful assembly, the assembler can produce an object file, a listing, and an errorprint file. Each of these files is optional; certain assembler controls allow you to specify them. Other assembler controls allow you to regulate their contents.

Source Program Restrictions

The assembler places quantitative restrictions on certain items within source programs, such as the number of characters in a source line. Most of these restrictions are listed in Table 2-3.

If your program exceeds a limit, the assembler returns an error. Most quantities in Table 2-3 are upper limits, but some items show both upper and lower limits. The table also points out some items for which there are no limits.

Table 2-3. Assembler Program Restrictions

Item	Limit
Source lines/programs	No limit
Characters per line Characters per identifier	255 including CR/LF and nonprinting characters 31 unique, up to 255 total
Symbols per module	2700 standard 3200 with Intel Above™ Board
Continued lines per statement	No limit
Characters per string	255 including enclosing quotation marks
Segment size	64K bytes (USE16 segments) 4 gigabytes (USE32 segments)
Number of bytes	Size cannot exceed 1K that can be duplicated
Procedure nesting	20 levels per segment
Items in each PUBLIC, EXTRN, and PURGE directive	Limited by the number of symbols
Parameters in all macro calls	255
Combined total of macro calls, active macro expansions, and nested INCLUDEs	64 levels
Items per storage initialization list	No limit
Fields per record	32
Record size	32 bits
Structure size	64K - 1 bytes (USE16) 4 gigabytes -1 (USE32)
Fields per structure	255
Parameters per codemacro definition	15
Bytes generated by each codemacro	255

Output Files

There are three possible output files: the object file, the listing or print file, and the errorprint file.

The object file contains assembled processor machine code, data initializations, symbol and type information, and the information necessary for combining the object module with other modules. After processing by the processor system utilities, the object file for a source module becomes part of an executable program. The assembler produces the object file by default unless you specify the `NOOBJECT` control.

The object file can optionally contain symbol and type information that is useful for symbolic debugging and inter-module type checking. The information is included if you specify the assembler's `DEBUG` and `TYPE` controls.

The listing file (or print file) contains the source lines, expanded macro source code, object code, and any source file error or warning messages produced by the assembler. Several assembler controls determine the format and content of the file. For example, the `SYMBOLS` control directs the assembler to include an optional list of symbols defined in the source program, called the symbol table. The assembler produces the listing by default unless you specify the `NOPRINT` control. 4.

The errorprint file is a summary of the errors and warnings encountered during assembly. It contains the source lines in which the errors occurred and the error messages. By default, this summary goes to the console output with the logical name `:CO:`. If the `ERRORPRINT` file has not been assigned to another file, it is directed to the screen. The assembler does not produce an errorprint file unless you specify the `ERRORPRINT` control. Figure 2-1 shows the assembler's logical files.

See also: `NOOBJECT`, `DEBUG`, `TYPE`, `SYMBOLS`, `NOPRINT`, and `ERRORPRINT` controls, Chapter 3
listing file, Chapter 4

The DOS manual indicates that the maximum number of characters in a pathname is 63, but in practice various products seem to restrict pathnames to less than 63 characters. To ensure compatibility with all products, make sure that all output pathnames do not exceed 43 characters. A fatal error is generated if your output pathname is too long, and the translator or R & L tool aborts.

In iRMX systems, you can use the **attachfile** command to assign a logical name to a long pathname. For example:

```
af /directory/subdirectory/subdirectory as f
```

You can then use the logical name as follows:

```
asm386 :f:file.asm
```

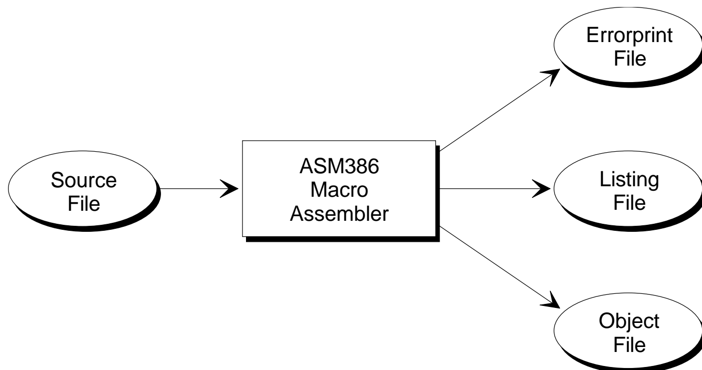


Figure 2-1. Macro Assembler Logical File

Work Files

The assembler creates temporary work files while processing the source and deletes them when the assembly is completed. These files are allocated to the `:WORK:` logical device and they do not conflict with any other files.

Under DOS, you may use the `SET` command to select the drive on which work files are placed. For example, this command places them in the root directory on drive `D:`

```
C>SET :work:=D:\
```

This capability is useful when a virtual disk in memory has been created with the DOS `VDISK.SYS` device driver.

Confusion may occur on user-defined logical names. The default assignments for :F0: through :F9: and :WORK: are not in the user manuals. If these logical devices are not defined with the SET command, the default assignments for :F0: through :F9: are to devices A through J. The default assignment for :WORK: is the current default disk. Use the SET command to assign the desired logical devices.

Under iRMX, *:work:* is defined by the operating system as the *:SD:work* directory. The user can establish other logical names (such as :F0:, :F1:, etc) using the **attachfile** command.

Files consisting of an 8-digit hexadecimal number with no extension may be left in the current :WORK: directory after you type CTRL-BREAK to abort an Intel translator or R & L tool or a user program converted to DOS with UDI2DOS.EXE. These are temporary files created by these programs to store intermediate data. They are normally deleted at the end of a program's normal execution. Delete or ignore the files. No important information is contained in them.

Messages

After invocation, the macro processor within the assembler scans the source file for macros and processes them. In the next phase, referred to as pass 1, the assembler constructs the symbol table. Lastly, during pass 2, the assembler completes the actual translation. Most assembler errors are detected during pass 1. If the assembler detects an error in pass 2, the error message contains "`(PASS 2)`".

Immediately after invocation, the assembler displays the following sign-on message:

```
system-id Intel386 MACRO ASSEMBLER Vx.y
Copyright year(s) Intel Corporation
```

Where:

system-id	is the string returned by the operating system.
x.y	is the version number.
year(s)	lists the copyright year(s).

If the assembler did not detect any fatal errors during assembly, it displays the following sign-off message:

```
ASSEMBLY COMPLETE, n WARNINGS m ERRORS
```

where *n* and *m* represent the number of warning and nonfatal error conditions, respectively, that occurred during processing.

When the assembler detects certain severe errors, it stops processing the source file and exits to the operating system without producing an object or listing. It produces an error message ending with "`ASM386 TERMINATED`".

See also: Assembler error messages, Appendix A

Automation of Program Invocation and Execution

DOS allows you to invoke and execute multiple programs either by using batch files or command files. The following sections provide examples that demonstrate how to use these files with the assembler and the binder, BND386.

See also: For details on BND386 and the other utilities, *Intel386 Family Utilities User's Guide*.

DOS Batch Files

A batch file contains one or more invocation commands or DOS batch commands that DOS executes one at a time. All batch files must have the extension BAT. This section explains how to create batch files that invoke several programs at once and that can operate on different sets of input files.

See also: DOS batch files, in your DOS manual

Note
DOS batch files cannot be nested. If a batch file references another batch file, control passes directly to the other batch file, but control does not return to the referring batch file.

Passing Parameters to Batch Files

It is possible to pass parameters to a DOS batch file when the file executes. The batch file can do similar work on a different program or set of data each time the batch file is executed. The following example illustrates this use of a batch file.

In the following example, the batch file ASM.BAT contains the command sequence that invokes the assembler and BND386 for two modules. Any assembler source file with the extension ASM can be passed as a parameter to ASM.BAT. Each percent sign and its accompanying digit in the batch file is replaced with the parameters specified on the command line that invokes the batch file. For instance:

```
ASM386 %1.ASM
ASM386 %2.ASM
BND386 %1.OBJ, %2.OBJ
```

Invoke the batch file by typing the name of the batch file without the BAT extension, followed by the names of the source files to be translated, without the ASM source file extension, as follows:

```
C>ASM PROG1 PROG2
```

ASM.BAT invokes the assembler to assemble PROG1.ASM and PROG2.ASM and passes the resulting files PROG1.OBJ and PROG2.OBJ to BND386. BND386 then links the two files and, by default, produces one executable file named PROG1.

Using Batch Commands

In addition to program invocation commands, batch files can contain DOS batch commands (or subcommands) such as FOR, IF, and GOTO. Such commands enable you to write a batch file that executes programs conditionally or repeatedly.

See also: Batch file commands, in your DOS manual

DOS Command Files

Under DOS version 2.0 or later, it is possible to invoke the DOS command-line interpreter program, COMMAND.COM, with input that is redirected from a file (called a command file). This file can contain DOS commands and invocation commands for programs such as the assembler. A command file must contain the DOS EXIT command.

For example, assume you created a command file named MAKEPROG.CMD that contained the following information:

```
ASM386 MAIN.ASM
ASM386 IO.ASM
PLM386 UTIL.PLM
BND386 MAIN.OBJ,IO.OBJ,UTIL.OBJ
EXIT
```

You can redirect the commands in this file to COMMAND.COM by entering the following:

```
C>COMMAND <MAKEPROG.CMD
```

COMMAND.COM then invokes all commands listed in the file MAKEPROG.CMD.

The following considerations apply when invoking COMMAND.COM with input that is redirected from a command file:

- Command files can only contain fixed sequences of commands; you cannot pass parameters to COMMAND.COM.
- Command files cannot support conditional DOS batch commands such as IF and GOTO; commands are always executed sequentially.

- Command files can be nested by reinvoking COMMAND.COM from the primary command file with input redirected from a secondary command file. The secondary command file must contain an EXIT command as its final line. When the EXIT command is executed, control returns to the point in the primary file immediately following the point from which the secondary file was invoked.
- Command files, unlike DOS batch files, can contain continuation lines. For example, the following is a valid command file:

  ```
  BND386 FILE1.OBJ,FILE2.OBJ,FILE3.OBJ,&
  NOTYPE OJ(PROG1.OBJ)
  EXIT
  ```

  The ampersand is the line continuation character.
- Output from a command file may be redirected to another file in order to obtain a complete log of all console output created during the command file's execution, including the invocation line for each program executed in the command file. For example, the following command invokes the command file MAKEPROG.CMD and creates a log file named MAKEPROG.LOG.

  ```
  C>COMMAND <MAKEPROG.CMD>MAKEPROG.LOG
  ```

Redirection of Command Input to Batch Files

DOS batch files can contain multiple invocation lines, but each invocation line must fit on a single line. No line continuation characters (such as the ampersand) are allowed within batch files. To process continuation lines in batch files, you must redirect the input from a file that contains continuation lines to a batch file. The following example shows how to do this.

In this example, two files are created: the batch file LINKBIG.BAT and LINKBIG.CON, which contains continuation lines. LINKBIG.CON is redirected to LINKBIG.BAT upon execution.

LINKBIG.BAT contains the line:

```
BND386 MODULE1.OBJ,MODULE2.OBJ <LINKBIG.CON
```

LINKBIG.CON contains the continuation line:

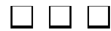
```
MODULE3.OBJ,MODULE4.OBJ FASTLOAD NODEBUG NOEP & NOPR NOTYPE
```

When LINKBIG.BAT is executed, BND386 is invoked, linking the four modules with the specified set of BND386 controls.

The sample file, LINKBIG.CON could also be redirected to a batch file that contains multiple invocation lines, as long as this batch file contains no continuation lines. For example, the batch file that follows, GENBIG.BAT, contains several invocation lines:

```
ASM386 MODULE1.ASM
ASM386 MODULE2.ASM
ASM386 MODULE3.ASM
ASM386 MODULE4.ASM
BND386 MODULE1.OBJ,MODULE2.OBJ,& <LINKBIG.CON
```

At execution, all of the modules in GENBIG.BAT are assembled and then linked with the set of BND386 controls specified in LINKBIG.CON.



Assembler Control Reference 3

This chapter contains a detailed description of each assembler control, listed in alphabetical order. Each description contains the syntax, default value, type (primary or general), abbreviation, and an explanation of its usage. Syntax descriptions include a command-line form and a source-file form. In many cases, the same syntax is used for both forms.

Certain controls override others, causing them to be ignored by the assembler. For example, if the `NOPRINT` control is in effect, the assembler ignores a `SYMBOLS` control because it cannot create a symbol table if it does not create a print file. Each control description explains the overrides for the control, if any.

See also: Command-line syntax, Chapter 2

DATE

Syntax

Command Line	`DATE(`*`date`*`)`
Source File	`DATE(`*`date`*`)`
Abbreviation	`DA`

Default

System time

Type

Primary

Discussion

The `DATE` control is supplied for compatibility with ASM86. The control is processed but the date parameter is ignored. The date that appears in the print file is obtained through a call to the operating system.

DEBUG

Syntax

Command Line	`DEBUG` `NODEBUG`
Source File	`DEBUG` `NODEBUG`
Abbreviation	`DB` `NODB`

Default

`NODEBUG`

Type

Primary

Discussion

`DEBUG` directs the assembler to include local symbol information for variables and labels in the object file for use in symbolic debugging. In addition, with the `DEBUG` control in effect, the assembler includes `LINES` and `SRCLINES` information for debugger support. `NODEBUG` directs the assembler to omit debugging information from the object file.

Depending on the contents of your source file, `DEBUG` can significantly increase the size of the object file produced.

The `NOOBJECT` control overrides the `DEBUG` and `NODEBUG` controls.

In ASM386 V4.0, the `DEBUG` primary control generates debug information in the object module necessary to support source-level display by debuggers.

If the source level debug support is not desired, the size of the loadable object file may be reduced by specifying the default `NODEBUG` control.

If symbolic debug information is desired without source level debug support, the `SRCLINES` and `LINES` debug information may be purged from a loadable object module using `MAP386`.

See also: `MAP386` and the `OBJECTCONTROLS` option, *Intel386 Family Utilities User's Guide*

EJECT

Syntax

Source File EJECT

Abbreviation EJ

Type

General

Discussion

EJECT directs the assembler to create a new page in the source listing, beginning with the next source line. Additional EJECT controls on a single control line are ignored. The EJECT control is not allowed on the command line.

The following is a list of interactions between EJECT and other assembler controls:

- If EJECT appears in a line suppressed by an earlier NOLIST control, a new page begins when the listing is started again by the appearance of LIST.
- If the NOPRINT control is in effect, EJECT is ignored.
- The NOPAGING control overrides any EJECT controls.

ERRORPRINT

Syntax

Command Line	`ERRORPRINT[(`*file-spec*`)]` `NOERRORPRINT`
Source File	`ERRORPRINT[(`*file-spec*`)]` `NOERRORPRINT`
Abbreviation	`EP` `NOEP`

Default

NOERRORPRINT

Type

Primary

Discussion

ERRORPRINT directs the assembler to create a file containing only error messages and their corresponding source lines or to print such a summary on the terminal screen. Each line and error message summary has the same form as in the full listing. For example:

```
system-id  Intel386
MACRO  ASSEMBLER  x.y  ASSEMBLY OF MODULE MYPROG
OBJECT MODULE PLACED IN MYPROG.OBJ
ASSEMBLER INVOKED BY: ASM386 ERRORPRINT MYPROG.ERR MYPROG.ASM
LOC   OBJ                LINE   SOURCE
-------                   31   DATA  ENDS
*** WARNING #354 IN 31, SEGMENT CONTENTS DO NOT AGREE WITH ACCESS-TYPE
```

If you do not supply a file specification, the assembler prints an error summary similar to the above on the terminal screen, followed by the standard assembler sign-off message.

See also: Error message formats, Appendix A

NOERRORPRINT directs the assembler not to create the error summary file.

ERRORPRINT and NOPRINT can be specified for the same assembly because the assembler can generate an errorprint file without generating a print file.

Place quotation marks around errorprint file names that are specified in the source file if they contain spaces or any of the following characters:

```
' , ( ) = # ! $ % \ ~ + - & | < > [ ] ;
```

For example:

```
$ERRORPRINT("ERROR(s).OUT")
```

Errorprint files that do not contain any of the characters above should appear as follows:

```
$ERRORPRINT(ERROR.OUT)
```

GEN/NOGEN/GENONLY

Syntax

Command Line	GEN NOGEN GENONLY
Source File	GEN NOGEN GENONLY
Abbreviation	GE NOGE GO

Default

GENONLY

Type

General

Discussion

The GEN, GENONLY, and NOGEN controls determine the mode of listing assembly source text, macro calls, and macro expansion text in the print file, as follows:

- GEN produces a listing that contains all source text, all macro calls, all macro expansions (i.e., the macro text), and object code.
- GENONLY produces a listing that contains source file nonmacro text and the final resulting text of all macros called, but omits the actual macro calls. Object code generated inside any macro calls is listed.
- NOGEN produces a listing that contains only the source file text (macro definitions and calls), not the macro expansions. Object code, if any, is listed after the line containing the macro call.

GEN produces the most complete and continuous source listing because it provides a trace of the entire macro call and expansion process. Expansions appear on the line below the call, indented to the same column as the call. (Horizontal tabs in macro calls or expansion lines are not expanded.)

This makes the GEN listing useful for debugging macros. However, GEN may produce an inconveniently large print file for programs that contain many macro calls.

Both the NOMACRO and NOPRINT controls override the GEN, GENONLY, and NOGEN controls. When any combination of the three controls -- GEN, NOGEN, and GENONLY -- appears on the same control line within the source file, the last setting takes effect.

Example

Only one of the GEN, GENONLY, or NOGEN controls can be in effect at one time in the source listing, although you can specify the controls at selected points to change the listing mode. The following example shows how the macro MAC is called in each of the three modes:

- In GEN mode, line 8, the listing includes the macro call and its expansion.
- In GENONLY mode, line 17, the call to MAC (%MAC(4,5,6)) is suppressed, but the resultant text is listed.
- In NOGEN mode, line 22, only the call to MAC is listed. The expansion lines are skipped.

```
LOC       OBJ     LINE              SOURCE
                    1       NAME XXX
                    2       $NOGEN
                    3       %*DEFINE(MAC(A,B,C) ) (DW %A
                    4       DW %B
                    5       DW %C
                    6       )
                    7       DATA SEGMENT RW
--------            8   +1  $GEN
                    9       %MAC (1,2,3)
00000000 0100      10   +1  DW %A
                   11   +2     1
00000002 0200      12   +1  DW %B
                   13   +2     2
00000004 0300      14   +1  DW %C
                   15   +2     3
                   16   +1
                   17   +1  $GENONLY
00000006  0400     18   +2  DW 4
00000008  0500     19   +2  DW 5
0000000A  0600     20   +2  DW 6
                   21   +1
                   22      $NOGEN
                   23      %MAC (7,8,9)
0000000C  0700
0000000E  0800
00000010  0900
--------           24      DATA ENDS
                   25      END
```

Figure 3-1. Sample Listing for GEN/NOGEN/GENONLY

INCLUDE

Syntax

Command Line `INCLUDE(`*`file-spec`*`)`

Source File `INCLUDE(`*`file-spec`*`)`

Abbreviation `IC`

Type

General

Discussion

`INCLUDE` directs the assembler to insert the contents of a file into the source file. If `INCLUDE` appears in the invocation line, the contents of the include file are inserted before the contents of the main source file. If `INCLUDE` appears in a control line, input from the include file begins following the control line and continues until the end of the include file is reached. At that time, input resumes from the file that was being processed when the `INCLUDE` control was encountered.

`INCLUDE` need not be the last command in a control line; however, it does not take effect until the end of the control line is reached. The following restrictions govern the use of `INCLUDE`:

- Only one `INCLUDE` control is allowed per line.
- No more than 64 combinations of macro calls and `INCLUDE` controls can be in effect at the same time in any one module.
- The maximum nesting level for included files is nine.
- An included file can contain lines specifying primary controls.

Included files specified in the source file must be enclosed within quotation marks if they contain spaces or any of the following characters:

```
' , ( ) = # ! $ % \ ~ + - & | < > [ ]
```

For example:

```
$INCLUDE("Clude(s).INC")
$INCLUDE("Lot#4.Inc")
```

LIST

Syntax

Command Line	LIST NOLIST
Source File	LIST NOLIST
Abbreviation	LI NOLI

Default

LIST

Type

General

Discussion

LIST directs the assembler to resume the source listing in the print file with the next source line read. NOLIST suppresses the listing, beginning with the next line read, until the next occurrence (if any) of LIST. Specifying NOLIST at invocation suppresses the listing, beginning with the first line.

The following is a list of interactions between LIST/NOLIST and other assembler controls:

- NOPRINT, XREF, and SYMBOLS override the LIST/NOLIST controls.
- PRINT does not override NOLIST. If both NOLIST and PRINT are in effect, only the assembler messages for source lines containing errors appear in the print file. Otherwise, the file contains only a header (assuming SYMBOLS or XREF is not in effect).
- If an EJECT control appears in a line suppressed by NOLIST, a new page begins when the listing is started again by LIST.
- NOLIST affects the setting of the PAGELENGTH control only if NOLIST is still in effect when the end of the source listing is reached. If NOLIST suspends the listing in the middle of a page and a subsequent LIST begins the listing again, source lines are added to the page until it reaches the specified length.

MACRO

Syntax

Command Line	MACRO NOMACRO control
Source File	MACRO[(*parameter*)] NOMACRO
Abbreviation	MR NOMR

Default

MACRO

Type

Primary

Discussion

The MACRO control directs the assembler to recognize and process macros.

Macros can appear anywhere in the source file, including control lines. Refer to Chapter 2 for an example of a macro call within a control line. In effect, any occurrence of the macro metacharacter (% by default) in the source is considered a macro call. The parameter has no effect for the assembler but is allowed for compatibility with existing ASM286 files.

Macros can also appear in the invocation line. Use the *%macro-string* statement.

See also: The *%macro-string* statement, Chapter 2
macro processing language, *ASM386 Assembly Language Reference*

NOMACRO directs the assembler to scan macros only as normal assembly language text, which usually causes assembler errors. It may speed up the assembly if no macros are used and the NOMACRO control is in effect.

NOMACRO overrides the GEN/GENONLY/NOGEN controls.

MOD386/MOD376/MOD486

Syntax

Command Line	`MOD386`
	`MOD376`
	`MOD486`

Source File	`MOD386`
	`MOD376`
	`MOD486`

Default

`MOD386`

Type

Primary

Discussion

The `MOD386`, `MOD376`, and `MOD486` controls direct the assembler to ensure that the input file meets the requirements of the Intel386, 376, and Intel486 processors, respectively. By default, the assembler accepts assembly language source code that executes on the Intel386 processor.

The Intel486 or Pentium processor architecture is fully compatible with the Intel386 processor architecture. All Intel386 processor modes are available to the Intel486 or Pentium processor. The Intel486 processor also has an expanded instruction set and additional registers which are supported by the assembler with the `MOD486` control specified.

The `MOD486` primary control provides the following support for Intel486 microprocessor software development using ASM386 V4.0:

- Enables the test registers TR3, TR4, and TR5 which are defined on the Intel486 microprocessor.
- Enables the forms of the `MOV` instruction to load and store the TR3, TR4, and TR5 registers.
- Generates machine code for all forms of the Intel486 microprocessor instructions.

See also: Differences between Intel386 and Intel486 processors, *ASM386 Assembly Language Reference*

The 376 processor architecture is a subset of the Intel386 processor architecture: the 32-bit protected mode is available, but real address mode, virtual 8086 mode, and paging are not available. The segmentation-based memory management and protection features are available on the 376 processor. 286 processor call, interrupt, or trap gates or 286 processor TSSs are not supported on the 376 processor.

See also: Differences between the 376 and Intel386 processors, *ASM386 Assembly Language Reference*

When assembling for the 376 processor, make sure the input file contains only `USE32` code or stack segments. `USE16` code segments are not executable on the 376 processor. The assembler issues an error message if the `USE16` segment directive is in effect for a code or a stack segment; `USE16` data segments may be included in the input file. An error is also issued if the `USE16` keyword is used in an `EXTRN` directive of type `NEAR` or `FAR`.

Because the 376 processor has a 24-bit address bus, a segment must be no larger than 16 megabytes. The assembler issues a warning when you specify `MOD376` and the source file contains a segment exceeding 16 megabytes.

See also: Errors and warnings, Appendix A

N387/N287

Syntax

Command Line	`N387` `N287`
Source File	`N387` `N287`

Default

`N387`

Type

Primary

Discussion

By default, the assembler generates code for Intel387 floating-point coprocessor instructions. The Intel387 floating-point coprocessor includes all the instructions for the Intel287 plus `FSINCOS`, `FSIN`, `FCOS`, `FUCOMPP`, `FUCOM`, `FUCOMP`, and `FPREM1`.

`N287` directs the assembler to detect the instructions not supported on the Intel287 and to issue an error message for each line that contains an instruction unique to the Intel387 floating-point coprocessor.

See also: Instructions for the Intel387 floating-point coprocessor, *ASM386 Assembly Language Reference*

OBJECT

Syntax

Command Line	`OBJECT[(`*`file-spec`*`)]` `NOOBJECT`
Source File	`OBJECT[(`*`file-spec`*`)]` `NOOBJECT`
Abbreviation	`OJ` `NOOJ`

Default

`NOOBJECT`

Type

Primary

Discussion

`OBJECT` directs the assembler to create an object file during assembly of the specified source file. If severe errors are found, the object file is not produced.

See also: Errors that affect the creation of an object file, Appendix A

If you do not specify `NOOBJECT` or you specify `OBJECT` without the object-file parameter, the assembler creates an object file with the same file name as the source file and the extension `OBJ`.

`NOOBJECT` directs the assembler not to create an object file.

`NOOBJECT` overrides the `DEBUG/NODEBUG` and `TYPE/NOTYPE` controls.

Object file names specified in the source file must be enclosed within quotation marks if they contain spaces or any of the following characters:

```
' , ( ) = # ! $ % \ ~ + - & [ < > [ ] ;
```

For example:

```
$OBJECT("TOP(s).OBJ")
```

PAGELENGTH

Syntax

Command Line	`PAGELENGTH(`*`length`*`)`
Source File	`PAGELENGTH(`*`length`*`)`
Abbreviation	`PL`

Default

`PAGELENGTH(60)`

Type

Primary

Discussion

`PAGELENGTH` directs the assembler to create print file pages of a specified length. The value of length may be an unsigned decimal integer from 10 to 65535 representing the number of lines per page of the print file. The total number of lines per page includes any header lines on the page. The minimum page length is 10 lines.

`PAGELENGTH` is ignored if the `NOPRINT` or `NOPAGING` control is in effect.

`NOLIST` affects the setting of `PAGELENGTH` only if `NOLIST` is in effect when the end of the source listing is reached. If `NOLIST` suspends the listing in the middle of a page and a subsequent `LIST` begins the listing again, source lines are added to that page until it reaches the specified length.

PAGEWIDTH

Syntax

Command Line	PAGEWIDTH(*width*)
Source File	PAGEWIDTH(*width*)
Abbreviation	PW

Default

PAGEWIDTH(120)

Type

Primary

Discussion

PAGEWIDTH directs the assembler to create print and errorprint file pages of a specified width. The value of *width* may be an unsigned decimal integer that specifies the number of characters on a line of the print and errorprint files.

The minimum page width is 60 characters; the maximum is 132.

The NOPRINT control overrides PAGEWIDTH.

PAGING

Syntax

Command Line PAGING
NOPAGING

Source File PAGING
NOPAGING

Abbreviation PI
NOPI

Default

PAGING

Type

Primary

Discussion

PAGING directs the assembler to format the print file into pages, as follows:

- Every page is initiated with a form feed character.
- Each page begins with a header containing the assembler name, title, date, and page number.
- The symbol table listing, if present, begins on a new page, following the source listing.

The length of the page depends on the setting of the PAGELENGTH control.

NOPAGING prevents the print file from being paged. Instead, a single header is printed at the beginning of the file and the listing is continuous until the symbol table (if any), which is separated from the source listing by four blank lines.

The following is a list of interactions between PAGING/NOPAGING and other assembler controls:

- NOPRINT overrides PAGING and NOPAGING.
- NOPAGING overrides PAGELENGTH.
- NOPAGING overrides EJECT.

PRINT

Syntax

Command Line `PRINT[(file-spec)]`
`NOPRINT`

Source File `PRINT[(file-spec)]`
`NOPRINT`

Abbreviation `PR`
`NOPR`

Default

`PRINT` *`source-file`*`.LST`

Type

Primary

Discussion

`PRINT` directs the assembler to create a source listing during assembly and write it to the listing file or to the screen. If you do not specify `NOPRINT` or you specify `PRINT` without the file-spec parameter, the source listing appears in a file with the same file specification as the source file with the `LST` extension.

`NOPRINT` directs the assembler not to create a source listing.

`NOPRINT` overrides all controls affecting the print file (`EJECT`, `SYMBOLS`, etc.), but does not affect controls related to the object file (`DEBUG`, `TYPE`, etc.).

If `NOLIST` is used while `PRINT` is in effect, the listing contains only the header, error messages, and those source lines containing errors. Correct source lines do not appear unless the listing is begun again by the `LIST` control.

Print files specified in the source file must be enclosed within quotation characters if they contain spaces or any of the following characters:

```
' , ( ) = # ! $ % \ ~ + - & | < > [ ] ;
```

For example:

```
$PRINT("New(s).LST")
```

SAVE/RESTORE

Syntax

Source File SAVE
RESTORE

Abbreviation SA
RS

Type

General

Discussion

SAVE directs the assembler to save the current settings of the LIST/NOLIST and GEN/GENONLY/NOGEN controls on a stack. The current setting is the setting in effect at the beginning of the SAVE control line. RESTORE specifies that the most recently saved settings on the stack become the current settings of those controls. SAVE and RESTORE are not allowed on the command line. The maximum nesting level of SAVEs is eight.

The SAVE and RESTORE controls do not function correctly when used under the following conditions:

- Fewer than two lines exist between the SAVE followed by RESTORE.
- The control lines containing SAVE/RESTORE are in an include file.
- SAVE or RESTORE is combined with either the GEN or NOGEN control.

Example

SAVE and RESTORE can be used to regulate the listing of macros. For example, you may want a listing that contains both macro calls and their results. This listing would be comparable to a combination of the results of the NOGEN and GEN controls. The call line is listed in NOGEN mode; both the call line and the expansion are listed in GEN mode. The following example shows the use of SAVE/RESTORE to regulate the listing of a macro MAC for two calls.

```
LOC       OBJ      LINE      SOURCE
                     1       NAME SAVE_TEST
                     2
                     3  +1   $GEN
                     4  +1   $SAVE  ;SAVES GEN'S SETTING ON STACK
                     5       %*DEFINE(MAC(A, B) )(
                     6       MOV AX, %A
                     7       MOV BX, %B
                     8       )
--------             9       DATA SEGMENT RW
00000000  0400      10       D1  DW 4
--------            11       DATA  ENDS
--------            12       CODE  SEGMENT  EO
                    13       ASSUME DS:DATA
                    14       $NOGEN
                    15       ;NOGEN SETTING IS IN EFFECT
                              FOR NEXT MACRO CALL
                    16       %MAC (40H, 50H)
00000000  66B84000
00000004  66B85000
                    17       $RESTORE  ;GEN SETTING FROM STACK
                                        IS IN EFFECT FOR NEXT CALL
                    18       %MAC (70H, 80H)
                    19
00000008  66B8700C
                    20       MOV  AX, %A
                    21       70H
0000000C  66BB8000
                    22       MOV BX, %8
                    23       80H
                    24
--------
                    25       CODE ENDS
                    26       END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Figure 3-2. Sample Listing for SAVE/RESTORE

SYMBOLS

Syntax

Command Line	SYMBOLS NOSYMBOLS
Source File	SYMBOLS NOSYMBOLS
Abbreviation	SB NOSB

Default

NOSYMBOLS

Type

Primary

Discussion

SYMBOLS directs the assembler to append a symbol table to the source listing in the print file. The symbol table is an alphabetical list of all assembler identifiers defined within the source, with their attributes. Assembler identifiers do not include macro identifiers.

See also: Symbol table, Chapter 4

NOSYMBOLS directs the assembler to suppress the symbol table.

NOPRINT overrides SYMBOLS; XREF overrides NOSYMBOLS.

TITLE

Syntax

Command Line	`TITLE(`*`title`*`)`
Source File	`TITLE(`*`title`*`)`
Abbreviation	`TT`

Default

`TITLE(`*`module-name`*`)`

Type

General (source file)
Primary (command line)

Discussion

`TITLE` directs the assembler to place a character string in the header on the first line of each page of the print file.

In the command line, `TITLE` functions as a primary control and sets the title for each page of the file. When specified in a control line, `TITLE` functions as a general control. The specified title appears on the page where the `TITLE` control line occurs and on all subsequent pages until changed by another `TITLE` control.

`TITLE` does not cause a new page to start. The `EJECT` control or normal paging determine the start of a new page.

The maximum title length is 60 printable ASCII characters. If the title does not fit within the specified page width, the assembler truncates the title on the right. No error message is generated for titles up to 80 characters. Titles over 80 characters long generate incorrect error messages:

```
ERROR #520:  BAD CONTROL PARAMETER
ERROR #612:  EXPECTED A RIGHT PARENTHESIS
```

Titles specified in control lines must be enclosed within quotation marks if they contain spaces or any of the following special characters:

```
' , ( ) = # ! $ % \ ~ + - & | < > [ ]
```

For example:

```
$TITLE("Section 2")
```

If the title itself contains a quotation character or apostrophe, enclose the title in the other type of quote mark. For example:

```
$TITLE('Nancy"s')
$TITLE("Nancy's")
```

In the absence of any `TITLE` controls, the assembler uses the module name specified with the `NAME` directive as the title.

`NOPRINT` overrides `TITLE`. The `NOPAGING` control overrides `TITLE` controls that appear after the primary control lines, because no new pages are created.

Use of the `TITLE` control on the command line overrides use of the `TITLE` control on the primary control line in the source file.

TYPE

Syntax

Command Line	`TYPE` `NOTYPE`
Source File	`TYPE` `NOTYPE`
Abbreviation	`TY` `NOTY`

Default

`NOTYPE`

Type

Primary

Discussion

`TYPE` directs the assembler to include type information about the `PUBLIC` variables and labels in the symbol records of the object module. `NOTYPE` directs the assembler to omit type information from the object module. The `NOOBJECT` control overrides `TYPE`.

Type information for variables declared `PUBLIC` is used primarily to assist debuggers in displaying symbols. Although the type produced for a variable may not exactly correspond to the intended contents of that memory location, the type is sufficient to specify the number of bytes to be displayed for the variable. Also, in the case of an array or structure, the format and size associated with the symbol are included.

The type information produced by the assembler can also be used for inter-module type checking by BND386, the binder for processor modules. BND386 compares the use of variables in different modules to ensure that every use of a variable is consistent with its type. However, the utility of `TYPE` is limited because the assembler does not produce type information for external symbols, nor does it support perfect type matching with high-level languages.

See also: Assembler data types, *ASM386 Assembly Language Reference*

USE32/USE16

Syntax

Command Line	`USE32` `USE16`
Source File	`USE32` `USE16`
Abbreviation	`U32` `U16`

Default

`USE32`

Type

Primary

Discussion

`USE16` directs the assembler to generate 16-bit addresses and offsets, as well as the appropriate operand sizes and address mode override prefixes. In this mode, which is compatible with ASM286, the assembler supports a maximum segment size of 64K. If `USE16` is in effect, the assembler can translate ASM286 assembly language modules without source modification.

ASM386 performs 32-bit arithmetic even when you specify `USE16`. ASM386 does not support register expressions that use scale with `USE16`.

`USE32`, the default, directs the assembler to generate 32-bit addresses and offsets, as well as the appropriate operand sizes and address mode override prefixes. In this context, the assembler supports a maximum segment size of 4 gigabytes.

The assembly language includes the `USE16` and `USE32` segment `USE` attributes that perform the same functions as the `USE16` and `USE32` controls, respectively. `USE32` is the default.

The following rules also apply:

- Only one member of the control pair USE32 and USE16 can be specified in the invocation line, that is, if USE16 is specified USE32 cannot be specified.
- If both USE16 and USE32 are specified in source control lines, the last one specified is in effect.

See also: Segment USE attributes, *ASM386 Assembly Language Reference*

WORKFILES

Syntax

Command Line `WORKFILES(`*`dir1`*`[,`*`dir2`*`])`

Source File `WORKFILES(`*`dir1`*`[,`*`dir2`*`])`

Abbreviation `WF`

Default

`WORKFILES(:WORK:,:WORK:)`

Type

Primary

Discussion

`WORKFILES` specifies logical names for devices or directories for storage of assembler-created temporary files. These intermediate files are deleted at the end of assembly. A single name may be specified as the parameter; this is equivalent to specifying that name twice.

This is provided for compatibility with earlier assemblers.

See also: Work Files, Chapter 2

XREF

Syntax

Command Line	`XREF` `NOXREF`
Source File	`XREF` `NOXREF`
Abbreviation	`XR` `NOXR`

Default

`NOXREF`

Type

Primary

Discussion

`XREF` directs the assembler to append a symbol table listing, including cross-reference line numbers, to the source listing in the print file. This table has the same format as the table produced by the `SYMBOLS` control, with an additional field entitled `XREFS`. The `XREFS` field contains the numbers of the lines in which a symbol is defined, referenced, or purged.

See also: Symbol table, Chapter 4

`NOXREF` directs the assembler to omit the cross-referenced field from the print file.

`XREF` overrides the `SYMBOLS` and `NOSYMBOLS` controls. `NOPRINT` overrides `XREF`.

The Listing (Print File) 4

The listing, sometimes referred to as the print file, provides information on the assembly of a module, such as a listing of the source code and object code, and any errors or warnings produced by the assembler. This chapter describes the fields of information in the print file and the file's optional symbol table listing.

Figure 4-1 is a sample listing for an assembler module named MYPROG, which contains errors to illustrate the assembler error reporting. The four main fields of information in the print file are LOC (location counter), OBJ (object code), LINE (line number), and SOURCE (source text). Other kinds of information may appear in a print file, depending on the nature of the source program. In general, information generated by the assembler appears to the left of the line number and source code appears to the right of the line number.

The Default Print File

If you do not specify any assembler controls that govern the format of the print file, the file has the following characteristics:

- The file specification is the source file's name with LST extension.
- The file is divided into pages 60 lines long and 120 characters wide. The first line of each page contains the assembler name, the title (the module name specified with the NAME directive), the time and date, and the page number.

For macros, the file contains the source file's text and the final resulting text of all macros, but not the actual macro calls. All object code generated inside macro calls is listed.

```
Intel386 MACRO ASSEMBLER MYPROG     time     mm/dd/yy    PAGE 1

system-id Intel386 MACRO ASSEMBLER Vx.y ASSEMBLY OF MODULE MYPROG
OBJECT MODULE PLACED IN MYPROG.OBJ
ASSEMBLER INVOKED BY: ASM386 SYMBOLS PAGEWIDTH 73 MYPROG.ASM

LOC    OBJ                     LINE      SOURCE
                                  1   NAME    MYPROG
                                  2
REG                               3   COUNT   EQU CX
-0800                             4   IVAL    EQU -800H
0100                              5   AR_SIZE EQU 100H
   #                              6   R17    RECORD SIGN:1, LOW7:7
                                  7
                                  8   EXTRN   SYSTEM:FAR
                                  9   PUBLIC  INIT
-------                          10   FLOAT   STRUC
0000000                          11   EXPONENT DB 0
0000001                          12   MANTISSA DD 0
-------                          13   FLOAT   ENDS
                                 14
C MACRO                          15   CODEMACRO D7 VALUE:D
   #                             16   R17   <0, VALUE>
   #                             17   ENDM
                                 18
-------                          19   STSEG  STACKSEG 100
                                 20
-------                          21   DATA   SEGMENT RW USE32
00000000 03                      22   INITIAL FLOAT <3,5>
00000001 05000000
00000005 03                      23   TOP    DB 3, 10
00000006 0A
                                 24   WOMBAT
***---------------------------------------^
*** ERROR #1 IN 24, SYNTAX ERROR
```

Figure 4-1. Sample Print File Page

```
00000007 414243              25   STRNG  DB  'ABC'
0000000A (10                 26          DW 10 DUP (1,3,44H)
         0100
         0300
         4400
         )
00000046 05000000          R 27   ITOP   DW TOP
0000004A 46000000          R 28   IITOP  DD ITOP
0000004E 07                  29          D7 07H
0000004F ----              R 30   ES_SEL DW EXTRA
--------                     31   DATA   ENDS
                             32
--------                     33   EXTRA  SEGMENT RW USE32
AAAAAAAA                     34          ORG 0AAAAAAAAH
AAAAAAAA (256                35   ARRAY  DD AR_SIZE DUP (?)
         ????????
         )
--------                     36   EXTRA  ENDS
                             37
AAAAAAB4:[]                  38   AR1BX  EQU ES:ARRAY1
         [EBX+10]
                             39
--------                     40   CODE   SEGMENT ER
                             41          ASSUME DS:DATA
                             42
00000000 ----              R 43   DS_SEL DW DATA
                             44
00000002                     45   INIT   PROC FAR
00000002 66B9F600            46          MOV COUNT,AR_SIZE - 10
```

Figure 4-1. Sample Print File Page (continued)

```
00000006 6689CB               47        MOV BX, COUNT
00000009                      48   INITLOOP:
00000009 26C783B4AAAAAAO00F8FF
                            R 49        MOV AR1BX, IVAL FF
00000014 E2F3                 50         LOOP INITLOOP
00000016 CB                   51         RET
00000017                      52   INIT   ENDP
                              53
00000017 2E8E1D00000000    R  54   START: MOV DS, DS_SEL
0000001E 8E054F000000      R  55          MOV ES, ES_SEL
00000024 9A02000000----    R  56          CALL INIT
0000002B 9A00000000----    E  57          CALL SYSTEM
--------                      58   CODE   ENDS
                              59
    ----                      60   CODE_16 SEGMENT EO
                                   USE16
   0000 B8----             R  61   MOV AX, EXTRA
   0003 8ECO                  62   MOV ES, AX
   0005 666726C7843BAAAAAAAA
                           R  63   MOV ES:ARRAY1[EBX][EDI],
        0FFFFFFFFH  FFFFFFFF
    ----                      64   CODE_16 ENDS
                              65
                              66 +1 $INCLUDE
                                 (WOMBAT.INC)
                          =1  67  WOMBAT
*** ------------------------------------^
*** ERROR #1 IN (WOMBAT.INC, LINE 67), SYNTAX ERROR
                              68  END START, DS:DATA, SS:STSEG
                              69
```

Figure 4-1. Sample Print File Page (continued)

Print File Headers

The first line of each page of the print file contains the assembler name, the title (either the module name or the name you specified with the `TITLE` control), the time and date determined by the operating system, and the page number. The first page contains an additional header in the following form:

```
system-id Intel386 MACRO ASSEMBLER Vx.y
ASSEMBLY  OF  MODULE  module-name
OBJECT  MODULE  PLACED  IN  object-file
ASSEMBLER  INVOKED BY:  ASM386 [controls] source-file
```

If no object file is requested or if errors prevent an object module from being created, the second line of the header contains a message `NO OBJECT FILE REQUESTED` or `NO OBJECT MODULE CREATED`. The last header line lists the controls used in the assembler invocation.

See also: Command Syntax, Chapter 2

Location Counter (LOC)

The program location counters track the current offset within the segment being assembled. The LOC field contains the location counter. For code in USE32 segments, the location counter is an eight-digit hexadecimal number. For code in USE16 segments, the upper four digits are blank and the location counter appears in the last four columns.

For source lines that generate object code and for labels (LABEL or PROC), the LOC field contains the location counter value effective at the beginning of the line.

For source lines containing the ORG directive, the LOC field contains the new value specified by the ORG statement.

The LOC field is blank for lines containing comments, directives, controls, macro definitions, or record definitions. If the object code for a source statement appears on more than one line, all other fields of the continued lines are blank, including the location counter.

For record definitions, a pound sign (#) appears in the rightmost column of the LOC field to signal that assembly is not taking place. For example:

```
    #                  6  R17    RECORDSIGN:1, LOW7:7
```

When a STRUC, SEGMENT, STACKSEG, or ENDS line has been coded, the LOC field contains the notation --------. For a USE16 SEGMENT, the notation is ----. The notation signals a break in the flow of the location counter. For example:

```
--------              21  DATA  SEGMENT RW  USE32
```

Equated Symbols (EQU Directive)

Equated symbols are on the left-hand side of a statement containing the EQU directive. Information about equated symbols appears in the last half of the LOC field and the first half of the OBJ field, starting in column three.

If the symbol is equated to a variable or label, this area contains the hexadecimal offset of the symbol. Variable or label equates can have segment override and indexing attributes. A colon (:) after the offset indicates an override attribute; brackets ([ ]) indicate an indexing attribute, as in the following example:

```
AAAAAAB4:[]     38  AR1BX   EQU ES:ARRAY1 [EBX+10]
```

If the symbol is equated to a number, this area contains the hexadecimal value of the number. If the symbol is equated to one of the following, the item's identifier appears in this area:

Item	**Identifier**
register	REG
macro	MACRO
codemacro	C MACRO
segment	SEGMENT
external variable	EXTRN
record	RECORD
record field	RFIELD
structure	STRUC
structure field	SFIELD
instruction	INSTRUCTION
keyword	KEYWORD

For example, the `LOC` field contents for a symbol equated to a register are shown in this line:

```
REG                     3   COUNT  EQU  CX
```

Floating-point Stack Elements (ST)

A floating-point stack element is indicated by `ST(`*`i`*`)`, where *`i`* is the numerical index value beginning in column 3 if the element is indexed, or by `ST` if the index is 0.

COMM Variables and Labels

The word `COMM` appears in columns 3 through 6 for each line of a data definition given the `COMM` attribute with the `COMM` statement.

Object Code (OBJ)

The object code is displayed as hexadecimal starting in column 10 and is filled as follows:

- The maximum size of an instruction is 15 bytes, even if all five prefix bytes are present.
- The field contains the notation ---- if segment selector values were assembled.

If an assembler statement spans several lines, object code produced for completed constructs on a continued line prints with the continued line. The assembler does not wait until a statement is completed to display all the object code.

Whenever a `DUP` field begins, a left parenthesis appears in the left column of the `OBJ` field, followed by the count in decimal numbers. The content bytes are left justified on the lines that follow, ending with a right parenthesis in the leftmost column. For example:

```
0000000A (10        25        DW  10  DUP  (1,3,44H)
         0100
         0300
         4400
         )
```

For nested `DUP`s, the left parenthesis, number, and the right parenthesis are indented one column for each nesting level, but the content bytes are never indented.

Relocatable or External Code (R, E)

The object code can be followed by a relocation indicator, which is the letter `R` if relocatable object code is generated on the current line, or the letter `E` if external code is generated. The `E` appears on lines containing code that is both external and relocatable. For example:

```
00000019 9A02000000----  R  55         CALL  INIT
00000020 9A00000000----  E  56         CALL  SYSTEM
```

Include Nesting Indicator (=)

An equal sign (=) followed by a number from 1 to 9 appears between the object code and the line number for all source lines that come from include files. The number indicates the level of nesting. An asterisk (*) appears if the include nesting level exceeds 9.

```
    66    $INCLUDE  (WOMBAT.INC)
=1  67   WOMBAT
```

Line Numbers (LINE)

The LINE field is five characters long. The line numbers begin with 1 and are incremented for every source or macro expansion line listed.

Macro Expansion Indicator (+)

The first column following the line number field of a macro expansion line contains a plus (+). The next two columns contain a number that indicates the nesting level of the macro, except for expansions, in which case these two columns are blank.

```
=1 85      %INC1(EXAMPLE,SIMPLE)
=1 86 +1
=1 87 +1   ;THIS %NOUN
=1 88 +2      EXAMPLE IS %ADJ
=1 89 +2              SIMPLE
```

Source Statements (SOURCE)

The source text is a copy of the source line of macro-generated text (as determined by the setting of the GEN/NOGEN/GENONLY controls).

Tabs in your source are reproduced so that the source text looks the same in the listing. If the GEN control is in effect, tabs are not expanded in lines containing macro calls (or parts of calls) or in macro expansion lines; instead, tabs appear as single spaces.

If a source statement exceeds the specified page width, it continues on the next line and the continued lines contain only source text, as shown:

```
00000009 6626C783B4AAAAA00F850     INITLOOP:MOV ES:ARRAY1
                              [EBX+10],IVAL
```

An error or warning message appears immediately after an erroneous line. The message contains an error or warning number, a listing line number, a pass number (if other than the first pass), and a brief description.

See also: Interpreting and correcting errors, Appendix A

The Symbol Table

The DOS symbol table capacity is approximately 4500 seven-character symbols when expanded memory and at least 568K conventional memory are available.

If the `SYMBOLS` or `XREF` control is in effect, the symbol table follows the source listing. `SYMBOLS` generates the standard table; `XREF` generates the same table with the addition of the numbers of each line in which a particular symbol was referenced. The example in Figure 4-2 was generated with `SYMBOLS`.

The symbols are in alphabetical order using the ASCII character order, except for the underscore (_), which comes first. Reserved names are not included unless they have been redefined or purged.

If the `PAGING` control is in effect, the symbol table begins on a new page; otherwise, it is separated from the source listing by four blank lines. The final message of the print file, which signals the end of assembly and shows the number of warnings and errors, appears after the symbol table.

See also: Symbol table fields and examples, Chapter 4

```
Intel386 MACRO ASSEMBLER MYPROG time   mm/dd/yy  PAGE 1

SYMBOL TABLE LISTING
- - - - - - - - - - -
NAME        TYPE      VALUE       ATTRIBUTES
AR1BX......V DWORD   AAAAAAB4H   ES:[EBX]
ARRAY1.....V DWORD   AAAAAAAAH   (256) EXTRA ES:
AR_SIZE....NUMBER    0100H
CODE.......SEGMENT               SIZE=00000032H ER USE32
CODE_16....SEGMENT               SIZE-00000013H E0 USE32
COUNT......REG       CX
D7.........C MACRO               DEFS=1
DATA.......SEGMENT               SIZE=00000051H RW USE32
DS_SEL ....V WORD    00000000H   CODE
ES_SEL ....V WORD    0000004FH   DATA
EXPONENT...V BYTE    00000000H   SFIELD
EXTRA......SEGMENT               SIZE=AAAAAEAAH RW USE32
FLOAT......STRUC                 SIZE=00000005H #FIELDS=2
IITOP......V DWORD   0000004AH   DATA
INIT.......P FAR     00000002H   CODE WC=0 PUBLIC
INITIAL....V  STRUC  00000000H   DATA
INITLOOP...L NEAR    00000009H
ITOP.......V DWORD   00000046H   DATA
IVAL.......NUMBER    FFFFF800H
LOW7.......R FIELD   00000000H   WIDTH=7
MANTISSA...V DWORD   00000000H   SFIELD
R17........RECORD                SIZE=1 WIDTH=8 DEFAULT=00H
SIGN.......RFIELD    00000007H   WIDTH=1
START......L NEAR    00000017H
STRNG......V BYTE    00000007H   (3) DATA
STSEG......STACK                 SIZE=00000064H RW PUBLIC USE32
SYSTEM.....L FAR     00000000H   EXTRN
TOP........V BYTE    00000005H   (2) DATA
VALUE.....------                 --UNDEFINED--
WOMBAT.....------                --UNDEFINED--

END OF SYMBOL TABLE LISTING
ASSEMBLY COMPLETE, NO WARNINGS, 2 ERRORS
```

Figure 4-2. Sample Symbol Table

Symbol Table Fields

The fields of the symbol table are NAME, TYPE, VALUE, ATTRIBUTES. The XREF field occurs when the table is generated by the XREF control.

In the NAME field, the name of the symbol appears as it was entered. The width of the field depends on the size of the longest name in the table, up to a maximum of 31 unique characters. Spaces and periods are added to fill out the field for short names.

The TYPE field appears after the NAME field. The possible types are described in the following sections.

The VALUE field contains the symbol's value, which is eight hexadecimal digits long for symbols within USE32 segments or four digits for symbols within USE16 segments. Not every type of symbol has a value displayed. Except for floating-point stack elements and register names, all displayed values are in hexadecimal.

The ATTRIBUTES field contains other pertinent information about the symbol, depending on its type. For example, the ATTRIBUTES field for a codemacro contains the number of its definitions.

The last part of the ATTRIBUTES field contains cross-reference information if the XREF control is in effect. The field contains one line number for each appearance of the symbol in the program. A pound sign (#) follows the number if the line contains a definition of the symbol. A P follows the number if the symbol was purged on that line. If the ATTRIBUTES/XREF field overflows a line, the field continues on subsequent lines.

The assembler lists as many cross-references as available memory allows. If memory is exhausted while the assembler is sorting cross-references, an error message appears at the beginning of the symbol table.

Code macros (C MACRO)

C MACRO in the TYPE field indicates a codemacro. The VALUE field is blank. The ATTRIBUTES field contains the notation DEFS=*n*, where *n* is the number of the codemacro's definitions.

Public and External Symbols (PUBLIC, EXTRN)

Public symbols have the PUBLIC attribute after all other attributes.

The TYPE field for external symbols contains the type that appears in the EXTRN statement. The VALUE field always contains 00000000H and the ATTRIBUTES field contains EXTRN.

Floating-point Stack Elements (F STACK)

`F STACK` in the `TYPE` field indicates a floating-point stack element. The `VALUE` field contains `ST(`*i*`)` if the element is indexed, where *i* is the numeric index value, or `ST` if the element is not indexed. The `ATTRIBUTES` field is blank.

Instruction

`INSTRUCTION` in the `TYPE` field indicates an instruction. The `VALUE` field contains the name of the instruction. The `ATTRIBUTE` field is blank.

Keyword

`KEYWORD` in the `TYPE` field indicates a keyword. The `VALUE` field contains the name of the keyword. The `ATTRIBUTE` field is blank.

Labels (L NEAR, L FAR)

`L FAR` and `L NEAR` in the `TYPE` field indicate labels. The `VALUE` field contains the label's offset. The `ATTRIBUTES` field contains the segment name, if known.

Numbers (NUMBER)

`NUMBER` in the `TYPE` field indicates a number. The `VALUE` field contains a hexadecimal number, which can be negative only for an integer. The `ATTRIBUTES` field contains `RELOC` for symbols equated to relocatable numbers and `REAL` for symbols equated to floating point numbers.

Procedures (P NEAR, P FAR)

Procedures are identified by `P NEAR` or `P FAR` in the `TYPE` field. The `ATTRIBUTES` field contains its size in bytes, the segment name, and the word count, if one was specified.

Records and Record Fields (RECORD, R FIELD)

`RECORD` and `R FIELD` indicate records and record fields, respectively.

The `VALUE` field for a record is blank. The `ATTRIBUTES` field contains the size of the record in bytes, the number of bits (width) required for that record, and its default value.

The `VALUE` field for a record field contains its shift count. The `ATTRIBUTES` field contains the name of the record containing the field and the field's bit width.

Registers (REG)

REG in the TYPE field indicates a register. The VALUE field contains the register name and the ATTRIBUTES field is blank.

Segments (SEGMENT)

SEGMENT in the TYPE field indicates a code or data segment. The VALUE field is blank. The first entry in the ATTRIBUTES field is the segment size. The remaining attributes duplicate the attributes declared in the segment definition line, including the defaults.

Stack Segments (STACK)

STACK in the TYPE field indicates a stack segment. The VALUE field is blank. The ATTRIBUTES field contains the segment size and information about the attributes, such as whether they are read-write (RW) or PUBLIC. The remaining attributes duplicate the attributes declared in the segment definition line, including the defaults.

Structures and Structure Fields (STRUC, S FIELD)

STRUC in the TYPE field indicates a structure. The VALUE field is blank. The ATTRIBUTES field contains the structure size in bytes, and the number of its fields.

If a symbol is a structure field, its type appears in the TYPE field and SFIELD appears in the ATTRIBUTES field. The VALUE field contains the hexadecimal offset from the start of the structure in which the field was defined.

Undefined Symbols (--------)

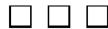
A symbol that was never defined, or was purged and then referenced without definition, is indicated by -------- in the TYPE field. The VALUE field is blank and the ATTRIBUTES field contains --UNDEFINED--.

Variables (V BIT . . . V n)

The types for variables are V BIT, V BYTE, V WORD, V DWORD, V PWORD, V QWORD, VTBYTE, V STRUC, and V *n* (where *n* is the type value).

The VALUE field shows the variable's offset. The ATTRIBUTES field contains the segment name, if known, and PUBLIC or EXTRN, if appropriate. Variable names defined in an EQU statement can have indexing and segment override attributes. The override is displayed as the segment register name. Any indices are indicated by the index register name. If the variable is defined as an array, the item count appears in the ATTRIBUTES field as a decimal number in parentheses.

V ABS in the TYPE field indicates an external absolute number. The VALUE field contains a zero and the ATTRIBUTES field contains EXTRN.



Error Messages A

This appendix begins with a description of the types of assembler errors and their error message formats. Following the descriptions is a numerical list of the assembler source file error and warning messages and their explanations.

Fatal Errors

Fatal errors cause the assembler to stop processing the source file, display an error message, and return control to the operating system. There are three types of fatal errors: invocation control errors, I/O errors, and internal errors. These errors cause the assembler to stop processing the source module without producing an object module. The following sections explain the types of fatal errors and their message formats.

Invocation Control Errors

Invocation control errors occur when controls or their parameters are specified incorrectly on the invocation line. The error messages have the following basic format:

```
ASM386 CONTROL ERROR
  CONTROL: control
  PARAMETER: parameter
  DELIMITER: character
  ERROR: description
ASM386 TERMINATED
```

The *parameter* and *delimiter* lines appear if you specify an incorrect delimiter or control parameter.

The error *descriptions* are the same as source file nonfatal errors. They are explained at the end of this appendix in numerical order.

I/O Errors

I/O errors indicate problems in using external files or devices. I/O error messages have the following format:

```
ASM386 I/O ERROR
  FILENAME = filename
  ERROR = error number and description
ASM386 TERMINATED
```

Where:

filename	is the name of the file containing the error.
error number	is the operating system error number.

Internal Errors

An assembler internal error indicates that an internal consistency check has failed. If an internal error occurs, contact Intel, following the instructions on the inside back cover of this manual. Please save the exact text of the error message, which has the following form:

```
*** ASM386 INTERNAL ERROR : description
```

Nonfatal Errors and Warnings

The remaining errors are nonfatal and occur within the source file itself. When a nonfatal error occurs, the error line assembly is usually wrong; subsequent lines, however, can still be assembled correctly.

The basic nonfatal error message contains an error number, a source line number, and a brief description. No line number is given if the assembler detected the error before the first source line. The message appears in the print and errorprint files after the line on which the error was detected.

The following are the message formats:

```
*** ERROR n IN l, description
*** ERROR n IN l, type description
*** ERROR n IN l, (LINE m) description
```

Where:

n	is a decimal number.
l	is the number of the listing line in which the error occurred.
type	is one of the following:

(PASS 2)	indicates an error in pass 2 of the assembler.
(MACRO)	indicates a macro error.
(CONTROL)	indicates an assembler control error.
(LINE *m*)	is the line number of an error.

See also: Assembler passes, Chapter 1

Syntax Errors

A syntax error indicates that the program does not conform to the assembly language's grammar rules.

The syntax error message has this form:

```
*** ------------/\
*** ERROR 1 IN l, SYNTAX ERROR
```

The assembler usually discards the remainder of the line following the syntax error. If the error occurs within a codemacro definition, the assembler exits definition mode, causing the `ENDM` statement to produce another syntax error, which is eliminated when the first error is corrected.

The pointer normally indicates the location of the syntax error. For example:

```
ASSUME ES
```

produces a syntax error after `ES`, indicating that the line is missing a colon followed by a segment name at the end. However, the assembler may not detect the error until one or more characters later. For example:

```
AAA DB 0
```

produces a syntax error at `DB` although `AAA`, already defined as an instruction (ASCII adjust for addition), is the actual error. The assembler interprets the line as an `AAA` instruction with `DB 0` as the operand field, and because the keyword `DB` is not a legal parameter, the assembler flags it as the error.

The assembler treats codemacro, register, and record names as unique syntactic entities. If you use these kinds of names improperly, you often receive a syntax error. For example:

```
ES EQU 7
```

is a syntax error because `ES` is a register name and is therefore syntactically distinct from an undefined symbol.

Syntax errors can occur for lines that by themselves are syntactically correct, but are misplaced within the program. For example, if the following statement is appropriately placed, it is syntactically correct:

```
FOO ENDS
```

However, if it were placed as follows:

```
DATA      SEGMENT
             .
             .
FOO       ENDS
```

it would produce an error, because a syntax error occurs if a `SEGMENT` or `PROC` statement does not have a corresponding `ENDS` or `ENDP` statement.

Warnings

Warnings occur when the assembler has assembled a source line without producing an assembler error, but in a way that could later cause errors during object module processing or execution. The warning message format is basically the same as the nonfatal error message format:

```
*** WARNING #n IN l, description
```

Macro Errors

When assembling source files, the assembler processes macros first if the `MACRO` control (the default) is in effect. Macro errors are errors detected during this macro pass. An example of a macro error is

```
UNDEFINED MACRO NAME
```

which indicates that the text following a metacharacter (% by default) is not a recognized user function name or built-in macro function.

Macro errors are followed by a trace of the macro call, which is a series of lines containing the names of the primary source file and currently nested include files, and every pending or active macro call.

Control Errors

A control error occurs in a source file control line (or in the invocation line, as discussed earlier). One example is

```
UNKNOWN CONTROL
```

which indicates that a specified control is not legal.

Source File Error and Warning Messages

The remainder of this appendix is a numerical list of the assembler source file error and warning messages and their explanations.

`***    ERROR #1 SYNTAX ERROR`

Your program does not conform to the assembly language's grammar rules.

`***    ERROR #2 TOKEN TOO LONG`

The maximum token length is 255 characters.

`***    ERROR #3 ORDINAL NUMBER TOO LARGE`

Some 64-bit integer values cannot be represented in packed-decimal form. The approximate range of 64-bit binary numbers is -1.8×10^{19} to 1.8×10^{19}, where the range of values that can be represented by the packed-decimal format is $-10^{18} -1$ to $10^{18} -1$.

`***    ERROR #4 BAD ASM386 CHARACTER`

The assembler found an illegal character in the input file. An unprintable ASCII character (which is shown as an up arrow) may cause this error. If the unprintable character is in a string or comment, the string or comment is terminated, and processing continues with the next character; a syntax error may occur.

A printable character that has no function in the assembly language can also cause this error. This often occurs when macro calls, beginning with the macro metacharacter, appear in a file that is assembled with the `NOMACRO` control.

`***    ERROR #5 REAL NUMBER TOO LARGE`

The hexadecimal real number specified does not fit the size of the defined variable.

See also: Ranges of variables, *ASM386 Assembly Language Reference*

`***    ERROR #6 DECIMAL CONVERSION ERROR`

A precision underflow or overflow occurred when converting decimal to extended precision real.

See also: Ranges of variables, *ASM386 Assembly Language Reference*

`***` ERROR #7 ARITHMETIC OVERFLOW IN EXPRESSION OR LOCATION COUNTER

This error occurs when an answer to a calculation does not fit the corresponding storage (for example, not between -128 and 127 or 0 to 255). Such instances include:

- Expressions with large answers or intermediate values
- Division by zero
- Oversize constants

The error also occurs when the evaluation of the location counter gives a result greater than the maximum value (64K for `USE16` segments or four gigabytes for `USE32` segments).

For example, `X DW 80000001H DUP (0)` means duplicate 2G+1 times, a word whose content is 0. The length of a word is 2, therefore the location counter must be incremented by 2 x 80000001H or 4G+2. This is not a valid 32-bit number and error #7 is issued.

Certain floating-point values incorrectly elicit an arithmetic overflow message. These hexadecimal real values are:

SINGLE PRECISION REALS (DD):

```
07FFFFFFFR, 07F800001R, 07F800000R, 0007FFFFFR, 000000001R,
000000000R, 080000000R, 080000001R, 0807FFFFFR, 0FF800000R,
0FF800001R, 0FFC00000R, 0FFFFFFFFR
```

DOUBLE PRECISION REALS (DQ):

```
07FF0000000000001R, 07FF0000000000000R, 0000FFFFFFFFFFFFFR,

00000000000000001R, 00000000000000000R, 08000000000000000R,

08000000000000001R, 0800FFFFFFFFFFFFFR, 0FFF0000000000000R,

0FFF0000000000001R, 0FFF8000000000000R, 0FFFFFFFFFFFFFFFFR
```

`***` ERROR #8 STACK OVERFLOW; STATEMENT TOO COMPLEX

The assembly statement is too complex for the assembler to process. Simplify your statement.

`***` ERROR #9 STACK OVERFLOW; STATEMENT TOO LONG

The assembly statement is too long for the assembler to process. Simplify your statement.

*** ERROR #10 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 are relocatable. If both sides are relocatable, they must both be declared within the same segment, and neither can be external.

*** ERROR #11 UNDEFINED SYMBOL; ZERO USED

An undefined symbol has occurred in an expression. Zero is used in its place, which may cause other errors.

*** ERROR #12 STORAGE INITIALIZATION EXPRESSION IS OF THE WRONG TYPE

The only kinds of expressions allowed in initialization lists are variables, labels, strings, formals, and numbers. This error also occurs when the expression's value is too large for the allocated storage.

*** ERROR #13 ABSOLUTE OPERAND REQUIRED IN THIS EXPRESSION

Certain expression operators require their operands to be absolute numbers. These operators include unary minus, divide, multiply, AND, MOD, NEG, OR, SHL, SHR, XOR, LOW, HIGH, LOWW, HIGHW.

*** ERROR #14 SIZE OF STACK SEGMENT HAS INCREASED PAST 64K

A USE16 stack segment has been specified more than once using the STACKSEG directive with the same stack name. The stack sizes given for each specification are added together to form a total stack size for that particular stack segment. The latest specification has caused the total stack size to exceed 64K.

*** ERROR #15 SIZE OF STACK SEGMENT HAS INCREASED PAST FOUR GIGABYTES

A USE32 stack segment has been specified more than once using the STACKSEG directive with the same stack name. The stack sizes given for each specification are added together to form a total stack size for that particular stack segment. The latest specification has caused the total stack size to exceed 4 gigabytes.

*** ERROR #16 SEGMENT USED TO INITIALIZE CS MUST BE TYPE EO OR ER

The segment containing the label used to initialize the CS register in the END statement must be executable. Therefore, the segment must have an access-type of EO or ER.

`***` `ERROR #17 COMBINE-TYPE DOES NOT MATCH ORIGINAL SEGMENT DEFINITION`

If more than one `SEGMENT-ENDS` pair exists for the same segment in the program, they must have the same combine-type. For example, you cannot specify the first one without a combine-type (private), and declare a subsequent one to be `PUBLIC`. Leaving the combine-type blank for subsequent `SEGMENT` declaratives in the same module is acceptable; the combine-type given in the first declarative is used.

`***` `ERROR #18 SEGMENT CANNOT BE TYPE EO`

The segment used to initialize the `DS` register in the `END` statement, or the `DS` or `ES` register in the `ASSUME` statement cannot have access-type `EO`.

`***` `ERROR #19 SEGMENT USED TO INITIALIZE SS MUST BE TYPE RW`

The segment used to initialize the `SS` register in the `END` or `ASSUME` statement must be writable, and therefore have access-type `RW`.

`***` `ERROR #20 SEGMENT ACCESS-TYPE TO RW`

The segment has been declared to have a data part (`SEGMENT` directive) and a stack part (`STACKSEG` directive). Because the stack part is always `RW`, the data part must also be `RW`.

`***` `ERROR #21 --- FILE DOES NOT EXIST`

In DOS systems, this message may be issued even though the file does exist. The PC/DOS Operating System is installed incorrectly. Re-install the Operating System and make sure that it is DOS V3.0 or later. DOS V3.0 or greater has a different COMMAND.COM file.

`***` `WARNING #21 CS-(E)IP AND/OR SS-SP AND/OR DS NOT INITIALIZED; REQUIRED FOR MAIN MODULE`

The `END` statement has no `CS` and/or `SS` and/or `DS` register initialization. All three of these initializations are necessary for a main module.

`***` `ERROR #22 MISSING END OF SEGMENT STATEMENT`

A segment definition must end with a statement in the form:

`name` `ENDS`

Where:

`name` is the segment name given in the corresponding `SEGMENT` directive.

`***` `ERROR #23 MISSING END OF MODULE STATEMENT`

The `END` directive is required as the last statement in all the assembler modules.

`***` ERROR #24 MISSING NAME STATEMENT; DEFAULT MODULE NAME USED

Every module must contain the NAME directive to include a name on the list file header and in the object module. If the NAME directive is omitted, the name "ANONYMOUS" is used.

`***` ERROR #25 MISSING END OF STRUCTURE STATEMENT

A structure definition must end with a statement in the form:

name ENDS

Where:

name is the structure name given in the corresponding STRUC directive.

`***` ERROR #26 MISSING END OF CODEMACRO STATEMENT

The definition of a codemacro must end with the ENDM statement.

`***` ERROR #27 UNDEFINED SEGMENT IN INITIALIZATION

All segment references within an initialization must be to a defined segment.

`***` ERROR #28 NO DEFINITION FOR PUBLIC SYMBOL

A public symbol must be defined within the module.

`***` ERROR #29 ILLEGAL OPERAND TO THIS OPERATOR

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

`***` ERROR #30 IDENTIFIER MUST BE A LABEL FOR A CS-(E)IP INITIALIZATION

The identifier used in the CS-IP or CS-EIP initialization must be a label. Check the definition of the indicated identifier.

`***` ERROR #31 SEGMENT WITH SAME NAME AS STACK MUST BE PUBLIC AND TYPE RW

The segment has been declared to have a data part (via the SEGMENT directive) and a stack part (via the STACKSEG directive). Because the stack part is always PUBLIC and has access-type RW, the data part must also be PUBLIC and RW.

`***` ERROR #32 VARIABLES NOT ALLOWED IN REGISTER INITIALIZATION

Variables cannot be used to initialize the DS or SS segment registers in the END statement. Only segment names can initialize segment registers in this context. A label is required to initialize the CS segment registers.

`***` ERROR #33 OPERANDS TO LOGICAL OPERATORS MUST BE ABSOLUTE NUMBERS

Other types of operands are not allowed.

***   ERROR #34 OPERAND TO BITOFFSET OPERATOR MUST BE A VARIABLE OR STRUCTURE FIELD

BITOFFSET allows you to convert variables or structure fields to numbers. If you receive this error message, you probably already have a number.

*** ERROR #35 NO DEFINITION FOR COMM SYMBOL

A COMM symbol must be defined within the module.

*** ERROR #36 OPERAND TO TYPE OPERATOR MUST BE A VARIABLE, STRUCTURE FIELD, OR LABEL

TYPE can only be used with a variable, structure field, or label. Any other parameter is illegal.

*** ERROR #37 OPERAND TO LENGTH OPERATOR MUST BE A VARIABLE OR STRUCTURE FIELD

LENGTH can be used only with a variable or a structure field. Any other parameter is illegal.

*** ERROR #38 OPERAND TO SIZE OPERATOR MUST BE A VARIABLE OR STRUCTURE FIELD

SIZE can be used only with a variable or a structure field. Any other parameter is illegal.

*** ERROR #39 OPERAND TO WIDTH OPERATOR MUST BE A RECORD

You cannot obtain the width of anything else.

*** ERROR #40 OPERAND TO MASK OPERATOR MUST BE A RECORD FIELD NAME

MASK of anything else has no meaning.

*** ERROR #41 OPERAND TO STACKSTART OPERATOR MUST BE A STACK SEGMENT

The operand to STACKSTART must be defined with the STACKSEG directive.

*** ERROR #42 OPERAND TO OFFSET OPERATOR MUST BE A VARIABLE OR LABEL

The OFFSET operator allows you to convert variables or labels to numbers. If you receive this error message, you probably already have a number.

*** ERROR #43 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 generates this error:

```
VAR DT 0
PUSH VAR
```

Because VAR is a TBYTE variable, it cannot be pushed on the stack with PUSH.

*** ERROR #44 OPERAND NOT REACHABLE FROM SEGMENT REGISTERS

This error occurs when the ASSUME statement is used incorrectly. For every code segment reference to a variable that is not defined in the current segment, the segment in which that variable is defined must be assumed to be accessible from one of the segment registers. For most programs, a single ASSUME statement at the top of the program for segment registers DS, ES, FS, GS, and SS is sufficient.

*** ERROR #45 BAD SCALE FACTOR; MUST EVALUATE TO THE ORDINAL VALUE 1, 2, 4, OR 8

The only values allowed for index scaling are 1, 2, 4, or 8.

*** ERROR #46 PWORD IS A BAD SCALE FACTOR; MUST EVALUATE TO THE ORDINAL VALUE

A pword is not allowed as an index scale value.

*** ERROR #47 TBYTE IS A BAD SCALE FACTOR; MUST EVALUATE TO THE ORDINAL VALUE

A tbyte is not allowed as an index scale value.

*** ERROR #48 32-BIT AND 16-BIT ADDRESSING CANNOT BE COMBINED

The address expression has both 32-bit and 16-bit elements. For example:

```
MOV AX, [EAX  BX]
```

is not legal.

*** ERROR #49 SYMBOL ALREADY DEFINED; CURRENT DEFINITION IGNORED

A symbol has an illegal multiple definition.

*** ERROR #50 ILLEGAL CIRCULAR EQU CHAIN

The following is an example of a circular chain of EQU statements:

```
VAR_1 EQU MYVAR
MYVAR EQU VAR_1
```

*** ERROR #51 EQU EXPRESSION CANNOT CONTAIN A FORWARD REFERENCE

You cannot equate to expressions containing forward references.

*** ERROR #52 BAD EQU EXPRESSION

An illegal expression occurred in an EQU statement. For example, the following statement causes this error:

```
VAR EQU [BX - SI]
```

`*** ERROR #53 EQU FORWARD REFERENCE CAN ONLY BE A VARIABLE, LABEL, OR EQU SYMBOL`

You can equate to simple forward-reference names, but not to an expression containing forward references.

`*** ERROR #54 COMBINING BIT OFFSET AND BYTE DISPLACEMENT EXCEEDS MAXIMUM SEGMENT SIZE`

The result from adding the bit offset and the byte displacement is greater than the 64K limit for `USE16` segments or the 4 gigabytes allowed for `USE32` segments. The following example shows this case:

```
BT VAR, 0FFFFH
```

where `VAR` is at offset 0FFFFH in a `USE16` segment.

`*** ERROR #55 STRING CONSTANT CANNOT EXCEED EIGHT CHARACTERS`

A string constant used to initialize a dword can contain at most eight characters.

`*** ERROR #56 RELATIVE DISPLACEMENT TOO LARGE FOR A USE16 SEGMENT`

A relative displacement greater than 16K is not allowed in a `USE16` segment. The target would not be reachable using a 16-bit relative displacement.

`*** ERROR #57 RELATIVE DISPLACEMENT TOO LARGE FOR A USE32 SEGMENT`

A relative displacement greater than 4 gigabytes is not allowed in a `USE32` segment. The target would not be reachable using a 32-bit relative displacement.

`*** ERROR #58 ADDITION OF DISPLACEMENT CAUSED OVERFLOW`

The result of the displacement evaluation is either greater than 64K in a `USE16` segment, or greater than 4 gigabytes in a `USE32` segment.

`*** ERROR #59 IMMEDIATE DWORD OVERFLOW`

An expression has a value that is out of range for storage in a dword.

`*** ERROR #60 IMMEDIATE WORD OVERFLOW`

An expression has a value that is out of range for storage in a word.

`*** ERROR #61 DISPLACEMENT TOO LARGE FOR A USE16 SEGMENT`

The displacement computed by the assembler is greater than 64K.

`*** ERROR #62 DISPLACEMENT TOO LARGE FOR A USE32 SEGMENT`

The displacement computed by the assembler is greater than 4 gigabytes.

*** ERROR #63 INVALID SYMBOL TYPE

The symbol type does not match the required operand type for the given instruction. For example, if `F1` is a structure field, then the following statement is an invalid specification:

```
PUSH F1
```

*** ERROR #64 LABEL DECLARED NEAR IS NOT IN THE CURRENT SEGMENT

A label referenced in the current segment is not local to that segment; it was declared in another segment. Change the label type to `FAR`.

*** ERROR #65 TWO REPEAT PREFIXES ARE ILLEGAL

Delete one `REPEAT` prefix.

*** ERROR #66 TWO LOCK PREFIXES ARE ILLEGAL

Delete one `LOCK` prefix.

*** ERROR #67 SEGMENT SIZE EXCEEDED

The location counter has become greater than 64K for a `USE16` segment or greater than 4 gigabytes for a `USE32` segment. Split the segment into smaller segments.

*** ERROR #68 PASS TWO INSTRUCTION SIZE EXCEEDED PASS ONE ESTIMATE

This error occurs when the instruction contains a forward reference and the assembler overestimates the amount of code the forward reference causes the instruction to generate. Overestimating usually occurs when:

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

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

- Otherwise, the assembler assumes that it is not needed.
- The forward reference is a `FAR` label. Explicitly provide the type in this case:

  ```
  JMP FAR PTR FWD_LABEL
  ```

  Otherwise, the assembler assumes `NEAR`.
- `SHORT` is indicated, or an instruction is used that takes only `SHORT` displacements. Change the code so that it does not use a `SHORT` jump.

*** ERROR #69 BAD OPERAND TO MONADIC INSTRUCTION

Monadic means an instruction with one operand. The type of the operand does not match the type required for this instruction.

*** ERROR #70 CURRENT SEGMENT NOT EXECUTABLE

You cannot include instructions in a non-executable segment. Change the segment attribute in the SEGMENT declarative.

*** ERROR #71 FIRST OPERAND IS ILLEGAL

The type of the first operand does not match the type required for this instruction.

*** ERROR #72 FIRST OPERAND CONTAINS AN UNDEFINED SYMBOL

An undefined symbol was included in the expression used as the first operand for this instruction.

*** ERROR #73 SECOND OPERAND IS ILLEGAL

The type of the second operand does not match the type required for this instruction.

*** ERROR #74 SECOND OPERAND CONTAINS AN UNDEFINED SYMBOL

An undefined symbol was included in the expression used as the second operand for this instruction.

*** ERROR #75 ILLEGAL OPERAND COMBINATION

The type of one of the operands to the instruction does not match the type required for the other operands. For example, the following sequence generates this error:

```
VAR DW 0
MOV BL, VAR
```

Because VAR is a WORD variable, it cannot be moved into the register BL.

*** ERROR #76 IMMEDIATE EXCEEDS 31; ONLY THE LOWER FIVE BITS WILL BE USED

The number or expression used as an immediate value is greater than 31, which is illegal in this context.

*** ERROR #77 IMMEDIATE EXCEEDS LIMITS IN THIS CONTEXT

The number or expression used as an immediate value is greater than the legal value for this context.

*** ERROR #79 SECOND OPERAND MUST BE CL

The second operand for this instruction cannot be anything other than the 8-bit general register CL.

*** ERROR #80 FIRST OPERAND MUST BE DX OR EDX

The first operand to this instruction cannot be anything other than the DX or the EDX register.

*** ERROR #81 THIS INSTRUCTION REQUIRES AT LEAST ONE OPERAND

This instruction must be specified with one or more operands. Some instructions such as RET accept one or no operands; others, such as ADD, require two operands.

*** ERROR #82 THIS INSTRUCTION DOES NOT ACCEPT ONE OPERAND

Check the description of this instruction in the *ASM386 Assembly Language Reference*.

*** ERROR #83 THIS INSTRUCTION DOES NOT ACCEPT TWO OPERANDS

Check the description of this instruction in the *ASM386 Assembly Language Reference*.

*** ERROR #84 THIS INSTRUCTION DOES NOT ACCEPT THREE OPERANDS

Check the description of the instruction in the *ASM386 Assembly Language Reference*.

*** ERROR #85 UNDEFINED SYMBOL

The symbol used has not been defined. Add a declaration for the symbol or check for a misspelling of the symbol.

*** ERROR #86 SECOND OPERAND MUST BE AX

The second operand for this instruction cannot be anything other than the AX register.

*** ERROR #87 SECOND OPERAND MUST BE EAX

The second operand for this instruction cannot be anything other than the EAX register.

*** ERROR #88 THIRD OPERAND IS ILLEGAL

The third operand for this instruction is of an incorrect type.

*** ERROR #89 THIRD OPERAND CONTAINS AN UNDEFINED SYMBOL

An undefined symbol is included in the expression used as the third operand for this instruction.

*** ERROR #96 STATEMENT NOT ALLOWED OUTSIDE SEGMENT BOUNDARIES

The statement must be included within a SEGMENT/ENDS pair. Otherwise, it is ignored by the assembler.

*** ERROR #97 EIGHT-BIT REGISTER IS ILLEGAL IN A REGISTER EXPRESSION

8-bit registers are not allowed in a register expression.

*** ERROR #98 ILLEGAL REGISTER EXPRESSION

The register expression contains some illegal operations. For example, the following is an illegal register expression:

```
PUSH WORD PTR DS:[BX] + AX
```

*** ERROR #99 NO MORE THAN TWO REGISTERS ALLOWED IN A REGISTER EXPRESSION

Up to two registers can be specified in a register expression.

*** ERROR #100 ILLEGAL SYMBOLIC REFERENCE IN A REGISTER EXPRESSION

You cannot mix a symbolic reference within a register expression.

*** ERROR #101 ILLEGAL OPERATION ON SYMBOLIC REFERENCE WITHIN SQUARE BRACKETS

Symbols cannot be specified within bracketed register expressions. For example, the following is an illegal operation:

```
PUSH WORD PTR[EBX + VAR]
```

*** ERROR #102 SCALED INDEX REGISTER MUST BE IN SQUARE BRACKETS

Index registers used with scale specifications must be within square brackets.

*** ERROR #103 OPERAND TO DOT OPERATOR MUST BE A STRUCTURE FIELD

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

*** ERROR #104 ILLEGAL FLOATING POINT STACK ELEMENT VALUE; ZERO USED

Stack elements can be specified only as `ST` or `ST(`*i*`)`, where *i* is in the range of 0 to 7.

*** ERROR #105 REGISTER EXPRESSION ILLEGAL OUTSIDE OF SQUARE BRACKETS

A register can undergo arithmetic inside square brackets; the operations are performed on the memory address represented by the bracketed expression. The arithmetic makes no sense outside the brackets, and is flagged. For example, the following is illegal:

```
JMP BX + 3
```

but the following is legal:

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

`*** ERROR #106 ESP CANNOT BE USED AS AN INDEX REGISTER`

Any general register except `ESP` can be used as an index register.

`*** ERROR #107 EXPRESSION CANNOT BE USED AS A FLOATING POINT STACK ELEMENT; ZERO USED`

The expression cannot be used as a stack element index.

`*** ERROR #108 ILLEGAL OPERAND TO SEG OPERATOR`

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

`*** ERROR #109 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN ES`

The destination operand of a string instruction must be accessible through the `ES` register.

`*** ERROR #110 DEFAULT ES SEGMENT REGISTER CANNOT BE OVERRIDDEN`

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

`*** WARNING #111 USE OF THE EVEN DIRECTIVE IN THIS CONTEXT DISABLES CODE OPTIMIZATION`

The assembler does not attempt to optimize instructions containing forward references after specification of the `EVEN` directive.

`*** ERROR #112 ILLEGAL INDEX REGISTER USED IN SECOND OPERAND; MUST BE EDI OR DI`

Only the general registers `EDI` or `DI` are allowed as index registers for the second operand of this instruction.

`*** ERROR #113 ILLEGAL INDEX REGISTER USED IN SECOND OPERAND; MUST BE ESI OR SI`

Only the general registers `ESI` or `SI` are allowed as index registers for the second operand of this instruction.

`*** ERROR #114 ILLEGAL INDEX REGISTER USED IN FIRST OPERAND; MUST BE ESI OR SI`

Only the general registers `ESI` or `SI` are allowed as index registers for the first operand of this instruction.

`*** ERROR #115 ILLEGAL INDEX REGISTER USED IN FIRST OPERAND; MUST BE EDI OR DI`

Only the general registers `EDI` or `DI` are allowed as index registers for the first operand of this instruction.

`*** ERROR #116 ILLEGAL INDEX REGISTER USED; MUST BE EDI OR DI`

Only the general registers `EDI` or `DI` are allowed as index registers in this context.

`*** ERROR #117 ILLEGAL INDEX REGISTER USED; MUST BE ESI OR SI`

Only the general registers `ESI` or `SI` are allowed as index registers in this context.

`*** ERROR #118 NEAR USE16 CALL OR JUMP ILLEGAL IN A USE32 CONTEXT`

In a `USE32` segment, you cannot specify `JMP [AX]` because a `NEAR USE16` jump uses only the lower 16 bits of `EIP`.

`*** ERROR #119 EXCEEDED NUMBER OF OPERANDS ALLOWED FOR CODEMACROS`

The maximum number of operands for a codemacro is 15.

`*** ERROR #120 NO IMPERATIVE OR CODEMACRO DEFINED WITH THIS NAME`

You have coded an undefined instruction.

`*** ERROR #121 OPERANDS DO NOT MATCH ANY IMPERATIVE OR CODEMACRO`

The number of operands specified for the instruction does not match the number required for any known imperatives or codemacros.

`*** ERROR #122 INSIDE A CODEMACRO, THE OPERAND TO THE DOT OPERATOR MUST BE A RECORD FIELD`

You have used the `DOT` operator with a variable of a type other than `RECORD`.

`*** ERROR #123 FORWARD REFERENCE INSIDE A CODEMACRO IS NOT ALLOWED`

Forward references cannot be included within codemacros.

`*** ERROR #124 CANNOT SHIFT A RELOCATABLE VALUE`

This error results when a relocatable value is passed as an operand to an instruction which shifts the operand. Shifting a relocatable value is not allowed.

`*** ERROR #125 NUMBER OF BYTES GENERATED BY A CODEMACRO IS LIMITED TO 255`

The codemacro is too long; the maximum is 255 bytes.

`*** ERROR #126 RELATIVE DISPLACEMENT WILL NOT FIT IN A BYTE`

This instruction expects a relative displacement within the range of -128 to +127.

`*** ERROR #127 RELATIVE DISPLACEMENT WILL NOT FIT IN A WORD`

This instruction expects a relative displacement within the range of -32768 to +32767.

`*** ERROR #128 RELATIVE DISPLACEMENT WILL NOT FIT IN A DWORD`

This instruction expects a relative displacement within the range of -2^{31} and $+2^{31}-1$.

*** ERROR #129 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN S

*** ERROR #130 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN DS

*** ERROR #131 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN GS

*** ERROR #132 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN FS

*** ERROR #133 ILLEGAL OPERAND SEGMENT ASSUMPTION TO OTHER THAN SS

For errors #129-133, the assembler requires that the operand for the instruction be reachable through the register indicated in the error message. These errors are generated when conditions specified in a user-defined codemacro using the NOSEGFIX statement have been violated.

*** ERROR #134 INSTRUCTION WAS PURGED

A purged symbol remains undefined until it is redefined.

*** ERROR #135 ILLEGAL OPERANDS INSIDE OF 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 #136 CANNOT ADD TWO RELOCATABLE NUMBERS

Only absolute numbers can be added.

*** ERROR #137 CANNOT SUBTRACT TWO RELOCATABLE NUMBERS IN DIFFERENT SEGMENTS

Two relocatable numbers can be subtracted only if they have been defined in the current module and in the same segment.

*** ERROR #138 CANNOT HAVE TWO INDEX REGISTERS IN A REGISTER EXPRESSION

Two-register expressions are legal only with one base register and one index register (and an optional displacement).

*** ERROR #139 CANNOT HAVE TWO BASE REGISTERS IN A REGISTER EXPRESSION

Two-register expressions are legal only with one base register and one index register (and an optional displacement).

*** WARNING #140 ILLEGAL USE OF THE CS REGISTER IN AN ASSUME

Unlike ASM86, the assembler automatically assumes that the selector of the current segment is in the CS register. CS is allowed in the ASSUME statement only if followed by NOTHING.

*** ERROR #141 N287 CONTROL SPECIFIED: 80387 INSTRUCTION IS ILLEGAL

When the N287 primary control is specified, any source code lines that contain Intel387 floating-point coprocessor instructions, not supported on the Intel287 coprocessor, are flagged as errors.

*** ERROR #142 INVALID OPERAND TO THE SHORT OPERATOR

The short operator cannot be used in address expressions which represent memory references. The short operator is used in LABEL expressions to indicate that jump is going to be (+127 bytes to -128 bytes) from the end of the current instruction.

*** ERROR #143 SEGMENT OVERRIDE NOT VALID IN A LABEL EXPRESSION

A segment override cannot be used in a label expression where the label type is NEAR or FAR. A segment override is used in an operand which represents a reference to memory.

*** ERROR #144 OPERAND TO LOW MUST BE A NUMBER OR ABS EXTRN

The LOW operator requires as an operand a constant expression that evaluates to a 16-bit number. Other types of operands (e.g., variables, labels, segment names, structure names, or record names) are not allowed.

*** ERROR #145 CANNOT USE SEGMENT OVERIDE WITH A REGISTER

Segment override may only be used with a variable name, a label that is not of type NEAR or FAR, or an address expression.

*** WARNING #146 MORE THAN ONE FORWARD REFERENCE SYMBOL IN AN EXPRESSION

More than one reference in an expression has been made to symbols declared after the expression. Declare the symbol before the reference is made.

*** ERROR #200 80376 DOES NOT SUPPORT USE16 CODE OR STACK SEGMENTS

When the MOD376 control is specified, a USE16 segment directive cannot be used for a code or stack segment in the input file. Nor can a USE16 keyword be used in an EXTRN directive of type NEAR or FAR. USE16 data segments may be included in the input file. The assembler continues processing after detecting this error, but the object file will be invalid.

*** ERROR #201 80376 PHYSICAL ADDRESS SIZE EXCEEDED

The 376 processor has a 24-bit address bus. Thus, a segment must be no larger than 16 megabytes. When the MOD376 control is used, the assembler detects any segments that are too large and issues this error. The assembler continues processing, but the segment wraps to low memory, possibly overwriting segments that are in low memory.

*** ERROR #202 RELOCATABLE CONSTANT EXPRESSIONS NOT ALLOWED

The instruction or operator requires an expression that takes only absolute constants as a value.

*** ERROR #203 VALUE LARGER THAN 256 BYTES NOT ALLOWED

The instruction or operator requires an expression that evaluates to a number in the range of 1 to 256.

*** ERROR #204 TEST REGISTER IS NOT VALID UNLESS MOD486 IS SPECIFIED

The test registers TR3, TR4, and TR5 are only valid when the MOD486 control is specified.

*** ERROR #205 INSTRUCTION IS NOT VALID UNLESS MOD486 IS SPECIFIED

The instructions BSWAP, CMPXCHG, INVD, INVLPG, WBINVD, and XADD are only valid when the MOD486 control is specified.

*** WARNING #206 NO SOURCE DEBUG INFORMATION FOR CODE SEGMENT

There should only be one code segment in a module when the DEBUG control is specified. Source debug information is only generated for one code segment per module.

*** ERROR #207 LOCK PREFIX IS NOT VALID WITH THIS INSTRUCTION

The LOCK prefix is only valid with the memory forms of the following instructions: ADD, ADC, AND, BT, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, XOR.

*** ERROR #300 BINARY ORDINAL REQUIRED IN A DBIT; ZERO USED

For a DBIT initialization, the values must be specified in binary format.

*** ERROR #301 SYMBOL ALREADY DEFINED; THIS DEFINITION IGNORED

This error message appears when a symbol is given an illegal multiple definition.

*** ERROR #302 STORAGE INITIALIZATIONS NOT ALLOWED OUTSIDE OF USER-DEFINED SEGMENT

All storage initializations (DBIT, DB, DW, DD, DP, DQ, DT, structure allocation, and record allocation) must appear within a user-defined segment (a SEGMENT/ENDS pair) or a codemacro definition.

*** ERROR #303 CANNOT HAVE A VARIABLE OR A LABEL IN A DBIT, DB, DQ, OR DT; ZERO USED

The variable or label used has the wrong type for the context. Although conversion to the offset number automatically occurs for variables of type DW, DD, and DP, it does not occur for those of type DBIT, DB, DQ, or DT. You must explicitly provide the OFFSET operator and be sure that the resulting number is absolute. In the case of a DB variable, the resulting number must also be small enough to fit in a byte.

*** ERROR #304 EXTERNAL NOT ALLOWED FOR INITIALIZATION

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

*** ERROR #305 MISMATCHED LABEL ON ENDS

ENDS requires a label that matches the corresponding SEGMENT or STRUCTURE declarative. If this error occurs, one of several things could be wrong. You could have a typographical error, a missing ENDS for a nested segment, or an error in the corresponding SEGMENT or STRUCTURE statement, in which case, this error is eliminated when the other is fixed.

*** ERROR #306 IDENTIFIER IS NOT A STRUCTURE OR RECORD NAME

In this form of data initialization, only structures or records are allowed.

*** ERROR #307 UNDEFINED STRUCTURE OR RECORD IDENTIFIER

You probably used the DOT operator with a structure or record whose name has not yet been defined. Alternately, you could have tried to initialize an undefined structure or record.

*** ERROR #308 TOO MANY OVERRIDING INITIALIZATIONS

When using a structure to allocate and initialize storage, the number of overriding expressions between angle brackets exceeded the number of fields in the structure. All extra values at the right end of the list are ignored. For example:

```
S STRUC
a DB 0
b DB 3
c DW 999H
S ENDS
foo S<1,4,0AAAH      ; This is correct.
baz S<2,5,0BBBH,93>  ; This is incorrect.  It has
                     ; four overriding values and
                     ; only three fields.
abc S<, , , 88>      ; This is also bad.
                     ; Although only one value
                     ; appears, the commas force
                     ; it into the fourth
                     ; position --- but
                     ; the structure has no
                     ; fourth field.
```

*** ERROR #309 STRUCTURE FIELD CANNOT BE OVERRIDDEN

Only structure fields initialized with a single expression, a single question mark, or a single string can be overridden.

*** ERROR #310 OVERRIDING STRING TOO LARGE FOR FIELD

If a structure field is initialized with a single string, the field can be overridden with a string that is less than or equal to it in length. If the overriding string is too long, it is truncated so that it fits into the field. (If it is too short, it is padded by the necessary last characters from the initializing string.)

*** ERROR #311 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 #312 RELOCATABLE VALUE DOES NOT FIT IN ONE BYTE

Relocatable numbers cannot be operands for the `DB` directive.

*** ERROR #313 CANNOT USE A RELOCATABLE NUMBER FOR THIS INITIALIZATION

Relocatable numbers cannot be used in this initialization because it is impossible to determine at assembly-time how to sign-extend the number into the high-order bytes.

*** ERROR #314 STRING LONGER THAN FIELD SIZE ALLOWED ONLY IN DB

All strings outside the DB context are treated as absolute numbers; therefore, strings longer than the field size are overflow quantities.

*** ERROR #315 IDENTIFIER MUST BE A LABEL OR AN EQUATE

In this context, only a label or equate is allowed.

*** ERROR #316 CANNOT HAVE NESTED STRUCTURE DEFINITIONS

Structures cannot be nested.

*** ERROR #317 CANNOT USE A REAL NUMBER FOR DB, DW, OR DP INITIALIZATION

The DB, DW, and DP data initialization directives do not accept real numbers as operands.

*** ERROR #318 CANNOT USE A NEGATIVE DUP FACTOR; ONE USED

The repetition count of a DUP directive must be a positive number, greater than zero. The value 1 is used if the specified value is a negative number.

*** ERROR #320 DUP COUNT MUST BE GREATER THAN ZERO; ONE USED

The repetition count of a DUP directive must be a positive number, greater than zero. The value 1 is used if the specified value is zero.

*** ERROR #350 WORDCOUNT MAY ONLY BE USED WITH FAR PROCEDURES; IGNORED

A wordcount has meaning only for FAR procedures and therefore cannot be specified for NEAR procedures.

*** ERROR #351 WORDCOUNT MAY NOT BE GREATER THAN 31; IGNORED

If the specified wordcount is greater than 31, it is ignored. The procedure is considered to have a wordcount of 0.

*** ERROR #352 DOES NOT MATCH CURRENT PROC NAME IDENTIFIER

ENDP requires a label that matches the corresponding PROC declarative. One of several things could be wrong: a typographical error, a missing ENDP for a nested procedure, or an error in the corresponding PROC line, in which case this error is eliminated when the other is fixed.

*** ERROR #353 CANNOT HAVE MORE THAN ONE NAME DECLARATIVE

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

*** ERROR #354 SEGMENT CONTENTS DO NOT AGREE WITH ACCESS-TYPE

Either the segment contains executable code and has an access-type of RO or RW, or the segment contains data and has an access-type of EO.

*** ERROR #355 ACCESS-TYPE SET ACCORDING TO SEGMENT CONTENTS

After a SEGMENT declarative is processed, the assembler keeps track of whether code and/or data is contained in the segment. If the segment's access-type has not been set by the time the first ENDS is encountered, the information about the segment's contents is used to set the access-type.

*** ERROR #356 MISSING END OF PROCEDURE STATEMENT

A labelled ENDP statement was expected. You probably have specified an ENDS (end of segment) or an END (end of module) statement before closing the procedure definition.

*** ERROR #357 CODEMACRO NAME WAS PREVIOUSLY DEFINED AS A NON-CODEMACRO

Having non-codemacro definitions of a codemacro identifier is illegal. If a codemacro name has already been defined as something other than a codemacro, 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 #358 TWO CODEMACRO FORMALS HAVE THE SAME NAME

All formals must have different names within a given codemacro definition.

*** ERROR #359 CANNOT HAVE MORE THAN 15 FORMAL PARAMETERS

This limitation is imposed by the internal codemacro coding formats.

*** ERROR #360 ILLEGAL SPECIFIER/MODIFIER FOR A CODEMACRO FORMAL

The only specifier letters allowed are A, C, D, E, F, M, R, S, T, and X. The only modifier letters allowed are B, BIT, D, DN, P, Q, T, and W (or none may be specified).

*** ERROR #361 SECOND PARAMETER MUST BE A FORMAL

The MODRM statement requires that the second parameter must be a formal parameter in the required format for this codemacro. For example, the following is in error:

```
CODEMACRO USR_MODRM FORMAL1:X

MODRM 0,0
ENDM
```

See also: Codemacro reference, *ASM386 Language Reference*

See Section 9.2 of the for details.

*** ERROR #362 ILLEGAL NESTED CODEMACRO DEFINITIONS

Nested codemacro definitions are not allowed.

*** ERROR #363 ILLEGAL CODEMACRO SPECIFIER RANGE VALUE

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

*** ERROR #364 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 the assembler encounters something other than a formal parameter, this error message appears.

*** ERROR #365 STATEMENT MAY NOT APPEAR OUTSIDE A CODEMACRO DEFINITION

The directive used (RELB, WARNING, etc.) can be specified only within a macro definition.

*** ERROR #366 CODEMACRO NAME MUST BE AN IDENTIFIER

A codemacro name must follow the same rules as any other assembler identifier. For example, it cannot begin with a digit.

*** ERROR #367 FIRST PARAMETER MUST BE A FORMAL OR A NUMBER

MODRM requires the first parameter to be the name of a formal parameter or an absolute number. For example, in the following, the first parameter AX to the MODRM statement is illegal:

```
CODEMACRO USER-MODRM FORMAL1:X
MODRM AX, FORMAL1
ENDM
```

*** ERROR #368 PARAMETER MUST BE A FORMAL WITH AN E, M, OR X SPECIFIER

This message signals an incompatibility between the type of a formal parameter and its usage.

*** ERROR #369 SECOND PARAMETER MUST BE A FORMAL WITH AN M OR X SPECIFIER

This message signals an incompatibility between the type of a formal parameter and its usage.

*** ERROR #370 FIRST PARAMETER MUST BE A SEGMENT REGISTER

NOSEGFIX requires the first parameter to be a segment register.

***   ERROR #371 PARAMETER MUST BE A FORMAL WITH CB, CW, CD, OR CDN SPECIFIER

A relative displacement statement in a codemacro definition requires the parameter to be a formal parameter list with the corresponding specifiers.

***   ERROR #372 FORMAL PARAMETER HAS ILLEGAL SPECIFIER TYPE

Specifiers can have only certain types. For example, a PREFIX67 statement could not use a formal with an A specifier.

***   ERROR #373 PARAMETER MUST BE A FORMAL

The parameter to this codemacro statement must be a formal. For example:

```
PREFIX67 0
```

is illegal.

***   ERROR #374 ACTUAL PARAMETER HAS ILLEGAL TYPE

The type of the actual parameter does not match that of the formal definition.

***   ERROR #375 NEGATIVE NUMBER NOT ALLOWED IN THIS CONTEXT

Negative numbers are not allowed in certain contexts, such as STACKSEG declaratives and DUP counts.

***   ERROR #376 MEMORY REFERENCE CANNOT BE REACHED WITH GIVEN SEGMENT REGISTER

The code is probably missing an ASSUME statement, so that the assembler cannot determine the segment base.

***   ERROR #377 SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)

The assembler has encountered one or more privileged instructions in the segment. There are two types of privileged instructions: instructions that can be executed only at privilege level 0, and instructions whose execution is restricted to IOPL level or more trusted.

The instructions that can be executed only at level 0 are LGDT, LLDT, LIDT, LTR, LMSW, CLTS, and HLT. The instructions whose execution is restricted to IOPL level or more trusted are INSB, INSW, OUTSB, INS, OUTS, IN, OUT, CLI, and STI.

Additional instructions that can be executed only at level 0 include MOV to or from CR0, CR2, CR3, DR0-3, DR6, DR7, TR3, TR4, TR5, TR6, and TR7.

The lowest privilege level that can execute these instructions is indicated by the I/O privilege level value in the flag register.

***   ERROR #378 ILLEGAL COMM VARIABLE TYPE

Only variables or labels can be declared as COMM and they cannot be initialized.

*** ERROR #379 CANNOT PURGE PUBLIC OR EXTRN VARIABLE

PUBLIC or EXTRN symbols cannot be purged.

*** ERROR #380 CANNOT PURGE UNDEFINED SYMBOL

The symbol you attempted to purge is undefined. (It may already have been purged.)

*** ERROR #381 CANNOT LIST MORE THAN 255 EXTERNALS IN A SINGLE STATEMENT

A single assembler statement may not contain more than 255 symbols declared to be EXTRN.

*** ERROR #383 SEGMENT ACCESS-TYPE HAS BEEN CHANGED

This message is a reminder that you have reopened a segment with a different access-type, which is legal as long as the access-types are compatible.

*** ERROR #384 SEGMENT REOPENED WITH CONFLICTING ACCESS OR USE ATTRIBUTE

The compatible sets of access-types are RO and RW, with a resulting type of RW, or any combination of RO, EO, and ER with a resulting type of ER. The USE attribute of a segment cannot be changed.

*** ERROR #385 SYSTEM ERROR CAUSED BY ACCESS TO OBJECT MODULE

An error occurred while the object module was being output. It could be an internal error or an I/O error.

*** ERROR #500 UNDEFINED MACRO NAME

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

*** ERROR #501 ILLEGAL EXIT MACRO

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

*** ERROR #502 FATAL SYSTEM ERROR

The macro processor discovered a loss of hardware and/or software integrity. Contact Intel, following the instructions on the inside back cover of this manual.

*** ERROR #503 ILLEGAL EXPRESSION

A numeric expression was required as a parameter to one of the built-in macro functions. The function call has been terminated and processing continued with the character following the illegal expression.

*** `ERROR #504 MISSING "FI"`

The `IF` built-in function did not end with `FI`.

*** `ERROR #505 MISSING "THEN"`

A call to the `IF` macro function requires a `THEN` statement following the `IF` conditional expression clause. The call to `IF` has been aborted and processing continued at the point in the string at which the error was discovered.

*** `ERROR #506 ILLEGAL ATTEMPT TO REDEFINE A MACRO`

You cannot redefine a built-in macro function or parameter name. A user-defined macro cannot be redefined inside an expansion of itself.

*** `ERROR #507 MISSING IDENTIFIER IN DEFINE PATTERN`

In a `DEFINE` statement, the occurrence of an at sign (@) indicates that an identifier type delimiter follows. No such delimiter existed and the `DEFINE` was aborted. Scanning continued from the point at which the error was detected.

*** `ERROR #508 MISSING BALANCED STRING`

The macro processor expected a balanced-text string. The macro call was aborted and scanning continued from the point at which the error was detected.

*** `ERROR #509 MISSING LIST ITEM`

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

*** `ERROR #510 MISSING DELIMITER`

A required delimiter was not present. The macro call was aborted and scanning continued from the point at which the error was detected. This error occurs only if a user macro was defined with a call-pattern containing two adjacent delimiters. If the first delimiter was scanned but was not immediately followed by the second, this error occurs.

*** `ERROR #511 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 was omitted, causing the macro processor to scan to the end of the file searching for the missing delimiter. This error can occur even if the closing delimiter of a macro call was given (and any preceding delimiters were not given) because the macro processor searches for delimiters one at a time.

*** `ERROR #512 DYNAMIC STORAGE (MACROS OR ARGUMENTS) OVERFLOW`

Either a macro argument was too long (possibly because of a missing delimiter) or enough space is not available because of the number and size of the macro definitions. All pending and active macros and include control lines are popped and scanning continues in the primary source file.

`***` `ERROR #513 MACRO STACK OVERFLOW`

Excessive recursion in macro calls, expansions, or include control lines has caused the macro stack to overflow. All active macro calls (macros whose values are currently being read, as well as various temporary strings used during the expansion of some built-in macro functions), all pending macro calls (calls to macros whose arguments are still being scanned), and all includes are popped, and scanning continues in the primary source file.

`***` `ERROR #514 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 that for `ERROR #513 MACRO STACK OVERFLOW`.

`***` `ERROR #516 LONG PATTERN`

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

`***` `ERROR #517 ILLEGAL METACHARACTER`

You attempted to change the macro processing metacharacter to an illegal character (a blank, letter, numeral, parenthesis, or asterisk). The current metacharacter remains unchanged.

`***` `ERROR #518 UNBALANCED ")" IN ARGUMENT TO USER-DEFINED MACRO`

The macro processor encountered an unmatched right parenthesis while scanning a user-defined macro. The macro call is aborted and scanning continues from the point at which the error was detected.

`***` `ERROR #519 ILLEGAL ASCENDING CALL`

A macro call beginning inside the body of a user-defined macro or built-in macro function was incompletely contained inside that body (possibly because of a missing delimiter for the macro call). The call is aborted.

`***` `ERROR #520 BAD CONTROL PARAMETER`

A control parameter is out of bounds, of the wrong type, or missing. Check for typographical errors.

See also: Control descriptions, Chapter 3

`***` `ERROR #521 MULTIPLE INCLUDE`

Only one `INCLUDE` control is allowed on a single line. Only the first (leftmost) `INCLUDE` is processed; the rest are ignored.

*** ERROR #600 ASM386 INTERNAL ERROR

An internal consistency check has failed. Contact Intel, following the instructions on the inside back cover of this manual.

*** WARNING #601 TOO MANY ERRORS; FURTHER ERROR MESSAGES SUPPRESSED

After the twentieth 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 #602 MISSING INPUT FILE

The assembler found the end of the invocation line before a source file specification was scanned.

*** ERROR #603 INVALID SYNTAX

Check the manual syntax description for this control.

*** ERROR #604 ILLEGAL DELIMITER

The assembler found a character in a control line or the invocation line that is not a legal delimiter. Check to see that the correct characters were used and that all the parameters were correctly entered.

*** ERROR #605 MISPLACED PRIMARY CONTROL

Primary controls must appear at the start of the source file before all comments and blank lines.

*** ERROR #606 UNKNOWN CONTROL

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

*** ERROR #607 EXPECTED LEFT PARENTHESIS

The assembler expected a left parenthesis as the delimiter for a control parameter.

*** ERROR #608 RESTORE WITHOUT SAVE

A RESTORE control was encountered without a corresponding SAVE control.

*** ERROR #609 INVALID NUMERIC VALUE

An invalid number was used as a parameter for a control.

*** ERROR #610 PAGE WIDTH OUT OF RANGE

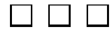
The parameter for PAGEWIDTH control must be a decimal integer from 60 to 132.

*** ERROR #611 PAGE LENGTH OUT OF RANGE

The parameter for the PAGELENGTH control must be a decimal integer from 10 to 65535.

*** ERROR #612 EXPECTED RIGHT PARENTHESIS

The assembler expected a right parenthesis as the delimiter for a control parameter.



System Hardware and Software Requirements B

This chapter describes the hardware and software requirements, and the procedure for making required modifications to the operating system.

Hardware and Software Requirements

- Hardware -- IBM PC XT or IBM PC AT or fully equivalent system
- Operating system -- DOS Version 3.0 or later
- Fixed disk storage capacity -- sufficient for the size of the product (360K bytes)
- System memory requirements -- 512K bytes minimum RAM for ASM386 v3.0, 357K bytes for ASM386 v4.0

Modifying the System Configuration

Before using Intel Software Development Tools from DOS, the system configuration file `CONFIG.SYS` must be created or modified to include the `FILES` and `BUFFERS` commands. The `FILES` command specifies the maximum number of the files that can be opened at the same time. The `BUFFERS` command specifies the number of disk buffers allocated in memory. To use Intel Software Development Tools, set the value of `FILES` to 12 (or greater) and set the value of `BUFFERS` to 10 (or greater).

Follow these steps to create the `CONFIG.SYS` file using the `DOS COPY` command:

1. Type:

   ```
   copy con \config.sys <CR>
   ```

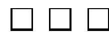
2. Enter the commands:

   ```
   FILES=12 (or greater) <CR>
   BUFFERS=10 (or greater) <CR>
   ```

3. To save the file, press the F6 key and then press <ENTER>.

4. Reboot the system.

If this file already exists on the system, use an editor to add or modify the existing file by including the commands `FILES=12` (or greater) and `BUFFERS=10` (or greater).



Index

A

addresses and offsets, 59
assembler errors, 79
attributes, 68
ATTRIBUTES field, 74

B

binder (BND386), 11
blank lines, 16
builder (BLD386), 11

C

codemacro, 74
COMM attribute, 69
command, 13
command files, 30
command lines, 14
comment lines, 16
control
- errors, 83
- line indicator ($), 22
- lines, 16, 20, 22
- parameter delimiters, 14
- parameters, 20
- precedence, 19, 20

controls, 13, 14, 16, 19, 20
- general, 16
- in commands, 14
- in macros, 23
- within macros, 22

D

DEBUG control, 35
debuggers, 58
descriptor tables, 11
DOS batch files, 29

E

EJECT control, 36
equated symbols, 68
ERRORPRINT control, 37
errorprint file, 26, 50
external symbols, 74

F

fatal errors, 79
floating-point coprocessor, 47
floating-point stack element, 69, 75

G

GEN control, 39
general controls, 19
GENONLY control, 39

H

hardware/software requirements, 113
header, 43, 49, 51, 56, 67

I

I/O errors, 79, 80
in-circuit emulators, 10
INCLUDE control, 42
indexing attribute, 68
input source, 14
instruction, 75
internal errors, 79, 80
invocation, 29
    commands, 14
    examples, 15
    syntax, 13
invocation control errors, 79

K

keyword, 75

L

labels, 75
librarian (LIB386), 11
limits, 25
line numbers, 63, 71
LIST control, 43
listing, 43, 52, 55, 62, 63
listing file, 26
location counter, 63, 68
logical files, 26
logical names, 61

M

macro, 26, 63
    call, 83
    calls, 39, 42, 44
    definitions, 20, 21
    errors, 83
    expansion, 39, 71
    metacharacter (%), 13
    nesting, 22
    processor, 22, 28
    string, 13
MACRO control, 44
mapper (MAP386), 11
maximum
    nesting level, 42
    nesting level of SAVEs, 53
    page width, 50
    segment size, 59
    title length, 56
minimum page width, 50
MOD376 control, 45
MOD386 control, 45
MOD486 control, 45
multiple controls, 20

N

N287 control, 47
N387 control, 47
NAME field, 74
nesting indicator, 70
NODEBUG control, 35
NOERRORPRINT control, 37
NOGEN control, 39
NOLIST control, 43
NOMACRO control, 44
nonfatal errors, 81
NOOBJECT control, 48
NOPAGING control, 51
NOPRINT control, 52
NOSYMBOLS control, 55
NOTYPE control, 58
NOXREF control, 62
numbers, 75

O

object code, 26, 63, 70
OBJECT control, 48
object file, 26, 35, 48
object files, 11
object module, 58
object modules, 9
OMF-386, 11
operand sizes, 59
output file, 14, 26
output files, pathname limitations, 26
override attribute, 68
override prefixes, 59

P

page tables, 11
PAGELENGTH control, 49
PAGEWIDTH control, 50
PAGING control, 51
parameters, 14
pathnames, limitations, 26
primary controls, 16, 18
PRINT control, 52
print file, 26, 43, 49, 50, 51, 52, 55, 62, 63
procedures, 75
program restrictions, 25
public symbols, 74

R

record definitions, 68
records and record fields, 75
registers, 76
relocation indicator, 70
RESTORE control, 53

S

SAVE control, 53
scanning modes, 22
segment and system descriptors, 11
segments, 76
severe errors, 28
sign-off message, 28
sign-on message, 28
source code, 63
source file, 13, 16, 28, 42
source file controls, 20
source lines, 26, 37
source statements, 71
source text, 39, 63
stack, 53
stack segments, 76
structures and structure fields, 76
symbol table, 28, 55, 62, 72
symbol table fields, 74
symbolic debugging, 35
SYMBOLS control, 55
syntax errors, 81, 82
system utilities, 11

T

task state segments, 11
temporary files, 61
temporary work files, 27
TITLE control, 56
TYPE control, 58
TYPE field, 74

U

undefined symbols, 76
USE16 control, 59
USE32 control, 59
utilities, 9

V

VALUE field, 74
variables, 77

W

warnings, 81, 82
WORKFILES control, 61

X

XREF control, 62

If you need to contact Intel Customer Support

Contacting us is easy. Be sure that you have the following information available:

- Your phone and FAX numbers ready
- Complete description of your hardware or software configuration(s)
- Your product's product code
- Current version of all software you use
- Complete problem description

Type of Service		How to contact us
FaxBACK* fax-on-demand system 24 hrs a day, 7 days a week	Using any touch-tone phone, have technical documents sent to your fax machine. Know your fax number before calling.	U.S. and Canada: (800) 628-2283 (916) 356-3105 Europe: +44-1793-496646
Intel PC and LAN Enhancement Support BBS 24 hrs a day, 7 days a week	Information on products, documentation, software drivers, firmware upgrades, tools, presentations, troubleshooting.	U.S and Canada: (503) 264-7999 Europe: +44-1793-432955 Autobaud detect 8 data bits, no parity, 1 stop
CompuServe* Information Service 24 hrs a day, 7 days a week	Worldwide customer support: information and technical support for designers, engineers, and users of 32-bit iRMX® OS and Multibus product families.	Worldwide Locations: (check your local listing) Type: GO INTELC once online.
Customer Support	Intel Multibus Support engineers offering technical advice and troubleshooting information on the latest Multibus products.	U.S. and Canada: (800) 257-5404 (503) 696-5025 FAX: (503) 681-8497 Hrs: M-F; 8-5 PST Europe: +44-1793-641469 FAX: +44-1793-496385 Hrs: M-F; 9-5:30 GMT
Hardware Repair	Multibus board and system repair.	U.S. and Canada: (800) 628-8686 (602) 554-4904 FAX: (602) 554-6653 Hrs: M-F; 7-5 PST Europe: +44-1793-403520 FAX: +44-1793-496156 Hrs: M-F; 9-5:30 GMT
Sales	Intel Sales engineers offering information on the latest iRMX and Multibus products and their availability.	Worldwide: Contact your local Intel office or distributor U.S. and Canada: (800) 438-4769 (503) 696-5025 FAX: (503) 681-8497 Hrs: M-F; 8-5 PST
Correspondence Mail letters to:	Worldwide: Intel Customer Support Mailstop HF3-55 5200 NE Elam Young Parkway Hillsboro, Oregon 97124-6497	Europe: European Application Support Intel Corporation (UK) Ltd. Pipers Way Swindon, Wiltshire England SN3 1RJ

* Third-party trademarks are the property of their respective owners.