Notice

Hewlett-Packard 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. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.


This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Microtec is a registered trademark of Microtec Research Inc.

SunOS, SPARCsystem, OpenWindows, and SunView are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Hewlett-Packard Company
P.O., Box 2197
1900 Garden of the Gods Road
Colorado Springs, CO 80901-2197, U.S.A.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause in DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are set forth in FAR 52.227-19(c)(1,2).
About this edition

Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition dates and the corresponding HP manual part numbers are as follows:

Edition 1  64871-90901, October 1988 E1088
Edition 2  64871-97004, July 1991
Edition 3  B1449-97000, April 1993

B1449-97000 incorporates information which previously appeared in 64871-97004, 64871-97005, 64871-97006, and 64871-92003.

Certification and Warranty

Certification and warranty information can be found at the end of this manual on the pages before the back cover.
Features

The as86 assembler:

• generates code for the complete Intel 8086/8088 and 80186/188 instruction set
• generates code for NEC V20/V30 extensions to 8086/186 instruction set
• supports Intel 8087 floating-point coprocessor instructions
• permits repeated definition of the same or of different code, data, and constants segments within a single source file
• has high-level data structures for structured data types, bit fields, and symbolic memory references
• allows control of the assembly process (conditional assembly, structured control, listing and output control) through a flexible set of assembly control statements
• produces extensive program listings that can include symbol table/cross reference information
• as part of the HP B1449 8086/186 Advanced Cross Assembler/Linker package, is well-integrated with the HP B1493 8086/186 C Advanced Cross Compiler
• comes with a powerful, string-oriented macro preprocessor. The macro preprocessor adds even more flexibility to the assembler with its features (including support for recursive macros).

The ld86 linking loader:

• produces relocatable object modules for later re-linking (incremental linking)
• produces absolute object modules in HP-OMF 86 format absolute, Intel Hexadecimal Object file format absolute, or HP 64000 format absolute
• can include symbols in the absolute object module for symbolic debugging
• allows independent specification of all relocatable segment load addresses
• allows specification of relocatable segment loading order
• supports segment groupings into either GROUP or CLASS
• allows definition of public symbols, or changes to the memory locations of previously-defined public, at load-time (except for incremental links)
• can generate a cross reference table of public symbols and the modules in which they are referenced, and also a memory map
• can load object modules from a user library created by the ar86 librarian
• can make a copy of initialized data values that can be placed in ROM
• gives detailed, well-documented error messages

The ar86 librarian:
• produces libraries that can be loaded by the linking loader
• can add, delete or replace individual modules in a library
• can display library directories
• supports batch command line input and return codes for make-type procedures
• has an optimized structure for fast access during load-time
• can be run in interactive mode as well as batch mode
• can accept Intel library files as input
In This Book

This book is organized into three parts:

**Part 1. Quick Start Guide**  
A short lesson to get you started and summaries of command syntax.

**Part 2. Assembler/Macro Preprocessor Reference**  
Detailed information about the assembler and macro preprocessor.

**Part 3. Linker/Librarian Reference**  
Detailed information about the linker and librarian.

**Part 4. Error Messages Reference**  
Lists of the error messages you might see during the assembly/linking process, and what to do about them.
Contents

Part 1 Quick Start Guide

1 Getting Started
   Objectives of the Example Program  2
   Description of the Example Program  3
   The "mov_mesg.s" Program Module  3
   The "transfer.s" Program Module  7
   The "delay.s" Program Module  8
   Assembling the Program Module Source Files  9
   Starting the Macro Preprocessor  9
   Starting the Assembler  9
   Viewing the Assembler Listing Files  9
   Creating an Example Library File  15
   Linking the Program Module Relocatable Object Files  17
   Linking the Object Modules  18
Contents

2 Command Syntax
   as86(1)  23
   ap86(1)  30
   ld86(1)  32
   ar86(1)  45

Part 2 Assembler/Macro Preprocessor Reference

3 Assembler Introduction
   Instruction Set  52
   Target Microprocessors  52
   Assembler Operation  52
   File Formats  53
   Input File Characteristics  53
   Output File Characteristics  53

4 Assembler Syntax
   Assembler Character Set  56
   ASCII Codes  58
   Symbols  59
   Symbol Formation  59
   Keywords  60
   Instruction Mnemonics  62
   Codemacro  62
   Label  62
5 Symbol and Expression Attributes

- TYPE 73
- OFFSET 73
- BASE 74
- INDEX 74
- SEGMENT 75
- SEGMENT RELOCATION 75
- RELOCATION TYPE 75
- SEGMENT ADDRESSABILITY 76
- CS ADDRESSABILITY 77

6 Assembler Directives

- Segmentation Directives 81
- Program Segmentation 81
- Default Segment - ??SEG 82
Contents

Data Definition Directives 83
Data Objects 83

Linkage Directives 84
Program Linkage 84
ASSUME 86
DB, DW, DD, DQ, DT 88
END 94
EQU 96
EVEN 99
EXTRN 100
GROUP 103
LABEL 105
NAME 107
ORG 108
PROC/ENDP 109
PUBLIC 111
PURGE 112
RECORD 114
SEGMENT/ENDS 118
STRUC/ENDS 123

7 Expressions
Reference Syntax Conventions 128

Expression Overview 128
Absolute Expression 129
Relocatable Expression 130
External Expression 130

Expression Operands 131
Numeric Values 131
Memory and Register Expressions 133
EQU 136

Expression Operators Introduction 137

Arithmetic Operators 137
Unary Plus, Unary Minus 137
Binary Addition, Subtraction  138
[ ] Square Brackets  139
. (Dot operator)  140
Multiplication, Division, Modulo  141
SHL, SHR  142
HIGH, LOW  143

Logical Operators  144
AND, OR, XOR  144
NOT  145
EQ, NE, LT, LE, GT, GE  145

Memory Operators  146
SHORT  146
THIS  146
PTR  147
Segment or Group Override  148
OFFSET  149
SEG  149
TYPE  150
LENGTH  151
SIZE  152

Record Operators  154
MASK  154
WIDTH  154

Segment and Group Operators  156
SEGOFFSET  156
GRPOFFSET  156
SEGSIZE  157
GRPSIZE  158

Operator Precedence  159

8 Instructions and Operands

Operands  162
Accepted Operands  162
9 Assembler Controls

General Syntax for Assembler Controls 195
Primary and General Controls 195
Controls on the Command Line 195
Control Conflicts 196
Controls and File Names 196
Control Abbreviations 196
Controls and the Macro Preprocessor (ap86) 196

Primary Controls 197
[NO]CASE 197
DATE(string) 197
[NO]DEBUG 198
[NO]ERRORPRINT (filename) 198
[NO]EXTERN_CHECK 198
GEN 199
GENONLY 199
[NO]GROUP_INFO 199
[NO]HLASSYM 200
[NO]MACRO(string) 200
MOD086 200
Contents

MOD186 201
MODV20 201
[NO]OBJECT (filename) 201
[NO]OPTIMIZE 202
PAGELENGTH(n) 202
PAGEWIDTH(n) 202
[NO]PAGING 203
[NO]PRINT(filename) 203
[NO]SYMBOLS 204
[NO]TYPE 204
[NO]UNREFERENCED_EXTERNALS 204
[NO]WARNING 205
WORKFILES( ... ) 205
[NO]XREF 205

General Controls 206
EJECT 206
[NO]GEN 206
GENONLY 206
INCLUDE(filename) 207
[NO]LIST 207
RESTORE 207
SAVE 208
TITLE(string) 208

Operational Differences in the Different Modes 209
8086 Mode 209
80186 Mode 209
V20 Mode 209
10 Assembler Listing Description

Assembly Listing 212

Cross Reference and Symbol Table Format Description 213

11 Codemacros

Referencing Codemacros 218

Codemacro Directives 219
CODEMACRO 219
ENDM 221

Codemacro Matching 221

The Specmod Field 223

Range Specification 227

Codemacro Matching Examples 229

Expressions in Codemacros 231

Directives within Codemacros 232
DB, DD, DW 233
MODRM 234
NOSEGFIX 235
ONLY186 (186 Mode Only) 236
Record Name Initialization 237
RELB, RELW 238
RFIX, RFIXM, RNFIX, RNFIXM, RWFIX 239
SEGFIX 241
12 Macro String Preprocessor Introduction

Input Source Characteristics 244
The Metacharacter ' % ' And The Call Pattern 245
Metacharacter Syntax 246
Literal Character * 247
Input Parsing 248
Output Buffering 248
Include Files 248
Macro Expressions 250
Operators 250
Numbers 251
Symbols 251
Balanced Text String (baltex) 252

13 Pre-Defined Macro Functions

Pre-Defined Macro Functions 254
% ' (Comment Function) 255
%n and % (Escape and Bracket Functions) 255
%EQS, %NES, %LTS, %LES, %GTS, %GES 256
%EVAL 257
%EXIT 258
%IF (Conditional Assembly Function) 258
%LEN 259
%MATCH 260
%METACHAR 262
%REPEAT 262
%SET 263
%SUBSTR 264
%WHILE 264
Contents

Example Problem  265

14 User-Defined Macros
  %DEFINE  269
  Macro Reference  271
  Referencing Macro-time Symbols  273

15 Assembler versions
  Version 3.10  276
  New Product Numbers  276
  New Assembler Controls  276
  New Linker/Loader Controls  276
  New Assembler Defaults  276
  New Location for Man Pages  276
  Version 3.00  276
  Demo Directory Change  276
  New Assembler Controls  277
  New Assembler Operators  277
  New Linker Commands  277
  Other Linker Changes  277

16 Converting HP 64853 Assembly Language Programs
  acvt86 Introduction  280
  Assembler Differences  281
  External Declarations  283
  Porting Procedure—Main Files with INCLUDE Files  284
  acvt86 Warnings, ap86 Errors, as86 Errors  285
  Code Substitution  286
  Byte ordering for BIN, DECIMAL, HEX, OCT  287
  Manual Macro Translations  287
  Macro Calls  288
Contents

acvt86(1) Command Syntax 290

Comparison of Keywords 294
ALIGN 294
ASSUME 294
COMN 294
DATA 295
< EOF> 295
EQU 295
EXPAND 296
EXT 296
Label Field 296
LIST 297
MASK 297
NAME 297
NOWARN 297
Operator Field 297
ORG 298
PROC 299
PROG 301
REAL 301
Reserved Words 301
SPC 301
WARN 302
*(Comment) 302

Linking to 64853 Programs 303
L_to_o86(1) 304

nm64(1) 305

17 8086/186 Instructions in Hexadecimal Order

18 8086/186 Instruction Set Summary
Footnotes 357
Part 3 Linker/Librarian Reference

19 Linker/Loader Introduction

Linking And Loading From Libraries 362
Linking to the 8087 362
M:_WST, M:_WT, M:_NST, and other Floating Point Externals 363

20 Linker/Loader Operation

Primary Functions 368
Incremental Linking 368

Segments and Load Addresses 369
Logical Segment 369
Base Address 369
Physical Segment 369
Absolute Segment 370
Relocatable Segment 370
Paragraph (Segment) Number 370
Class 370
Group 371
Group Base Address 371
Module 372

Complete Name 372

Segment Attributes 372
Combine-type Attributes 373
Align-type Attribute 374
Segment Alignment 375
21 Loader Commands

Loader Commands Introduction 384
Command Symbols 384
Complete Name 385
Order of Commands 385
Command Length 385

Loader Command Descriptions 386
ALIGN 387
Comment (*) 389
END 389
ERROR, WARN, NOERROR 389
EXIT 390
FORMAT 391
GROUP 392
INITDATA 394
LENGTH 395
LIST, NLIST 396
LISTABS 400
LISTMAP 400
LOAD 402
NAME 403
ORDER 403
PUBLIC 405
RESADD, RESNUM 406
SEG 407
SEGSIZE 409
START 410
TYPEMERGE 411
WIDTH 412
Contents

Commands Summary 441
Shorthand Names 442
ADDLIB 444
ADDMOD 444
CLEAR 445
CREATE 445
DELETE 446
DIRECTORY 446
END 447
EXTRACT 447
FULLDIR
LIST 448
HELP 449
OPEN 449
REPLACE 450
SAVE 450

25 Librarian Listing Description

Librarian Sample 1 453
Librarian Sample 2 456

Part 4 Error Messages Reference

26 Error Message Formats

Warning 462
Error 462
Fatal Error 462
Interactive and Non-Interactive Conditions  462

27  Assembler Error Messages
    Syntax Errors   466

28  Macro String Preprocessor Error Messages
    Error Codes and Messages   494

29  Loader Error Messages
    Warning Messages       500
    Error Messages         506
    Fatal Error Messages   510

30  Librarian Error Messages
    Librarian Error Messages  514
Getting Started

A short example of the process of assembling, creating libraries, and linking several program modules.
Objectives of the Example Program

This list of topics covered by the example program is provided here to give you an idea of why the program is written the way it is. The example program is designed to show some of the basic features of the B1449 8086/186 Advanced Cross Assembler/Linker. Consequently, the example programs:

- Contain 8086 assembly language instructions written in a manner that makes use of relocatable program sections.
- Contain a few of the most used assembler directives.
- Contain an example of a simple macro definition.
- Contain an example of structured control statements.
- Show how the relocatable program sections are used with the Linking Loader (ld86).
- Show how to link two or more program modules.
- Show how to link object files from a library file.
The example programs in this chapter have been included with your Assembler/Linker/Librarian software and can be found in directory: /usr/hp64000/demo/languages/as86

Description of the Example Program

The example program moves data from three different memory locations to a fourth memory location. The program will be written in three modules to show how several program modules are linked together.

The **mov_mesg.s** program module is made up of a data table which contains the messages to be transferred, the main program which will define a macro and call "transfer" and "delay" subroutines, and a RAM location where the messages will be transferred.

The **transfer.s** program module contains the "transfer" subroutine which is called by the main program. The **transfer.s** subroutine will transfer a message from the data table to the destination memory location. The address of the message to be transferred will be passed in register SI, and the length of the message will be passed in register CX.

The **delay.s** program module contains the "delay" subroutine which is called by the main program. The **delay.s** subroutine will delay for the number of seconds which are passed in register CX.

The **delay.o** and the **transfer.o** relocatable object files will be placed into an example library file called **exlib.a**.

### The "mov_mesg.s" Program Module

The example program of this chapter will move three messages which are contained in a data table to another memory location. The three messages are labeled MESSAGE_1, MESSAGE_2, and MESSAGE_3. The ends of the messages are also labeled so that the program will know how many words of data to transfer. The destination memory location is labeled VIDEO_RAM.

The example program will (1) move the first message to VIDEO_RAM, where it will be displayed for about 10 seconds, (2) move the second message to
Chapter 1: Getting Started
Description of the Example Program

VIDEO_RAM, where it is displayed for about 7 seconds, and (3) move the third message to VIDEO_RAM, where it is displayed for about 4 seconds. At this point the program will loop back and display the second and third messages, one after the other, repeatedly. The mov_mesg.s source file is shown in Figure 1-1.

```
$XREF
NAME MOV_MESG
PUBLIC START, VIDEO_RAM
EXTRN TRANSFER:FAR
EXTRN DELAY:FAR

TABLE SEGMENT
MESSAGE_1 DB 'The example program moves ' DB 'this and two additional ' DB 'messages to a RAM location. '
MESSAGE_1_END LABEL BYTE
MESG_1_LENGTH EQU MESSAGE_1_END - MESSAGE_1

MESSAGE_2 DB 'The first message is' DB 'displayed for a medium ' DB 'length of time. '
MESSAGE_2_END LABEL BYTE
MESG_2_LENGTH EQU MESSAGE_2_END - MESSAGE_2

MESSAGE_3 DB 'The second message is ' DB 'displayed for a shorter ' DB 'length of time. '
MESSAGE_3_END LABEL BYTE
MESG_3_LENGTH EQU MESSAGE_3_END - MESSAGE_3

TABLE ENDS
```

Figure 1-1. The "mov_mesg.s" Source File
Figure 1-1. The 'mov_mesg.s' Source File (Cont’d)
PUBLIC Definitions.

The first thing the mov_mesg.s program module does is define the symbols which can be referenced by other program modules. These definitions are made with the PUBLIC assembler directive. The label VIDEO_RAM is defined as public because the transfer.s program module will reference the destination memory locations. The label START is defined as a public for program debugging convenience.

External Definitions.

The EXTRN assembler directive allows you to use labels or variables which are defined in other program modules. In the mov_mesg.s program module, the CALL TRANSFER and the CALL DELAY instructions use labels which are defined in the transfer.s and delay.s program modules, respectively. Therefore, TRANSFER and DELAY must be declared as external references.

The TABLE Program Segment.

The TABLE program segment contains the ASCII bytes of the three messages which are written to the destination memory location. The DB assembler directive is used to define the ASCII data. The lengths of the three messages are assigned to labels with the EQU assembler directive.

The M_CODE Program Segment.

The executable code of the mov_mesg.s program module is found in the M_CODE segment. After the user segment registers and stack pointer are loaded, the SET_UP macro is defined. The three parameters in the macro definition (ADDRESS, LENGTH, and COUNT) are assigned actual values in the macro calls. Each time the macro is called, assembly code is generated which calls the CLEAR, TRANSFER, and DELAY subroutines. (Parameters are moved into registers before the TRANSFER and DELAY calls.) After the macro is defined, it is called three times. The CLEAR subroutine, which moves ASCII spaces to the destination memory locations, appears at the end of the M_CODE program section.

The DATA Program Segment.

Storage locations are defined in the DATA program segment with the DW assembler directive. This storage location is the destination of the three messages and is labeled VIDEO_RAM.
The STACK Program Segment.

Storage locations are defined in the STACK program segment with the DB assembler directive. This storage location is used for data stack.

The "transfer.s" Program Module

The main program branches to the subroutine contained in the transfer.s program module. The "transfer" subroutine will move the data from the address passed in SI to the destination memory location VIDEO_RAM. Notice that the executable code in this module appears in a program segment named T_CODE. Also, notice the public definition of the label TRANSFER (which allows the main program to branch to this label) and the external reference of the variable VIDEO_RAM which was defined in the main program module. The transfer.s source file is shown in Figure 1-2.

``` assembly
$XREF
NAME TRANSFER
PUBLIC TRANSFER
EXTRN VIDEO_RAM: WORD
T_CODE SEGMENT
ASSUME CS:T_CODE
TRANSFER PROC FAR
MOV DI, OFFSET VIDEO_RAM; point to destination
REP MOVSW; move words until CX=0
RET
TRANSFER ENDP
T_CODE ENDS
END
```

Figure 1-2. The "transfer.s" Source File
The "delay.s" Program Module

The main program branches to the "delay" subroutine contained in the delay.s program module. The "delay" subroutine is used to display the various messages for the number of seconds passed in register CX. This program module's executable code is placed in a program segment named D_CODE. Notice the public definition of the DELAY label so that other program modules can refer to this subroutine. The delay.s source file is shown in Figure 1-3.

```
$XREF
NAME DELAY
PUBLIC DELAY
D_CODE SEGMENT
ASSUME CS:D_CODE
DELAY PROC FAR
    MOV AX, 553; load delay constant
    MUL CX; multiply twice by delay count
    MUL CX
    DLOOP: DEC AX; decrement value until 0
    JNZ DLOOP
    RET
DELAY ENDP
D_CODE ENDS
END
```

Figure 1-3. The "delay.s" Source File
Assembling the Program Module Source Files

Assembling program module source files will create object files. The commands to assemble the source files follow.

Starting the Macro Preprocessor

The macro preprocessor must be run for mov_msg.s before it is assembled because it contains macro definitions. The output of the macro processor—the file mov_msg.ap—is then used as input to the assembler. The command to start the macro preprocessor:

\$ ap86 mov_msg.s -s > mov_msg.ap

Starting the Assembler

The output of the macro preprocessor and the other two source files are assembled with the following commands:

\$ as86 -Lh mov_msg.ap > mov_msg.lis
\$ as86 -Lh transfer.s > transfer.lis
\$ as86 -Lh delay.s > delay.lis

The -L in the assembler commands above causes an assembler listing to be sent to the standard output. The -h option in the assembler commands above specifies that the assembler create HP 64000 assembler symbol files (with .A suffixes). The " > " in the commands above redirects the standard output to a file.

Viewing the Assembler Listing Files

You can view the assembler listings (files with the ".lis" extensions as specified above) with the more command. For example, to view the "mov_msg.lis" file, enter the command below.

\$ more mov_msg.lis

Assembler listings for each of the program modules are shown in Figures 1-4 through 1-6.
Chapter 1: Getting Started
Assembling the Program Module Source Files

Figure 1-4. The “mov_mesg.lis” Assembly Listing
Figure 1-4. The "mov_msg.lis" Assembly List (Cont'd)
Figure 1-4. The "mov_mesg.lis" Assembly List (Cont'd)
Chapter 1: Getting Started
Assembling the Program Module Source Files

Figure 1-5. The "transfer.lis" Assembly Listing
Chapter 1: Getting Started
Assembling the Program Module Source Files

Figure 1-6. The "delay.lis" Assembly Listing
Creating an Example Library File

One of the objectives of this chapter is to show how object modules can be linked from libraries. Before we can link from a library file, a library file must be created. To create an example library file consisting of the "transfer.o" and "delay.o" relocatable object modules, enter the following command:

```
$ ar86 -a delay.o,transfer.o -L exlib > exlib.lis
```

The -a option in the command above specifies that the files which follow are to be added to the library. The -L option in the command above specifies that a library listing file be sent to the standard output (which is redirected to the "exlib.lis" file). The library listing file is shown in Figure 1-7.

Notice the warning message. Warning messages announce something that might be a problem. Since you are creating a new library file, you already know that "exlib.a" does not yet exist, so you can ignore the warning. The warning in the first line of the listing appears on the display, not in the listing file.
Chapter 1: Getting Started
Creating an Example Library File

WARNING: (107) file exlib.a does not exist.
Hewlett-Packard ar86
HPB1449-19302 A.03.10 24Mar93 Copr. HP 1988
OPEN exlib.a
WARNING: (107) file exlib.a does not exist.
ADDMOD transfer.o
ADDMOD delay.o
LIST exlib.a
Hewlett-Packard ar86 Mon Mar 29 08:36:11 1993
HPB1449-19302 A.03.10 24Mar93 Copr. HP 1988

Library being built exlib.a

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSFER ...</td>
<td>290</td>
</tr>
</tbody>
</table>

***** PUBLIC DEFINITIONS *****

TRANSFER

***** EXTERNAL REFERENCES *****

VIDEO_RAM
Public Count = 1
External Count = 1

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY ...</td>
<td>280</td>
</tr>
</tbody>
</table>

***** PUBLIC DEFINITIONS *****

DELAY

***** EXTERNAL REFERENCES *****

Public Count = 1
External Count = 0

Module Total = 2
SAVE
END
Warnings = 1
Errors = 0

Figure 1-7. The "exlib.lis" Library Listing
Linking is the process in which program modules are joined together to form a single absolute file which can then be executed or debugged. Because you can link several object modules to form an executable file, the Linking Loader is sometimes called the "Linker". Also, because you can specify the load addresses of various program sections, the Linking Loader will sometimes be referred to as the "Loader". Either name is correct; the ld86 tool does both.

There are two ways that you can specify object files to be linked:
• enter the names of the files on the command line
• specify the names of the object files in a linker command file

The linker command file shown in Figure 1-8 will be used to link the three object modules in the example program.

```
* Demo 8086 loader command file
NAME DEMO
ORDER M_CODE, T_CODE, D_CODE
SEG TABLE=1000H
SEG M_CODE=1400H
SEG DATA=1800H
LOAD mov_mesg.o, exlib.a
END
```

**Figure 1-8. The "demo.k" Linker Command File**
Chapter 1: Getting Started
Linking the Program Module Relocatable Object Files

Linking the Object Modules

The command to link the example program object modules is shown below. The -c option specifies that a linker command file will be supplying information to the Linking Loader.

\$ ld86 -c demo.k -Lh > demo.lis

The -L option in the command above specifies that an output load map listing file be sent to the standard output (which is redirected to the "demo.lis" file). The -h option specifies that the linker create HP 64000 format output files (demo.X is the absolute file and demo.L is the linker symbol file). The load map listing file is shown in Figure (1-9). The resulting executable (demo.X), along with the linker symbol file (demo.L) and assembler symbol files (mov_mesg.A, transfer.A, and delay.A), may be loaded into an emulator or downloaded into ROM for execution.

* Demo 8086 loader command file
NAME DEMO
ORDER M_CODE, T_CODE, D_CODE
SEG TABLE=1000H
SEG M_CODE=1400H
SEG DATA=1800H
LOAD mov_mesg.o, exlib.a
END

OUTPUT MODULE NAME: DEMO
OUTPUT MODULE FORMAT: HP64000 absolute

START ADDRESS: 00140:00000 -> 01400

Figure 1-9. The "demo.lis" Load Map Listing
Figure 1-9. The "demo.lis" Load Map Listing (Cont'd)

This completes the "Getting Started" example. For a complete description of the as86, ap86, ld86, and ar86 commands and their options, refer to the "Assembler/Macro Preprocessor/Linker/Librarian Command Syntax" chapter that follows.
Chapter 1: Getting Started
Linking the Program Module Relocatable Object Files
Command Syntax

Syntax for the assembler, macro preprocessor, linker, and librarian.
Options may be entered on the command line to control generation of the output listing and object module, and to turn internal assembler flags on and off.

The information on the following syntax pages can be accessed on your workstation via the man command. For example, to view the as86 on-line manual page, just type in the following command:

```
$ man as86
```

If this command doesn’t work ("No manual entry"), check that the MANPATH environment variable includes the path $HP64000/man.
as86 - cross assembler for the Intel 8086/186 microprocessors

Synopsis
/usr/hp64000/bin/as86 [ options ] file

Description
as86 assembles the named file.

as86 attempts to open the file named on the command line. If this fails and
the file does not have a suffix (does not contain a period), as86 appends a .s to
the file name and attempts to open that file.

The output is a relocatable file containing Intel 8086/186 instructions and
symbolic data. The format of the output file is HP's extension of the INTEL
8086 OMF relocatable file format.

An HP 64000 format assembler symbol file is also produced when the -h
option is used. The asmb_sym file name will have a .A suffix added to the
source file name.

The name of the object file may be specified with the -o option. If it is omitted,
the output file name is created by stripping off the ending suffix from the input
file name and appending .o in place of the suffix. Any full path prefix is also
stripped from the beginning of the input file name. The output files are placed
in the local directory, unless the file named with an -o option specifies a
different path.

The -L option may be used to obtain an assembler listing on standard output.
Standard output may be redirected into a listing file. This listing contains
offsets, instruction codes, symbol table information, symbol table cross
reference, and other useful information.

Options
The following command line options are recognized by as86:

-f flaglist
The flags in flaglist are used to select and change the internal assembler
control switches.
The flags recognized and their meanings are defined below. Each flag may be set (or unset) either on the command line using the -f option as described here or by entering the option in the assembler source program.

Groups of flags following the -f option must be separated by commas or separated by white space and quoted. Any option that contains white space must be quoted. For example, the following sets the flags debug, ty, title (my title), and xref:

```
-f debug,ty -f "title (my title) xref"
```

A flag may be unset (turned off) by preceding the flag value with no. A negated flag may not have a parameter. The following flags may not be negated: include, pagelength, pagewidth, title, date, workfile, and optimize. For example, the following turns off the debug and object flags:

```
-f nodebug,noobject
```

-**H asmb_sym_file** specifies a file name to override the default file name for the HP 64000 format assembler symbol file. (See the -h option below.)

If asmb_sym_file has a suffix, then the name is used as is. Otherwise, a .A is appended to form asmb_sym_file.A.

-**h** specifies that an HP 64000 format assembler symbol file should be produced. The assembler symbol file name will have a .A suffix added to the source file name. The source file name will have all preceding directories and the trailing suffix stripped off before the .A is added. If the assembler symbol file is to be used in an HP 64000 station, recall that file names in the HP 64000 are restricted to nine characters in length and must begin with an upper case letter. The default asmb_sym file name may be overridden with the -H option.

When writing the asmb_sym file, all identifiers in the source program are converted to legal HP 64000 identifiers. Illegal characters within identifiers such as ‘?’ or ‘@’ will be converted to an ‘_’. Identifier names longer than 15 characters will be truncated to 15 characters. No attempt is made to search for duplicate symbols created by the truncation.

-**L** specifies that an assembler listing file be written to standard output.

-**o objfile** specifies the name of the output file. This overrides the default file name for the HP-OMF 86 format relocatable file produced.
The following flags may be specified using the -f flaglist option. Flags may be specified as either upper or lower case. All flags have a two-letter abbreviation that may be used. Flags on the command line are set from left to right, so the rightmost setting for a particular flag will be used. Some flags may be used anywhere within a source file, which means the value of a flag might be changed later in the source. These flags are also called general controls. Other flags may only be used on either the command line or the first lines of the source file. These flags are also called primary controls. A primary control used on the command line will override a primary control used in the source file. The last occurrence of a general control will be the one which takes effect. This means that any occurrence of a general control in the source file overrides the general control setting in the command line beginning at the point in the source file where the general control occurs to either the end of the assembly source file or until another duplicate control is found.

**Case Sensitivity Flags**

- **case**
  - Causes symbols to be case sensitive. That is, upper and lower case characters will be assumed different. The option nocase means that upper and lower case characters in symbols are treated as upper case. Note that INTEL-generated OMF is case insensitive (all upper case). This option, or its negated form, may also be entered on the first assembly source lines. (Default: case)

- **debug**
  - Causes debug and type information to be stored in the resulting relocatable file. This option, or its negated form, may also be entered on the first assembly source line. (Default: debug)

- **eject**
  - Causes a page eject to occur and a new page heading to be printed. This option is only useful if a listing is being generated and paging is in effect. This option may be used anywhere within the assembly source.

- **errorprint**
  - Causes error and warning information to be displayed on standard error. If a filename is used with the errorprint control, the filename is ignored. The noerrorprint control suppresses error and warning messages from being displayed on standard error. The nowarning control may be used to suppress warning messages while allowing error messages to be displayed. (Default: errorprint)

- **extern_check**
  - Causes use of external symbols to be checked such that an assume register has been defined that can reference the external symbol. An error is generated if this condition cannot be met. The noextern_check control causes the assembler to allow any use of an external symbol without verifying that the symbol is accessible through an assume register. (Default: extern_check)
**Chapter 2: Command Syntax**

as86(1)

**gen**

Supplied for Intel compatibility. The assembler does no macro processing. This is done by the macro preprocessor, ap86(1). Therefore, this control has no effect.

**go**

Supplied for Intel compatibility. The assembler does no macro processing. This is done by the macro preprocessor, ap86(1). Therefore, this control has no effect.

**group_info**

Causes the debug information emitted from the assembler to associate group information to all symbols that belong to segments belonging to a group. Only one group will be assigned, regardless of how many groups a given segment belongs to. The nogroup_info control will only associate group information to labels and procedures; variables will NOT have group information associated with them. (Default: group_info)

**hlassym**

Causes as86 to generate low-level symbol information for static procedures, static data, and embedded code. This option is useful when compiler-generated output is to be debugged in an emulator. If the output is to be debugged in AxDB or AxDE, then the negated form of this option is recommended. (Default: nohlassym)

**include('filename')**

Causes the indicated file to be included into the assembly code before any other source is assembled. This option may be used anywhere within the assembly source.

**list**

Causes assembly source to be displayed in the listing while it is being assembled. The nolist option turns off the listing function until the next list option. This option is only useful if a listing is produced. This option, or its negated form, may be used anywhere in an assembly source file. (Default: list)

**mod086**

Causes iAPX86 instruction set to be recognized. Errors or warnings will be issued when instructions from conflicting instruction sets are encountered. (Default: mod086)

**mod186**

Causes iAPX186 instruction set to be recognized. Therefore, BOUND, ENTER, INS, INSB, LEAVE, OUTS, OUTSB, OUTSW, POPA and PUSHA are predefined symbols. The iAPX86 instructions will still be recognized.

**modv20**

Causes v20/v30 support.
Note
Except for the specific instructions that are V20/V30 extensions, as86 uses Intel mnemonics. as86 uses Intel syntax for all instructions.

object
"(['filename')]"
oj
Causes an object (or relocatable) file to be created. The created file will have the same name as the input file, only with a .o extension, unless the -o flag was used. In that case, the object file will have the filename specified with that flag. If a filename is specified with the object control, that filename is ignored. The noobject option causes no relocatable file to be created. This option, or its negated form, may also be entered on the first assembly source lines. (Default: object)

optimize
"op"
Causes extra processing of the input file to remove extraneous NOPs. These NOPs are generated when the assembler encounters certain forward references in instructions. In those cases, the assembler does not know how many bytes the instruction will require, so it allocates the maximum number of bytes needed. If the instruction requires fewer bytes, then the assembler must pad the object code with NOPs, so the addresses of following symbols remains consistent. The optimize option causes the assembler to process the input file to remove these extra NOP bytes, so as to generate the smallest amount of object code possible.

pagelength
"(['NUMBER')]"
pl "(['NUMBER')]"
Sets the maximum number of lines per listing page. This option is only useful if a listing is produced and paging is enabled. This option may be entered on the first assembly source lines. (Default: pagelength(55))

pagewidth
"(['NUMBER')]"
pw "(['NUMBER')]"
Sets the maximum width of each line in the listing. This option is only useful if a listing is produced. This option may be entered on the first assembly source lines. (Default: pagewidth(120))

paging
"pi"
Causes page ejects to occur whenever the maximum number of lines per listing page is generated. The nopaging option causes no page ejects to occur throughout the listing, due to page lengths. This option is only useful if a listing is produced. This option, or its negated form, may be entered on the first assembly source lines. (Default: paging)

print
"(['filename'])"
pr "(['filename'])"
Prints the assembly listing. The noprint control suppresses the generation of the listing file including error messages and symbol table listings. You cannot override noprint with a list control occurring later in the program; however, a list control with no preceding print or noprint implies print. The filename parameter is accepted for Intel compatibility, but it is ignored by
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>restore rs</td>
<td>Restores a previously saved state for the list/nolist flag. This option is only useful if a listing is being generated. It may appear anywhere within a source file.</td>
</tr>
<tr>
<td>save sa</td>
<td>Saves the current status of the list/nolist flag. These settings may then be restored later by using the restore flag. Up to 64 saves may be made. This option is only useful if a listing is being generated. It may also appear anywhere within a source file.</td>
</tr>
<tr>
<td>symbols sb</td>
<td>Causes an alphabetically-sorted list of symbols to be appended to the listing. This option differs from the xref option in that no cross-reference information is placed in this list. Using the xref option overrides either symbols or nonsymbols. This option is only useful if a listing is produced. This option, or its negated form, may also be entered on the first assembly source line. (Default: symbols)</td>
</tr>
<tr>
<td>title(TEXT)</td>
<td>Causes the TEXT to become the new title, which is printed at the top of each listing page. This option is only useful if a listing is produced. This option may be used anywhere in an assembly source file. (Default: title (MODULE NAME))</td>
</tr>
<tr>
<td>warning wa</td>
<td>Causes warning messages to be displayed on standard error. The negated form suppresses warning messages from being sent to standard error. The errorprint control overrides either use of this control. (Default: warning)</td>
</tr>
<tr>
<td>xref xr</td>
<td>Produces a symbol table in the listing with source line definition and usage cross referencing. This option is only useful if a listing is produced. This option, or its negated form, may be entered on the first assembly source lines. (Default: noxref)</td>
</tr>
</tbody>
</table>

**Files**

- `file.o`: HP-OMF 86 format relocatable object file.

See Also

HP B1449 8086/186 Assembler/Linker/Librarian User’s Guide, ld86(1), ar86(1), ap86(1).

Diagnostics

as86 returns zero if no errors are detected in the assembly source. Otherwise, it returns non-zero.

Diagnostic messages including optional lines containing assembly errors are displayed on standard error.
**Name**
ap86 - macro preprocessor for the Intel 8086/186 microprocessors

**Synopsis**
/usr/hp64000/bin/ap86 file [-i includepath] [-s] [-e] [-c]

**Description**
ap86 reads the named file and performs macro preprocessor replacements or operations upon this file. The resulting text may be sent to standard output, for redirection to another file for storage. Error messages for macro operations may also be sent to standard error.

ap86 accepts the macro preprocessor language that is described in the Intel 8086 Assembler Reference Manual. This macro language allows the definition and use of macros, evaluation and replacement of expressions, loop control, and including of other text files. Correct use of a macro preprocessor can simplify the task of writing assembly language source when redundant operations are performed or code is shared between files.

**Options**
The following command line options are recognized by ap86:

- `-i includepath`
  This option causes the macro preprocessor to search that directory for any include files that are referenced in the source file. If this option is not used, the current directory is searched. The search only occurs for file names that use relative paths for the include file. If the path names start with a `'/`, then no search is required.

- `-s`
  This option causes the macro preprocessor to send the processed file to standard output. If this option is not used, the input file is processed, but no text output is generated. If this option is used, the output should usually be redirected to a file for use as input to the as86 assembler.

- `-e`
  This option causes the macro preprocessor to display lines that contain errors and the error message that was generated by processing that line. This text is sent to standard error, so as not to interfere with the `-s` option. This option is useful since the text generated on standard output does not always display erroneous text in the most identifiable manner.

- `-c`
  This option causes the macro preprocessor to be case insensitive. The default is case sensitive. For example, if ap86 is started in default mode, then
%SET(var1,-1) and %SET(VAR1,-2) cause substitutions for var1 of -1 and for
VAR1 of -2. But if ap86 is started with the -c (case insensitive) option, then
the last %SET will cause substitutions for both var1 and VAR1. Note that
predefined macro functions written in lower case letters, such as %set and
%define, are currently only recognized when the -c option is used. However,
predefined macro functions written in upper case letters are always
recognized.

Files

file.s  Assembly language source file.

See Also

HP B1449 8086/186 Assembler/Linker/Librarian User’s Guide, ld86(1),
ar86(1), as86(1).

Diagnostics

ap86 returns zero if no errors are detected in the macro source. Otherwise, it
returns non-zero.

Diagnostic messages including optional lines containing assembly errors are
displayed on standard error.
ld86(1)

Name
ld86 - cross linker/loader for Intel 8086/186 microprocessors

Synopsis
/usr/hp64000/bin/ld86 [ options ] [ files ]

Description
ld86 takes one or more relocatable object files as input and combines them to produce a single output file. In doing so, it resolves references to external symbols, assigns final addresses to procedures and variables, revises code and data to reflect new addresses, and updates symbolic debug information (when it is present in a file). ld86 accepts relocatables in Intel 8086 OMF format and HP-OMF 86 format, as well as archive input files in ar86 format. These files may be produced by a cross assembler (as86), or the archive file librarian (ar86). While ar86 libraries are not in strict OMF format, ar86 can read in existing Intel OMF format library files.

By default, the output is HP-OMF 86 absolute. HP-OMF 86 format is HP’s implementation of 8086 binary OMF. This file contains Intel 8086/186 instructions and symbolic data. Options to ld86 may be used to create output files in HP 64000 format absolute or the Intel Hexadecimal Object file format absolute.

Typically, the output file contains instructions and data in absolute form. That is, address information has been supplied to locate the program in target memory.

The -i option may be used to specify a relocatable output file in a process called incremental linking. In an incremental link, the input relocatable files are simply combined into an output relocatable file. Incremental linking is only supported for HP-OMF 86 format relocatable. Therefore, the -i option cannot be used with either the -h option or the -H link_symb_file option.

The operation of ld86 is controlled by LINKER COMMANDS (described below). Linker commands specify the input relocatable and archive files, the location and order of relocatable sections, and the content and format of the output files.
ld86 reads commands from a command_file or from standard input if no input filenames were specified on the command line. A command_file is specified using either the -c user_cmd or -d (default cmd_file) options. If a command_file is not specified (using either the -c or -d options), then ld86 reads standard input. If standard input is a tty, ld86 enters interactive mode and prompts for commands.

Input files may be specified in "LOAD" commands or on the command line. The order of specification of the input files is significant to the operation of the linker. If input files are specified on the command line, these files are loaded before files specified in "LOAD" commands in the command_file.

If the input file names have a suffix, then the name is used as is. Otherwise, ld86 appends .o to the name on the command line to form an input file name.

The basic name of the output file is determined in the following way. The default basic file name is the command_file stripped of any preceding pathname (up to and including the last '/') and stripped of any suffix (including '.'). The default file name may be overridden by specifying the output file name with the -o option. If no command_file is used, it is an error not to specify the output file name with the -o option.

Depending on the format and type of output file, a suffix is appended to the basic file name to form the output file name.

If the output is HP-OMF 86 format absolute, then the suffix is .x.
If the output is HP 64000 format absolute, then the suffixes are .X for the absolute file and .L for the linker symbol file.
If performing an incremental link, then the output is in HP-OMF 86 format relocatable and the suffix is .o.

**Options**

The following command line options are recognized by ld86:

- **-c command_file**
  The option specifies the name of the command_file to be used to supply information to ld86. The file name part of the command file path, with suffix stripped, is used to form the default names of output files unless the name is specified with the -o option.

- **-d**
  Use the default linker command file. ld86 examines the environment variable LD86CMD to find the name of the linker command file. If LD86CMD
doesn't exist or is all blank, then the loader attempts to open
/usr/hp64000/etc/ld86cmd as the loader command file.

-f flaglist

The flags in flaglist are used to select and change the internal linker control
switches.

The flags recognized and their meanings are defined below. A more complete
explanation may be found in the HP B1449 8086/186
Assembler/Linker/Librarian User’s Guide. Each flag may be set (or reset) in
either of two ways. A flag may be set on the command line using the -f option
described here. A flag may also be set using the LIST linker command and
reset using the NLIST linker command.

Groups of flags following the -f option must either be separated by commas or
separated by white space and quoted. For example, the following option sets
the flags c, d, s, and x:

-f c,d -f "s x"

A flag may be reset (turned off) by preceding the flag with no. For example,
the following option turns off the o and p flags:

-f noo -f nop

-H link_sym_file

This option overrides the default file name for the HP 64000 format linker
symbol file (.L file) and absolute file (.X file). (See the -h option below.)

-h

The option indicates that the linker should produce HP 64000 format output
files. There are two output files, the absolute file and the linker symbol file.
The default name for the absolute file is command_file.X and the default
name for the linker symbol file is command_file.L. The -fd flag is implied
through the use of the -H or -h flags.

It is the user’s responsibility to assure that all identifiers (i.e. global symbol
definitions and external symbol references) are converted to legal HP 64000
identifiers before being used. For example, Intel assembly language identifiers
may contain the characters _ (underscore), ? (question mark), and @ (at
sign) and have a maximum of 31 significant characters. To produce legal HP
64000 identifiers, all question marks, and at signs could be converted to _
(underline). Identifiers will also need to be truncated to 15 characters
maximum.
Note: Conversion to HP 64000 symbols may have unexpected side effects. Duplicate symbol errors may occur. ld86 does perform name translations, but will NOT warn if duplicate symbols have been created. File names must also not exceed 9 characters for the HP 64000 and must begin with an upper case letter.

-i specifies that an incremental link is to be performed. The relocatable input files are combined to produce a relocatable output file. Any linker commands which specify location (e.g. ORDER, GROUP) cause a linker error. The name of the relocatable output file defaults to command_file.o. Incremental linking is only supported for the HP-OMF 86 format. An error will be issued if the -i option is used with either the -h option or the -H option.

-L specifies that an output load map listing be written to standard output.

-o objfile specifies the name of the output file. This overrides the default file name for the HP-OMF 86 format absolute file, the HP-OMF 86 format relocatable file, and the HP 64000 format absolute file.

Linker Commands: The linker/loader recognizes the following commands in command files or in interactive mode. Square brackets, [ ], enclose optional parameters. Ellipsis or ‘...’ indicate the preceding item may be repeated.

* comment text... designates a comment.

SEG SEG segment= address
SEG segment= paragraph,offset
SEG /class= address
SEG /class= paragraph,offset

The SEG command specifies the base address of a user’s logical segment (LSEG). The user may also wish to use the ORDER command to control the placement of segments which were not specified in the SEG command.

The 'segment' portion is the name of a relocatable segment which may have a classname attached with a slash, such as 'SEGNAME/CLASSNAME'. If a segment has an associated classname, this classname must be specified or ld86 will not find the correct segment.
The 'class' is the name of a class. A classname preceded by a slash may appear in place of the segment name, whereupon the first segment whose class attribute is 'classname' will be assigned the base address.

The 'address' is the 20 bit address specifying where the segment will begin. The range for 'address' is 0 through 0FFFFFH. A segment register pointing to the segment should have the value of 'address/16'.

The 'paragraph,offset' base address will be '16*paragraph+offset' where paragraph or offset may be 0 through 0FFFFH.

**Note**
Addresses are not rounded to conform with an alignment attribute from an ALIGN command.

**SEGSIZE**

SEGSIZE segment= length

SEGSIZE /class= length

The SEGSIZE command specifies the length of a segment in bytes. Although SEGSIZE can be used to set the length of any segment, SEGSIZE is typically used to set the size of a stack segment. A warning message is issued if the segment does not have a combine type of STACK or COMMON.

**GROUP**

GROUP group= address

GROUP group= paragraph,offset

GROUP specifies the absolute base address of a group, which must be a multiple of 16 because it always lies on a paragraph boundary. The GROUP command does not specify the base address of any segments within the group. All such segments should lie within 'group address' through 'group address' plus 0FFFFH.

The 'group' is the name of the group.

The 'address' is the 20 bit address specifying where the group will begin. The range for 'address' is 0 through 0FFFFFH. A segment register pointing to the group should have the value of 'address/16'.

The 'paragraph,offset' base address will be '16*paragraph+offset' where paragraph or offset may be 0 through 0FFFFH.

**ALIGN**

ALIGN segment= [blank,B,P,I,G,W]
The 'align' command may be used to override the alignment type of an input module without reassembling. A named segment can be specified as Byte, Page, Inpage, Paragraph, or Word relocatable regardless of the type specified by the assembler. This could be used to place all segments on page boundaries while debugging and then to create the final program as byte relocatable without reassembling.

The 'segment' is the name of a relocatable segment. The segment name may have a classname following it, separated by a slash. If a segment has an associated classname, this classname must be specified or ld86 will not find the correct segment.

The 'blank' specifies that the alignment type is to be what the assembler specified.

The 'B' specifies byte alignment.

The 'P' specifies page alignment.

The 'I' specifies inpage alignment.

The 'G' specifies paragraph alignment.

The 'W' specifies word alignment.

**Note**

Do not put blanks between the '=' and the alignment designator because a blank is a legal alignment designator.

**FORMAT**

FORMAT [ASCII,HP,OMF86,NOABS] [INCREMENTAL,LIMITED,LTL]

This command instructs the linker as to what format should be used in the created output file. If the NOABS option is specified, then no object file is generated. If ASCII is specified, then the output file will be in Intel Hexadecimal format. If HP is specified, then the output will be in HP64000 file format. If OMF86 is specified, then the output will be in HP’s implementation of Intel OMF file format.

Different forms of OMF can be generated through three modifiers. If no modifier is used, then the output is an absolute file. If the INCREMENTAL modifier is used, then the output is a relocatable object file. If LIMITED is specified, then the output will be an absolute file, but all non-commented records will conform strictly to the Intel absolute file
format document. Finally, the LTL modifier will cause a load-time loadable file to be generated. These options can also be specified through the use of command line options or through the use of the LIST command.

The LIMITED and LTL modifiers are only usable with an output file format type of OMF86.

INITDATA

INITDATA segment [segment [, ...]] [, address]

Segment is the name of a relocatable segment expressed as either 'segmentname', or 'segmentname/classname', or '/classname'.

Address indicates the beginning address of the ??DATAn/??INIT segment and may be an absolute address from 0 through 0FFFFFFH or the address may be in paragraph, offset from where paragraph and offset are from 0 through 0FFFFFFH. Leading zeros are required on hexadecimal addresses in the initdata command when the hex value begins with A-F.

The INITDATA command is used to specify that the indicated segments and classes will be initialized in memory at run time.

ORDER

ORDER element[, ...]

The first segment specified in the ORDER command will begin at address 0 and subsequent ones immediately after the preceding one. Addresses will not be assigned which conflict with absolute segments, areas specified in RESNUM or RESADD commands, or segments specified in a SEG command. The ORDER command will not override the base address of an absolute segment or one assigned with SEG. However, segments which appear in the ORDER command following one of these segments will be assigned space in memory above it.

The 'element' may be any of the following: a segment name; a classname preceded by a slash; a segment name followed by a slash and a classname; a classname followed by one or more segment names separated by hyphens (as in CLASSNAME-SEG1-SEG2-...-SEGn). A classname preceded by a slash specifies all segment names with that class attribute in the order that the loader finds them. The classname-segmentname-segmentname... element would cause the loader to move the specified segments to the beginning of the class. Any remaining unspecified segments would then immediately follow.
If the first segment in a class has been assigned a base address with the SEG command and an ORDER command has also been used, then the classname should also be placed in the ORDER command so that segments in the class will be assigned adjacent memory.

A segment name may not appear more than once in an ORDER command. This includes both the explicit case of SEGMENT/CLASSNAME and the implicit case of CLASSNAME. The same classname may not appear more than once following a comma, but it may appear in a SEGMENTNAME/CLASSNAME combination as often as needed.

If an ampersand is encountered while the loader is expecting either a comma or a hyphen, then the next line will be considered a continuation line. Only the last ORDER command is effective. A warning noting that only the last ORDER command is effective will be issued if more than one ORDER command is used.

**START**

START CS-value,IP-value  
START address

START specifies the starting values for CS and IP, otherwise they will be taken from the END record of the first main program. If no main program is present, they would be zero.

CS-value and IP-value must be between 0 and 0FFFFH. The address value must be between 0 and 0FFFFFFH.

**LOAD**

LOAD (-)module1[(-)module2,...(-)moduleN]

LOAD specifies input object modules to be loaded. Multiple LOAD commands are allowed.

The 'module' is a relocatable object file name or a library file name. Any 'library file' preceded by a minus sign will cause all object modules within the library to be read until an EOF is encountered.

Libraries not preceded by a minus sign will load only those modules needed to resolve undefined EXTRNs.

A library should be loaded after all other non-libraries or else EXTRNs to a library from a subsequently loaded file may not be resolved correctly. Backward EXTRNs within a library are resolved correctly.
END  The END command causes the load to be finished and an output module produced. This command should be included as the last command in a command stream.

[NO]ERROR  [NO]ERROR [ UNREF,UNRES,OVERLAP,number ] [, ...]

The ERROR and NOERROR commands specify that the message or message number indicated is to be treated as an error or a non-error. The undefined external reference message is denoted by the UNREF argument. The unreferenced external message is denoted by the UNRES argument. The memory overlap message is denoted by the OVERLAP argument. If a number is given as an argument then it must corresponds to a particular error or warning number of the linker. These commands have a global effect from the point at which the linker processes the information contained in the command. A subsequent ERROR or NOERROR command overrides any values set by a previous one.

EXIT  The EXIT command causes the linker to exit without finishing the load and without producing an output module. An error message is issued to remind the user that ld86 was terminated early.

PUBLIC  PUBLIC sym1= value1[,sym2= value2]...

The PUBLIC command may be used to define and/or change the value of a public definition.

The 'symN' specifies a user defined public symbol definition which is considered absolute instead of relative to a segment or a group.

The 'value' is the 20 bit value to be assigned to the symbol.

LIST  LIST/NLIST {flag [,flag] ...}

LIST sets linker flags. NLIST is the opposite of LIST and suppresses the listing of the elements specified. The flags may also be set on the command line and are defined below.

"a" creates an Intel Hexadecimal Object format absolute output file.

"b" creates an HP-OMF 86 format absolute output file.

"c" prints the identifier cross reference table in the load map. (Default: noc)

"d" causes public symbols to be put into the output object module. (Default: d)
"e" causes warning messages to be generated for any remaining undefined external symbols during an incremental link. (Default: noe)

"i" causes incremental linking to occur resulting in a relocatable object file. (Default: noi)

"l" causes warning messages to be printed for any unreferenced, unresolved external references. (Default: nol)

"o" specifies that an object module is produced. (Default: o)

"p" specifies that any symbols present in the input modules (local) be placed in the loader symbol table. Its purpose is to exclude symbols from certain input modules from the output module. One does this by surrounding LOAD commands with NLIST P and LIST P commands. (Default: p)

"q" causes ld86 to produce a 'limited' form of Intel binary OMF which is strictly compatible with Intel's binary OMF document. (Default: noq)

"s" specifies that the local symbols be written into the object module and may be used for debugging. (Default: s)

"t" specifies that the local symbol table be listed in the load map. (Default: not)

"u" disables case sensitivity for matching public symbols, external symbols, segment names, group names, and class names. This also causes all symbols (except module names) to be converted to upper case. (Default: nou)

"v" causes an expanded segment summary that lists the modules where the segment parts came from. (Default: nov)

"w" causes ld86 to display all warning messages. (Default: w)

"x" causes symbols defined in PUBLIC commands to appear in the load map. (Default: nox)

**LISTABS**

```bash
LISTABS [ [NO]INTERNALS,[NO]PUBLICS ] [, ...]
```

The LISTABS command controls the output of certain items to the output object module. Multiple LISTABS commands can be specified and have an accumulative effect.

"INTERNALS" causes local symbols to be written to the output file. This is equivalent to the LIST S command. (default: INTERNALS)
'PUBLICS' causes globally defined symbols to be written to the output file. This is equivalent to the LIST D command. (default: PUBLICS)

The LISTABS command will eventually replace the LIST/NLIST D and LIST/NLIST S commands.

LISTMAP option [, option] ...

The LISTMAP command controls the output of certain items to the linker’s map or listing file. The LISTMAP command options have a global effect. Multiple LISTMAP commands that do not have any inconsistencies with previous LISTMAP commands can be specified and have an accumulative effect.

The valid values for option are as follows:

"[NO]CROSSREF" controls whether or not a cross-reference will appear in the linker listing file. (default: NOCROSSREF)

"[NO]INTERNALS [/BY_NAME,/NAME]" controls the listing of the non-public symbol table to the listing file. If /BY_NAME or /NAME is specified, the symbol table will be sorted by symbol name. (default: NOINTERNALS)

"LENGTH number" controls the page length of the linker listing file. The argument, number, must be between 5 and 255. (default: LENGTH 55)

"[NO]MODULE" controls the output of the module summary to the linker listing file. (default: MODULE)

"[NO]PUBLICS [/BY_ADDR,/ADDR,/BY_NAME,/NAME]" controls the listing of the public symbol table to the listing file. If /BY_NAME, /NAME, or nothing is specified, the symbol table will be sorted by symbol name. If /BY_ADDR, or /ADDR is specified, the symbol table will be sorted by address values. (default: NOPUBLICS)

"[NO]SEGMENT" controls the output of the segment summary to the linker listing file. (default: SEGMENT)

"[NO]VERBOSE" indicates whether or not additional information is to be included in the segment summary portion of the linker listing file. If the LISTMAP NOSEGEMENT option has been selected then the setting for VERBOSE is irrelevant. (default: NOVERBOSE)
"[NO]WARNINGS" controls the output of warning messages to the linker listing file. (default: WARNINGS)

"WIDTH number" controls the page width of the linker listing file. The argument, number, must be between 20 and 255. (default: WIDTH 80)

The LISTMAP command will eventually replace some of the functionality of the LIST command.

NAME name

NAME allows the user to specify the module name in the module header record of the output file.

RESADD

RESADD lowaddress,highaddress
RESNUM lowaddress,number

RESADD and RESNUM allow the user to declare certain areas of memory off limits to the loader.

The 'lowaddress' is a 20 bit address which is the lowest address of the memory that may not be used.

The 'highaddress' is a 20 bit address which is the highest address of the memory that may not be used.

The 'number' is a 20 bit value indicating the number of bytes (including lowaddress) that may not be used.

TYPEMERGE

TYPEMERGE [ ALL | SIMPLE ]
NOTYPEMERGE

The TYPEMERGE command removes redundant type information from the resulting executable. Normally, no type information is removed by the linker. However, since the HP-OMF 86 file format can only store up to 32k type definitions, it may be necessary to remove some redundant types for larger executables.

The ALL option causes all redundant types to be removed, while the SIMPLE option causes only redundant simple types to be removed.

The NOTYPEMERGE command prevents the linker from removing any redundant types. NOTYPEMERGE is the default operating mode for the linker.

WARN

WARN [ UNREF,UNRES,OVERLAP,number ] [, ...]
The WARN command specifies that the message or message number indicated is to be treated as a warning. The undefined external reference message is denoted by the UNREF argument. The unreferenced external message is denoted by the UNRES argument. The memory overlap message is denoted by the OVERLAP argument. If a number is given as an argument then it corresponds to a particular error or warning number of the linker.

These commands have a global effect from the point at which the linker processes the information contained in the command. A subsequent WARN command overrides any values set by a previous one.

Files

/usr/hp64000/etc/ld86cmd: Default 8086/80186 linker command file
file.x: HP-OMF 86 format absolute object file
file.X: HP 64000 format absolute file
file.L: HP 64000 format linker symbol file
file.o: HP-OMF 86 format relocatable object file from incremental link

See Also

HP B1449 8086/186 Assembler/Linker/Librarian User’s Guide, ar86(1), as86(1), ap86(1).

Diagnostics

ld86 returns zero if no errors are detected while linking, otherwise it returns non-zero. Diagnostic messages are displayed on standard error.

Bugs

Using the -h or -H options will cause global and external identifiers to be converted to legal HP 64000 identifiers. Conversion can cause duplicate symbols to be created.
ar86(1)

Name

ar86 - archive/library maintainer for Intel 8086/186 microprocessors

Synopsis

/usr/hp64000/bin/ar86
/usr/hp64000/bin/ar86 [options][action] ... archivefile

Description

ar86 maintains groups of relocatable files combined into a single archive file. The archive files may then be used by ld86, the 8086/186 linker/loader, to form executable programs for the Intel 8086/186 processors.

Individual relocatable files are inserted without change into the archive file. In addition, there is a library symbol table which is used by the linker/loader, ld86, to effect multiple passes over the library in an efficient manner.

Individual relocatable files define modules which have modulenames. The modulename is usually the same as the name of the assembly source file (with preceding pathname and suffix stripped). However, the assembler, as86, could change the modulename if a NAME directive is used. The modulename is used to identify the various modules that may exist within an archive file.

ar86 operates in either of two modes. The mode is determined by the presence (or absence) of the archivefile name.

In the first mode,

ar86

an archivefile is not specified. ar86 reads librarian commands from standard input. If the standard input is a terminal device, then ar86 operates in interactive mode, prompting the user for librarian commands.

The librarian commands are defined below. The commands completely control the operation of ar86. The commands specify the name of the archive file and the actions to be performed on the modules which constitute the library.

In the second mode,
ar86 [options] [action] ... archivefile

all the control information is contained on the command line.

Archivefile names the archive file to be operated on. If the archivefile does
not exist, then an empty archive file is created before the actions are
performed.

If the archive file name contains a suffix (i.e. contains a period), then the name
is used as is to access the archive file. If the archive file name has no suffix,
then .a is appended to the name before accessing the archive file.

[action] is one of the following:

-a filelist  The modules contained in the relocatable files in filelist are added to the
library contained in the archive file. If a module which already exists in the
library is added, it is an error (see -r to replace modules).
-d modulelist The modules in the modulelist are deleted from the library.
-r filelist  The modules contained in the relocatable files in filelist replace modules of
the same name in the library.
-e modulelist The modules in the modulelist are extracted (i.e. copied) and put into
relocatable files. The name of the file is the same as the name of the module
but with the suffix .o appended.

These actions are applied in the following order, regardless of their order on
the command line: -a, -d, -r, -e.

In filelist (or modulelist), individual file names (or module names) may be
separated by commas or separated by white space with the whole list quoted.

If the file names in filelist have a suffix (i.e. contain a period), then the name is
used as is to access the relocatable input file. If the name has no suffix, then
.o is appended to the name to obtain the name of the input file.

The following option is recognized by ar86:

-L  specifies that a library listing file be written to standard output. This output is
in the same format as that produced from the LIST command documented
below and is for the result of the archive session.
Commands

In the interactive mode, ar86 recognizes the following commands. In the syntax descriptions below, square brackets [ ] enclose optional items. Ellipsis ‘...' indicate that the preceding item may be repeated.

ADDLIB archivefile [(module [, ...] )]

Add one or more modules from the named library to the present library. If no modules are specified, the entire library is included.

ADDMOD filename [, ...]

Add the module(s) contained in one or more relocatable files to the present library.

CLEAR

Resets the librarian to await the creation or opening of a library. All information about the previous state of the librarian is lost.

CREATE archivefile

Specify the name of a new archive file to be created.

DELETE module [, ...]

Delete one or more modules from the current library.

DIRECTORY archivefile [(module [, ...] )] [listfile]

Obtain a brief listing of the modules in a library. If no modules are specified, the entire library is listed. If listfile is not specified, the listing goes to standard output.

END

Exit the librarian without saving the current library. Use SAVE to save the results of the current session.

EXTRACT module [, ...]

Copy one or more modules to individual relocatable object files. The name of the object file is the module name with .o appended.

FULLDIR archivefile [(module [, ...] )] [listfile]
LIST archivefile [(module [, ...] )] [listfile]

Obtain a detailed listing of the modules in a library. If no modules are specified, the entire library is listed. If listfile is not specified, the listing goes to standard output.
Chapter 2: Command Syntax

HELP
Displays a list of commands that may be executed at the current time. Only commands that are valid at the current time are displayed.

OPEN archivefile
Specify the name of an existing archive file to be opened. An archive file must be opened before commands like ADDMOD, DELETE, EXTRACT, and REPLACE can be used.

REPLACE filename [, ...]
Replace one or more existing modules in the present library with the modules from the named files.

SAVE
Saves the current library to disk. Use END to exit the librarian. The END command will not save the current library before exiting, so the SAVE command should be used before exiting the librarian if the library is to be saved or updated.

Files
archivefile.a Relocatable archive file.
file.o Relocatable file produced by as86(1) or ld86(1).

See Also
HP B1449 8086/186 Assembler/Linker/Librarian Reference Manuals, as86(1), ld86(1), ap86(1).

Diagnostics
ar86 returns zero if no errors are detected. Otherwise it returns non-zero if errors are detected.
Diagnostic messages including optional lines containing assembly errors are displayed on standard error.
Part 2

Assembler/Macro Preprocessor Reference
Assembler Introduction

Overview of the instruction set, target microprocessors, input and output file formats, and other similar information.
Chapter 3: Assembler Introduction

Instruction Set

The as86 assembler supports Intel instruction mnemonics, op codes, and syntax for the target microprocessors and thus is compatible with those used in Intel software and documentation.

The supported instruction set is listed in the chapter titled "Instructions and Operands." For further information about the instruction set, refer to the Intel iAPX 86/88, 186/188 User's Manual Programmer's Reference.

Target Microprocessors

The as86 supports the Intel 8086/186 chip family. The 8086/186 family includes the 8086, 8088, 80186, and 80188. In addition to the 8086/186 family, the as86 assembler will accept NEC V20/V30 extensions to the 8086/186. For these instructions, as86 accepts NEC mnemonics, but uses Intel syntax. For overlapping instructions (instructions found in both the 8086/186 and V20/V30), as86 accepts only Intel mnemonics and syntax. Unless an assembler control changes the microprocessor mode, as86 defaults to the 8086 mode.

The as86 assembler also translates instructions specific to the Intel 8087 or 80187 floating-point coprocessor for coprocessor execution.

Assembler Operation

as86 is a two pass assembler. On the first pass, labels, variables, and other user-defined symbols are examined and placed in an internal symbol table. Additionally, structure definitions are stored.

On the second pass, as86 generates the object code, resolves symbolic addresses, and outputs the object module if the assembly was error free. If it was not error free, then as86 displays errors on the output listing device and also a cumulative error count. In addition to the object module, as86 can also output an HP 64000 format assembler symbol file for use in analysis tools.
The assembly listing produced during pass two contains information pertaining to the assembled program, including opcodes, assembled data, and the original source statements. Based on command line options, as86 may also output a symbol table or cross reference table which gives further information not found in the standard assembly listing. Refer to the chapter titled "Assembler Listing Description" for a more complete explanation of the assembly listing and cross reference or symbol table information.

File Formats

Input File Characteristics

The source file input for the as86 assembler is a text file containing 8086/186 instructions, assembler directives, and assembler controls. This file can be produced from an editor or the output file from another component of the HP B1449 package, the ap86 macro preprocessor.

Output File Characteristics

HP-OMF 86

as86 produces a relocatable output object file in HP-OMF 86 format relocatable. HP-OMF 86 format relocatable is a superset of Intel Binary OMF relocatable. HP-OMF 86 format relocatable contains extensions to facilitate code integration and debugging. This format has not been verified to be strictly compatible with Intel Binary OMF relocatable. HP-OMF 86 format relocatable files, therefore, may not work correctly with tools or systems designed to consume Intel Binary OMF relocatable.

HP 64000 Assembler Symbol File

as86 can optionally produce an HP 64000 format assembler symbol file. This file is used by analysis tools. The purpose of the assembler symbol file is to preserve the relationship between symbolic names that appeared in the original source file and the memory locations that they referenced.
Chapter 3: Assembler Introduction
File Formats
Assembler Syntax

The basic elements of assembler language.
Chapter 4: Assembler Syntax

Assembler Character Set

Assembly language, like other programming languages, has a character set, a vocabulary, rules of grammar, and conventions that allow for definition of new words or elements. The rules that describe the language are referred to as the "syntax" of the language. This chapter describes the basic elements of assembler language:

- the character set
- symbols
- constants
- delimiters

These basic elements, in turn, are put together to form assembler statements. This chapter also gives the general syntax of those statements.

Input source lines over 1024 characters in length will be truncated and an error message will be generated.

Assembler Character Set

The assembler recognizes the characters in the following tables.

The characters are case sensitive by default. If case sensitivity is turned off, then all lower case alphabetic characters are treated as if they were upper case, unless they appear in quoted strings.

### Alphabetic Characters

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

### Numeric Characters

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

56
Chapter 4: Assembler Syntax
Assembler Character Set

Special Characters

- blank
- $ dollar sign
- ' single quote
- ) right parenthesis
- - minus sign
- : colon
- # double quote
- ? question mark
- [ left bracket
- ` accent grave
- | vertical bar
- horizontal tab
- < less than
- ( left parenthesis
- + plus sign
- . period
- ! exclamation point
- = equal sign
- \ back slash
- ] right bracket
- { left brace
- ~ tilde
- / slash
- > greater than
- * asterisk
- , comma
- @ commercial at
- & ampersand
- ; semicolon
- % percent
- _ underscore
- caret (uparrow)
## ASCII Codes

<table>
<thead>
<tr>
<th>Char.</th>
<th>ASCII</th>
<th>Char.</th>
<th>ASCII</th>
<th>Char.</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>20</td>
<td>@</td>
<td>40</td>
<td>‘</td>
<td>60</td>
</tr>
<tr>
<td>!</td>
<td>21</td>
<td>A</td>
<td>41</td>
<td>a</td>
<td>61</td>
</tr>
<tr>
<td>&quot;</td>
<td>22</td>
<td>B</td>
<td>42</td>
<td>b</td>
<td>62</td>
</tr>
<tr>
<td>#</td>
<td>23</td>
<td>C</td>
<td>43</td>
<td>c</td>
<td>63</td>
</tr>
<tr>
<td>$</td>
<td>24</td>
<td>D</td>
<td>44</td>
<td>d</td>
<td>64</td>
</tr>
<tr>
<td>%</td>
<td>25</td>
<td>E</td>
<td>45</td>
<td>e</td>
<td>65</td>
</tr>
<tr>
<td>&amp;</td>
<td>26</td>
<td>F</td>
<td>46</td>
<td>f</td>
<td>66</td>
</tr>
<tr>
<td>'</td>
<td>27</td>
<td>G</td>
<td>47</td>
<td>g</td>
<td>67</td>
</tr>
<tr>
<td>(</td>
<td>28</td>
<td>H</td>
<td>48</td>
<td>h</td>
<td>68</td>
</tr>
<tr>
<td>)</td>
<td>29</td>
<td>I</td>
<td>49</td>
<td>i</td>
<td>69</td>
</tr>
<tr>
<td>*</td>
<td>2A</td>
<td>J</td>
<td>4A</td>
<td>j</td>
<td>6A</td>
</tr>
<tr>
<td>+</td>
<td>2B</td>
<td>K</td>
<td>4B</td>
<td>k</td>
<td>6B</td>
</tr>
<tr>
<td>,</td>
<td>2C</td>
<td>L</td>
<td>4C</td>
<td>l</td>
<td>6C</td>
</tr>
<tr>
<td>-</td>
<td>2D</td>
<td>M</td>
<td>4D</td>
<td>m</td>
<td>6D</td>
</tr>
<tr>
<td>.</td>
<td>2E</td>
<td>N</td>
<td>4E</td>
<td>n</td>
<td>6E</td>
</tr>
<tr>
<td>/</td>
<td>2F</td>
<td>O</td>
<td>4F</td>
<td>o</td>
<td>6F</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>P</td>
<td>50</td>
<td>p</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Q</td>
<td>51</td>
<td>q</td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>R</td>
<td>52</td>
<td>r</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>S</td>
<td>53</td>
<td>s</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>T</td>
<td>54</td>
<td>t</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>U</td>
<td>55</td>
<td>u</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>V</td>
<td>56</td>
<td>v</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>W</td>
<td>57</td>
<td>w</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>X</td>
<td>58</td>
<td>x</td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>Y</td>
<td>59</td>
<td>y</td>
<td>79</td>
</tr>
<tr>
<td>:</td>
<td>3A</td>
<td>Z</td>
<td>5A</td>
<td>z</td>
<td>7A</td>
</tr>
<tr>
<td>;</td>
<td>3B</td>
<td>[</td>
<td>5B</td>
<td>{</td>
<td>7B</td>
</tr>
<tr>
<td>&lt;</td>
<td>3C</td>
<td>\</td>
<td>5C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>3D</td>
<td>]</td>
<td>5D</td>
<td>}</td>
<td>7D</td>
</tr>
<tr>
<td>&gt;</td>
<td>3E</td>
<td>^</td>
<td>5E</td>
<td>~</td>
<td>7E</td>
</tr>
<tr>
<td>?</td>
<td>3F</td>
<td>_</td>
<td>5F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Symbols

Symbol Formation

A symbol is a sequence of characters. The first character must be

- A-Z or a-z (alphabetic)
- ? (question mark)
- @ (commercial at sign)
- _ (underscore)

The second and following characters can be any of these characters or the numerals 0-9. Symbols can be up to 255 characters in length, but only the first 31 characters are significant.

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), and so on.

Examples of valid symbols:

- LAB1
- @mask
- LOOP_NUM
- L2345678901234567890123456789012345

In the last symbol, the entire symbol is stored, but only 31 characters are used for comparison.

Examples of invalid symbols:

- ABORT* ;contains special character
- 1LAR ;begins with a numeric
- PAN N ;embedded blank, symbol is PAN

Different symbols represent different kinds of data objects. In general, only a few kinds of symbols are allowed in any particular syntactic construct. Any of the following elements are considered to be symbols.
Keywords

Keywords (also called Reserved Words) are symbols pre-defined by the assembler which you can reference in certain acceptable constructs. Keywords are not user-definable, nor can you create a user-defined symbol with a name that conflicts with a keyword. Keywords include directives and register names, among others. Keywords are not case-sensitive. The full list of assembler keywords appears in the following table. Although the keywords in the table are in upper case, there is no requirement that they appear in upper case in the source code.

Table 4-1. as86 Assembler Keywords and Instructions

<table>
<thead>
<tr>
<th>Segment</th>
<th>AAA</th>
<th>AAD</th>
<th>AAM</th>
<th>AAS</th>
<th>ABS</th>
<th>ADC</th>
<th>ADD</th>
<th>ADD4S</th>
<th>AH</th>
<th>AL</th>
<th>AND</th>
<th>ASSUME</th>
<th>AT</th>
<th>AX</th>
<th>BH</th>
<th>BL</th>
<th>BOUND</th>
<th>BP</th>
<th>BRKEM</th>
<th>BX</th>
<th>BYTE</th>
<th>CALL</th>
<th>CBW</th>
<th>CH</th>
<th>CL</th>
</tr>
</thead>
</table>
### Table 4-2. as86 Assembler Keywords and Instr. (Cont’d)

<table>
<thead>
<tr>
<th>CLC</th>
<th>CLD</th>
<th>CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR1</td>
<td>CMC</td>
<td>CMP</td>
</tr>
<tr>
<td>CMP4S</td>
<td>CMPS</td>
<td>CMPSB</td>
</tr>
<tr>
<td>CMPSW</td>
<td>CODEMACRO</td>
<td>COMMON</td>
</tr>
<tr>
<td>CS</td>
<td>CWD</td>
<td>CX</td>
</tr>
<tr>
<td>DAA</td>
<td>DAS</td>
<td>DB</td>
</tr>
<tr>
<td>DD</td>
<td>DEC</td>
<td>DH</td>
</tr>
<tr>
<td>DI</td>
<td>DIV</td>
<td>DL</td>
</tr>
<tr>
<td>DQ</td>
<td>DS</td>
<td>DT</td>
</tr>
<tr>
<td>DUP</td>
<td>DW</td>
<td>DWORD</td>
</tr>
<tr>
<td>DX</td>
<td>END</td>
<td>ENDM</td>
</tr>
<tr>
<td>ENDP</td>
<td>ENDS</td>
<td>ENTER</td>
</tr>
<tr>
<td>EQ</td>
<td>EQU</td>
<td>ES</td>
</tr>
</tbody>
</table>
Instruction Mnemonics

A full set of instruction names (mnemonics) is pre-defined by the assembler. Instruction names can be removed from the symbol table with the PURGE directive and re-defined as something else. If you do this, the original meaning of the instruction is lost. There are six instructions (the operators AND, NOT, OR, SHL, SHR and XOR) that cannot be removed. A full list of the pre-defined instruction mnemonics, including the argument combinations acceptable for each, appears at the end of the chapter titled "Instructions and Operands."

Codemacro

A codemacro is a user-defined instruction or prefix to an instruction. The output generated from a codemacro can be a new instruction, a mixture of normal instructions, or just about anything that a customer might want (some assemblers define the normal instructions through the use of codemacros). A codemacro can be defined with the same name as an existing instruction or it can have a completely unique name that describes a new operation. Codemacros can be used anywhere that a predefined instruction can be used.

Label

A label is a user-defined symbol denoting the address of an instruction. Labels can be referenced only in the JMP and CALL instructions and variations thereof. A label can be defined with the PROC directive or with the LABEL directive, but there is another way to define a label that is used most often.

The most common way of defining a label is to place a name (followed by a colon) before an instruction mnemonic, which defines it as a label. Labels have certain attributes, but a discussion of those aspects of labels is left to the chapter titled "Symbol and Expression Attributes." Example:

```
THIS_IS_A_LABEL: MOV AX, 2
```

Variable

A variable is a user-defined symbol denoting the address of a location to be used for data storage. Unlike many other assembly languages, as86 distinguishes between a label and a variable. They are defined according to syntax and cannot be used interchangeably in expressions or instructions.
However, when the LABEL directive is used with the keywords BYTE, WORD, DWORD, QWORD, TBYTE, or with a variable that is a structure name or record name, it defines a variable. When the LABEL directive is used with the type designator NEAR or FAR, it defines a label. Variables have certain attributes, which are discussed in the chapter titled "Symbol and Expression Attributes."

**Structure Name**

A structure is a user-defined template describing the manner in which a block of storage is to be broken up into elements. A structure template does not have a storage area associated with it which means that a structure name, while it is still a symbol, is not a variable. A structure template name does not have attributes associated with it.

**Structure Field Name**

The individual elements of the structure template are called structure fields. Structure fields may be optionally assigned names, but again, since the structure template does not occupy storage, the structure field name is not a true variable. A structure field name, when a structure is allocated using the template, can be used with the dot operator to access an element of the structure, but the structure field name cannot be used alone. Structure field names do not have attributes associated with them.

**Record Name**

A record is a user-defined template describing how a one- or two-byte block of storage is to be broken up into bit fields. A record template does not have a storage area associated with it which means that a record name is not a variable. Record names do not have attributes associated with them.

**Record Field Name**

Each bit field describes a number of bits and has a name associated with it. Record field names are not variables, however, and do not have any attributes associated with them.
Segment Name

A segment is a user-defined logical division of the assembly source program. A logical segment can contain code, data, or stack information. Logical segments have names associated with them. These names are used to identify the logical segments to the assembler and loader so that they will eventually be placed together in the same physical segment in memory.

Group Name

A group name identifies a collection of logical segments gathered together because of some common factor. At load time, a group will be placed in memory such that any segment that is a member of the group will be within 64 kilobytes of the base of the group. Group names are also significant to the assembler and loader.

EQU Symbols

EQU symbols are names associated with other symbols or expressions through the use of the EQU assembler directive. EQU symbols are simply "replacement names" that can be used anywhere the symbols or expressions they replace could be used. Unlike symbols, however, EQU symbols are not variables and are not allocated storage.

Constants

A constant is an invariant quantity that can be either an arithmetic value or a character constant. Arithmetic values can be represented in either integer or floating-point format.

This section describes integer constants, real constants, and character constants.
Integer Constant

Decimal (base-10) constants can be defined as a sequence of numeric characters optionally preceded by a plus or a minus sign. If unsigned, the value is positive by default.

Internally, the assembler performs arithmetic on 17-bit quantities. A 17-bit value is 16-bit value with the 17th bit (the leftmost bit) as a sign bit. This value may range from -65535 to 65535 (-0FFFFH to 0FFFFH). However, integer constants are only allocated 16 bits when the assembler stores them in the output code. The 17-bit value can be interpreted as a signed or unsigned value and stored in one or two bytes.

A one byte constant can contain an unsigned number with a value from 0 to 255. A two byte unsigned number can range from 0 to 65535. When a constant is negative, its equivalent two's complement representation is generated and placed in the field specified. A 1-byte two's complement number can range from -128 to +127. A 2-byte two's complement number can range from -32768 to +32767. Whether or not a number is interpreted as a two's complement or an unsigned number is typically up to you.

Integer constants outside this range (-65535 to +65535) can appear only in the DD, DQ and DT directives, and on the right side of an EQU directive. The legal range is different for each directive, as discussed in the chapter called "Assembler Directives."

Other Bases

Constants with bases other than decimal are defined by specifying a coded descriptor after the constant. In addition, the base may restrict or expand the accepted digits for the constant. The following list is of the available descriptors and their meanings and the range of acceptable digits for each kind of constant. If no descriptor follows a constant, the number is decimal by default.

- B - a binary constant - digits must be either 0 or 1
- O - an octal constant - digits are 0-7 inclusive
- Q - an octal constant - digits are 0-7 inclusive
- D - a decimal constant (the default if no descriptor appears) - digits are 0-9 inclusive
Chapter 4: Assembler Syntax

Constants

- H - a hexadecimal constant - digits are 0-9 inclusive and the letters A-F (or a-f — either are allowed regardless of case sensitivity)

**Note**

Hexadecimal constants may not begin with the letters A-F (a-f). In those cases, prefix the constant with a zero.

Examples of acceptable constants:

- `10011B  ;binary constant`
- `25       ;defaults to decimal constant`
- `-0FFH    ;hex constant - notice leading 0`
- `1377Q    ;octal constant`
- `2d9fh    ;hex constant`

**Real Constant**

Real constants can only appear in DD, DQ, DT and EQU directives. There are three syntactically distinct ways of defining real numbers.

**Decimal Real Without Exponent**

See the following figure for the syntax diagram of decimal reals with exponents.

![Figure 4-1. Syntax for Decimal Real Without Exponent](image)

Examples:

- `1.234`
- `.1234`
- `1234.`
Decimal Real With Exponent

See the following figure for the syntax diagram for decimal reals with exponents.

![Syntax for Decimal Real with Exponent]

This format is interpreted to mean that the number to the left of the E is multiplied by 10 raised to the power of the number to the right of the E.

Examples:

\[
3.14159E-27 \quad ;\text{means } 3.14159 \times 10^{-27} \\
-1e4 \quad ;\text{means } -10000.
\]

Hex Real

The syntax is 8, 16, or 20 hex digits followed by the letter R (or 9, 17, or 21 hex digits if a 0 must be prefixed to constants with leading hex digits of A-F).

Note that no sign is permitted. This format represents the actual bit pattern to be placed in a variable of type DWORD (8 or 9), QWORD (16 or 17), or TBYTE (20 or 21). (Intel’s documentation describes the bit patterns used to represent real numbers.) Examples:

\[
40490FDBR \\
0c0000000r
\]

Character Constant

An ASCII character constant is specified by enclosing one or two characters within single or double quotation marks. The constant is encoded as a 16-bit number stored in different ways depending upon usage.
A character string of arbitrary length can be specified with the DB assembler directive.

A more complete discussion of character constants is contained in several of the chapters that follow.

**Delimiters**

The characters "blank" and "tab" are referred to as delimiters.

**Note**

There must be at least one delimiter between adjacent symbols and/or numeric constants to prevent them from being interpreted as a single item.

Delimiters are significant in character strings. Delimiters are not required between characters that have special meaning to the assembler (such as [, +, =, $, and so on).

**Assembler Statements**

**General Syntax**

The basic elements just described are put together to create statements and instructions that the assembler understands. The rules that govern the ways that statements may be formed are called syntax rules. The general syntax for an as86 assembly language instruction statement is as follows:

```
[ label : ] [ prefix ] keyword [ operand [ , ... ] ] [ ; comment ]
```

Each field in the general syntax has one or more of the delimiters discussed in the previous section between it and adjacent fields. Each field has a different purpose.
Chapter 4: Assembler Syntax
Assembler Statements

Label
The label is optional and, if present, identifies or marks the offset of the instruction. This label may be used as a destination in CALL, JMP or conditional branch instructions. Notice the colon following the label. It must be present if the label is present.

Prefix
The prefix, if present, causes looping with string instructions or forces a bus lock during the instruction's execution. New prefixes can be defined through the use of codemacro definitions.

Keyword
Keywords can be any of the instruction mnemonics (a list of instruction mnemonics appears at the end of the chapter titled "Instructions and Operands"), codemacros defined by the user, or an EQU symbol set to an instruction or codemacro name.

Operand
An operand is an argument to the instruction in the keyword field. Commas separate multiple operands. Operands are discussed more completely in the chapter titled "Instructions and Operands."

Comment
The comment begins with a semicolon and continues until the end of the line. Comments are used to make "notations" about the assembly language code so that you or others may better understand the purpose of the code or how it works.

Comment
Comments can appear after instructions, assembler directives, control statements, macro definitions, or on lines by themselves. In fact, comments can appear anywhere in the assembly source file as long as they are preceded by semicolons. Comments are not processed by the assembler, but are passed through to the assembler listing.
When a comment is on a line by itself, a leading semicolon must be the first non-blank character (tabs are considered blank characters) on the line. Blank lines are treated like comments.

Continuation Lines

Some assembler statements will not fit on a single line. If a statement will not fit on a single line, it may be continued to the next line by beginning the next line with the ampersand (&) character. The ampersand must be in column one of the next line. Symbols, numbers, and strings cannot be broken across lines. It is not acceptable to use the ampersand to continue a comment line. In most cases, an error is likely to occur. Simply begin the new line with a semicolon to make it another comment line.
Symbol and Expression Attributes

An introduction to attributes.
Symbols and expressions have certain attributes that determine where they may be used with an instruction and what object code will be generated if they are used. Most attributes are only important when a symbol or expression involves a relocatable or external value. Absolute values will not involve most attributes since absolute values are not modified by the loader.

There are nine attributes that a symbol or expression can have. They are

- **TYPE**
- **OFFSET**
- **BASE**
- **INDEX**
- **SEGMENT**
- **SEGMENT RELLOCATION**
- **RELOCATION TYPE**
- **SEGMENT ADDRESSABILITY**
- **CS ADDRESSABILITY**

Not all attributes will apply in all cases, however. The following sections discuss the different attributes and how they affect symbols and expressions.
The TYPE attribute may belong to either a variable, label, or memory expression. The fixed types are:

- BYTE (1 byte)
- WORD (2 bytes)
- DWORD (4 bytes)
- QWORD (8 bytes)
- TBYTE (10 bytes)
- FAR (same or different segment)
- NEAR (same segment)

User-defined types are also possible and are created when a record or structure template is defined. See the chapter titled "Assembler Directives" for more about records and structures.

It is possible for a memory expression to not have a type. Instead, the type is determined by using the expression. These explicitly typeless memory expressions are the so-called anonymous references.

The OFFSET attribute for a variable, label, or memory expression is the offset from the start of a segment or group. It is simply the number of bytes from the start of the segment or group. If the variable or label belongs to a noncombinable segment or if the expression was generated from a numeric value, the offset will be absolute. If the variable or label belongs to a combinable segment or to a group, the offset will be relocatable.
BASE

The BASE register may be set as part of a memory reference. If a base register is used as part of an expression, the expression is known as a register expression, to set it apart from the simpler memory expression.

The base registers are BX and BP. Only one of these registers may be present in a any single register expression, although an index register may be present with the base register. If a base register is used in a memory expression, its contents are added to the memory offset at run-time to calculate a final offset for a memory location. If both a base and index register are present in the memory expression, then their values are first added together and then added to the offset to produce the memory reference. If the memory expression does not have a SEGMENT attribute (i.e., no variable, label, or segment override was used as part of the expression), then a default segment register will be used depending upon which base register appears in the register expression. If the BX register is used, DS is the default segment register. If BP is used, the default is SS. The default to SS for BP holds even if an index register is also present in the memory expression.

INDEX

The INDEX register may also be used as part of a memory reference. If an index register is used as part of an expression, either with or without a base register, then the expression is known as a register expression, to set it apart from the simpler memory expression.

The valid index registers are SI and DI. Only one index register can be present in a single register expression. It is also possible, of course, that no index register will be used. If an index register is used in a register expression, its contents are added, at run-time, to a memory offset to calculate a final offset for a memory location. If both an index and base register are used in a register expression, both registers are added to the offset to calculate the final offset. If the memory expression does not have a SEGMENT attribute and no base register is used, then the DS segment register is used as a default.
The SEGMENT attribute determines which segment a variable, label, or memory expression belongs to. The segment attribute is the base value of that segment. The base value is absolute if the segment has been placed using the AT keyword. Otherwise, it is a relocatable value until load time. (This attribute is also the value that is returned by using the SEG operator.)

**SEGMENT RELOCATION**

The SEGMENT RELOCATION attribute becomes important when a variable, label, or memory expression belongs to a group. In contrast to the SEGMENT attribute, this attribute determines which group the item belongs to. The SEGMENT attribute identifies which segment within the group the item belongs to. These two values must be known to correctly calculate offsets for a memory expression. Normally, this attribute is the same as the SEGMENT attribute unless the expression contains a group override. This attribute can be ignored unless groups are used.

**RELOCATION TYPE**

The RELOCATION TYPE is determined by a combination of the type of an expression and by operators that are applied to it. This value will be null if the expression can be completely determined at assembly time. This is true of offsets within non-combinable segments and for segment bases of segments that use the AT keyword. This value will be set, however, if the item is an offset from either a combinable segment or a segment base for a non-located segment or group. The possible types of relocation are:

- **OFFSET:** This type of relocation will generate the offset of a variable, label, or memory expression as part of the object code. A 16-bit offset value will be calculated by the loader and inserted into the object code. The offset will be calculated relative to the base of the segment or, if a
group override is used, relative to the base of the group. It is possible to add a 17-bit value to this offset.

- **BASE**: This type of relocation causes a 16-bit base value to be written directly to the object code. The base will be the base address of the segment that the variable, label, or memory expression belongs to unless a group override is used. In that event, the base will be the base address of the group. It is possible to add a 17-bit value to this base.

- **HIGH**: This type of relocation causes the upper 8-bit portion of an offset to be written to object code. The offset is calculated using the same rules as noted above, but only the high byte will be written out. It is possible to add an 8-bit value to this byte.

- **LOW**: This type of relocation causes the lower 8-bit portion of the offset to be written to object code. The offset is calculated using the same rules as noted above, but only the low byte will be written out. It is possible to add an 8-bit value to this byte.

---

**SEGMENT ADDRESSABILITY**

The **SEGMENT ADDRESSABILITY** of a memory location is determined by the segment the memory location belongs to and by any segment or group overrides applied. If a segment override is used to name a specific segment register, that register is used to address the memory location. Otherwise, the values found in the ASSUME directives must be tested. If the segment or group is found through the current ASSUME values, then that segment register is used to address that memory location. If no match is found, an error is generated, since the memory cannot be accessed.

It is possible to have a memory location that does not belong to a segment or group. This would be true of an anonymous memory reference, which looks like

```
[BX][SI]
; base and index registers
```

In such a reference, the segment addressability will be determined by using the default segment registers defined for the base and index registers. Recall that
the default segment register will be DS unless the BP base register is used, in which case the default will be the SS segment register.

---

**CS ADDRESSABILITY**

The CS ADDRESSABILITY of a label is determined from both the current ASSUME value for the CS register, and any segment or group overrides that are applied to the label.
Assembler Directives

Alphabetical description of assembler directives.
This chapter describes the as86 assembler directives. In an assembly language program, assembler directives are written as any other program statement might be, but directives are not translated into equivalent machine language instructions. Instead, assembler directives are interpreted as *instructions to the assembler* to control the program assembly process itself.

In this chapter, directives are organized in alphabetical order for easy reference. (The DB, DW, DD, DQ, and DT directives are described together because of their similarity.) However, assembler directives may also be grouped into three broad categories—Segmentation Directives, Data Definition Directives, and Program Linkage Directives— which identify the parts of the assembly process the different directives are designed to affect. Segmentation Directives inform the assembler about the logical organization of your program. Data Definition Directives control the allocation and initialization of data, variables, and labels. Program Linkage Directives make it possible to create modular assembly language programs. The first sections of this chapter list the directives grouped by these three categories, briefly describe their functions, and more thoroughly discuss some concepts important to understanding how these directives work.
Segmentation Directives

ASSUME informs the assembler of the contents of the segment registers.

GROUP combines several logical segments together.

SEGMENT/ENDS defines a logical segment in the assembly language program code.

These directives control program segmentation (the dividing of the assembly program into logical parts). To better understand program segmentation, read the following discussion.

Program Segmentation

The 8086 can directly address one megabyte of memory. This memory is viewed by the CPU through four segments, known as physical segments, each containing up to 64K bytes. The start of each segment is defined by a value, called a paragraph number, placed in one of the four special registers known as segment registers. A paragraph number, or boundary, is located at a memory address which is divisible by 16 (that is, the least significant hexadecimal digit of the address is 0H). A physical segment is said to be active if one of the segment registers contains the base address of the start of the segment.

The four segments are classified as the code, data, stack, and extra segments. They are each pointed to by a separate segment register:

- CS for code
- DS for data
- SS for stack
- ES for extra

Executable instructions will be in a physical segment defined by the value in CS. Any stack operation will occur within the segment defined by SS. Data is generally found in the segment pointed to by DS, but it can also be placed in any of the other segments. The segment accessed through the ES register will usually hold data also.
Chapter 6: Assembler Directives

A logical segment is a segment as defined within a single assembly file. The linking loader can combine this logical segment with other segments of the same name to form a single physical segment. The size of the physical segment is limited to 64K, so the sum of the logical segments cannot exceed this limit. The collection of segments into a group is another form of physical segment.

**Default Segment - ??SEG**

All code and data within a source file must exist within some segment. Any code or data defined outside of segment directives within a source file will be assigned to a segment automatically created by the assembler. This segment is named ??SEG and exists in all object files. The ??SEG segment is defined to be public, so it is combined with all other ??SEG segments from other modules. It is also defined to be paragraph aligned.
Data Definition Directives

**DB** defines one byte of storage.

**DW** defines one word (two bytes) of storage.

**DD** defines one double word (four bytes) of storage.

**DQ** defines one quad word of storage (eight bytes - 8087 data types).

**DT** defines one tbyte (ten bytes - 8087 data types) of storage.

**EQU** assigns a particular value to a symbol.

**EVEN** aligns code or data with a word boundary.

**ORG** adjusts the location counter within the current segment.

**PROC/ENDP** assigns a label to a sequence of instructions.

**PURGE** causes a user-defined symbol to become undefined.

**RECORD** defines a record template.

**STRUC/ENDS** defines a structure template.

Data Definition Directives control the definition and initialization of data and/or storage as labels, variables, records, or structures.

**Data Objects**

The two most referenced data objects are variables and labels. With the Data Definition Directives, you may define these and other data objects in your program. Variables are data items, or areas of memory where values are stored. Labels allow you to "mark" locations or sections in your code that may be JMPed to or CALLed. One use of labels is to define "subroutine" locations in order to create structured programs. Unlike high-level language subroutines, however, scoping of names does not occur and you can "fall into" an embedded "subroutine."

Records and structures may also be defined by this category of directives. Records and structures are alike in that they are user-defined templates for storage allocation and initialization, they are not allocated storage at
definition time, the assembler "remembers" what they look like, they can be referenced as often as you like, and each reference generates one or more copies of storage in the format of the template. At the time of the reference, records and structures may optionally have certain of their definition-time default values replaced.

Records and structures are different, however, in their basic makeups. When you define a structure, you specify how many bytes the template covers, how the bytes will be broken up into variables, and what default values will be placed into those bytes at allocation-time. In contrast, a record must be a one or two byte collection of bit fields. When defining a record, you specify how the record is to be broken up into bit fields, and any default values to be placed in the bit fields at allocation-time. The record size depends upon the sum of the number of bits in all the bit fields, which means the total may not exceed 16 bits.

**Linkage Directives**

- **END** specifies the end of an assembly module.
- **EXTRN** specifies symbols defined in other modules.
- **NAME** assigns a name to an assembly module.
- **PUBLIC** specifies which symbols are public.

Program Linkage Directives make it possible for you to create modular assembly language programs. Refer to the discussion of program linkage that follows to better understand the use of these directives.

**Program Linkage**

as86 supplies the necessary directives to support multi-module programs. A program may be composed of many individual modules that can be separately assembled or compiled. Each module may define variables or labels that other modules may use. The Program Linkage Directives are the mechanisms in as86 for communicating symbol information from module to module, for identifying those symbols within the current module that may be used by other modules, for stating what symbols (defined elsewhere) can be used within the
current module, and for uniquely naming different object modules that are to be linked together. Using these directives, you may specify a "main module," that is, a module which contains the code that will be initially executed upon loading the program (the address the loader will use to initialize the start address of the program). At the same time, you may also supply initialization values for other segment registers.
ASSUME

The ASSUME directive is used to inform the assembler of the contents of the segment registers.

Syntax:

ASSUME segreg:segpart [, ...]
(or)
ASSUME NOTHING

Where:

segreg is one of the segment registers CS,DS,ES or SS.

segpart is one of the following:

- A segment name. The base address of the segment is assumed to be in the named register. All data (or code) in the segment is addressable through this register.
  Example:

  ASSUME CS:CODE, DS:DATA

- A group name (must have been previously defined). The base address of the group is assumed to be in the named register. All code or data in all segments in the group are addressable through this register. Example:

  ASSUME CS:CODEGRP, DS:DATAGRP

- A forward reference. Forward references with ASSUME are only allowed for symbols which will be defined as segment names later in the program. When the segment name is later defined, then it may be used to address memory within the segment. Failure to define the segment name will cause an error to be reported.

- The keyword SEG followed by the name of a previously-defined label, variable or external symbol. The base address of the segment containing the symbol (which may not be known until link-time) is assumed to be in the named register. The specified symbol and any other data known to be
in the segment are addressable through the register. (For an external symbol defined outside a segment, no such data is known.) Example:

ASSUME CS:SEG START, DS:SEG COUNT

- The keyword NOTHING. The register is assumed to contain garbage. The register will not be used to address any memory. The format

Assume NOTHING

is also legal; this is equivalent to

Assume CS:NOTHING, DS:NOTHING, ES:NOTHING, SS:NOTHING

Description: ASSUME is used by the assembler to

- determine if the code or data your program references is addressable
- decide whether a segment override byte should be generated.

Initially, the segment registers contain NOTHING (garbage) by default. The assembler assumes the contents of each segment register has not changed — since initialization or the last ASSUME — unless an ASSUME for that register is encountered. ASSUME itself, however, does not alter the value in the segment register. For example, the statement ‘ASSUME DS:DATA’ does not alter the contents of DS. You must, at some point, follow the ASSUME with a MOV instruction to DS in order to access data in the DATA segment without error.

CS register initialization, since it is done by the loader, does not require a MOV, but CS still requires an ASSUME before it may be used.

Note

There is an exception to the requirement that the CS register must have an ASSUME before it is used. When a JMP instruction is used without a current CS-ASSUME value, the default is to ASSUME the current segment. The segment registers will not be checked. This only applies to NEAR references, since a JMP to a FAR label requires that the CS register be updated.
Chapter 6: Assembler Directives
DB, DW, DD, DQ, DT

DB, DW, DD, DQ, DT

The DB, DW, DD, DQ, and DT directives are used to define variables and/or initialize memory.

Syntax:

1 byte (Byte) initialization:
[name] DB init [,...]

2 byte (word) initialization:
[name] DW init [,...]

4 byte (dword) initialization:
[name] DD init [,...]

8 byte (qword) initialization:
[name] DQ init [,...]

10 byte (tbyte) initialization:
[name] DT init [,...]

(or)

[name] Dx repeatval DUP(init,[,...])
(where x is B, W, D, Q, T)

Where:

name is a unique as86 symbol. Its associated attributes will be:

- segment - current segment
- offset - current location counter
- type - type of data initialization unit

init may take on many possible values depending upon what type of initialization you wish to do. Init may be any of the following:

- A constant expression.
Chapter 6: Assembler Directives
DB, DW, DD, DQ, DT

- **DB** - 1 byte initialization. An integer constant or an expression which fits into 8 bits (either 0-extended or sign-extended) when stored in two's complement format. The range is -255 to +255. High and low relocatable numbers (created by the HIGH and LOW operators) are also acceptable scalars. Other relocatable numbers, such as the offset of a variable, are not acceptable.

Examples:

```
DB 0
```

```
DB 65535 ;not accepted, out of range
```

```
DB -1 ;these are equivalent
DB 255 ;both generate hex FF
```

- **DW** - 2 byte initialization. A constant or expression that evaluates to a number (either absolute or relocatable) which must fit into 16 bits (either 0-extended or sign-extended) when stored in two's complement format. The range is -65535 to +65535. Examples:

```
DW 0
```

```
DW 65536 ;not accepted, out of range
```

```
DW -1 ;these are equivalent
DW 65535 ;and generate hex FFFFH
```

- **DD** - 4 byte initialization. An integer constant or an expression that evaluates to an absolute number. The value must fit into 16 bits (either 0-extended or sign-extended). The range is -65535 to +65535. The 16-bit value is stored in the lower 2 bytes in two's complement format (least significant byte first) and the higher 2 bytes are sign-filled. Relocatable numbers are not permitted (it is impossible to determine how to fill the higher 2 bytes at assembly-time).

Use DD for an integer constant in the range -4,294,967,295 to +4,294,967,295 (from -(2^{32} + 1) to +(2^{32} -1), but not small enough to qualify for DW. Note that an expression cannot yield a value this large; all expressions evaluate to 17-bit numbers. The value is stored as a 32-bit two's complement integer, low byte first.
Chapter 6: Assembler Directives

DB, DW, DD, DQ, DT

A decimal real. The valid range is roughly -3.4E38 to -1.2E-38, 0, 1.2E-38 to 3.4E38.

A hex real of 8 digits (or 9 digits if its leading digit is 0).

Examples of the possibilities:

- **DD** - 8 byte initialization. An integer constant, or an expression whose value resolves to a 17-bit absolute number. The range of constants is -(2^64 + 1) to + (2^64 -1). Such integer values are stored in 64-bit twos complement format.

A decimal real number which has an approximate legal range of values is -1.7E308 to -2.3E-308, 0, 2.3E-308 to 1.7E308.

A hex real number consisting of 16 digits (or 17 digits if its leading digit is 0).

- **DT** - 10 byte initialization. An integer constant, or an expression that resolves to a 17-bit absolute number. The range of constants is -(10^18 + 1) to + (10^18 -1). All integer values are stored in 80-bit signed-magnitude packed decimal (BCD) format, least significant byte in the lowest address.

A decimal real number that has an approximate range of -1.1E4932 to -3.4E-4932, 0, 3.4E-4932 to 1.1E4932.

A hex real number consisting of 20 digits (or 21 digits if its leading digit is 0). Examples:

- **DQ** - 8 byte initialization. An integer constant, or an expression whose value resolves to a 17-bit absolute number. The range of constants is -(2^64 + 1) to + (2^64 -1). Such integer values are stored in 64-bit twos complement format.

A decimal real number which has an approximate legal range of values is -1.7E308 to -2.3E-308, 0, 2.3E-308 to 1.7E308.

A hex real number consisting of 8 digits (or 9 digits if its leading digit is 0).

Examples:

- `DD 0 ; yields 00000000`
- `DD 65535 ; yields FFFF0000 (low byte first)`
- `DD -1 ; yields FFFFFFFF`
- `DD 65537 ; yields 01000100 (low byte first)`
- `DD 3.14159 ; another decimal real`
- `DD 0C0000000R ; a hex real`
- `DD 0C0000000R ; a hex real`

- **DQ** - 8 byte initialization. An integer constant, or an expression whose value resolves to a 17-bit absolute number. The range of constants is -(2^64 + 1) to + (2^64 -1). Such integer values are stored in 64-bit twos complement format.

A decimal real number which has an approximate legal range of values is -1.7E308 to -2.3E-308, 0, 2.3E-308 to 1.7E308.

A hex real number consisting of 8 digits (or 9 digits if its leading digit is 0).

Examples:

- `DD 0 ; yields 00000000`
- `DD 65535 ; yields FFFF0000 (low byte first)`
- `DD -1 ; yields FFFFFFFF`
- `DD 65537 ; yields 01000100 (low byte first)`
- `DD -65537 ; yields FFFFFEFF (low byte first)`

- **DT** - 10 byte initialization. An integer constant, or an expression that resolves to a 17-bit absolute number. The range of constants is -(10^18 + 1) to + (10^18 -1). All integer values are stored in 80-bit signed-magnitude packed decimal (BCD) format, least significant byte in the lowest address.

A decimal real number that has an approximate range of -1.1E4932 to -3.4E-4932, 0, 3.4E-4932 to 1.1E4932.

A hex real number consisting of 20 digits (or 21 digits if its leading digit is 0). Examples:

- `DT 65535 ; generates 35550600000000000000000H (low byte first)`
- `DT -65535 ; generates 355506000000000000000080H (low byte first)`

- The character "?" for indeterminate initialization.
In situations where you wish to reserve storage but do not need to initialize the area to any particular value, use the special character "?" instead of a value. The area will be reserved with an indeterminate value. Examples:

```
ABYTE DB ? ; reserve a byte
AWORD DW ? ; reserve a word (2 bytes)
ADWORD DD ? ; reserve a double word (4 bytes)
AQWORD DQ ? ; reserve a quad word (8 bytes)
ATBYTE DT ? ; reserve a tbyte (10 bytes)
```

- **An address expression.**

  Assume registers are not checked when these directives are used with address expressions. Therefore, the only way to get a group-relative reference is to use a group override in the address expression.

  - **DW** - 2 byte initialization. DW may be used with a variable name, a label name, a group name, or a segment name. Using DW with a variable or label name causes the offset of a variable or label (relative to its segment or, if a group override is used, to its group) to be stored. Using DW with a group or segment name causes the paragraph number of that group or segment to be stored. Examples:

    ```asm
    DW COUNT     ; COUNT is a variable or label
                 ; store offset of COUNT from its segment
    DW DATAGRP :COUNT ; store offset of COUNT from its
                     ; group (DATAGRP)
    DW CODE      ; CODE is a segment or group name
                 ; store the paragraph number
    ```

  - **DD** - 4 byte initialization. DD may be used with a variable name, a label name, a group name, or a segment name. Using DD with a variable or label name causes the offset (relative to its segment or, if a group override is used, to its group) of the variable or label to be stored in the low order word and the segment or group base address for the label or variable to be stored in the high order word. Using DD with a group or segment name causes the paragraph number of that group or segment to be stored in the
low order word. The high order word will be set to 00H. Using DD with a variable or label name is equivalent to storing a pointer to the variable or label address. Examples:

\[
\begin{align*}
\text{DD COUNT} & \quad ; \text{COUNT is a variable or label, a pointer to it is stored} \\
\text{DW COUNT} & \quad ; \text{store offset of COUNT} \\
\text{DW SEG COUNT} & \quad ; \text{store COUNT’s segment}
\end{align*}
\]

- **Initialize with a string.**

  - **DB** - 1 byte initialization. A string of up to 1024 characters may be specified with the DB directive. Each character in the string, left to right, is assigned one byte of memory, low address to high address. The string must be enclosed within single or double quotes. A single quote may be embedded in the string by using two consecutive quotes. Examples:

    \[
    \begin{align*}
    \text{ALPHABET DB 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'} \\
    \text{WITHQUOTE DB 'THIS AIN’T HARD!’} & \quad ; \text{inserting single quote in string}
    \end{align*}
    \]

  - **DW, DD, DQ**. You may use these directives to code a string of 1 or 2 characters. Such a string is interpreted as a 17-bit number that is stored differently than it would be if DB were used. If two characters are stored, the second character in the string appears in the low byte of the storage and the first character appears in the next higher byte of the storage. If only one character is stored, the low byte of the storage contains the character. With either a 1 or 2 byte string, if any bytes of the storage remain unfilled, they are set to 00H. Using more than 2 characters in a string results in a warning message and only the first 2 characters are used.

  - **DT** DT can also code a two character string, but it does it in a way different from the other directives. DT stores the string in BCD packed decimal format. If a single character is stored, its decimal ASCII value is stored in the low byte of storage. The remaining bytes are set to 00H. If two characters are stored, however, it becomes more complicated. It is done as follows:

    The 17-bit hexadecimal number representing the string is converted to its decimal equivalent. (The 17-bit hex number is formed by placing the ASCII hex value of the first character of the
2 character string in the leftmost byte of the 17-bit word and placing the ASCII hex value of the second character in the rightmost byte of the 17-bit word. The sign bit is zero.)

Beginning with the rightmost digit of the resulting decimal value, the decimal representation is stored 2 digits per byte, working from right to left in the decimal value, until all digits are stored.

Any remaining bytes of storage are set to 00H.

Examples:

- **DB '01'** ; generates 3031H (shown low byte first)
- **DW '01'** ; generates 3130H (shown low byte first)
- **DD '01'** ; generates 3100H (shown low byte first)
- **DQ '01'** ; generates 3100 0000H (shown low byte first)
- **DT '01'** ; generates 3723 0100 0000 0000H

Repeating value. The special construct, DUP, can initialize an area of memory with a repeated value or a repeated list of values.

- **repeatval** specifies the number of data initialization units (from 1 to 65535) to be filled (bytes, words, dwords, qwords, or tbytes depending upon whether Dx is DB, DW, DD, DQ, or DT).
- **init** (as an argument to DUP) may be a single occurrence of the possibilities that were acceptable for **init** in the non-repeating-value syntax, including another DUP, or **init** may be a list of these same values. DUPs may be nested to eight levels deep. Below are some examples:

```
WORD1 DB 2 DUP (?)    ; two consecutive bytes form word
DD 2 DUP ('01')       ; generates 3130000031300000H
NESTEDDUP DB 3 DUP (4 DUP (1, 6 DUP (0))))
; 60 occurrences of 1, 6 DUP (0)
```

If an indeterminate initialization is repeated, the memory reserved by that data directive will NOT be initialized to 0. Also, repeating a relocatable value (such as a location in memory) will result in only the first value being assigned correctly. So this practice is discouraged.

Description: The DB, DW, DD, DQ, and DT directives are used to define variables and/or initialize memory. The descriptions of the parts of the syntax adequately describe these directives.
The END directive is used to inform the assembler that the last source statement has occurred and optionally to specify initial values for selected registers.

**Syntax:**

```
END [regint [, ...]]
```

**Where:**

- **regint** This field defines the contents for a segment register (and possibly the IP and SP registers). To initialize the segment registers, the field may take one of the following forms:
  
  - `[CS:] segname[:labelname]`
  - `DS: segname`
  - `SS: segname[:varname]`

  where

  - **segname** is either a segment name or a group name. It identifies the paragraph number to be loaded into the segment register.
  
  - **labelname** is the name of a label defined in the module. If it is used alone, its segment will be used to initialize the CS register and its offset will initialize the IP. If it is used with a segname, then just its offset will be used to initialize IP.
  
  - **varname** is the name of a variable defined in the module. Its offset will be used to initialize SP.

**Description:**

An END directive is required for all assembly language programs. Any statements that follow the END directive will not be processed. Specifying a load address with the END directive also informs the loader that the current module is the main program. The main program defines the start of execution because execution begins at the address specified with the END directive for the main program. If multiple load modules are combined by the loader, only one module can specify a load address and therefore be considered the main program.
The END directive may also be used to define the initial contents of the DS and SS segment registers by specifying values to be placed in these registers by the linker/loader at load-time.

**Note**
If the code is to be targeted for HP 64000 format absolute, you may only initialize the CS:IP register with END. Initialize the other registers explicitly within the code.

**Examples**
The following examples show the proper syntax for initializing different segment registers.

**CS (code) segment register initialization:**

- `END labelname` (initializes CS and IP; (the segment part of the label is used for CS)
- (or)
- `END CS:labelname` (same as ‘labelname’)
- (or)
- `END CS:segname:labelname` (the segment part (paragraph number) to be loaded into CS is taken from segname)

**SS (stack) segment register initialization:**

- `END SS:segname` (SP will be initialized to be equal to the size of the segment)
- (or)
- `END SS:segname:varname` (initializes SS and SP; (SP will be initialized to the offset of varname)

**DS (data) segment register initialization**

- `END DS:segname` (initializes DS
Chapter 6: Assembler Directives

EQU

EQU

The EQU directive causes the assembler to assign a particular value to a symbol.

Syntax:

equate_symbol EQU expression

Where:

equate_symbol is a mandatory symbol defined by this statement.

expression is one of the following items:

- A numeric constant or numeric expression. The value of the expression must be determined at assembly time. Any symbols used in the expression must have been previously defined. See the Description section below for more discussion about real constants. Examples:

```
PI EQU 3.14159 ; real constant stored with 10 byte precision
DD PI ; 4 byte floating point
DQ PI ; 8 byte floating point
DT PI ; 10 byte floating point
E1 EQU 2 + 3 ; numeric expression
E2 EQU E1 AND 4 ; E1 previously defined
E3 EQU (E1 - E2) / 12 ; E1 and E2 previously defined
```

- A variable or label name (which may be a forward reference).

```
ALABEL EQU ALAB ; ALAB not defined yet
```

- A register name, including ST registers. Example:

```
COUNT EQU CX
POINTER EQU BX
MOV COUNT, 10 ; CX = 10
MOV POINTER, OFFSET ARRAY ; BX = offset of array
FREQ EQU ST(1)
FADD ST, FREQ
```

- An instruction or codemacro name.

```
BUMP EQU INC ; instruction name
BUMP AX ; same as INC AX
```

- A register expression. These may be single register expressions, or they may also include a segment override. This construct is useful when
defining data items to be accessed on the stack. Refer to the Description section for a more information about the use of register expressions.

**Examples:**

```
STACKWORD EQU WORD PTR SS:[BP + 2]
AVAR EQU [BX + 3]
ANEXTVAR EQU ES:[BX]
```

**Description:**

The EQU directive in as86 is more powerful than the EQU found in most other assemblers. All the various attributes of address expressions are stored, and any missing attributes may be added later with expression operators at the time the EQUed symbol is referenced.

*Decimal real numbers* are stored in a full 10-byte format to prevent a loss of precision; they may be used in DD, DQ, or DT directives later in your code. Hex real numbers, however, are stored in as many bytes as the specification indicates; they can be used later only in the single directive that accepts a hex real of that size.

It is possible for a symbol to appear as a forward reference before it is defined in an EQU. When this happens, the assembler assumes that the forward reference will resolve to a number, variable or label. If this turns out not to be the case, an error may occur on pass 2 if the assembler did not leave enough room for an instruction on pass 1.

Symbol chaining (defining a symbol in terms of another symbol which is in turn defined by another symbol) can be accomplished with the EQU directive, but the chain must eventually end as a numeric or address expression. An error occurs if the definition ends in a register or real number expression. Circular EQU definitions are also errors. Example:

```
A EQU B
B EQU A ;ERROR! circular reference
```

A symbol defined by an EQU to an *address expression* consisting of more than one symbol (for example, BYTE PTR VBL) is stored as a variable or label, if possible. The entire EQU expression takes its attributes from the sub-expression on the right-side of the EQU. However, not all attributes will be set if attributes are missing from the right-side sub-expression. If that is the case, missing attributes must be supplied when the symbol on the left-side of the EQU is used elsewhere in an expression.

**Examples:**
A EQU [BX][SI][5] ; anonymous reference — type
; information must be supplied
; when A used elsewhere
B EQU WORD PTR 10 ; segment information must be
; supplied later
EVEN

The EVEN directive causes the Location Counter to be aligned to an even value (a word boundary).

Syntax:

EVEN

Description: The assembler brings about alignment by generating a NOP (90H) instruction if the current location counter contains an odd address value. The EVEN directive cannot be used in a byte aligned segment. Doing so will cause an error message to be generated.
The EXTN directive is used to declare certain symbols as external references.

Syntax:

```
EXTN name:type [, ...]
```

Where:

- **name** is a symbol, declared PUBLIC (see PUBLIC directive later in this chapter) in another module, to be defined as an external reference. Its associated attributes are the following:
  - **segment** - unknown unless defined within a SEGMENT/ENDS pair
  - **offset** - unknown
  - **type** - type declared in type argument
  - **relocation type** - external

- **type** is one of the following:
  - The keyword **BYTE, WORD, DWORD, QWORD**, or **TBYTE** for a variable which is one of these types.
  - A **structure name**. Names a variable whose type is equal to the number of bytes allocated in a preceding structure definition.
  - A **record name**. Names a variable whose type will be either byte or word depending on the preceding record definition.
  - **NEAR** or **FAR**. A label of type near or far.
  - **ABS**. A constant (17-bit number), always of type word.

Description: Symbols declared as EXTN are not expected to be defined in the current module (they cannot be), but are passed to the loader to be matched against symbols declared PUBLIC in other modules. In as86, the EXTN directive will specify the name of the symbol and its associated type.
Note

The type declaration must agree with the type of the symbol declared PUBLIC, but the loader does not do type-checking. It is your responsibility to maintain type compatibility.

The type ABS is used to declare a constant. Despite the mnemonic ABS, this number can prove to be offset relocatable or absolute when it is resolved depending upon how it was defined as a PUBLIC symbol. In either case, name can be used and treated like a constant value.

You must be careful in the placement of the EXTRN directive in relation to the definition of the program segment. If you know the segment in which the external symbol was defined as PUBLIC, place the EXTRN directive between a SEGMENT/ENDS pair that is identical to the SEGMENT/ENDS pair in which the object was defined in the other module. An external symbol defined in this manner will be addressable through the segment register containing the segment in question. In particular, a NEAR label defined EXTRN must be defined in segment identical to the one where it is defined PUBLIC because of the NEAR type restrictions.

Example:

In module "A"

```
DATA SEGMENT WORD PUBLIC
COUNT DB 0                ;declared as byte through DB
PUBLIC COUNT
DATA ENDS
```

In module "B"

```
DATA SEGMENT WORD PUBLIC ;different module, but same
EXTRN COUNT:BYTE          ;segment declaration
;typed as byte
DATA ENDS
```

If you do not know the segment in which the external symbol is defined, or if the segment in which it is defined is non-combinable, place the EXTRN directive outside of all SEGMENT/ENDS pairs in your program. To address the external symbol you must load the segment part (paragraph number) of the symbol into a segment register using the SEG operator and then either use an ASSUME directive to verify addressability or use a segment override for each use of that symbol.
The 8086/186 linker does NOT verify that the definition of an external symbol matches the definition of its resolving public symbol. It is up to the user to make sure that external symbol definitions are placed within the correct segment or they should NOT be placed in a segment at all.

Example:

```
MOV AX, SEG COUNT
MOV ES, AX ;loads segment

(then)

ASSUME ES:SEG COUNT ;verify addressability
MOV DL, COUNT ;use symbol
(or)
MOV DL, ES:COUNT ;use segment override
```
GROUP

The GROUP directive is used to specify several logical segments that are to be placed in the same physical segment.

Syntax:

```
name GROUP segpart [, ...]
```

Where:

- **name** is a mandatory, unique, user-defined name for the group.
- **segpart** is one of the following:
  - A segment name.
  - The keyword SEG followed by the name of a previously-defined variable, label, or external symbol. This construct refers to the segment in which the specified symbol lies. For externals, this may not be discovered until link-time.
  - An undefined symbol that must be defined later in the program as a segment name or the assembler reports an error.

Description:

At assembly-time you may specify that certain logical segments will all go in the same physical segment so the assembler will know that all such segments may be accessed from the same segment register. Such a collection of segments is called a group. The ordering of the segments in a GROUP directive will not necessarily control or represent the ordering of the segments in memory nor are the segments in a group necessarily adjacent in memory. GROUPing them only implies that they should lie within the same physical segment.

The total address space covered by all segments in a group must be less than or equal to 64K bytes. The size of the group is equal to the sum of the sizes of all segments in the group. The assembler does not check whether the size of the group is greater than 64K bytes, but the loader does.

A group has a base address. The base address of a group refers to the lowest memory address of any segment in that group. The loader sets the group base
address, and all segments in the group are addressable from this same group base address.

Forward references to group names are not allowed.

One of the uses of the group name is with the ASSUME directive. If, for example, you have defined many different data segments that you intend to form into one physical segment when the program is located in memory, you could combine these segments with the GROUP directive. Since the contents of all these data segments will be addressable through DS during the execution of the program, you may use the group name in the ASSUME and to initialize DS. For example,

```assembly
DATAGRP GROUP DATA1, DATA2 ;DATA1 and DATA2 not defined yet

DATA1 SEGMENT
   ABYTE DB 0
DATA1 ENDS

DATA2 SEGMENT
   AWORD DW 0
DATA2 ENDS

ASSUME DS:DATAGRP, CS:CODE ;use group name in ASSUME

CODE SEGMENT
   MOV AX, DATAGRP ;AX = base address of group
   MOV DS, AX ;initialize DS
   MOV AX, AWORD ;addressable through DS
   ...
CODE ENDS
```

**Use of the OFFSET Operator With Groups**

When using the OFFSET operator with a variable or label whose segment is in a group, you must use the group name as a segment override in an expression which references that variable or label, as in

```assembly
MOV BX, OFFSET DATAGRP:COUNT
```

Also, if you wish to store the paragraph number of a variable or label defined with a group, you must use a group override. Otherwise, the paragraph number of the segment that contains the variable is stored instead. Example:

```assembly
DW SEG DATAGRP:COUNT
DD DATAGRP:COUNT
```
LABEL

The LABEL directive is used to create a name for the current location of assembly, whether it is data or code.

Syntax:

```
name LABEL type
```

Where:

- **name** is a unique user-defined symbol. Its associated attributes are the following:
  - **segment** - current segment
  - **CS-assume** - current CS-assume value (labels only)
  - **offset** - current location counter
  - **type** - as specified below

- **type** is one of the following:
  - The keyword **BYTE, WORD, DWORD, QWORD, or TBYTE** to create a variable which is one of these types.
  - **A structure name** Creates a variable whose type is equal to the number of bytes allocated in a structure definition.
  - **A record name** Creates a variable whose type will be either byte or word depending on the record definition.
  - **NEAR** or **FAR** To create a label of type near or far.
Chapter 6: Assembler Directives

LABEL

**Description:**

The LABEL directive and the idea of a "label" should not be confused. The LABEL directive creates a label or variable at the current location being assembled. A label is a name for a location in the code that can be JMPed to or CALLed.

The LABEL directive is used primarily to address the same data item or same piece of code as different types. As a rule, as86 requires that the type of a data reference match the type of the data definition (known as strong typing), which makes this dual addressing difficult. If you want to access a variable either as a word or as 2 bytes depending upon the context, the following would allow you to do so:

```assembly
WORDNAME LABEL WORD
LOWBYTE DB 0
HIBYTE DB 0
```

The LABEL directive also allows you to define two labels of different types (for instance, both NEAR and FAR) but be careful that the right RET is executed for the type of CALL made. The following (potentially fatal) example illustrates this use:

```assembly
AFARLABEL LABEL FAR
NEARLAB: MOV AX, BX
;would be near, so some information
;would be left on the stack
RET
```

as86 does not, in general, permit data storage at label locations—that makes writing self-modifying code difficult.
NAME

The NAME directive is used to assign a name to an object module.

Syntax:

NAME module_name

Where:

module_name is a user-defined identifier. The name identifier can be any length, but only the first 40 characters are meaningful.

Description:

Every object module produced by as86 has a name; if you do not provide one, the assembler issues a warning and gives the file a special name. The special name is the source file base name stripped of any path and suffix. A module name is not stored as a symbol. You can therefore duplicate a keyword or a user-defined label without conflict. Module names are not affected by the case control. They are always case-sensitive.

The linker does not require that modules have unique names, but it identifies its input files by module name on its listing map. For this reason, assign each module a unique name for clarity.

The librarian program does identify its modules by name. Every module used as input to the librarian must have a unique name or an error will result.
Chapter 6: Assembler Directives

ORG

---

**ORG**

The ORG directive is used to alter the value of the Location Counter within the current segment.

**Syntax:**

```
ORG expression
```

*expression* evaluates to

- an absolute number (modulo 65536) that does not contain forward references or
- an offset relocatable number (modulo 65536) that is only relocatable from the current segment. Using the offset of `$` (dollar sign is the special character for the current location counter value) in a PUBLIC segment is an example of this form of ORG.

**Description:**

The ORG directive is used to locate code or data at a particular location (offset) within a segment. Using ORG with an absolute segment allows you to specify an actual memory location at which the code or data will be located.

**Note**

Avoid expressions of the form

```
ORG OFFSET ($-1000)
```

since this particular expression will overwrite your last 1000 bytes of assembly (or will re-ORG high in the current segment if the expression evaluates to a negative number). An expression with the syntax "$+ 1000" will produce an error because this expression evaluates to a label, not to a number. To achieve what is intended, the expression "OFFSET ($+ 1000)" can be used.
The PROC/ENDP directive pair is used to delimit a section of code which can then be called from elsewhere in the program, much like a procedure in a high-level language.

Syntax:

```
name PROC [type]
.
.
(instructions)
.
.
name ENDP
```

Where:

- **name** is a unique user-defined symbol providing a label for the beginning of the PROC. The name on the ENDP directive must match that on the most recently defined PROC for which an ENDP was not already encountered. The ENDP directive signals the end of a PROC definition to the assembler. The attributes of the PROC name are the following:
  - **segment** - current segment
  - **CS-assume** - current CS-assume
  - **offset** - current location of PROC directive
  - **type** - depends on type indicated
  - **relocation type** - depends on enclosing segment

- **type** is the type of the label defined at the beginning of the PROC. Type can be NEAR or FAR. NEAR is the default if no type is specified.
Description: The primary use of the PROC/ENDP pair is to give a type to the RET instruction enclosed by the pair. A RET instruction generates a NEAR return or a FAR return depending on whether the most recently defined PROC is NEAR or FAR. A RET or IRET outside of a PROC/ENDP pair or inside a pair which has no type specified is, by default, of type NEAR. Therefore, any code you wish to CALL FAR and then successfully RET from should be enclosed in a PROC/ENDP pair typed FAR.

Code execution begins at the instruction immediately following the PROC Directive when PROCs are CALLED or JMPed to.

Nested PROCs

When a PROC is defined inside another (nested), it does not necessarily have the same type assigned to its RET or IRET instruction as does the enclosing PROC. For instance, an enclosing PROC may be typed FAR. When the next PROC occurs, it might be a NEAR. For the duration of that PROC until the ENDP, the type of any return instruction will be NEAR and not FAR. When the ENDP is found for the nested PROC, however, the type reverts to the type of the enclosing PROC, in this case FAR. Having a NEAR PROC inside a FAR PROC, then, does not affect the enclosing PROC.

Differences Between PROCs and "Subroutines"

The code in a PROC/ENDP pair is not a procedure in the same sense as it is in high-level languages. A few differences are of note:

- In contrast to the scoping of names in block-structured languages, all labels and variables within the PROC/ENDP pair are not local to the "subroutine", but are global to the entire file.

- It is possible for execution to "fall into" a PROC from the previous instruction; it is not necessary to CALL a PROC to execute it. Executing a RET or a IRET from a "fallen into" PROC can cause unpredictable results.

- The ENDP does not function as a return-from-procedure; it marks the end of the PROC for the assembler. It is possible for execution to "fall out of" a PROC through the ENDP into the next instruction. To return from a CALL, a RET or IRET instruction must be used.
The PUBLIC directive is used to specify symbols, defined in one module, that are available to other modules at link time.

**Syntax:**

```
PUBLIC name [, ...]
```

**Where:**

- `name` is the name of the symbol defined in the current module.

**Description:**

Symbols designated PUBLIC will be placed in the object file and used by the loader to resolve external references (made with the EXTRN directive) from other modules.

PUBLIC symbols must be variables, labels or 17-bit constants defined by using EQU; any other types will generate an error. A 17-bit constant can be absolute or offset relocatable only; other relocation types are not allowed.
PURGE

The PURGE directive places a flag on the specified user-defined symbol in the symbol table so that the symbol is no longer recognized.

Syntax:

PURGE symbol [, ...]

*symbol* can be any keyword or user-defined symbol, *except*
- *register names*
- *segment names* (including ??SEG).
- *group names*
- *hands-off keywords* (see keyword list in chapter titled "Assembler Syntax")
- *any user-defined symbol that appears in a PUBLIC statement*

Description:

A PURGEd symbol can be redefined following the PURGE statement. A reference to the symbol following the PURGE statement, but before a re-definition, is treated as a forward reference to the second definition. If a PURGEd symbol is never redefined, references to the symbol following the PURGE statement are considered errors (reference to undefined symbol).

Purging symbols does not physically remove them from the symbol table and therefore PURGE cannot be used to deal with symbol table overflow.

If a variable or label that is defined in the current module but does not appear in a PUBLIC or EXTRN statement (that is, a local symbol) is purged, it will not appear in the object module. A PURGE directive, placed just before the END statement can—in combination with the $DEBUG assembler control statement—be used to pass on only a few selected symbols for debugging purposes.

Any variable, label or absolute number that was defined by an EXTRN statement can be purged, but the symbol will still appear in the object module as an external reference.
If a symbol is defined by an EQU to another symbol (not an expression), a PURGE on one of the symbols can cause unexpected results. The rule is that if a symbol in an EQU chain is PURGEEd, it and all symbols that precede it to the beginning of the chain are also PURGEEd.

Given the EQU chain that follows:

```
A EQU B
B EQU C
C DW 0 ;EQU chain resolving at C
```

The following PURGEs, which should not be considered as sequential code but as separate lines somewhere in the assembly source program, would have the described effects.

<table>
<thead>
<tr>
<th>PURGE A</th>
<th>;purges only A (B and C are still defined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURGE B</td>
<td>;purges A and B (C still defined)</td>
</tr>
<tr>
<td>PURGE C</td>
<td>;purges A, B, and C</td>
</tr>
</tbody>
</table>
RECORD

The RECORD directive defines a record template.

Syntax:

name RECORD recfieldname:nnn[=datum] [,...]

Where:

- name is a mandatory user-defined name for the record template.
- recfieldname is a mandatory user-defined name for a bit field.
- nnn is an integer constant, or an expression containing no forward references, that evaluates to an absolute number. The range of nnn is from 1 to 16, inclusive, and denotes how many bits will be in a bit field. Bits are counted from high bit to low bit within the full byte or word. Thus, the first bit field following the RECORD keyword is the most significant field of the record.
- datum is an optional integer constant, or an expression containing no forward references which evaluates to an absolute number, specifying a default value for this bit field. This value can be overridden when the record is allocated. If no datum is present, zero is the default. If the datum is present, it must fit into the number of bits specified (nnn), zero-filled. For example, the legal default values for a 1-bit field are 0 and 1. Values that are either negative or too large are truncated to fit within a given field. A warning is also generated.

Description:

The RECORD directive always defines a 1-byte or 2-byte template. This definition only describes a record; it does not allocate any memory at definition time. If the total number of bits in a record template is one to eight, the unit used to allocate storage when the record template is used is 1 byte. If the number of bits is 9 to 16, then allocation is 2 bytes.

You might experience some confusion in those cases where the bit field allocation does not fill exactly 8 or 16 bits. Although bit counting begins with the most significant bit in cases where the byte or word is completely filled, partially allocated records (the number of bits in the bit fields do not total exactly 8 or 16 bits) will have their bit fields right-justified in the byte or word.
and the remaining most significant bits will be zero-filled. This means that the first bit in the left-most bit field where counting begins will not be the left-most bit of the byte or word. The following definition

```
REC1 RECORD R1:3=7,R2:5 ;generates 11100000B or E0H
```

defines an 8-bit pattern which has all 8 bits filled. Note that R2, because it is not initialized, is set to zero by default. However, the definition

```
REC2 RECORD R3:3=7,R4:3=3 ;generates 00111011B or 3BH
```

leaves two bits remaining in an 8-bit byte. The two three-bit bit fields are right justified, and the remaining two bits, the two most significant bits, are zero-filled. The following figure illustrates how, for the above example of record template REC2, the partial record is defined by the RECORD directive.

![Figure 6-1. "Partial" Record Definition](image)

Similarly, the two 16-bit record definitions below illustrate what happens to 16-bit partial records.

```
REC3 RECORD R5:3=7,R6:13=4095 ;generates 1110111111111111B or 0EFFFH
REC4 RECORD R7:1=1,R8:8=127 ;generates 0000000010111111 or 017FH
```

Remember, the RECORD directive only defines a template, it does not allocate storage. To see how to allocate storage using a record template, read the next section.

### Allocating Record Storage

After you have defined a record template, the template definition can be used in the following syntax to allocate storage:

**Syntax:**
Chapter 6: Assembler Directives

RECORD

[name] recname <[[datum],] [..]> (or) [name] recname repeatval DUP (<[[datum],] [..]>)

Where:

• name is an optional name to be declared as a variable with the following attributes:
  – segment - current segment being assembled
  – offset - current location counter value
  – type - total number of bytes in the record template (either 1 or 2)

• recname is the name assigned to a previously-defined record template
  repeatval is a 17-bit integer constant, or an expression containing no forward references and evaluating to a 17-bit absolute number, between 1 and 65535 inclusive. Repeatval specifies the number of copies of the record to allocate.

• datum is an optional value to be used instead of the default value provided in the template. All such values must be 17-bit integer constants, or expressions that evaluate to 17-bit absolute numbers. Relocatable values are not allowed.
  – The first datum replaces the default value of the first bit field within the record, the second datum replaces the default on the second bit field, etc. Null data items are permitted (separated by commas) to direct the assembler to use the default values; null data values are useful when a default value other than the first needs to be overridden. If a field is mmm bits wide, the least significant mmm bits of the two's complement representation of the datum are used. For example, if a 3-bit field is being overridden, values of 6, -2, and 14 will all generate the 3 bits 110.

Examples (using the REC1 definition shown above):

FIRSTREC REC1 <> ; no overrides to defaults, generates 0E0H
SECNDREC REC1 <4> ; overrides R1 - generates 080H
THIRDPREC REC1 <,5> ; overrides R2 - generates 0E5H
FIVERECS REC1 5 DUP (<>); 5 copies of default record

It is allowable to nest record allocations up to 10 deep.
SEGMENT/ENDS

The SEGMENT/ENDS directive pair is used to define a logical segment.

**Syntax:**

```
name SEGMENT [align-type][combine-type][‘classname’]
```

```
name ENDS
```

**Where:**

- **name** is a mandatory user-defined name that cannot conflict with any other symbol.
- **align-type** specifies what boundary the logical segment must be placed on. If the align-type is not specified, PARA is the default. Align-type may be any of the following keywords:
  - **BYTE** - byte alignment. Segment can start anywhere.
  - **WORD** - word alignment. The segment must start on an address divisible by 2. (An address which has a least significant bit of 0.)
  - **PARA** - an address divisible by 16. (An address which has its least significant hexadecimal digit equal to 0H.)
  - **PAGE** - page alignment. The segment must start on an address divisible by 256. (An address which has its two least significant hexadecimal digits equal to 00H.)
  - **INPAGE** - inpage alignment. The entire logical segment cannot be more than 256 bytes long; it cannot cross a page boundary (an address divisible by 256). It will be moved to start on an address divisible by 256 only if movement is necessary to prevent the segment from crossing a page boundary.
- **combine-type** specifies the way in which the linking loader combines this segment with other logical segments of the same name to form a physical segment in memory. If combine-type is not specified, the logical segment will not be combined with any other logical segment, not even one with the same name from a different module. Combine-type can be any of the following keywords:
Chapter 6: Assembler Directives

SEGMENT/ENDS

- **PUBLIC** - all segments of the same name defined to be PUBLIC will be concatenated to form a single physical segment. The loader controls the order of concatenation. The length of the resulting physical segment will be equal to the sum of the lengths of the segments that have been combined.

- **COMMON** - all segments of the same name defined to be COMMON will be overlapped, starting at the same physical address, to form a physical segment. The size of the resulting physical segment will be equal to the size of the largest segment of those overlapped.

- **STACK** - all segments of the same name defined to be STACK will be concatenated into a physical segment such that the combined segment will end at a certain physical address (overlaid against high memory) and will grow "downward." The length of the resulting segment will be the sum of the lengths of the combined segments. (STACK is not a true keyword. You can define a symbol to be STACK without conflicting with the usage in the SEGMENT directive.)

- **MEMORY** - all segments of the same name defined to be MEMORY will be combined so that the first memory segment encountered by the linker will be treated as the physical "memory" segment. In the list of linked modules, the first module that contains a "memory" segment will be used to define the physical "memory" segment. It will be located at an address above all other segments in the program. Any other segments of the type memory that are encountered by the linker will be treated as common with the first segment. The length of the physical memory segment will be equal to the length of the first memory segment encountered (Memory, like Stack, is not a true keyword. You can define a symbol to be MEMORY without conflicting with the usage in the SEGMENT directive).

- **AT nnn** - this segment will be placed at the paragraph number specified. The expression nnn cannot contain forward references and must evaluate to an absolute number. Absolute segments are not aligned by the linker; the various align-type keywords are syntactically correct when used in combination with AT but are ignored. Note that nnn represents a paragraph number, not an actual address; therefore if AT 0444H is specified, the segment will start at address 04440H. A segment created with AT will be non-combinable with segments from other modules.

'classname' is a name that may be used to indicate that segments are to be located near each other in memory. When assigning physical addresses to
these logical segments, the linking loader attempts to place logical segments with the same classnames close together. However, the classname cannot be used to combine segments such that they may be addressed through the same segment register.

The classname must be enclosed in single quotes, as shown, or in double quotes.

Classnames are not stored as symbols; they may duplicate symbol names (even keywords) without conflict. If a classname is to be assigned to a segment, assign it at the first occurrence of the segment in the source file.

**Description:**

The SEGMENT/ENDS directive pair is used to define a logical segment. This segment may be combined with other segments of the same name defined in either the same module or in other modules. These logical segments will form the physical segments, located in memory, that are pointed to by the segment registers. Within a source module, each occurrence of an equivalent SEGMENT/ENDS pair (with the same name) is viewed as being one part of a single program segment.

**Multiple Definitions of a Segment**

The assembler keeps the value of the offset from the current segment (i.e., the most recent SEGMENT directive) in an internal location called the location counter. The assembler saves the location counter for each segment when it finds an ENDS for that segment, or if it finds a new SEGMENT directive. Later, if the assembler finds another SEGMENT directive which uses the name of that previously defined segment, the earlier location counter is retrieved and used. For this reason, a segment may be broken into pieces within a module, or between modules if it is combinable, and those pieces will still be placed in the same physical segment.

The align-type, combine-type and classname need not be included with the second and later SEGMENT directives for a segment of the same name. If they are absent, the assembler takes the segment’s characteristics from the first definition. However, any keywords that are present must match the first definition, or an error is reported. If an absolute segment is broken into pieces and the AT keyword is used on a SEGMENT directive for the second or later piece, the absolute base address must match the first definition, even though the location counter is taken from the stored value. The second part of the segment will not start at the specified base address, but the AT value must match. Examples of breaking a segment:
Nested or Embedded Segments

It is legal to nest SEGMENT/ENDS pairs. Each ENDS must refer to the most recently-defined SEGMENT whose ENDS was not yet encountered. The fact that a segment is nested inside another does not mean that the code for the nested segment is placed inside the enclosing segment. The code is the same as it would be if no nesting occurred. Nesting helps you to define logical structures to make programming easier. Example:

Improper Nesting:

Maximum Number of Segments

If you use the default HP-OMF 86 object file format, you may use an unlimited number of segments. The HP 64000 (.X) object file format allows
only three named segments. Therefore, if you use the HP 64000 object file format (the -h command-line option), use three or fewer relocatable segments per module.

The first relocatable segment with code will be assigned the PROG segment. The first relocatable segment with data will be assigned the DATA segment, if that segment is not used for PROG. The next relocatable segment, whether it contains code or data, will be assigned the COMN segment.
The STRUC/ENDS directive pair is used to define a structure template.

**Syntax:**

```
name STRUC
.
.
<data directives>
.
.
nname ENDS
```

**Where:**

- **name** is a unique user-defined symbol that becomes the structure name. The name on the ENDS must match the name on the STRUC. Its type attribute is the following:
  - **type** - number of bytes defined in structure data directives

**Description:**

The structure definition only describes a given structure and its contents; it does not allocate any memory at that time. All statements between the STRUC and ENDS directives must be one of the following: DB, DW, DD, DQ, or DT directives, comment lines, blank lines, or assembler controls. Any assembler controls that are included within the STRUC/ENDS pair are not stored as part of the template and therefore are not executed anew each time the structure is referenced. Any symbols referenced in the argument field of any of the included directives must have been previously defined. Forward references are not allowed within a structure definition.

You will notice that the ENDS directive is also used to terminate a SEGMENT definition. This is unambiguous, since an ENDS closing a SEGMENT is not legal within a structure definition.

If a DB or other directive within a structure definition has a name in its name field (which must be unique, and cannot previously have been the object of a forward reference), this name is known as a structure field. It is not the same as a variable, and it is not associated with any particular storage location or segment. Structure names and structure fields can be used in very few syntactic
constructs. Forward references to structure names and structure fields are not allowed.

Structure field names do have associated attributes. They follow:

- **offset** - offset from the beginning of the structure definition
- **type** - type of data definition directive

## Allocating Structure Storage

After you have defined a structure template, it can be used in the following syntax to allocate storage:

**Syntax:**

```
[name] strucname <[[datum],] [...]>
(or)
[name] strucname repeatval DUP ( <[[datum],] [...] > )
```

**Where:**

- **name** is an optional name to be declared as a variable with the following attributes:
  - **segment** - current segment being assembled
  - **offset** - current location counter value
  - **type** - total number of bytes in the structure template
- **strucname** is the name assigned to a previously defined structure template.
- **repeatval** is a 17-bit integer constant, or an expression containing no forward references and evaluating to a 17-bit absolute number between 1 and 65535 (inclusive); it is the number of copies of the structure to allocate.
- **datum** is an optional scalar to be used in place of the default value provided in the template. The first datum replaces the default value on the first data definition directive within the structure, the second datum replaces the default on the second data definition directive, etc.
  - Null data (separated by commas) is permitted and directs the assembler to use the default value; this is useful when a value other than the first occurring value must be overridden. The legal values for these scalars are the same as in the data definition directive to which they apply, including the
indeterminate-initialization keyword ‘?’ . Note that repeated data (i.e., DUP expressions) cannot be used as an override.

– Not every default value can be overridden. Default values can be replaced only if the template defined just one unit of data for the data definition directive (structure field) that is to be overridden, or the template defined a character string in a DB directive. These conditions mean that such defaults as DB 1,2 and DW 10 DUP (0) cannot be overridden.

The number of bytes used in a DB string is fixed when the structure is defined. Such a string can be overridden only by another string. If a longer string is used to override, it is truncated, and a warning message is given. If a shorter string is used to override, it is filled out, using the characters at the end of the default string.

The structure definition

<table>
<thead>
<tr>
<th>STRUC/ENDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLUEPRINT</td>
</tr>
<tr>
<td>FIRST DW</td>
</tr>
<tr>
<td>0FFFEH</td>
</tr>
<tr>
<td>SECOND DW</td>
</tr>
<tr>
<td>BUFFER</td>
</tr>
<tr>
<td>THIRD DB</td>
</tr>
<tr>
<td>7, 5</td>
</tr>
<tr>
<td>FOURTH DB</td>
</tr>
<tr>
<td>‘A’</td>
</tr>
<tr>
<td>FIFTH DB</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>SIXTH DW</td>
</tr>
<tr>
<td>257</td>
</tr>
<tr>
<td>BLUEPRINT</td>
</tr>
<tr>
<td>ENDS</td>
</tr>
</tbody>
</table>

yields a structure template like this:

<table>
<thead>
<tr>
<th>.FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>.SECOND</td>
</tr>
<tr>
<td>.THIRD+1</td>
</tr>
<tr>
<td>.FIFTH</td>
</tr>
<tr>
<td>.SIXTH</td>
</tr>
</tbody>
</table>

Figure 6-2. Structure Definition and Allocation
The instruction
   B1 BLUEPRINT < >
allocates storage for B1 that looks like:

15

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
</table>

OFFSET (BUFFER)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th></th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>indeterminate</td>
<td></td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   | 0 | 1 | 0 | 1 |

The instruction
   B2 BLUEPRINT < ,0,,,255>
allocates storage for B2 that looks like:

15

<table>
<thead>
<tr>
<th>0</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
</table>

|   | 0 | 0 | 0 | 0 |

| 0 | 5 |   | 0 | 7 |

| F | F |   | 4 | 1 |

|   | 0 | 1 | 0 | 1 |

*Figure 6-2. Structure Def. and Allocation (Cont’d)*
Chapter 6: Assembler Directives

STRUC/ENDS
Expressions

The syntax and semantics of expressions.
This chapter describes the syntax and semantics of expressions. The early part of the chapter explains the kinds of expressions and discusses expression operands. The latter part lists the different expression operators and their uses. The end of the chapter has a table showing the precedence ranking of the expression operators.

Reference Syntax Conventions

The sections that include the references about the expression operators follow certain conventions:

1. The name of the operator (or a descriptive term for the operator) appears in the lefthand column.
2. The proper assembler syntax appears next under a heading of "Syntax."
3. A short description follows the syntax. The description explains the syntax and any arguments appearing in the syntax. There may also be other information relating to the operator itself or to using the operator.
4. Some expression operators may affect the attributes (see the "Symbol and Expression Attributes" chapter) of its operands. If that is so, a list of attributes and their values follows the description.
5. Some short examples that use the operator may follow the description or attributes sections.

Expression Overview

An expression is a simple or complex combination of operands that may be bound by operators. Operands can be numeric values or address expressions. Operators include conventional unary and binary arithmetic operators (+, -, *, /, MOD, etc.), logical operators (AND, OR, XOR, NOT), or special operators such as memory and record operators.

Expressions have certain attributes. Attributes are discussed thoroughly in the chapter named "Symbol and Expression Attributes."

Expressions are in turn used as operands to assembly language instructions and assembler directives. Expressions may be absolute, relocatable, or external.
Absolute Expression

An absolute expression is one whose value is known completely at assembly time. Assembly of absolute expressions results in object code that does not need to be further modified by the loader. An absolute expression will have an operand that is

- a numeric constant
- a constant memory expression (addresses which are known at assembly time)
- record allocation values
- a record bit field offset
- a segment base located during assembly time with the AT keyword (AT is discussed in the SEGMENT/ENDS directive in the "Assembler Directives" chapter)
- an offset for a variable or label from a segment which is non-combinable
- a register name
Relocatable Expression

A relocatable expression contains a relocatable operand as part of the expression. The value of a relocatable expression is not known at assembly time and must be assigned later by the loader. Relocatable expression values are 16-bit values unless modified by the HIGH or LOW operators to become 8-bit values. A relocatable expression will have an operand that is

- a segment base where the segment is combinable (including all groups, since their bases are not set until load time)

- a variable or label which belongs to a combinable segment

External Expression

An external expression is a relocatable expression which contains items that are not within the module being assembled. These expressions reference external variables, labels, or numbers. Their values must be assigned by the loader when the module containing the referenced item is available for relocating. External expressions, like relocatables, are assumed to be 16-bits in size, but may be modified with the HIGH or LOW operators to be 8-bit values. More information about external references appears in the chapter titled "Assembler Directives."

During the assembly process, the assembler uses 17-bit numbers to perform arithmetic and other operations involving expressions. A 17-bit number is a 16-bit number with an additional sign bit. The 17-bit number is used within the assembler so that negative numbers with large absolute values (to -65535) may be used in calculations. When the value is coded, the sign bit is discarded and is not output, since only 16-bit values are used in the object code.
Expression Operands

An expression may consist of only an operand, or operand(s) modified by one or more operator(s). Operands are broadly divided into two groups: numeric values and memory or register expressions. A numeric value will be directly represented in the assembled code. A memory or register expression is an indirect value because the assembler is coding a reference—or reserving a space that will be filled later—which points to a location in memory where the actual data resides. Expressions involving the EQU directive can be either a numeric or memory expression.

Numeric Values

Numeric values result from a variety of different operands. Numeric constants, obviously, are numeric values, but other, less clearly numeric operands also produce numeric values. Any of the following operands can generate numeric values:

- **A constant.** There are several ways that an absolute number, or constant, may be represented to the as86 assembler. The easiest and most straightforward way is to make the expression operand a decimal, octal, hexadecimal, or binary number. The various representations are as follows:

  - A decimal number is a series of digits, ranging from 0 to 9, that optionally ends with the character 'D'. Decimal numbers are base-10 and are the numbers people are most familiar with.

  - An octal number is a base-8 number represented by a series of digits, ranging from 0 to 7, and ending with either the character 'O' or 'Q'.

  - A hexadecimal number is a base-16 number represented by a series of digits, ranging from 0 to 9, or by characters, ranging from 'A' to 'F' (or 'a' to 'f'). These numbers must end with the character 'H'. A hexadecimal number may not begin with a character; in those instances, place a leading zero in front of the hex number.

  - A binary number is a base-2 number represented by a series of digits, either 0 or 1, and ending with the character 'B'.

Examples of numeric constants:
Chapter 7: Expressions
Expression Operands

MOV AX, 35 ;decimal number
MOV AX, 12D ;decimal number with optional ;following 'D'
MOV AX, 37O ;octal number with the letter 'O'
MOV AX, 12Q ;octal number with following 'Q'
MOV AX, 12H ;hexadecimal number
MOV AX, 0A34H ;hexadecimal number with leading 0
MOV AX, 0110101B ;binary number

- **Quoted string.** A one or two character quoted string which is used as an expression operand will be stored as a hexadecimal number in a two byte word. Each byte contains the ASCII value of the character it stores. If two characters are stored in a word, the first character is represented in the high byte of the word and the second character is represented in the low byte. If only a single character is stored, it is represented in the low byte and the high byte is set to 00H. A quoted string always evaluates to a positive 17-bit value. This method of representing numbers is cumbersome and not very useful. It is also much more difficult to verify that the value is correct. Examples:

MOV AX, 'A#' ;generates 04123H
MOV AL, HIGH 'B' ;generates 00H

- **Record template.** The chapter titled "Assembler Directives" discusses the record structure. A record is a series of bit fields which may be defined within a one or two byte structure called a template. Template definition does not allocate storage, but specifying an occurrence of a record can allocate memory, much like a DB (define byte) or a DW (define word) directive might allocate memory. A record template may also be used as an expression operand, but in this usage no memory is allocated. Instead, the operand is evaluated to be a positive 17-bit value and used the same as any number.

Examples:

R1 RECORD F1:3, F2:5, F3:2 ;the RECORD directive
MOV AX, R1<> ;defines record template ;no defaults specified ;in template definition
MOV AX, R1<2,14,3> ;value is 0013BH
MOV AX, R1<2,14,3> + 5 ;value is 00140H

- **Record field.** You may also use a record field name by itself as an expression operand. If the field name is used without a MASK or WIDTH operator, then the assembler replaces the field name with a number which is the shift value required to move the lower bit of its bit field to the 0th bit position. For example, using the record template definition above, the
value that would be replaced for F1 is 7 since there are 7 bits of data to the right of the field F1. The shift value, combined with the MASK operator described later in this chapter, may be used to extract field values from a record.

- **Segment or group name.** When used as an expression operand, the name becomes an immediate value that is the paragraph number for the segment or group. Since most segments and all groups are not assigned this value by the assembler, it will usually be relocatable. Only segments that use the AT keyword will have a fixed paragraph number known by the assembler. These values may be used as is—to initialize a segment register, for instance—or used wherever a relocatable number may be used (except with HIGH and LOW). Examples:

```
MOV AX, SEG1 ;load paragraph number for segment
MOV DS, AX   ;initialize DS register
MOV AX, GRP1 ;load paragraph number for group
```

## Memory and Register Expressions

There are several ways to reference memory in assembly source files. Memory might be referenced with operands which are any of the following:

- **Variables or labels.** Variables are defined through data directives and structure or record allocations. Labels are defined through assembly instructions or PROC directives. Either variables or labels may also be defined through EXTRN statements or LABEL directives. Given the variable and label definitions in the first three lines of the example below, the last two lines use those definitions as memory operands:

```
WMEM DW 2         ;word variable
R1 RECORD F1:3, F2:4 ;record template definition
U1 R1 <>          ;byte variable, from
                   ;a record allocation
L1: MOV AX, WMEM  ;NEAR label, using a word
                   ;variable
MOV AL, U1        ;uses byte variable as operand
JMP L1            ;uses NEAR label as operand
```

- **Variable with offset.** Variables used as memory operands may have offsets added to them in order to refer to memory locations near the memory location of the variable. The variable with offset operand may be expressed in two ways. Examples of both:

```
MOV AX, WMEM + 5  ;adds 5 to variable address
                   ;accesses memory 5 bytes higher
                   ;than location of variable WMEM
MOV AX, 5 + WMEM  ;same result from slightly different
                   ;way of expressing it
```
**Chapter 7: Expressions**

**Expression Operands**

MOV AX, WMEM[5] ; same result from very different ; way of expressing it

- **Structure field.** Much the same as using an added offset to a variable, using a structure field name as part of a memory operand allows access to memory that is near a variable. Offset is from the variable named when storage using the structure template was allocated. Using a structure field name as a memory operand also changes the type of the memory expression to that of the field. Example:

  ```
  ST1 STRUC
  BFIELD DB ? ; field offset value from ST1 is 0
  WFIELD DW ? ; field offset value from ST1 is 1
  ST1 ENDS
  MOV AX, BMEM.WFIELD ; adds 1 to offset, word type
  ```

- **Register indirect reference.** The 8086/186 processors also allow an instruction to indirectly refer to memory by using base and/or index registers. The contents of these registers are added to a variable’s offset at runtime, which means a memory address can be created that is not known when the assembly code is written. A register expression operand can contain one base register name, one index register name, or one base and one index register name. Additionally, constants may be part of the operand along with the registers.

  The valid base registers are BX and BP and the valid index registers are SI and DI.

  Base or index registers used this way must be enclosed in square brackets in a register expression, but there are several different ways to represent expressions given this restriction.

  - A base and index register may be added together explicitly by using a ‘+’ sign within the brackets or added implicitly by enclosing each register name in separate, adjacent brackets.

  - A base or index register alone may have a constant added to it or subtracted from it in the same manner. (The ‘-’ sign must be used for subtraction, since adjacent brackets are, by default, added.)

  - A base and index register added together may also have a constant added using either a ‘+’ sign or adjacent brackets, or a constant may be subtracted by using a ‘-’ sign within the brackets.

  - A base and index register cannot be subtracted from one another, however.
Examples:

MOV AX, WMEM[BX]  ;one base register, no index register

MOV AX, WMEM[BP][SI]  ;these two slightly different expressions are equivalent
MOV AX, WMEM[BP+SI]  ;both add one base register and one index register

MOV AX, WMEM[SI]  ;no base register, one index register

MOV AX, WMEM[5][BP]  ;both of these expressions use an index register with a constant added
MOV AX, WMEM[5+BP]  ;an index register with a constant subtracted, no index register

MOV AX, WMEM[BX][DI][5]  ;one base and one index register added
MOV AX, WMEM[BX+DI+5]  ;with constant added also; both expressions equivalent

- **Anonymous reference.** This form of register expression operand contains only constants and registers and does not include a variable or label name. Because there is no variable or label name, no segment or type information is inherent in the expression.

This expression may be given a type and segment, using the PTR and segment override operators. Otherwise, default values are assumed, depending upon the instruction and the registers that are used. If the base register BP is used, the default segment register is SS. Otherwise, the DS segment register is the default segment register.

A default type value may be assumed if other operands to the instruction provide enough information to limit the type of the memory expression. Otherwise, an error is generated. For a constant to be used as a memory reference, it must be typed with the PTR operator so the assembler knows to treat the value as such. Otherwise, the constant is treated as an immediate value.

Examples:

MOV AX, [BX]  ;default is DS segment
MOV AX, [BP][SI]  ;default is SS segment
MOV AX, ES:[BX]  ;segment is ES
MOV AX, DS: WORD PTR 5  ;segment is DS
MOV AX, [BX].WFIELD  ;default is DS segment
The EQU directive, discussed in the chapter titled "Assembler Directives," allows you to assign a value to a symbol. Some of the possible assignments include register names, variables, memory expressions, or constants. The symbol on the left side of the EQU directive may be used in an expression as an operand. The result is the same as if whatever appears on the right side of the EQU were used as an operand instead. Examples:

E1 EQU BX ; 8086 register
MOV AX, E1 ; register to register
MOV AX, BX ; same as MOV AX, E1

E2 EQU WMEM ; variable
E3 EQU BMEM[BP][SI] ; register expression
E4 EQU 037B2H ; constant
MOV AX, WMEM[E1] ; register from memory
MOV AX, E2[E1] ; register from memory
MOV AX, E4 ; immediate value into register
MOV AX, E4 / 5 ; immediate value into register
Expression Operators Introduction

Operators are functions that take one or more operands and return a new value. Operators are used to build expressions that cannot be defined strictly as simple operands. Use operators to add numbers, change the type of a memory expression, or to cause segment overrides. You may use a complex expression involving operators anywhere a simple operand may be used if the value returned by the complex expression is equivalent to the value of the simple operand.

Arithmetic Operators

The arithmetic operators conform to the commonly understood notions of these operators. Arithmetic involving these operators is done using the full 17-bit representation of the operands. Negative number results are stored, however, in twos complement form.

Unary Plus, Unary Minus

Syntax:

- Unary Plus:  + operand
- Unary Minus:  - operand

Description: The unary operators '+' and '-' each take a single operand and return a single value as the result. The '+' operator may be applied to an absolute or relocatable value and the result will be an absolute or relocatable value. The '-' operator may only be applied to absolute values. The result will be the 2's complement of the value. These operators may be thought of as being the binary operators '+' and '-' with a lefthand operand of 0. Examples:

- MOV AX, + 5 ; result is 5 or 00005H
- MOV AX, - 2 ; result is -2 or 0FFFEH
- MOV AX, + WMEM ; result is memory expression
Chapter 7: Expressions
Arithmetic Operators

Binary Addition, Subtraction

Syntax:

Addition: operand1 + operand2
Subtraction: operand1 - operand2

Description: The binary operators '+' and '-' each take two operands and return a single value as the result. If memory addresses are used, the offset from the segment base is the value used as an operand. The types of operands that are allowed and the types of the results are shown in the following table.

Table 7-1. Binary Plus and Minus Results

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSNUM</td>
<td>+,-</td>
<td>ABSNUM</td>
<td>ABSNUM</td>
</tr>
<tr>
<td>RELOCNUM</td>
<td>+,-</td>
<td>RELOCNUM</td>
<td>RELOCNUM</td>
</tr>
<tr>
<td>ABSNUM</td>
<td>+</td>
<td>ABSNUM</td>
<td>RELOCNUM</td>
</tr>
<tr>
<td>ADDR</td>
<td>+,-</td>
<td>ADDR</td>
<td>ADDR</td>
</tr>
<tr>
<td>ABSNUM</td>
<td>+</td>
<td>ADDR</td>
<td>ABSNUM</td>
</tr>
<tr>
<td>ADDR</td>
<td>-</td>
<td>ADDR</td>
<td></td>
</tr>
</tbody>
</table>

The shorthand words in the table mean the following:

ABSNUM = absolute number, constant
RELOCNUM = relocatable number (OFFSET, external ABS, SEG)
ADDR = memory address, possibly relocatable or external

Note that ADDR-ADDR is only valid if both memory addresses are either absolute or relocatable. They must also belong to the same segment so that their offsets are relative to the same base value. This allows the result to be absolute. Neither address may be of an external reference, since its offset is not known at assembly time. Examples:

EXTRN EXABS: ABS  ; declared labels - variables
MEMSTART DB ?
WMEM DW 2
MEMEND DW ?

MOV AX, 5 + 15  ; result is 20 or 00014H
MOV AX, 3 - 12  ; result is -9 or 0FFF7H
MOV AX, WMEM + 5  ; result is offset of WMEM + 5
Square Brackets

Syntax:

address [ data_or_reg ]

Description: Square brackets give base and/or index attributes to an address expression or create a new address expression. The square brackets must occur in pairs. Such pairs cannot occur within angle brackets. However, more than one pair of square brackets can occur in a single expression.

The contents of the brackets are very limited. The only valid register names that can be used are BX, BP, SI, and DI. The first two, BX and BP, are base registers and only one of the two can be present within an entire expression. The SI and DI registers are index registers and, like base registers, only one of these registers can be present within an entire expression. It is valid to have both a base register and an index register in an expression. It is also possible to place numeric constants within the brackets.

The above items can appear singly within square brackets, as in:

mov AX, wmem[BX][SI][5]

It is also valid to replace '[' pairs with a '+' sign, as in:

mov AX, wmem[BX+SI+5]

The only time a minus sign is valid within square brackets is to subtract a constant, as in:

mov AX, wmem[BX+SI-5]
Chapter 7: Expressions
Arithmetic Operators

The constant expression part of the square brackets modifies the offset value of any memory value that is also part of the expression. The base and index registers are used to denote indirect addressing as part of an expression. The contents of the indicated registers are added to any memory expression offset in the expression to create a final memory address.

A memory address is not required to be part of an expression which has square brackets as part of itself. For example, take the following expression:

```asm
mov AX, [BX][SI][5]
```

This expression represents a memory location that is 5 bytes past the sum of the contents of the BX and SI registers at the moment of execution for that instruction. The segment register used for this instruction would be the DS register. The SS register is used if the BP base register is part of the expression. It is also valid to specify a different segment register through the use of a segment override, such as:

```asm
mov AX, ES: [BX][SI][5]
mov AX, SEG1: [BX][SI][5]
mov AX, GRP1: [BX][SI][5]
```

. (Dot operator)

Syntax:

```
address '.' struc_field
```

Description: This operand accepts an address expression as its left operand and a structure field as its right operand. The result of the operation is an address expression whose offset is equal to the offset attribute of the left operand plus the offset of the structure field within its structure template (in bytes). The type of the resulting memory expression is the type of the structure field. All other attributes are derived from the left operand. This operator is convenient for addressing fields within memory that contains one or more occurrences of a given structure. For example, suppose a structure was defined like this:
STRUCNAME STRUC
BYTEFLD DB 0
WORDFLD DW 5 DUP (3)
           DT 3.14159
STRINGFL DB 'DEFAULT'
STRUCNAME ENDS

The offset of BYTEFLD, WORDFLD, and STRINGFL within this structure template are 0, 1, and 21, respectively. These structure field names can be used to reference fields within a structure in memory, as in:

DATABLOCK STRUCNAME<> MOV AX, DATABLOCK.WORDFLD ; WORD type MOV CL, DATABLOCK.BYTEFLD ; BYTE type MOV DI, OFFSET DATABLOCK MOV AX, [DI].WORDFLD ; indirectly referencing memory

It is not valid to use the dot operator immediately after a digit, due to the possible confusion with a real number. Instead, the operator must be separated from the digit by parenthesis, such as:

(DATABLOCK + 2).WORDFLD ; valid DATABLOCK + 2.WORDFLD ; illegal

**Multiplication, Division, Modulo**

**Syntax:**

- Multiplication: `absval * absval`
- Division: `absval / absval`
- Modulo: `absval MOD absval`

**Description:** These three operators each take two absolute values as operands and return a single absolute value. The `*` operator multiplies the two operands and returns the result. The `/` operator divides the first operand by the second operand. The `MOD` operator returns the value of the first operand modulo the second operand. Modulo division discards the integer quotient and returns a value that is only the remainder. For either straight division (`/`) or modulo division, the righthand operand cannot have a value of 0. Examples:

- `MOV AX, 5 * 3` ;result is 15 or 0000FH
- `MOV AX, (-2) * 5` ;result is -10 or 0FF6H
- `MOV AX, 5 / 2` ;result is 2
- `MOV AX, 13 MOD 3` ;result is 1
SHL, SHR

Syntax:

```
absval SHL shiftvalue
absval SHR shiftvalue
```

Description: The SHL and SHR operators shift the first operand bitwise by the value of the second operand. The SHL operator shifts bits to the left and SHR shifts bits to the right. Bits that are shifted to the left beyond the leftmost bit and bits that are shifted to the right beyond the rightmost bit are lost. Bits with a value of 0 are shifted in to fill.

All 17 bits, including the sign bit, are shifted. Thus both operands must be absolute values, and the result is also absolute.

For example, the statement

```ml
MOV AX, 1FFFFH SHR 3
```

places the value 3FFFH in the AX register. The binary values look like this:

```
1 1111 1111 1111 1111       (1FFFFH, before SHR 3)
0 0011 1111 1111 1111       (3FFFH, after SHR 3)
```

Notice that the sign bit (the leftmost bit) of the argument in the example was shifted in when the shift right occurred.

It is possible for a shift to produce the invalid 17-bit number -65536 (10000H), which is automatically converted to 0.

If the count is negative, the shift is performed in the opposite direction. If the magnitude of the count is greater than 16, the result is 0.

Some other shifted values:

- `MOV AX, 5 SHL 2` ;result is 20 or 00014H
- `MOV AX, 13 SHR 2` ;result is 3
- `MOV AX, 44 SHL 11` ;result is 24576 or 06000H
- `MOV AX, (-54) SHR 3` ;result is 16377 or 3FF9H
HIGH, LOW

Syntax:

HIGH operand  
LOW operand

Description: These operators take either an absolute value or relocatable memory expression as an argument and return a BYTE-sized value of the same type. HIGH returns the high byte of the operand, LOW returns the low byte.

If the operand is a memory expression, it cannot contain index or base register names.

Attributes: relocation type - high or low

Examples:

MOV AL, HIGH 01234H ;result is 012H
MOV AL, LOW 01234H ;result is 034H
MOV AH, HIGH WMEM ;result is high byte of offset
MOV AL, LOW WMEM ;result is low byte of offset

EXTRN EXTABS:ABS
MOV AL, HIGH EXTABS ;result is high byte of external number

The following identities apply to HIGH and LOW.

High (High X) = 0H
Low (Low X) = Low X
High (Low X) = 0H
Low (High X) = High X
Logical Operators

The logical operators return values that are the result of comparing operands. (NOT can be seen as an exception.) AND, OR, and XOR compare the bits of their operands while EQ, NE, ..., GE all compare the values of their operands.

**AND, OR, XOR**

**Syntax:**

```
absval AND absval
absval OR absval
absval XOR absval
```

**Description:** These operators each take two absolute values as operands and return a single absolute value. If \( n \) is used to identify any given bit of the result, bit \( n \) has its value set differently depending on the operator used. The following rules apply:

- The AND operator will set bit \( n \) of the result to 1 if bit \( n \) of both operands is a 1; otherwise bit \( n \) is set to 0.
- The OR operator will set bit \( n \) of the result to 1 if bit \( n \) of either operand is a 1; otherwise bit \( n \) is set to 0.
- The XOR operator will set bit \( n \) of the result to 1 if bit \( n \) of each operand is different; bit \( n \) is set to 0 if both bits are the same.

The operations are performed on full 17-bit values. Examples:

```
MOV AX, 035H AND 3145H ;result is 5
MOV AX, 035H OR 3145H ;result is 3175H
MOV AX, 035H XOR 3145H ;result is 3170H
```
Chapter 7: Expressions
Logical Operators

NOT

Syntax:

NOT absval

Description: The NOT operator takes an absolute value as its operand and returns an absolute value that is the one's complement of the operand.

The one’s complement is derived by toggling the bits of the operand. If bit \( n \) of the operand is 1, then bit \( n \) of the result will be 0. Similarly, if bit \( n \) of the operand is 0, bit \( n \) of the result will be 1. The operation is performed on full 17-bit values.

Since the bitwise complement of \( 0FFFFH \) is \( 10000H \) (-65536) (which is not a valid 17-bit value), NOT \( 0FFFFH \) is defined to be 0.

Examples:

MOV AX, NOT 1 ;result is 0FFFEH
MOV AX, NOT 55 ;result is 0FFC8H

EQ, NE, LT, LE, GT, GE

Syntax:

equal: operand1 EQ operand2
not equal: operand1 NE operand2
less than: operand1 LT operand2
less than or equal: operand1 LE operand2
greater than: operand1 GT operand2
greater than or equal: operand1 GE operand2

Description: These operators each compare their operands and return a value that depends upon the result of the comparison. The result will be 0 if the comparison is false and the value will be \( 0FFFFH \) if the comparison is true. The operands must both be absolute numbers, both be memory expressions, or both be segment base values. Memory expressions may not contain base or index register names, may not refer to externals, and must reside in the same segment. It is the offset portion of the memory addresses that are compared. Offsets and absolute values are compared using 17-bit arithmetic.
Examples:
MOV AX, 15 GT 3 ;result is 0FFFFH
MOV AX, WMEM EQ BMEM ;result is 00000H
MOV AX, SEG WMEM EQ A ;result depends upon whether
;WMEM lies within segment A

Chapter 7: Expressions
Memory Operators

**Examples:**
MOV AX, 15 GT 3 ; result is 0FFFFH
MOV AX, WMEM EQ BMEM ; result is 00000H
MOV AX, SEG WMEM EQ A ; result depends upon whether
; WMEM lies within segment A

---

**Memory Operators**

**SHORT**

**Syntax:**
SHORT label

**Description:** The SHORT operator takes a label as its operand. The SHORT operator assures the assembler that the label will be within 127 bytes of the current location counter. SHORT is mainly used with the JMP instruction, where a forward reference to a label can result in either a one-byte or two-byte displacement. The SHORT operator informs the assembler that a one-byte displacement may be used (which only requires one byte of storage) where otherwise a two-byte displacement would result in extra object code size. It is up to you to ensure that the label is within 127 bytes because an error occurs if it is not. Example:

JMP SHORT FWDLAB

**THIS**

**Syntax:**
THIS type

**Description:** The THIS operator takes a type name as an operator and returns a memory reference of the given type. The memory referenced will be for the current location and segment. The length of the memory will be 1. The valid types for the operand are BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, and FAR. The result of this operator may be used as either the right-hand side of an EQU (in which case it acts the same as a LABEL.
directive) or as a memory reference in an instruction (which would be a rare use). Note that THIS NEAR is the same as '$'. (Dollar sign is the special character used to represent the location counter.)

Attributes:

- **segment** - current segment
- **offset** - current location counter
- **type** - as defined
- **relocation type** - depends upon current segment
- **segment** - current segment if defining variable
- **CS-assume** - current CS assume value if defining label

Examples:

LAB2 EQU THIS FAR ;create FAR label
LAB1: NOP
DATAN EQU THIS WORD ;allow word accesses to bytes
DATAEB DB 1
DATABH DB 2

**PTR**

Syntax:

type PTR operand

**Description:** The PTR operator is used to either set or change the type of its operand. The valid types that may be used are BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, and FAR. The resulting expression will behave as a variable, label, memory expression, or register expression of the given type. Valid operands depend upon the type used. For instance, it is not possible to change the type of a register expression to a NEAR or FAR label.

Attributes:

- **type** - as defined

Examples:

MOV AX, WORD PTR BMEM ;access as word
JMP NEAR PTR LABFAR ;use far label as NEAR
MOV AL, BYTE PTR [BP] ;typing an anonymous ;memory reference
MOV DS: WORD PTR 10, AX ;absolute offset typing
Segment or Group Override

Syntax: 

\[ \text{operand1} : \text{operand2} \]

Description: The segment override changes the segment attribute of the second operand to that of the first operand for the duration of the instruction statement. The first operand may be

- one of the segment registers (DS, ES, SS, or CS)
- the name of a segment
- the name of a group

The second operand must be a variable, label, memory expression, or register expression. If the first operand is a segment register, then the second operand’s segment addressability attribute is changed to that of the segment register and no further testing is done. If the first operand is a segment name or group name, then the ASSUME values are checked to see if a segment register has been assumed to point to the segment or to the group. If one is found, the segment relocation and addressability attributes are changed to that of the matching segment register. If one is not found, it is an error. Remember, segment overrides only affect the current instruction; the ASSUME directive should be used for more global overrides.

The group override is useful when referring to variables or labels that belong to segments in the group. If no override is used, all offsets are relative to the base of the segment that the memory belongs to. The group override must be used to make the offset relative to the base of the group, which is probably a different value.

Attributes:

- **segment relocation** - set to value of group or segment name used
- **segment addressability** - set for variables
  - **CS-assume** - set for labels if group or segment name used

Examples:

- MOV AX,  DS: WMEM  ;offset from DS, base of segment
- MOV AX,  SEG1: WMEM ;offset from base of SEG1, or group
  ;that SEG1 belongs to, depending upon
Chapter 7: Expressions
Memory Operators

OFFSET

Syntax:
OFFSET variable
OFFSET label

Description: The OFFSET operator takes a variable, label, or memory expression as its operand and returns the offset value from some base as the result. If no segment override appears in the operand, the offset will be from the beginning of the segment. If a group name is used as a segment override, then the offset will be from the group base. Remember that no checking is done against the ASSUME values for the registers. To get the offset from a group, an explicit group override must be used. In either case, the result is an immediate value, not a memory address. The value may be relocatable, depending upon whether the operand resides in a combinable segment or in a group. The result of an OFFSET operator occupies 2 bytes if it is a relocatable value. Otherwise, the number of bytes depends upon the value of the offset. Example:

MOV SI, OFFSET WMEM ;offset from segment base
MOV SI, OFFSET GRP1:WMEM ;offset from group base

SEG

Syntax:
SEG variable
SEG label

Description: The SEG operator takes a variable, label, or memory expression as its operand and returns a segment base as its result. The base may be relocatable, depending upon the type of the segment or group that the operand belongs to or on any overrides that have been applied to the operand. The memory expression may not contain index or base register names. Externals are allowed in the operand. The size of a relocatable segment base is always 2 bytes unless the segment definition used the AT keyword. In that
instance, the number of bytes may be 1 or 2, depending upon the segment location.

The SEG operator should not be used with operands that belong to a group. Instead, a segment register should be initialized to the group base so that all memory addresses will be offset from that base. Otherwise, the group is not being used correctly.

Note that the SEG operator may also be used in the ASSUME directive. See the reference about the ASSUME directive in the chapter titled "Assembler Directives" for more discussion on how SEG may be used with ASSUME.

Note
The SEG operator will also accept a segment name or a group name as an operator. Since segment names and group names do not have segment attributes, SEG with a segment or group name does not perform any function. The assembler ignores the SEG operator and acts as if only the segment or group name were used.

Attributes: relocation type - base

Example:
MOV AX, SEG WMEM; load base value into AX
MOV DS, AX; initialize DS register

TYPE

Syntax:

TYPE variable
TYPE label

Description: The TYPE operator takes a variable, label, structure name, or memory expression as its operand. TYPE returns an absolute value that represents the type of the operand.

For most operands, the result is equal to the number of bytes allocated by a single occurrence of the operand. This value could then be used for incrementing a pointer into a data array, for example. The following are the returned values for variables or labels of a given type:
• BYTE - returns 1
• WORD - returns 2
• DWORD - returns 4
• QWORD - returns 8
• TBYTE - returns 10
• NEAR - returns -1 in two’s complement form
• FAR - returns -2 in two’s complement form
• record - returns number of bytes described by an occurrence of record
• structure - returns the sum of the sizes of the directives within the structure

Examples:

MOV AX, TYPE WMEM ;result is 2
MOV AX, TYPE LABFAR ;result is -2 in two’s complement form (FFFEH)
REC1 RECORD F1:3, F2:5 ;record definition with RECORD directive
R1 REC1 <> ;storage allocation
MOV AX, TYPE REC1 ;result is 1
MOV AX, TYPE R1 ;result is 1
ST1 STRUC ;structure template ;definition
DB ?
DW ?
ST1 ENDS
SU1 ST1 <> ;storage allocation using ;structure template
MOV AX, TYPE ST1 ;result is 3
MOV AX, TYPE SU1 ;result is 3

LENGTH

Syntax:

LENGTH variable

Description: The LENGTH operator takes a variable as its operand. It returns an absolute value equal to the number of units that were defined with the variable. A unit may include several bytes allocated by a single occurrence
of a type, but it still counts as just one unit. For instance, a single word allocation occupies two bytes, but from the point of view of LENGTH, it is one unit (in this case one word). The length of external symbols is always defined to be 1, regardless of how it is defined in a different file. LENGTH does not operate on structure or record templates. Examples:

```
L1 DB 1
MOV AX, LENGTH L1 ;result in AX is 1

L2 DW 1,2
MOV AX, LENGTH L2 ;result in AX is 2

L3 DB 5 DUP (2)
MOV AX, LENGTH L3 ;result in AX is 5

L4 DW 1, 4 DUP (?)
MOV AX, LENGTH L4 ;result in AX is 5

REC1 RECORD F1:3, F2:5 ;record template definition
R1 REC1 <> ;variable declared using record
MOV AX, LENGTH R1 ;result in AX is 1

R2 REC1 5 DUP (<>) ;another variable with record
MOV AX, LENGTH R2 ;result in AX is 5

ST1 STRUC ;structure template def.
DB ?
ST1 ENDS
SU1 ST1 <> ;variable declared
MOV AX, LENGTH SU1 ;result in AX is 1
```

### SIZE

**Syntax:**

```
SIZE variable
```

**Description:** The SIZE operator takes a variable, structure name, structure field, or record name as its operand and returns an absolute value equal to the total number of bytes defined by the operand. The size is generally equal to the length of the operand multiplied by the operand’s type. Examples:

```
L1 DB 1
MOV AX, SIZE L1 ;result in AX is 1

L2 DW 1,2
MOV AX, SIZE L2 ;result in AX is 4

L3 DB 5 DUP (2)
MOV AX, SIZE L3 ;result in AX is 5
```
L4 DW 1, 4 DUP (?)
MOV AX, SIZE L4 ;result in AX is 10

REC1 RECORD F1:3, F2:5 ;record template definition
R1 REC1 <> ;storage allocation using record
MOV AX, SIZE R1 ;result placed in AX is 1
MOV AX, SIZE REC1 ;result placed in AX is 1

ST1 STRUC ;structure template def.
DB ?
STF1 DW ?
ST1 ENDS
SU1 ST1 <> ;variable declared using
MOV AX, SIZE ST1 ;structure template
MOV AX, SIZE SU1 ;result placed in AX is 3
MOV AX, SIZE STF1 ;result in AX is 2
Record Operators

Record operators are used with record structure templates and record allocations to isolate bit fields of records and to find the actual number of bits in a record.

MASK

Syntax:

MASK recfield

Description: The MASK operator takes a record field as its operand. It returns an absolute number that will mask all the bits in a record except for those that belong to the record field operand. A mask is a number that will have 1's for all bits within the record field and have 0's for all other bits. It can be either a byte- or word-sized value, depending upon the size of the record and the positioning of the field within the record.

The MASK operator is useful when combined with the shift value (see "Expression Operands" in this chapter) for a record field. Together, they allow you to extract the value of a field. First, mask the record to isolate the bits that belong to the field. Then, shift the field so that its least significant bit is in the 0th bit position. The value of the result will now be equal to the value in the record field. Example:

R1 RECORD F1:5, F2:2
U1 R1 <14,3>

; load record into register
MOV AL,U1
; mask out extra bits with MASK
AND AL,MASK F1
; put field shift value
MOV CL, F1
; in register
SHR AL,CL
; shift field to lowest bit
; position - AL now contains value of record field

WIDTH

Syntax:

WIDTH operand
**Description:** The WIDTH operator takes a record name or record field as its operand. It returns an absolute number that is the number of bits defined in the operand. For a record name, the value will be the sum of the bits in the record fields, and will not include unused bits. For a record field, the value is the number of bits within that particular field. Examples:

```
R1 RECORD F1:5, F2:2
MOV AX, WIDTH R1          ;result in AX is 7
MOV AX, WIDTH F1          ;result in AX is 5
```
Chapter 7: Expressions
Segment and Group Operators

Segment and Group Operators

These operators return values that are only known at link-time. They generally refer to the size or address of segments and groups within a program.

SEGOFFSET

Syntax:

SEGOFFSET segmentname

Description: The SEGOFFSET operator returns a value that is the offset of the indicated segment from the next-lowest paragraph boundary. This value is the same as the last hex-digit of the base address for the segment. If the segment is paragraph or page aligned or is at an absolute location, then this value will be 0. Otherwise, this value is a relocatable value that will be known at final link time. The value will be range from 0 to 15, but will be word-sized if it is relocatable. Example:

A SEGMENT BYTE
; LOAD PARAGRAPH VALUE FOR SEGMENT
MOV AX, A

; LOAD OFFSET OF SEGMENT FROM NEAREST
; PARAGRAPH. TOGETHER, THEY FORM THE
; START LOCATION FOR THE SEGMENT
MOV BX, SEGOFFSET A

GRPOFFSET

Syntax:

groupname GRPOFFSET segmentname

Description: The GRPOFFSET operator returns the offset of a segment’s base from the start of a group that it belongs to. The segment must be defined as part of the group or this operator will result in an error. Since the offset within the group is not known until link time, this operator will result in a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time, which will the offset of the segment’s base from the start of the group. Example:
GRGRP GROUP A,B

; POINT DS AT GROUP
MOV AX, GRGRP
MOV DS, AX

; SET UP POINTER TO START
; OF SEGMENT SO LOCATIONS
; WITHIN THE SEGMENT CAN BE
; REFERENCED FROM THE GROUP
; SELECTOR
MOV SI, GRGRP GRPOFFSET B

SEGSIZE

Syntax:

SEGSIZE segmentname

Description: The SEGSIZE operator returns a word-sized value that is the size of the indicated segment. Since this size is not known (usually) at assembly time, this operator generates a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time. Note that the linker will return the value 0 if the group size is 64K.

Example:

A SEGMENT PUBLIC

; LOAD SEGMENT SIZE.
; COULD BE USED TO MAKE
; SURE INDEX VALUES DON’T
; GO OUTSIDE OF A SEGMENT.
MOV AX, SEGSIZE A
GRPSIZE

**Syntax:**

GRPSIZE groupname

**Description:** The GRPSIZE operator returns a word-sized value that is the size of the indicated group. Since this size is not known at assembly time, this operator generates a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time, which will be the size of the group. Note that the linker will return the value 0 if the group size is 64K. Examples:

GRGRP GROUP A,B

MOV AX, GRPSIZE GRGRP
Operator Precedence

Complex expressions, or expressions that contain multiple operators, are evaluated according to operator precedence rules:

- Expressions enclosed within parentheses are evaluated from the innermost set of parenthesis to the outermost set. Within a set of parenthesis, operators conform to the other precedence rules below.

- Excluding parentheses, sub-expressions that have operators of higher precedence will be calculated before sub-expressions with operators of lower precedence. For example, a multiply operation is done before an addition operation.

- Excluding parentheses, sub-expressions which have operators of equal precedence (Operators that appear on the same line in the following table are of equal precedence.) are evaluated left-to-right. Left-to-right evaluation means that if two operators of equal precedence appear in the same expression, the operator which is closer to the leftmost end of the expression will be evaluated before an operator closer to the rightmost end. For instance, in the expression '6 * 5 / 3' the order of evaluation is to multiply 6 by 5 and then divide by 3. The result is 10.

The ranking of operators from higher to lower precedence is given in the following table.
### Chapter 7: Expressions

**Operator Precedence**

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher</td>
<td>(, [ ], &lt; &gt; . LENGTH, SIZE, WIDTH, MASK, SEGOFFSET, SEGSIZE, GRPOFFSET, GRPSIZE, PTR, OFFSET, SEG, TYPE, THIS, Segment Override</td>
</tr>
<tr>
<td></td>
<td>HIGH, LOW</td>
</tr>
<tr>
<td></td>
<td>*, /, MOD, SHR, SHL</td>
</tr>
<tr>
<td></td>
<td>Unary +, -</td>
</tr>
<tr>
<td></td>
<td>Binary +, -</td>
</tr>
<tr>
<td></td>
<td>EQ, NE, LT, LE, GT, GE</td>
</tr>
<tr>
<td></td>
<td>NOT</td>
</tr>
<tr>
<td></td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>OR, XOR</td>
</tr>
<tr>
<td>Lower</td>
<td>SHORT</td>
</tr>
</tbody>
</table>
Instructions and Operands

A discussion of operands and a list of recognized instructions.
Operands

You may recall that the general syntax of an assembler statement is as follows:

[ label : ] [ prefix ] keyword [ operand [ ,... ] ] [;comment]

This section concentrates on the operand field of this syntax.

Accepted Operands

A list of assembly language instructions and the operand combinations acceptable for each instruction is at the end of this chapter. Each allowable combination has a limited range of values. Any other combination results in an error condition.

Compatible Types

In most instances, if an instruction takes more than one operand, the operands must be of the same type. For example, it is only possible to move a WORD-sized value into a WORD destination. A mismatch error occurs if an instruction attempts to move a WORD into a BYTE. It is possible, however, to move a BYTE-sized immediate value into a WORD-sized destination. The immediate is either stored as a WORD or it is sign-extended during execution.

Some instructions allow operands to be of different types. It is best to check the list of instructions at the end of the chapter for allowable operand combinations.

Required Typing

Many instructions do require that the memory operand be typed. Instructions that take a single operand generate different object code depending upon the type of the operand. Or, perhaps the type of one operand does not restrict the valid type of the other operand. The assembler cannot decide what object code to output in these instances. The following instructions demonstrate some unacceptable operand combinations:

```
INC [BX] ;generate byte or word instruction?
```
The INC instruction accepts both BYTE and WORD memory operands. In the above example, the assembler could not decide which instruction to generate.

The ESC instruction also accepts BYTE and WORD memory operands. The immediate value 5, in the example above, does not help limit the type of the memory operand since the value is independent of the memory type.

For the MOV instruction above, the immediate value 2 is small enough to fit in either a BYTE or a WORD. Again, the immediate operand does not restrict the type sufficiently.

When in doubt, type these ambiguous expressions to avoid possible error conditions.

Anonymous References

Most instructions are able to accept operands that do not have type information—references known as anonymous memory references. These references do not have a variable or any type information associated with them, so the assembler must use other knowledge to determine the type. The assembler may type the anonymous operand to be the same as another operand in the instruction, or not require a type at all. The following examples are of typing the same as another operand:

- MOV AX, [BX] ;WORD since AX is a WORD-sized register
- MOV [BX], AL ;BYTE since AL is a BYTE-sized register
- MOV [BX], 1000 ;WORD since 1000 can’t be stored in BYTE

Assumed Type With Register

The assembler can easily determine the type of an anonymous reference if the other operand is an 8086 register. Notice in the above example when AX and AL were used. Another example of an instruction not needing a type (since it handles all memory operands the same) is an 8087 floating point instruction. Example:

FLDCW [BX]
Operands

Operand Positioning

If an instruction takes a single operand, the operand position (other than it must be in the proper place) is not critical. Instructions which accept two operands generally treat the first operand as the destination operand and the second operand as the source operand. The movement of data is then from the second operand into the first. The instruction

\[ \text{MOV AX, BX} \]

takes the contents of the BX register and places it in the AX register. There are exceptions. Some string instructions use the first operand as the source operand and the second operand as the destination operand. Check the usage of the operands when in doubt. The instruction list at the end of this chapter—and in the Intel iAPX 86/88, 186/188 User’s Manual—includes information on data movement between operands.

Immediate Values

Immediate values are operands in many assembly language instructions. In most cases, the immediate value is a source operand. This value is stored directly in the destination operand or used to modify a value already stored elsewhere, say in a register or memory location.

Immediate values are not always numbers. Immediate values are also generated in many non-obvious ways as shown in the "Expressions" chapter.

Range of Immediate Values

Immediate values can be absolute, relocatable, or external numbers. The size of the value is determined by the instruction used, by the value itself, and by what type is assumed for it.

An absolute immediate may range anywhere from -65535 to 65535 depending upon the instruction and the type of the operand. The INT (interrupt) instruction, for instance, can only take a value from 0 to 255 since that is the range of interrupt values for the 8086. A variable of type BYTE may take a value from -255 to 255. A variable of type WORD may take a value from -65535 to 65535.

A relocatable or external immediate is always assumed to be a 16-bit value unless modified with a HIGH or LOW operator.
Registers

A very common operand is a processor register. A processor register is a memory store that is internal to the 8086/186 processors, and the 8087 co-processor. Internal registers can be source operands or destination operands for data. Some registers have special tasks which restrict their uses in programs. Since some instructions may indirectly use or modify these restricted registers, take care their contents are not accidentally modified or misused.

The figure below shows the general purpose and special registers for the 8086/186 processor. Following the figure is a more detailed description of the various processor registers.
## DATA REGISTERS

<table>
<thead>
<tr>
<th>7</th>
<th>0 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH (HIGH BYTE OF AX)</td>
<td>AL (LOW BYTE OF AX)</td>
<td></td>
</tr>
<tr>
<td>BH (HIGH BYTE OF BX)</td>
<td>BL (LOW BYTE OF BX)</td>
<td></td>
</tr>
<tr>
<td>CH (HIGH BYTE OF CX)</td>
<td>CL (LOW BYTE OF CX)</td>
<td></td>
</tr>
<tr>
<td>DH (HIGH BYTE OF DX)</td>
<td>DL (LOW BYTE OF DX)</td>
<td></td>
</tr>
</tbody>
</table>

## POINTER AND INDEX REGISTERS

<table>
<thead>
<tr>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP (STACK POINTER)</td>
<td></td>
</tr>
<tr>
<td>BP (BASE POINTER)</td>
<td></td>
</tr>
<tr>
<td>SI (SOURCE INDEX)</td>
<td></td>
</tr>
<tr>
<td>DI (DESTINATION INDEX)</td>
<td></td>
</tr>
</tbody>
</table>

## SEGMENT REGISTERS

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CS (CODE)</td>
</tr>
<tr>
<td>DS (DATA)</td>
</tr>
<tr>
<td>SS (STACK)</td>
</tr>
<tr>
<td>ES (EXTRA)</td>
</tr>
</tbody>
</table>

Figure 8-1. 8086/186 Registers
16-bit Registers AX, BX, CX, DX, DI, SI, SP, BP

There are eight 16-bit (WORD-sized) general purpose registers located on the 8086/186 processors referenced by the unique register names AX, BX, CX, DX, DI, SI, BP and SP. AX, BX, CX, and DX are general purpose data registers. For most instructions that allow a register as an operand, these four registers are used. DI, SI, BX and BP are the index and base registers.

Some instructions explicitly use certain registers. The CX register, for instance, is used to control looping. Many string instructions use the SI as a source pointer and DI as a destination pointer. The SP register points to the top of stack and is modified whenever CALLs, PUSHs, or POPs occur. Data loss can occur through a side effect of these explicit usages. Be careful to protect the contents of these registers so they are not accidentally modified through the use of an instruction.

8-bit Registers AL, AH, BL, BH, CL, CH, DL, DH

There are also eight 8-bit (BYTE-sized) registers. The unique names given to them are AL, AH, BL, BH, CL, CH, DL, and DH. These registers are not separate registers; instead they are the byte-addressable upper and lower halves of the four 16-bit general-purpose data registers (AX, BX, CX, and DX). AX, for instance, is equivalent to AL+ AH. (Not the value, but the register.)

The 'L' in AL means the low byte of AX and the 'H' in AH means the high byte AX. If you refer to AL, the assembler understands that you mean the low byte of AX. If you refer to AX, the assembler understands that you mean the entire 16 bits of AX.

You may load data into these registers either as a single 16-bit quantity or as two 8-bit quantities. The resulting value in the register is the same.

Segment Registers CS, DS, SS, ES

8086/186 memory addresses are generated by offsetting from segment registers. To be able to address a particular location in memory, that address must be contained in one of the four, currently active physical segments. Each segment has a maximum size of 64K and each has a particular register that contains the base address (lowest memory location) of the segment. Each segment has a different purpose:
Chapter 8: Instructions and Operands
Operands

- Executable code (program code) is located in the Code segment and is addressable through the CS (Code Segment) register.

- Data is most often located in the Data segment (although it can be in any of the four segments) and is addressed through the DS (Data Segment) register.

- The program stack is located in the Stack segment and is addressed through the SS (Stack Segment) register.

- Data often is located in the Extra segment and is addressed via the ES (Extra Segment) register.

Memory Addressing  A memory address is a 20-bit value —allowing the 8086/186 to address 1 megabyte of memory— that is calculated from the segment base address located in one of the segment registers, and an offset supplied either by the IP (instruction pointer), or by operands contained in the instruction itself. To calculate the memory address, the 16-bit value in a given segment register is first shifted to the left 4 bits. Then the offset value (either a 16-bit or 8-bit value) is added to the shifted value to generate the 20-bit address necessary to access memory.

Segment Register Use  The four segment registers have restricted use. The only assembly instructions that may reference these registers as operands are the MOV, PUSH, and POP instructions.

Some Assembler Directives also use the register names as part of their syntax, but this use does not cause object code to be generated.

Other instructions indirectly reference the segment registers. LDS and LES, for instance, could change the segment register contents. CALLs and JMPs change the CS register if the branch takes execution out of the current segment. Finally, as noted in the chapter titled ‘Expressions,’ segment register names may be used as overrides in memory operands.

8087 Floating Point Registers ST(0)...ST(7)

The 8087 co-processor has eight floating point stack registers. They are referenced by the names ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7). ST(0) may be referenced as just ST without the appended (0). These registers are only used with some 8087 floating point instructions.
They are not directly accessible to the 8086/186 processors. Instead, 8087 instructions make the contents of these registers available in memory. The 8087 floating point stack registers are 80 bits in size and store their values in IEEE floating point format.

Memory Expressions and the MODRM Byte

Memory expressions may be either simple memory references (using a variable name by itself) or a complex expressions involving register indirection or offsets within structures. A simple memory reference will always take the type of the variable, so that type must either be compatible with an instruction or it must be re-typed with the PTR operator. Examples:

- `MOV AX, WMEM` ;simple variable
- `MOV AX, [BX][SI]` ;indirect anonymous memory
- `MOV AX, [BP].SFWORD` ;memory reference
- `MOV AX, WMEM[BP][DI]` ;indirect memory reference
- `MOV AX, STR1.SFWORD` ;structure field reference
- `MOV AX, WORD PTR DMEM` ;typed memory reference

Physical Address Calculation

The processor must generate a physical address for each memory reference. The offset part of the address—the value which is added to the shifted segment register address—may be coded into the instruction in one of four ways:

- As a direct 16-bit offset.
- As an indirect offset through a base register, BX or BP, optionally with an added (or subtracted) 8-bit or 16-bit displacement.
- As an indirect offset through an index register, SI or DI, optionally with an added (or subtracted) 8-bit or 16-bit displacement.
- As an indirect offset through the sum of one base register and one index register, optionally with an added (or subtracted) 8-bit or 16-bit displacement.

MODRM Byte

The information describing how the offset is derived is stored in the object code in a special byte called the MODRM byte. This byte has three fields:
Chapter 8: Instructions and Operands
Segment Addressability and Overrides

1 The first field describes how many bytes are required to hold the displacement portion of the address. This field can specify that 0, 1, or 2 bytes are required. If the value is a relocatable or external value, two bytes are always required.

2 The second field contains a register code or part of the code for the instruction; it is not relevant to this section.

3 The third field contains information describing what base and index registers are used, if any, when generating the address.

The MODRM byte, along with any displacement value, determines the offset of the memory address referenced in an instruction. Remember, the value is just the offset of the memory address. The base from which to offset must still be decided.

Single Memory Expression per Instruction

Each memory expression is either a source or destination for the instruction. Most instructions allow only a single memory expression, since the MODRM byte can only describe one. Some string instructions may have two memory expressions as operands, but these instructions are special cases because the operands are only used to check for segment addressability. Their offsets are not emitted as object code. Instead, the SI and DI registers are used for addressing the memory.

Segment Addressability and Overrides

The 8086 or 80186 processor generates a memory address by shifting the value from a segment register four bits to the left and then adding an offset to the shifted value. A segment of memory, up to a maximum of 64K bytes in size, is active only if one of the four segment registers points to that particular piece of memory.

Note that the segment is a physical segment, a physical piece of memory. These physical segments contain the logical segments of your assembly language program that you identified through SEGMENT/ENDS assembler directive pairs and other, similar means.
With the ASSUME assembler directive, you tell the assembler what values to assume as the base locations of the currently active segments. The ASSUME directive, then, lets you inform the assembler of the relationship between the logical segments you have defined in the program and the physical segments where they will eventually be located.

**Addressability Checking**

During assembly, if the assembler encounters an instruction that generates a memory reference, the assembler checks that reference against the value in the ASSUME for that segment. The assembler generates an error if the location in memory cannot be accessed through that particular segment register. The exception to checking against the ASSUME is when a memory reference contains a specific segment override.

NEAR and SHORT label references are also checked for addressability through the CS segment register to assure the assembler that the label can be reached during execution. A segment or group name may be used to override a label if the CS segment register value will be different than that currently assumed.

Addressability checking is done so that the correct object code may be generated. Unless a memory reference contains a segment override, the instruction is not preceded by a segment override byte in the generated object code. If no segment override byte is coded with the instruction, then the instruction memory reference defaults to a certain segment, depending upon the nature of the instruction.

**Default Segments**

If a memory reference does not specifically name a segment register through a segment override, there are default segment registers for memory references. The CS register is the default for instruction fetching. The DS segment register is the default for most memory data references, unless BP (a base register) is specified for register indirection. The SS segment register is the default if BP is used. Some string instructions default to the ES segment register with certain operands.

Although there are default segment registers for references, you must still use the ASSUME directive to inform the assembler where the bases of these segments are located; again, to specify the relationship between logical and physical segments and to aid in addressability checking.
Segment Overrides

An instruction may override these default registers by including a segment override in the instruction operand. There are two reasons why a segment override might be included in a memory reference:

- The memory location accessed is not located in the default segment that would be used with a particular instruction.
- The memory location accessed is located within a group in a segment. In this instance, the base of the group must be used for memory access, not the base of the segment.

The override holds for the duration of the instruction only. Segment overrides do not alter the contents of segment registers or the values specified in ASSUME directives.

Improper Uses of Segment Overrides

The section on default segments mentions that some string instructions default to the ES register. For these string instructions, you may not use segment overrides for string operands. You may use segment overrides, however, for the other memory operands in those instructions.

These and other exceptions are noted in the listing of instructions at the end of this chapter.

Segment Override Byte

When the assembler generates code for an instruction containing a segment override, the assembler precedes the instruction code with a segment override byte. (Whether it will appear or not is discussed below.) This override byte, if present, causes a specific segment register to be used to address that memory, regardless of which segment the variable belongs to. In the segment override byte, specific values are associated with specific registers. Examination of these values can tell you which segment the override has been generated for. The values are

- CS = 2EH
- DS = 3EH
- SS = 36H
- ES = 26H
 Overrides and Checking Against ASSUME

If a segment name is used to override the default segment value for a memory reference, then the ASSUME value for the override segment is checked to see if it has been set to either

- the segment named in the override, or
- to a group that contains the segment named in the override.

If a group name is used, then the group name must match exactly.

Examples of segment overrides:

```
MOV AX, SEG1: WMEM       ;matches segment or group
MOV AX, GRP1: WMEM       ;matches group only
```

Segment Override Byte Generation

A memory reference that includes a segment override generates a segment override byte depending upon the outcome of the following checks:

1. If the memory is addressable by the default segment register for that type of instruction and operand, then the instruction needs no override byte.

2. If this test fails, then the segment registers are checked in the following order: DS, ES, CS, and SS. If the memory expression is addressable by one of these registers, then an override byte is generated for that register.

3. If no register match occurs, an error is generated. The checks are specific. If the variable used in the memory expression was an external defined outside of a segment, it can only match an ASSUME segment that has been set to the SEG value of the external or to a group that includes that segment.
The Instruction Set

This section contains the instruction set accepted by the as86 assembler. All operand combinations are listed for each instruction. Some of these instructions or operand combinations are only valid in certain modes (such as 80186 or V20). These restricted instructions are explained in the notes at the end of the list of instructions.

A special code denotes what operand patterns are allowed for each instruction. If no operands are shown, then none are expected for that instruction. Otherwise, each operand will have a name, indicating what the operand does, followed by a colon and a code indicating what type of operand is to be used. If an operand is restricted to certain values, then these values will be listed in parenthesis after the code. If more than one restricted value is possible, then they will be separated by commas. Numeric ranges will be denoted by their boundary values.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>AL only</td>
</tr>
<tr>
<td>AW</td>
<td>AX only</td>
</tr>
<tr>
<td>CB</td>
<td>SHORT label with current segment and within 127 bytes of current location</td>
</tr>
<tr>
<td>CD</td>
<td>FAR label, offset and base</td>
</tr>
<tr>
<td>CW</td>
<td>NEAR label, within current segment</td>
</tr>
<tr>
<td>D</td>
<td>17-bit immediate value</td>
</tr>
<tr>
<td>DB</td>
<td>1-byte immediate value, from -255 to 255</td>
</tr>
<tr>
<td>DW</td>
<td>2-byte immediate value, from -65535 to 65535</td>
</tr>
<tr>
<td>EB</td>
<td>either an 8-bit register or BYTE-type memory expression</td>
</tr>
<tr>
<td>ED</td>
<td>DWORD-type memory expression</td>
</tr>
<tr>
<td>EW</td>
<td>either a 16-bit register or WORD-type memory expression</td>
</tr>
<tr>
<td>F</td>
<td>8087 floating point stack register</td>
</tr>
<tr>
<td>M</td>
<td>any type of memory expression</td>
</tr>
<tr>
<td>MB</td>
<td>BYTE-type memory expression</td>
</tr>
<tr>
<td>MW</td>
<td>WORD-type memory expression</td>
</tr>
<tr>
<td>MD</td>
<td>DWORD-type memory expression</td>
</tr>
<tr>
<td>MQ</td>
<td>QWORD-type memory expression</td>
</tr>
<tr>
<td>MT</td>
<td>TBYTE-type memory expression</td>
</tr>
<tr>
<td>RB</td>
<td>8-bit register</td>
</tr>
<tr>
<td>RW</td>
<td>16-bit register</td>
</tr>
<tr>
<td>S</td>
<td>segment register</td>
</tr>
<tr>
<td>T</td>
<td>ST(0); top of 8087 floating point register stack</td>
</tr>
<tr>
<td>XB</td>
<td>BYTE-type, simple memory expression; no register indirection</td>
</tr>
<tr>
<td>XW</td>
<td>WORD-type, simple memory expression; no register indirection</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>AAA</td>
<td></td>
</tr>
<tr>
<td>AAD</td>
<td></td>
</tr>
<tr>
<td>AAM</td>
<td></td>
</tr>
<tr>
<td>AAS</td>
<td></td>
</tr>
<tr>
<td>ADC dst:AB,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADC dst:AW,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADC dst:AW,src:DW</td>
<td></td>
</tr>
<tr>
<td>ADC dst:EB,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADC dst:EB,src:RB</td>
<td></td>
</tr>
<tr>
<td>ADC dst:EW,src:DB(-128,127)</td>
<td></td>
</tr>
<tr>
<td>ADC dst:EW,src:DW</td>
<td></td>
</tr>
<tr>
<td>ADC dst:EW,src:RW</td>
<td></td>
</tr>
<tr>
<td>ADC dst:RB,src:EB</td>
<td></td>
</tr>
<tr>
<td>ADC dst:RW,src:EW</td>
<td></td>
</tr>
<tr>
<td>ADD dst:AB,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADD dst:AW,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADD dst:AW,src:DW</td>
<td></td>
</tr>
<tr>
<td>ADD dst:EB,src:DB</td>
<td></td>
</tr>
<tr>
<td>ADD dst:EB,src:RB</td>
<td></td>
</tr>
<tr>
<td>ADD dst:EW,src:DB(-128,127)</td>
<td></td>
</tr>
<tr>
<td>ADD dst:EW,src:DW</td>
<td></td>
</tr>
<tr>
<td>ADD dst:EW,src:RW</td>
<td></td>
</tr>
<tr>
<td>ADD dst:RB,src:EB</td>
<td></td>
</tr>
<tr>
<td>ADD dst:RW,src:EW</td>
<td></td>
</tr>
<tr>
<td>ADD4S dst:M,src:M</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>ADD4S dst:AB,src:DB</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>AND dst:AW,src:DB</td>
<td></td>
</tr>
<tr>
<td>AND dst:AW,src:DW</td>
<td></td>
</tr>
<tr>
<td>AND dst:EB,src:DB</td>
<td></td>
</tr>
<tr>
<td>AND dst:EB,src:RB</td>
<td></td>
</tr>
<tr>
<td>AND dst:EW,src:DB</td>
<td></td>
</tr>
<tr>
<td>AND dst:EW,src:DB</td>
<td></td>
</tr>
<tr>
<td>AND dst:EW,src:RW</td>
<td></td>
</tr>
<tr>
<td>AND dst:RB,src:EB</td>
<td></td>
</tr>
<tr>
<td>AND dst:RW,src:EB</td>
<td></td>
</tr>
<tr>
<td>BOUND indx:RW,bptr:MD</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>BOUND indx:RW,bptr:MW</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>BRKEM vector:Db</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>CALL addr:CB</td>
<td></td>
</tr>
<tr>
<td>CALL addr:CD</td>
<td></td>
</tr>
<tr>
<td>CALL addr:CW</td>
<td></td>
</tr>
<tr>
<td>CALL addr:ED</td>
<td></td>
</tr>
<tr>
<td>CALL addr:EW</td>
<td></td>
</tr>
<tr>
<td>CBW</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td></td>
</tr>
<tr>
<td>CLR1 dst:Eb,off:D(0,7)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>CLR1 dst:Eb,off:Rb(CL)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>CLR1 dst:Ec,off:D(0,15)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>CLR1 dst:Ec,off:Rb(CL)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>CMC</td>
<td></td>
</tr>
<tr>
<td>CMP dst:AB,src:DB</td>
<td></td>
</tr>
<tr>
<td>CMP dst:AW,src:DB</td>
<td></td>
</tr>
<tr>
<td>CMP dst:AW,src:DW</td>
<td></td>
</tr>
<tr>
<td>CMP dst:EB,src:DB</td>
<td></td>
</tr>
<tr>
<td>CMP dst:EB,src:RB</td>
<td></td>
</tr>
<tr>
<td>CMP dst:EW,src:DB(-128,127)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>dst:EW, src:DW</td>
</tr>
<tr>
<td>CMP</td>
<td>dst:EW, src:RW</td>
</tr>
<tr>
<td>CMP</td>
<td>dst:EB, src:EB</td>
</tr>
<tr>
<td>CMP</td>
<td>dst:EW, src:EW</td>
</tr>
<tr>
<td>CMP4S</td>
<td>dst:M, src:M  (Note 3)</td>
</tr>
<tr>
<td>CMPS</td>
<td>SI__ptr:MB, DI__ptr:MB  (Note 1)</td>
</tr>
<tr>
<td>CMPS</td>
<td>SI__ptr:MW, DI__ptr:MW  (Note 1)</td>
</tr>
<tr>
<td>CMPSB</td>
<td></td>
</tr>
<tr>
<td>CMPSW</td>
<td></td>
</tr>
<tr>
<td>CWD</td>
<td></td>
</tr>
<tr>
<td>DAA</td>
<td></td>
</tr>
<tr>
<td>DAS</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>dst:EB</td>
</tr>
<tr>
<td>DEC</td>
<td>dst:RW</td>
</tr>
<tr>
<td>DIV</td>
<td>divisor:EB</td>
</tr>
<tr>
<td>DIV</td>
<td>divisor:EW</td>
</tr>
<tr>
<td>ENTER</td>
<td>disp:D(0,0FFFFH), level:D(0,255)  (Note 2)</td>
</tr>
<tr>
<td>ESC</td>
<td>opcode:DB(0,63), addr:EB</td>
</tr>
<tr>
<td>ESC</td>
<td>opcode:DB(0,63), addr:ED</td>
</tr>
<tr>
<td>ESC</td>
<td>opcode:DB(0,63), addr:EW</td>
</tr>
<tr>
<td>EXT</td>
<td>dst:Rb, count:D(0,15)  (Note 3)</td>
</tr>
<tr>
<td>EXT</td>
<td>dst:Rb, src:Rb  (Note 3)</td>
</tr>
<tr>
<td>F2XM1</td>
<td></td>
</tr>
<tr>
<td>FABS</td>
<td></td>
</tr>
<tr>
<td>FADD</td>
<td></td>
</tr>
<tr>
<td>FADD</td>
<td>dst:F, src:T</td>
</tr>
<tr>
<td>FADD</td>
<td>dst:T, src:F</td>
</tr>
<tr>
<td>FADD</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FADD</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FADDP</td>
<td>dst:F, src:T</td>
</tr>
<tr>
<td>FBLD</td>
<td>memop:MT</td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBSTP</td>
<td>memop:MT</td>
</tr>
<tr>
<td>FCHS</td>
<td></td>
</tr>
<tr>
<td>FCLEX</td>
<td></td>
</tr>
<tr>
<td>FCOM</td>
<td></td>
</tr>
<tr>
<td>FCOM</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FCOM</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FCOM</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FCOMP</td>
<td></td>
</tr>
<tr>
<td>FCOMP</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FCOMP</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FCOMP</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FCOMPP</td>
<td></td>
</tr>
<tr>
<td>FDECSTP</td>
<td></td>
</tr>
<tr>
<td>FDISI</td>
<td></td>
</tr>
<tr>
<td>FDIV</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FDIV</td>
<td>dst:T,src:F</td>
</tr>
<tr>
<td>FDIV</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FDIV</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FDIVP</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FDIVR</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FDIVR</td>
<td>dst:T,src:F</td>
</tr>
<tr>
<td>FDIVR</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FDIVR</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FDIVRP</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FENI</td>
<td></td>
</tr>
<tr>
<td>FFREE</td>
<td>fpst:F</td>
</tr>
<tr>
<td>F IADD</td>
<td>memop:MD</td>
</tr>
<tr>
<td>F IADD</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FICOM</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FICOM</td>
<td>memop:MW</td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FICOMP</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FICOMP</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FIDIV</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FIDIV</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FIDIVR</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FIDIVR</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FILD</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FILD</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FILD</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FIMUL</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FIMUL</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FINCSTP</td>
<td></td>
</tr>
<tr>
<td>FINIT</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FIST</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FIST</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FISTP</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FISTP</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FISTP</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FISUB</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FISUB</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FISUBR</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FISUBR</td>
<td>memop:MW</td>
</tr>
<tr>
<td>FLD</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FLD</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FLD</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FLD</td>
<td>memop:MT</td>
</tr>
<tr>
<td>FLD1</td>
<td>memop:M</td>
</tr>
<tr>
<td>FLDCW</td>
<td>memop:M</td>
</tr>
<tr>
<td>FLDENV</td>
<td>memop:M</td>
</tr>
<tr>
<td>FLDL2E</td>
<td>memop:M</td>
</tr>
<tr>
<td>FLDL2T</td>
<td>memop:M</td>
</tr>
<tr>
<td>FLDLG2</td>
<td>memop:M</td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDLN2</td>
<td></td>
</tr>
<tr>
<td>FLDPI</td>
<td></td>
</tr>
<tr>
<td>FLDZ</td>
<td></td>
</tr>
<tr>
<td>FMUL</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FMUL</td>
<td>dst:T,src:F</td>
</tr>
<tr>
<td>FMUL</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FMUL</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FMULP</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FNCLEX</td>
<td></td>
</tr>
<tr>
<td>FNDISI</td>
<td></td>
</tr>
<tr>
<td>FNENI</td>
<td></td>
</tr>
<tr>
<td>FNINIT</td>
<td></td>
</tr>
<tr>
<td>FNOP</td>
<td>memop:M</td>
</tr>
<tr>
<td>FNSAVE</td>
<td>memop:M</td>
</tr>
<tr>
<td>FNSTCW</td>
<td>memop:M</td>
</tr>
<tr>
<td>FNSTENV</td>
<td>memop:M</td>
</tr>
<tr>
<td>FNSTSW</td>
<td>memop:M</td>
</tr>
<tr>
<td>FPATAN</td>
<td></td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,127)</td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,15),addr:Mb</td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,15),addr:Md</td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,15),addr:Mq</td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,15),addr:Mt</td>
</tr>
<tr>
<td>FPO2</td>
<td>opcode:D(0,15),addr:Mw</td>
</tr>
<tr>
<td>FPREM</td>
<td></td>
</tr>
<tr>
<td>FPTAN</td>
<td></td>
</tr>
<tr>
<td>FRNDINT</td>
<td>memop:M</td>
</tr>
<tr>
<td>FRSTOR</td>
<td>memop:M</td>
</tr>
<tr>
<td>FSAVE</td>
<td>memop:M</td>
</tr>
<tr>
<td>FSSCALE</td>
<td></td>
</tr>
<tr>
<td>FSQRT</td>
<td></td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FST</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FST</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FST</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FSTCW</td>
<td>memop:M</td>
</tr>
<tr>
<td>FSTENV</td>
<td>memop:M</td>
</tr>
<tr>
<td>FSTP</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FSTP</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FSTP</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FSTP</td>
<td>memop:MT</td>
</tr>
<tr>
<td>FSTSW</td>
<td>memop:M</td>
</tr>
<tr>
<td>FSUB</td>
<td>dst:T,src:F</td>
</tr>
<tr>
<td>FSUB</td>
<td>dstF,src:T</td>
</tr>
<tr>
<td>FSUB</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FSUB</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FSUBP</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FSUBR</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FSUBR</td>
<td>dst:T,src:F</td>
</tr>
<tr>
<td>FSUBR</td>
<td>memop:MD</td>
</tr>
<tr>
<td>FSUBR</td>
<td>memop:MQ</td>
</tr>
<tr>
<td>FSUBRP</td>
<td>dst:F,src:T</td>
</tr>
<tr>
<td>FTST</td>
<td></td>
</tr>
<tr>
<td>FWAIT</td>
<td></td>
</tr>
<tr>
<td>FXAM</td>
<td></td>
</tr>
<tr>
<td>FXCH</td>
<td></td>
</tr>
<tr>
<td>FXCH</td>
<td>fpst:F</td>
</tr>
<tr>
<td>FXTRACT</td>
<td></td>
</tr>
<tr>
<td>FYL2X</td>
<td></td>
</tr>
<tr>
<td>FYL2XP1</td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td></td>
</tr>
<tr>
<td>IDIV</td>
<td>divisor:EB</td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDIV</td>
<td>divisor:EW</td>
</tr>
<tr>
<td>IMUL</td>
<td>dst:RW,src1:EW,src2:DB(-128,127) (Note 2)</td>
</tr>
<tr>
<td>IMUL</td>
<td>dst:RW,src1:EW,src2:DW (Note 2)</td>
</tr>
<tr>
<td>IMUL</td>
<td>dst:RW,src2:DB(-128,127) (Note 2)</td>
</tr>
<tr>
<td>IMUL</td>
<td>dst:RW,src2:DW (Note 2)</td>
</tr>
<tr>
<td>IMUL</td>
<td>mplier:EB</td>
</tr>
<tr>
<td>IMUL</td>
<td>mplier:EW</td>
</tr>
<tr>
<td>IN</td>
<td>dst:AB,port:DB</td>
</tr>
<tr>
<td>IN</td>
<td>dst:AB,port:RW(DX)</td>
</tr>
<tr>
<td>IN</td>
<td>dst:AW,port:DB</td>
</tr>
<tr>
<td>IN</td>
<td>dst:AW,port:RW(DX)</td>
</tr>
<tr>
<td>INC</td>
<td>dst:EB</td>
</tr>
<tr>
<td>INC</td>
<td>dst:EW</td>
</tr>
<tr>
<td>INC</td>
<td>dst:RW</td>
</tr>
<tr>
<td>INS</td>
<td>DI__ptr:EB,port:RW(DX) (Notes 1,2)</td>
</tr>
<tr>
<td>INS</td>
<td>DI__ptr:EW,port:RW(DX) (Notes 1,2)</td>
</tr>
<tr>
<td>INS</td>
<td>dst:Rb,count:D(0,15) (Note 3)</td>
</tr>
<tr>
<td>INS</td>
<td>dst:Rb,src:Rb (Note 3)</td>
</tr>
<tr>
<td>INSB</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>INSW</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>INT</td>
<td>itype:DB(3)</td>
</tr>
<tr>
<td>INT</td>
<td>itype:DB</td>
</tr>
<tr>
<td>INTO</td>
<td></td>
</tr>
<tr>
<td>IRET</td>
<td></td>
</tr>
<tr>
<td>JA</td>
<td>place:CB</td>
</tr>
<tr>
<td>JAE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JB</td>
<td>place:CB</td>
</tr>
<tr>
<td>JBE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JC</td>
<td>place:CB</td>
</tr>
<tr>
<td>JCYZ</td>
<td>place:CB</td>
</tr>
<tr>
<td>JE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JG</td>
<td>place:CB</td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JL</td>
<td>place:CB</td>
</tr>
<tr>
<td>JLE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JMP</td>
<td>place:CB</td>
</tr>
<tr>
<td>JMP</td>
<td>place:CD</td>
</tr>
<tr>
<td>JMP</td>
<td>place:CW</td>
</tr>
<tr>
<td>JMP</td>
<td>place:EW</td>
</tr>
<tr>
<td>JMP</td>
<td>place:MD</td>
</tr>
<tr>
<td>JNA</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNAE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNB</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNBE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNC</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNG</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNGE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNL</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNLE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNO</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNP</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNS</td>
<td>place:CB</td>
</tr>
<tr>
<td>JNZ</td>
<td>place:CB</td>
</tr>
<tr>
<td>JO</td>
<td>place:CB</td>
</tr>
<tr>
<td>JP</td>
<td>place:CB</td>
</tr>
<tr>
<td>JPE</td>
<td>place:CB</td>
</tr>
<tr>
<td>JPO</td>
<td>place:CB</td>
</tr>
<tr>
<td>JS</td>
<td>place:CB</td>
</tr>
<tr>
<td>JZ</td>
<td>place:CB</td>
</tr>
<tr>
<td>LAHF</td>
<td></td>
</tr>
<tr>
<td>LDS</td>
<td>dst:RW,src:ED</td>
</tr>
<tr>
<td>LEA</td>
<td>dst:RW,src:M</td>
</tr>
</tbody>
</table>
| LEAVE       |             | (Note 2)
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>dst: src:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LES</td>
<td>RW: ED</td>
</tr>
<tr>
<td>LOCK</td>
<td>PREFIX</td>
</tr>
<tr>
<td>LODS</td>
<td>SI_ptr: MB</td>
</tr>
<tr>
<td>LODS</td>
<td>SI_ptr: MW</td>
</tr>
<tr>
<td>LODSB</td>
<td></td>
</tr>
<tr>
<td>LODSW</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>place: CB</td>
</tr>
<tr>
<td>LOOPE</td>
<td>place: CB</td>
</tr>
<tr>
<td>LOOPNE</td>
<td>place: CB</td>
</tr>
<tr>
<td>LOOPNZ</td>
<td>place: CB</td>
</tr>
<tr>
<td>LOOPZ</td>
<td>place: CB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: AB, src: XB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: AW, src: XW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EB, src: DB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EB, src: RB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EW, src: DB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EW, src: DW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EW, src: RW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: EW, src: S</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: RB, src: EB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: RW, src: EW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: S(ES), src: EW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: S(SS, DS), src: EW</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: XB, src: AB</td>
</tr>
<tr>
<td>MOV</td>
<td>dst: XW, src: AW</td>
</tr>
<tr>
<td>MOV</td>
<td>DI_ptr: MB, SI_ptr: MB</td>
</tr>
<tr>
<td>MOV</td>
<td>DI_ptr: MW, SI_ptr: MW</td>
</tr>
<tr>
<td>MOV</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>mplier: EB</td>
</tr>
<tr>
<td>MUL</td>
<td>mplier: EW</td>
</tr>
<tr>
<td>NEG</td>
<td>dst: EB</td>
</tr>
</tbody>
</table>

(Note 1)
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>dst:EW</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>dst:EB</td>
</tr>
<tr>
<td>NOT</td>
<td>dst:EW</td>
</tr>
<tr>
<td>NOT1</td>
<td>dst:Eb,off:D(0,7)</td>
</tr>
<tr>
<td>NOT1</td>
<td>dst:Eb,off:Rb(CL)</td>
</tr>
<tr>
<td>NOT1</td>
<td>dst:Ev,off:D(0,15)</td>
</tr>
<tr>
<td>NOT1</td>
<td>dst:Ev,off:Rb(CL)</td>
</tr>
<tr>
<td>OR</td>
<td>dst:AB,src:DB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:AW,src:DB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:AW,src:DW</td>
</tr>
<tr>
<td>OR</td>
<td>dst:EB,src:DB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:EB,src:RB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:EW,src:DB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:EW,src:DW</td>
</tr>
<tr>
<td>OR</td>
<td>dst:EW,src:RW</td>
</tr>
<tr>
<td>OR</td>
<td>dst:RB,src:EB</td>
</tr>
<tr>
<td>OR</td>
<td>dst:RW,src:EW</td>
</tr>
<tr>
<td>OUT</td>
<td>port:DB,dst:AB</td>
</tr>
<tr>
<td>OUT</td>
<td>port:DB,dst:AW</td>
</tr>
<tr>
<td>OUT</td>
<td>port:RW(DX),dst:AB</td>
</tr>
<tr>
<td>OUT</td>
<td>port:RW(DX),dst:AW</td>
</tr>
<tr>
<td>OUTS</td>
<td>port:RW(DX),SI_ptr:EB</td>
</tr>
<tr>
<td>OUTS</td>
<td>port:RW(DX),SI_ptr:EW</td>
</tr>
<tr>
<td>OUTSB</td>
<td></td>
</tr>
<tr>
<td>OUTSW</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>dst:EW</td>
</tr>
<tr>
<td>POP</td>
<td>dst:RW</td>
</tr>
<tr>
<td>POP</td>
<td>dst:S(ES)</td>
</tr>
<tr>
<td>POP</td>
<td>dst:S(SS,DS)</td>
</tr>
<tr>
<td>POPA</td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source/Count</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>src:DB(-128,127)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>PUSH</td>
<td>src:DW</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>PUSH</td>
<td>src:EW</td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>src:RW</td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>src:S</td>
<td></td>
</tr>
<tr>
<td>PUSHA</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>PUSHF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EB,count:DB(0,31)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EB,count:DB(1)</td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EB,count:RB(CL)</td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EW,count:DB(0,31)</td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EW,count:DB(1)</td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>dst:EW,count:RB(CL)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EB,count:DB(0,31)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EB,count:DB(1)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EB,count:RB(CL)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EW,count:DB(0,31)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EW,count:DB(1)</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>dst:EW,count:RB(CL)</td>
<td></td>
</tr>
<tr>
<td>REP</td>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>REPC</td>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>REPE</td>
<td>PREFIX</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>REPNC</td>
<td>PREFIX</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>REPNE</td>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>REPNZ</td>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>REPZ</td>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>src:DB</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>src:DW</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>dst:EB,count:DB(0,31)</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>dst:EB,count:DB(1)</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>dst:EB,count:RB(CL)</td>
<td></td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Count</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROL</td>
<td>dst:EW</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>ROL</td>
<td>dst:EW</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>dst:EW</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL4</td>
<td>dst:Eb</td>
<td></td>
<td></td>
<td>(Note 3)</td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EB</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EB</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EB</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EW</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EW</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>dst:EW</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR4</td>
<td>dst:Eb</td>
<td></td>
<td></td>
<td>(Note 3)</td>
</tr>
<tr>
<td>SAHF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAL</td>
<td>dst:EB</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SAL</td>
<td>dst:EB</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAL</td>
<td>dst:EB</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAL</td>
<td>dst:EW</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SAL</td>
<td>dst:EW</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EB</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EB</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EB</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EW</td>
<td>count:DB(0,31)</td>
<td></td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EW</td>
<td>count:DB(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>dst:EW</td>
<td>count:RB(CL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:AB</td>
<td>src:DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:AW</td>
<td>src:DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:AW</td>
<td>src:DW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:EB</td>
<td>src:DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:EB</td>
<td>src:RB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:EW</td>
<td>src:DB(-128,127)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:EW</td>
<td>src:DW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst:EW</td>
<td>src:RW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBB</td>
<td>dst:RB, src: EB</td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>dst: RW, src: EW</td>
<td></td>
</tr>
<tr>
<td>SCAS</td>
<td>DI__ptr: MB</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>SCAS</td>
<td>DI__ptr: MW</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>SCASB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCASW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET1</td>
<td>dst: Eb, off: D(0,7)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>SET1</td>
<td>dst: Eb, off: Rb(CL)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>SET1</td>
<td>dst: Ew, off: D(0,15)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>SET1</td>
<td>dst: Ew, off: Rb(CL)</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EB, count: DB(0,31)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EB, count: DB(1)</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EB, count: RB(CL)</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EW, count: DB(0,31)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EW, count: DB(1)</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>dst: EW, count: RB(CL)</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EB, count: DB(0,31)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EB, count: DB(1)</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EB, count: RB(CL)</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EW, count: DB(0,31)</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EW, count: DB(1)</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>dst: EW, count: RB(CL)</td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOS</td>
<td>DI__ptr: MB</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>STOS</td>
<td>DI__ptr: MW</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>STOSB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOSW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>dst: AB, src: DB</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>dst: AW, src: DB</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>dst: AW, src: DW</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>dst: EB, src: DB</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>dst:EB,src:RB</td>
</tr>
<tr>
<td>SUB</td>
<td>dst:EW,src:DB(-128,127)</td>
</tr>
<tr>
<td>SUB</td>
<td>dst:EW,src:DW</td>
</tr>
<tr>
<td>SUB</td>
<td>dst:EW,src:RW</td>
</tr>
<tr>
<td>SUB</td>
<td>dst:RB,src:EB</td>
</tr>
<tr>
<td>SUB</td>
<td>dst:RW,src:EW</td>
</tr>
<tr>
<td>SUB4S</td>
<td>dst:M,src:M (Note 3)</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:AB,src:DB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:AW,src:DB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:AW,src:DW</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EB,src:DB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EB,src:RB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EW,src:DB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EW,src:DW</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EW,src:RW</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EB,src:RB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:EW,src:DB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:RW,src:EB</td>
</tr>
<tr>
<td>TEST</td>
<td>dst:RW,src:EW</td>
</tr>
<tr>
<td>TEST1</td>
<td>dst:Eb,off:D(0,7) (Note 3)</td>
</tr>
<tr>
<td>TEST1</td>
<td>dst:Eb,off:Rb(CL) (Note 3)</td>
</tr>
<tr>
<td>TEST1</td>
<td>dst:Eb,off:D(0,15) (Note 3)</td>
</tr>
<tr>
<td>TEST1</td>
<td>dst:Ec,off:Rb(CL) (Note 3)</td>
</tr>
<tr>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:AW,src:RW</td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:EB,src:RB</td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:EW,src:RW</td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:RB,src:EB</td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:RW,src:AW</td>
</tr>
<tr>
<td>XCHG</td>
<td>dst:RW,src:EW</td>
</tr>
<tr>
<td>XLAT</td>
<td>table:MB</td>
</tr>
<tr>
<td>XLATB</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>dst:AB,src:DB</td>
</tr>
</tbody>
</table>
Table 8-2. Assembler Instruction Set (Cont’d)

<table>
<thead>
<tr>
<th>Operaion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR dst:AW,src:DB</td>
<td></td>
</tr>
<tr>
<td>XOR dst:AW,src:DW</td>
<td></td>
</tr>
<tr>
<td>XOR dst:EB,src:DB</td>
<td></td>
</tr>
<tr>
<td>XOR dst:EB,src:RB</td>
<td></td>
</tr>
<tr>
<td>XOR dst:EW,src:DB</td>
<td></td>
</tr>
<tr>
<td>XOR dst:EW,src:DW</td>
<td></td>
</tr>
<tr>
<td>XOR dst:EW,src:RW</td>
<td></td>
</tr>
<tr>
<td>XOR dst:RB,src:EB</td>
<td></td>
</tr>
<tr>
<td>XOR dst:RW,src:EW</td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:**
- CMPS: second operand must be ES addressable.
- INS: operand must be ES addressable.
- MOVES: first operand must be ES addressable.
- SCAS: operand must be ES addressable.
- STOS: operand must be ES addressable. The register is always used to address these operands, even if the ASSUMEd contents of ES and DS are the same.

**Note 2:**
These instruction/operand combinations will generate code that works correctly on an 80186, or a V20/V30, but not an 8086. A warning is printed whenever one of these combinations is used in 8086 Mode.

**Note 3:**
These instruction/operand combinations are accepted only in V20/V30 Mode.
Assembler Controls

Description of assembler controls and control defaults.
Chapter 9: Assembler Controls

Assembler controls are internal assembler switches which let you enable and disable certain aspects of the assembly process. This chapter describes assembler controls and control defaults.

If a [NO] appears in the heading, it indicates that the word NO can be prefixed to a control to make it do the opposite of what the control does normally. For example, LIST turns on the output listing, but NOLIST turns off printing of the listing. (The -L command line option causes a listing to be generated.)
General Syntax for Assembler Controls

The syntax of a control line in the source code is:

$control[(parameter)] [...]

The dollar sign may be preceded by tabs or blanks. Separators must be included between adjacent controls. Examples:

$XREF
$INCLUDE(filename) DEBUG SYMBOLS
$PRINT ERRORPRINT(FILENAME)

Primary and General Controls

Assembler controls are classified as either primary or general. Primary control statements occur only on the first few lines of the source program before any non-control statements (other than comments and blank lines). Primary controls are not processed when they occur after any statement other than a control line; their presence after any statement other than a control line causes an error. General controls, however, can be specified at any time in the source program. In most instances, an error in either kind of control line causes all remaining controls on the line to be ignored.

Controls on the Command Line

Assembler controls may also be included on the command line when the assembler is invoked. If a primary control is entered on both the command line and in the first lines of the source file being assembled, the control from the command line overrides the control in the file for that particular assembly.

If a general control is entered on both the command line and in the file (since general controls can appear anywhere in the file, the general control might be far, relatively, from the beginning of the file), then the control from the command line is in effect until the control in the source file is found. At that point, the source file control overrides the command line control for the rest of the assembly.
Chapter 9: Assembler Controls
General Syntax for Assembler Controls

Control Conflicts

If a primary control conflicts with another primary control and both are in the source file, then the one that appears last takes effect. If the conflict is between a control on the command line and one in the file, then the control which appears on the command line overrides the one in the file.

If general controls conflict (whether both in the file or one on the command line and one in the file), then the control which appears last will be the one to take effect. Example:

`$MOD186
$MOD086 ;this control is last, it will be the one to take effect`

Controls and File Names

Certain controls accept a file name as a parameter. The file name parameters are optional, except with INCLUDE, and are ignored with all controls except INCLUDE. The [NO] form of these controls does not accept a file name.

Control Abbreviations

Each control can be abbreviated to a two-character equivalent; the abbreviations are listed with each control. Abbreviations may be negated if the full name of the control can be negated. Controls are not case-sensitive; upper-case and lower-case letters are equivalent. Remember that their arguments may be case sensitive, although the controls are not.

Controls and the Macro Preprocessor (ap86)

Most controls are used only by the assembler. The INCLUDE control acts differently, however, if the source file is processed by the macro preprocessor before assembly. If the source file contains INCLUDE controls and does go through the macro preprocessor, then the macro preprocessor will expand the INCLUDEs. The output from the preprocessor will then contain the include files and will no longer contain the INCLUDE controls (not a problem, since they are no longer necessary). The macro preprocessor does not act on any other assembler controls.
Primary Controls

[NO]CASE

shorthand = [NO]CA
default = CASE

Causes symbols to be case sensitive. That is, upper and lower case letters are not equivalent. If this control is negated, then all lower case characters in symbols will be treated as upper case. This control does not affect text within strings (except for class names).

Note

All Intel-generated OMF will contain case insensitive symbols.

DATE(string)

shorthand = DA
(No default necessary.)

The DATE control has no effect. It is supplied for Intel compatibility, and its use will not generate an error. The date printed on the listing and placed in the object file is obtained from the operating system.
[NO]DEBUG

shorthand = [NO]DB
default = DEBUG

Causes symbolic debug and type information to be placed into the output object module. By default, only non-PURGEd variables, labels and numbers are placed into the object module.

[NO]ERRORPRINT (filename)

shorthand = [NO]EP
default= ERRORPRINT

This control causes error and warning information to be displayed on standard error. The filename, if present, is ignored and is only allowed for Intel compatibility. The noerrorprint control suppresses error and warning messages from being displayed on standard error. The nowarning control may be used to suppress only warning messages while allowing error messages to be displayed.

[NO]EXTERN_CHECK

shorthand = [NO]EC
default = EXTERN_CHECK

This control causes the use of external symbols to check that an ASSUME register has been defined such that the external symbol can be referenced from the ASSUME register. The noextern_check control causes the assembler to allow any use of an external symbol without verifying that the symbol is accessible through whatever assume register is used to reference that symbol. This then requires the user to make sure that segment registers are correctly set up to reference the segment that a given external symbol belongs to.
GEN

shorthand = GE
(No default necessary.)

Supplied for Intel compatibility. The assembler does no macro processing. This is done by the macro preprocessor, ap86(1). Therefore, this control has no effect.

GENONLY

shorthand = GO
(No default necessary.)

Supplied for Intel compatibility. The assembler does no macro processing. This is done by the macro preprocessor, ap86(1). Therefore, this control has no effect.

[NO]GROUP_INFO

shorthand = [NO]GI
default = GROUP_INFO

This control causes the debug information emitted from the assembler to associate group information to all symbols that belong to segments that are members of a group. Only one group will be assigned to a given symbol, regardless of how many groups a given segment belongs to. The nogroup_info control will only associate group information to labels and procedures; variables will NOT have group information associated.
[NO]HLASSYM

shorthand = [NO]HA  
default = NOHLASSYM

Causes as86 to generate low-level symbol information for static procedures, static data, and embedded assembly code. This option is useful when compiler-generated output is to be debugged in an emulator. If the output is to be debugged in AxDB or AxDE, then the negated form of this option is recommended.

[NO]MACRO(string)

shorthand = [NO]MR  
(No default necessary.)

Enables or disables macro assembly. Since macro processing is accomplished by a separate program, this control has no effect in either the assembler or the macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.

MOD086

shorthand = M0  
default = MOD086

Identifies the target microprocessor as 8086. The assembler generates errors for instructions that are not part of the 8086 microprocessor instruction set.
Chapter 9: Assembler Controls
Primary Controls

MOD186

shorthand = M1
default = MOD086

Identifies the target microprocessor as 80186. The assembler generates errors for instructions that are not part of 80186 microprocessor instruction set.

MODV20

shorthand = MV
default = MOD086

Identifies the target microprocessor as V20. Causes the V20/V30/V40/V50 instruction set to be recognized. Errors or warnings will be issued when instructions from conflicting instruction sets are encountered. All instructions are accepted without error in this mode.

Note

In V20 mode, the as86 assembler accepts the extensions that are specific to the NEC V20/V30 microprocessor. For these extensions, as86 accepts NEC mnemonics, but uses Intel instruction syntax. Where instructions might overlap (the NEC equivalent of an Intel instruction), as86 accepts only the Intel instruction mnemonic and not the NEC mnemonic or syntax (which might be different for the same function). This assembler is targeted for the 8086/186 family of microprocessors and should not be considered to support the NEC V20/V30 microprocessor instruction mnemonics or syntax.

[NO]OBJECT (filename)

shorthand = [NO]OJ
default = OBJECT
Generates an output object module, but the optional file name is ignored and only allowed for Intel compatibility. The assembler gives the object file the same root name as the source file, with a ".o" (dot lower case o) default file name extension.

[NO]OPTIMIZE

shorthand = OP
default = NOOPTIMIZE

This control will cause the assembler to spend extra time processing the input file so the resulting object file has as few NOPs as possible. These NOPs are generated when forward references are used in expressions. The assembler does not always know how many bytes of output will be produced for a given instruction, so it reserves extra space. If the instruction turns out to be shorter than that size, then the assembler pads the rest of the length with NOP bytes. This control will allow the assembler to spend time removing these NOPs when they are generated under these conditions. Note that this control will cause the assembler to run for a longer time than it otherwise would.

PAGELENGTH(n)

shorthand = PL
default = 55 lines per page

Specifies the page length of the listing as "n" lines, where n= 20 or more lines.

PAGEWIDTH(n)

shorthand = PW
default = 132 characters per line
Specifies the listing page width in number of characters, where n is a number between 60 and 255, inclusive. Lines exceeding the current page width are wrapped to the next line.

**[NO]PAGING**

shorthand = [NO]PI  
default = PAGING

Formats the output listing so as to have headers at the top of each page. By default, the headers supply the assembler name, title, and the date. If NOPAGING is specified, then the listing does not contain page headers or page ejects (except for an initial header on the first page). This option is only useful if a listing is produced.

**[NO]PRINT(filename)**

shorthand = [NO]PR  
default = NOPRINT

Prints the assembly listing. The noprint control suppresses the generation of the listing file including error messages and symbol table listings. You cannot override noprint with a list control occurring later in the program; however, a list control with no preceding print or noprint implies print. The file name parameter is accepted for Intel compatibility, but it is ignored by the assembler. Any lines that precede the print control will not be output to the listing. (Default: noprint)
Chapter 9: Assembler Controls
Primary Controls

[NO]SYMBOLS

shorthand = [NO]SB
default = SYMBOLS

Prints an alphabetically sorted symbol table with the output listing. The listing will not contain cross-reference information. Cross-reference information is produced with the XREF control. If XREF is used, it will override this control and cross-reference information will be produced. This option is only useful if a listing is output.

[NO]TYPE

shorthand = [NO]TY
default = TYPE

This control is recognized for Intel compatibility only and its use will not have any effect. Whether type information is generated depends upon the DEBUG control being on.

[NO]UNREFERENCED_EXTERNALS

shorthand = [NO]UE
default = NOUNREFERENCED_EXTERNALS

This control will cause all external symbols, including those that are unreferenced, to appear in the generated object file. In certain cases, these externals may be used to cause certain object files to be linked at link time. If this control is not present or if the NOUNREFERENCED_EXTERNALS control is used, any unreferenced externals will be removed from the resulting object file. This form of the control is useful when using inline functions in the Hewlett-Packard C cross compiler. This will prevent unnecessary routines from being linked in that are being processed inline.
[NO]WARNING

shorthand = [NO]WA
default = WARNING

This control causes warning messages to be displayed along with any error messages that may appear on standard error. The nowarning control suppresses the warning messages so only error information is sent to standard error. The errorprint control overrides either form of this control in determining whether any information is sent to standard error or not.

WORKFILES(...)

shorthand = WF

(No default necessary.)

This control has no effect. It is supplied for Intel compatibility, and its use will not generate an error.

[NO]XREF

shorthand = [NO]XR
default = NOXREF

Prints a cross reference table on the output listing. If you use both the XREF and SYMBOLS controls, a cross reference table will be generated.
General Controls

EJECT

shorthand = EJ

(No default necessary.)

Advances the listing form to the beginning of the next page and prints a new header. This is only useful if a listing is being generated and paging is in effect.

[NO]GEN

shorthand = [NO]GE

(No default necessary.)

This control has no effect in either the assembler or macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.

GENONLY

shorthand = GO

(No default necessary.)

This control has no effect in either the assembler or macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.
**INCLUDE(filename)**

shorthand = IC

(No default necessary.)

Indicates that the specified file should be included in the source input before the next line of the current source file is processed. Unlike other controls, INCLUDE must appear on a line by itself. No other controls, or other INCLUDEs, can be on the same line.

**Note**

The default directory for INCLUDE is always the current working directory. To use a file in another directory, specify the complete path name.

---

**[NO]LIST**

shorthand = [NO]LI

default = LIST

Turns on assembly listing at any point in the program. If used in combination with NOLIST, you can list a portion of the source file. NOLIST overrides XREF and SYMBOLS. An error summary still goes to stdout and errors still go to stderr regardless of LIST setting.

---

**RESTORE**

shorthand = RS

(No default necessary.)

Restores, as the current settings, the most recently-saved settings for LIST/NOLIST that are on the stack. This control is used mainly to restore LIST/NOLIST settings after returning from INCLUDE files.
SAVE

shorthand = SA

(No default necessary.)

Saves the current settings of LIST and NOLIST controls on a stack up to 64 entries deep. This control remains in effect until explicitly changed. SAVE is typically used with RESTORE where LIST/NOLIST settings are saved before an INCLUDE control switches the input source to another file. RESTORE can be used to restore the settings at the end of the include file or upon returning from the include file.

TITLE(string)

shorthand = TT
default = module name

Enables you to define a title of up to 41 characters in a page header. Unquoted parentheses in "string" must be balanced. String may be quoted if "unusual" characters are used in the title. The length of the title is bound by PAGEWIDTH. If you want the title to appear on the first page, use the TITLE control on the first source line or the command line.
Operational Differences in the Different Modes

The as86 operates in one of three modes depending upon the choice of control: MOD086, MOD186, or MODV20. The 8086 mode is the default.

8086 Mode

The default 8086 mode is the simplest mode. It is intended for assembling code destined for an 8086 or 8088. The pre-defined instructions which work in the 80186, but not the 8086, are flagged with errors when they appear. The 80186 instructions that will be flagged:

BOUND, ENTER, IMUL with 2 or 3 operands, INS, INSB, INSW, LEAVE, OUTS, OUTSB, OUTSW, POPA, PUSH immediate, PUSH A. The shifts RCL, RCR, ROL, ROR, SAL, SAR, SHL, and SHR with a numeric second operand other than 1.

The extensions for the V20 are also flagged with errors.

80186 Mode

The 80186 mode differs from the 8086 mode in that the pre-defined instructions listed in the 8086 mode discussion above do not generate an error when they are found. The V20 instructions, however, do generate errors.

V20 Mode

The V20 Mode differs from the 80186 mode in that it accepts the additional V20 predefined instructions: EXT, ADD4S, CMP4S, SUB4S, ROL4, ROR4, TEST1, NOT1, CLR1, BRKEM, FPO2, REPC, REPNC, and two additional operand combinations for INS. Remember that for those instructions common to the V20 and 8086 processor families, as86 accepts only Intel mnemonics. In addition, as86 accepts only Intel syntax for all instructions.

If you are assembling programs for NEC V-Series microprocessors, you should consider obtaining the Hewlett-Packard V-Series Cross Assembler.
Assembler Listing Description

Description of assembler listings, including a description of the optional symbol table and cross reference format.
Assembly Listing

The as86 Assembler uses a two-pass process. During the first pass, labels, variables and other user-defined symbols are examined and placed in the symbol table. Additionally, structures are stored internally.

During the second pass, the object code is generated, symbolic addresses are resolved, and a listing and object module are produced. Errors detected during the assembly process will be displayed on the output listing with a cumulative error count. At the end of the assembly process a symbol table or a cross reference table can be displayed.

The listing contains information pertaining to the assembled program, including op codes, assembled data and the original source statements. The listing can be used as a documentation tool by including comments and remarks that describe the function of the particular program segment.

A sample assembler listing is provided in Chapter 1. Refer to the following points to examine and understand the listing.

1. The page headings show the time and date of the program run.
2. The column titled "Line" contains decimal numbers associated with the listing source lines. These numbers are referred to in the cross reference table.
3. The column titled "Offset" contains a value that represents the first memory address of any object code generated by this statement.
4. The columns under "Object-Bytes" show the object code generated by instructions and directives in the file. Bytes are output lowest address first.
5. To the right of the data bytes are the assembler relocation flags. The flags are 'R' for relocatable operand, and 'E' for external operand. If one operand is relocatable and the other is external, the 'E' flag will be displayed.
6. The original source statements are reproduced to the right of the object-bytes field.
7. At the end of the listing the assembler prints the number of assembler errors. The assembler substitutes NOPs when it cannot translate a particular opcode and therefore provides room for patching the program.
A symbol table or cross reference table can be generated at the end of the assembly listing if the option specifying its output is used. All user-defined symbols, in alphabetic order, along with the symbol’s value type and attributes, are listed in the symbol table.

Cross Reference and Symbol Table Format Description

By default, the assembler produces a symbol table at the end of each listing. If you want the assembler to produce a cross reference table in place of the symbol table, use the XREF option.

If SYMBOLS and XREF are both specified, a cross reference table is produced. The cross reference table includes all the information present in the symbol table, but with line references noted for each symbol. The symbol table listing and cross reference features can be turned on only at the beginning of a program, and once on, cannot be turned off at a later point.

<table>
<thead>
<tr>
<th>Label</th>
<th>In the symbol table or cross reference listing header, the Label field lists the symbol name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>The Type field describes the kind of symbol represented by the Label. This field may be any of the following:</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>segment name</td>
</tr>
<tr>
<td>GROUP</td>
<td>group name</td>
</tr>
<tr>
<td>CLASS</td>
<td>class name</td>
</tr>
<tr>
<td>LOCAL</td>
<td>local variable</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>public variable</td>
</tr>
<tr>
<td>EXTERN</td>
<td>external variable or label</td>
</tr>
<tr>
<td>LABEL</td>
<td>local far or near label</td>
</tr>
</tbody>
</table>
Chapter 10: Assembler Listing Description
Cross Reference and Symbol Table Format Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUC</td>
<td>structure definition</td>
</tr>
<tr>
<td>STR_FLD</td>
<td>structure field name</td>
</tr>
<tr>
<td>REC</td>
<td>record definition</td>
</tr>
<tr>
<td>REC_FLD</td>
<td>record field name</td>
</tr>
<tr>
<td>EQU</td>
<td>equate symbol</td>
</tr>
<tr>
<td>PROC</td>
<td>procedure name</td>
</tr>
<tr>
<td>UNDEF</td>
<td>undefined symbol</td>
</tr>
</tbody>
</table>
Value

The Value field appears to the right of the Type field and is used to indicate attributes of the symbol. These attributes further describe what the symbol is or where the symbol resides. The specific attributes shown depend upon the Types above.

SEGM
Size of segment (in bytes), followed by combine type (PUBLIC/MEMORY/STACK/COMMON), followed by alignment (BYTE/WORD/PARA/PAGE/INPAGE/AT nnn), followed by classname, if present.

GROUP
List of segments that belong to the group. If a SEG EXTRN was used, then the name of the external will be displayed.

LOCAL, PUBLIC, EXTERN, LABEL, PROC
Segment name (if known), and offset within segment, followed by type (BYTE/WORD/ DWORD/QWORD/TBYTE/ NEAR/FAR/ABSOLUTE).

STRUC
Size of structure, followed by number of fields.

STR_FLD
Offset within structure, followed by type of field (BYTE/WORD/ DWORD/QWORD/TBYTE).

REC
Size of record, followed by number of fields, followed by width of record in bits.

REC_FLD
Bit offset within record, followed by width of field in bits.

EQU
If EQU’ed to a register, the name of the register is shown.

If EQU’ed to a 17-bit value, NNNN.

If EQU’ed to a real number, REAL.

If EQU’ed to an instruction, INSTRUCTION.
If EQU’d to a memory expression, EXPRESSION.

The UNDEF and CLASS types do not have any attributes.

**Cross Reference**

If a cross reference is being generated in addition to the symbol listing, then line references will appear to the right of the Value field. Each line reference will be separated from the next by a space.

The line on which the symbol is defined will have a minus sign placed before it. All other line numbers indicate references to the symbol. It is possible for there to be more than one definition of a symbol (for example, a segment). Also, purged symbols may appear more than once in the table.
Codemacros define 8086, 8087, 8088, 80186, and V20 instructions. A codemacro is a template for generating code, with certain bits fixed and other bits that are supplied when the codemacro is referenced (much as a record or structure). You must define the codemacro using the CODEMACRO directive before referencing it.

Referencing Codemacros

Formal arguments can be defined on the call line and then referenced in the body of the codemacro. Forward references to codemacros are illegal.

A codemacro is referenced by using its name in the opcode field of a source statement. You must provide actual parameters at this time, which must match the parameters as to the sort of entity described (number, WORD address expression, segment register, etc.). Matching is described in detail below. If matching is successful for all arguments, the codemacro is used to generate code. At this time, the formal arguments in the codemacro body will be replaced with data derived from the corresponding actual parameters.

Multiple codemacros with the same name are legal. When the name is referenced, each of the defined codemacros is checked to determine whether its formal arguments match the actual parameters you provide. The first codemacro whose arguments match is used to generate code. Multiple codemacros are checked in reverse order; the most recently-defined codemacro is checked first. This feature permits a single symbol to generate a variety of different code, depending on the arguments provided. When defined, as86 compiles the codemacro into a compact internal form and stores it in virtual memory.


Chapter 11: Codemacros
Codemacro Directives

**Codemacro Directives**

**CODEMACRO**

Enters Codemacro Definition

**Syntax**

```
CODEMACRO cmac_name [formal:specmod[range]][,formal:specmod[range]]...
...
ENDM
```

or

```
CODEMACRO cmac_name PREFIX
...
ENDM
```

**Description**

- **cmac_name**
  The name associated with the defined codemacro. It may have been previously defined as a codemacro, but not as anything else. This name is stored as a symbol and should not conflict with reserved words. Note that using an instruction name in this field is legal and results in an additional codemacro to be searched for that name.

- **formal**
  An arbitrary symbol defining a formal argument to the codemacro. Formals are not stored as symbols, and can duplicate keywords or even the cmac_name without conflict. Formals have no existence outside their codemacro and do not appear in the symbol table listing, although two formal parameters to the same codemacro cannot have the same name. A codemacro can have at most 255 formal arguments.

- **specmod**
  A letter or pair of letters describing the actual parameters that will match this formal parameter.

The legal values for specmod are:
Upper- and lower-case letters are interchangeable for these values. The convention of one upper-case letter followed by one lower-case letter is used in this chapter for clarity and to avoid confusion with the directives DB and DW. The first letter of the specmod is referred to as the specifier and the second letter as the modifier. The meaning of the various specmods is described in the table on page 223.

range
An optional field that follows a parameter. It describes a range of values that limits the acceptable modules for the parameters matching the formal argument. The first letter of the specmod must be A, D, R, or S. Any other type of specmod is not permitted to have a range field. The syntax and meaning of range fields is further described later in this section.

PREFIX
A keyword that can appear instead of the formal arguments, indicating that the codemacro name cannot take parameters. Instead, it is used to precede another codemacro or instruction name. At the time the codemacro is referenced, an error is detected if another codemacro or instruction does not follow this one.

PREFIX is associated with the codemacro name as a whole rather than separately with each codemacro. If one codemacro uses PREFIX, another codemacro with the same name must also use PREFIX. The last codemacro defined controls in case of conflict. A formal argument cannot be named PREFIX.
The CODEMACRO directive lets you enter the codemacro definition mode and specifies the formal arguments associated with the new codemacro. The ENDM is used to terminate the codemacro definition mode. Each CODEMACRO directive must have a corresponding ENDM directive, and codemacro definitions cannot be nested.

Examples

<table>
<thead>
<tr>
<th>CODEMACRO</th>
<th>CMAC1</th>
<th>FORMAL1: Ew, FORMAL2: Db(10, 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEMACRO</td>
<td>CMAC2</td>
<td>FORMAL3: S</td>
</tr>
<tr>
<td>CODEMACRO</td>
<td>CMAC3</td>
<td></td>
</tr>
<tr>
<td>CODEMACRO</td>
<td>CMAC4</td>
<td>PREFIX</td>
</tr>
</tbody>
</table>

**ENDM**

Terminates Codemacro Definition

**Syntax**

```
ENDM
```

**Description** The ENDM directive terminates the codemacro definition mode. Each ENDM must correspond to a CODEMACRO directive. For more information on ENDM, see the description of the CODEMACRO directive in the previous section.

---

**Codemacro Matching**

The assembler performs two passes on the input file to match codemacro references to definitions.

1. During pass 1, all actual parameters are evaluated. Parameters containing undefined symbols are called “forward references,” and are treated differently from other expressions. as86 is much more liberal concerning what a forward reference can match than what a fully-evaluated expression can match. Forward references are considered to be typeless unless type information is specifically attached with PTR or SHORT.

2. The chain of codemacros corresponding to the instruction mnemonic is searched, beginning with the last one defined. as86 looks for a codemacro
with the same number of formal arguments as there are actual parameters, such that each actual parameter matches the corresponding formal as far as specmod and range goes. Matching is described in the "Range Specification" section. The first codemacro that matches is used as described in # 3 below. If none matches, an error is reported.

3 The number of bytes of object code is estimated by executing the codemacro and discarding the generated bytes. This estimate is used to update the location counter. By default, forward references do not require a segment override byte from the SEGFI$X$, RFIXM, and RNFI$X$M directives.

4 During pass 2, the codemacro chain search starts at the beginning again. Presumably, all forward references have now been resolved. If not, an error is issued and the absolute number 0 is substituted for the undefined symbol, which may in turn cause other errors. This resolution of forward references can cause a different codemacro to be matched than in pass 1. If none matches, an error is reported. If a codemacro matches in pass 1, it does not necessarily have to match in pass 2.

5 Code is generated using the matched codemacro. A different number of bytes of code can be generated than was called for in the estimate from pass 1. If more code is generated in pass 1 than in pass 2, the extra room allocated is filled with NOPs (90H). If more code is generated in pass 2 than in pass 1, an error message is issued and the entire space allocated is filled with NOPs.
The Specmod Field

The specmod field determines what actual parameters match each formal argument. In the table which follows, “variable” is an address expression with type BYTE, WORD, DWORD, QWORD, TBYTE, a structure name, or a record name, and “label” is an address expression with type NEAR or FAR. For the purpose of matching, forward references during pass 1 are treated as a special kind of expression that match certain specmods. Specmods match actual parameters as shown in the table.

Table 11-1. Specmods and Parameter Matches

<table>
<thead>
<tr>
<th>Specmod</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AX or AL.</td>
</tr>
<tr>
<td>Ab</td>
<td>AL.</td>
</tr>
<tr>
<td>Aw</td>
<td>AX.</td>
</tr>
<tr>
<td>C</td>
<td>Any label, or any forward reference of type NEAR or FAR or no_type.</td>
</tr>
<tr>
<td>Cb</td>
<td>Any NEAR label with the same segment definition attribute as the current assumed contents of CS via ASSUME and within the range -128 to + 127 from the beginning of the code macro reference, or any forward reference with SHORT attached.</td>
</tr>
<tr>
<td>Cd</td>
<td>Any FAR label, or any forward reference without a type or of type FAR.</td>
</tr>
<tr>
<td>CW</td>
<td>Any NEAR label with the same segment definition attribute as the current assumed contents of CS via ASSUME but farther away from the beginning of the codemacro reference than -128 to + 127, or any external NEAR label</td>
</tr>
<tr>
<td>D</td>
<td>Any 17-bit number, or any forward reference with no type.</td>
</tr>
<tr>
<td>Db</td>
<td>Any absolute number between -256 and 255, inclusive, or any number of relocation type high or low</td>
</tr>
</tbody>
</table>
### Table 11-1. Specmods and Parameter Matches (Cont’d)

<table>
<thead>
<tr>
<th>Specmod</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dw</td>
<td>Any absolute number not between -256 and 255 inclusive, or any number of relocation type offset or base, or any forward reference with no type.</td>
</tr>
<tr>
<td>E</td>
<td>Any variable, or any address expression without a type, or any register except segment registers, or any forward reference, except for those typed NEAR</td>
</tr>
<tr>
<td>Eb</td>
<td>Any variable with type BYTE</td>
</tr>
<tr>
<td>Ed</td>
<td>Any variable with type DWORD, or any forward reference of type DWORD or no type.</td>
</tr>
<tr>
<td>EW</td>
<td>Any variable with type WORD, or any 16-bit register, except segment registers, or any forward reference of type WORD or no type.</td>
</tr>
<tr>
<td>F</td>
<td>The floating-point stack or any element thereof: ST</td>
</tr>
<tr>
<td>M</td>
<td>Any variable or any address expression without a type, or any forward reference except those of type NEAR</td>
</tr>
<tr>
<td>Mb</td>
<td>Any variable with type BYTE, or any forward reference of type BYTE or no type.</td>
</tr>
<tr>
<td>Md</td>
<td>Any variable with type DWORD, or any forward reference of type DWORD or no type.</td>
</tr>
<tr>
<td>Mq</td>
<td>Any variable with type QWORD, or any forward reference of type QWORD or no type.</td>
</tr>
<tr>
<td>Mt</td>
<td>Any variable with type TBYTE, or any forward reference of type TBYTE or no type.</td>
</tr>
<tr>
<td>Mw</td>
<td>Any variable with type WORD, or any forward reference of type WORD or no type.</td>
</tr>
<tr>
<td>R</td>
<td>Any register except segment registers.</td>
</tr>
<tr>
<td>Rb</td>
<td>Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL).</td>
</tr>
<tr>
<td>Rw</td>
<td>Any 16-bit register except segment registers (AX, BX, CX, DX, SI, DI, BP, SP)</td>
</tr>
</tbody>
</table>
Table 11-1. Specmods and Parameter Matches (Cont’d)

<table>
<thead>
<tr>
<th>Specmod</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Segment registers (ES, DS, SS, CS)</td>
</tr>
<tr>
<td>T</td>
<td>The floating-point stack top: ST or ST(0) only.</td>
</tr>
<tr>
<td>X</td>
<td>Any variable or any address expression without a type, whose base and index attributes are null or any forward reference except those of types NEAR</td>
</tr>
<tr>
<td>Xb</td>
<td>Any variable of type BYTE whose base and index attributes are null, or any forward reference of type BYTE or no type.</td>
</tr>
<tr>
<td>Xd</td>
<td>Any variable of type DWORD whose base and index attributes are null</td>
</tr>
<tr>
<td>Xq</td>
<td>Any variable of type QWORD whose base and index attributes are null, or any forward reference of type QWORD or no type.</td>
</tr>
<tr>
<td>Xt</td>
<td>Any variable of type TBYTE whose base and index attributes are null, or any forward reference of type TBYTE or no type.</td>
</tr>
<tr>
<td>Xw</td>
<td>Any variable of type WORD whose base and index attributes are null, or any forward reference of type WORD or no type.</td>
</tr>
</tbody>
</table>

In addition, typeless address expressions such as [BX] will sometimes match the specmods Eb, Ew, Mb, and Mw. There must be enough information for as86 to infer the size of the operation. This condition is met if the codemacro has at least two formal arguments, and one or more of the actual parameters corresponding to the other argument(s) is not either another typeless address expression or a number that matches Db.

For example, suppose a codemacro has ARG1:Ew, ARG2:Ew as the formal arguments. The actual parameters [BX], AX match, since AX implies a WORD operation; however, the actual parameters [BX],[BX] do not match since the information to infer the size of the operation is insufficient. This condition means that any codemacro with a single formal parameter of specmod Eb, etc., cannot match a typeless address expression, including several of the built-in instructions (e.g., INC, FISUB, IMUL).
A few built-in instructions (e.g., FLDENV) have the specmod M on their single formal parameter and, therefore, will accept a typeless address expression.
Range Specification

A codemacro range is a parenthesized list of one or two expressions separated by a comma. The syntax of a range specification is:

(value1[ , value2])

Each value must be a register name or an expression evaluating to an absolute number (i.e. not an address). Registers are converted to absolute numbers according to the following table.

Table 11-2. Absolute Number Conversion for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL, AX, ES</td>
<td>0</td>
</tr>
<tr>
<td>CL, CX, CS</td>
<td>1</td>
</tr>
<tr>
<td>DL, DX, SS</td>
<td>2</td>
</tr>
<tr>
<td>BL, BX, DS</td>
<td>3</td>
</tr>
<tr>
<td>AH, SP</td>
<td>4</td>
</tr>
<tr>
<td>CH, BP</td>
<td>5</td>
</tr>
<tr>
<td>DH, SI</td>
<td>6</td>
</tr>
<tr>
<td>BH, DI</td>
<td>7</td>
</tr>
</tbody>
</table>

Some codemacros have specific limits on the range of parameters that can be used. This pertains to formals using specifiers A, D, R, or S.

When codemacros are referenced, the actual parameter is checked against the specified range, converting actual registers according to the table. If the range field contained a single value, the actual parameter must match it. If the range field contained two values, the actual parameter must be greater than or equal to the first and less than or equal to the second. Otherwise, the actual parameter does not match. Relocatable actual parameters and forward references never match a formal with a range field.
Chapter 11: Codemacros
Range Specification

Examples:

S(0,2)   Matches ES, CS, or SS.
S(0)     Matches only ES.
Db(2,-1) Generates error - invalid range.
Db(-1,2) Matches -1, 0, 1, or 2.
          255 does not match (9-bit comparison).
Db(-1,DL) Same as previous example.
Rw(DX)   Matches DX.
Rb(CL)   Matches CL.
Db(1)    Matches 1.
Codemacro Matching Examples

This table shows a list of the arguments on some example codemacros for the MOV instruction, in the order they are searched, along with actual parameters that will match each. WORDVAR is a variable of type WORD, and BYTEVAR is a variable of type BYTE.

Table 11-3. Arguments and Actual Parameters

<table>
<thead>
<tr>
<th>Codemacro Reference</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV WORDVAR,AX</td>
<td>MOV dst:Xw,src:Aw</td>
</tr>
<tr>
<td>MOV BYTEVAR,AL</td>
<td>MOV dst:xb,src:Ab</td>
</tr>
<tr>
<td>MOV AX,WORDVAR</td>
<td>MOV dst:Aw,src:Xw</td>
</tr>
<tr>
<td>MOV AL,BYTEVAR</td>
<td>MOV dst:Ab,src:Xb</td>
</tr>
<tr>
<td>MOV SS,WORDVAR</td>
<td>MOV dst:S(SS,DS),src:Ew</td>
</tr>
<tr>
<td>MOV WORDVAR,CS</td>
<td>MOV dst:Ew,src:S</td>
</tr>
<tr>
<td>MOV CX,WORDVAR</td>
<td>MOV dst:Rw,src:Ew</td>
</tr>
<tr>
<td>MOV CL,BYTEVAR</td>
<td>MOV dst:Rb,src:Eb</td>
</tr>
<tr>
<td>MOV DS:[BX],AX</td>
<td>MOV dst:Ew,src:Rw</td>
</tr>
<tr>
<td>MOV DS:[BX],AL</td>
<td>MOV dst:Ew,src:Rb</td>
</tr>
<tr>
<td>MOV CX,1000</td>
<td>MOV dst:Rw,src:Dw</td>
</tr>
<tr>
<td>MOV CX,20</td>
<td>MOV dst:Rw,src:Db</td>
</tr>
<tr>
<td>MOV CL,20</td>
<td>MOV dst:Rb,src:Db</td>
</tr>
<tr>
<td>MOV WORDVAR,1000</td>
<td>MOV dst:Ew,src:Dw</td>
</tr>
<tr>
<td>MOV WORDVAR,20</td>
<td>MOV dst:Ew,src:Db</td>
</tr>
<tr>
<td>MOV BYTEVAR,20</td>
<td>MOV dst:Ew,src:Db</td>
</tr>
</tbody>
</table>
Chapter 11: Codemacros
Codemacro Matching Examples

The following is a list of some instructions that do not match the formal argument pairs.

MOV CS,WORDVAR ; CS is not between SS and DS, ; and not equal to ES.
MOV ES,BYTEVAR ; No such 8-bit operation appears.
MOV WORDVAR,BL ; In general, 8-bit and 16-bit operands ; cannot mix.
MOV BL,WORDVAR ; Mixed 8- and 16-bit operands.
MOV BL,1000 ; Mixed 8- and 16-bit operands. 1000 won’t fit in BL.
MOV BYTEVAR,1000 ; Mixed 8- and 16-bit operands. 1000 won’t fit in ; BYTEVAR either.
Expressions in Codemacros

Only a small subset of the usual expressions is available within codemacro definitions. The following are allowed:

- Absolute numbers, and expressions which evaluate to absolute numbers. No forward references are allowed within such expressions.
- Segment registers.
- Formal argument names.
- Shifted formal arguments.

Syntax:

```
formal_name.recordfield
```

where formal_name and recordfield are symbols. This means to perform a right shift of the actual parameter corresponding to the formal_name at the time the codemacro is referenced, by the number of bits given by the shift count of the recordfield. The actual parameter must be an expression that evaluates to an absolute number. If the actual parameter is a relocatable number, an error is reported at the time the codemacro is referenced. The predefined ESC instruction uses this construct.

PROCLEN

PROCLEN has the value 255 if the most recently defined PROC at the time of codemacro reference was declared FAR. It has the value 0 otherwise. Thus, if the codemacro reference is not in a PROC, PROCLEN yields 0.
Directives within Codemacros

Only a few directives are legal within a codemacro definition, and these are listed below. Instructions are not allowed within a codemacro definition, but assembler controls and comments are; however, the assembler control is not considered part of the codemacro. None of these directives are allowable outside a codemacro definition unless so described elsewhere in this manual (e.g. DB, DW, DD, and record names).

The following pages describe directives within codemacros.

Table 11-4. Directives within Codemacros

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>Generates byte of immediate data.</td>
</tr>
<tr>
<td>DD</td>
<td>Generates 4 bytes of immediate data.</td>
</tr>
<tr>
<td>DW</td>
<td>Generates 2 bytes of immediate data.</td>
</tr>
<tr>
<td>MODRM</td>
<td>Generates ModRM byte.</td>
</tr>
<tr>
<td>NOSEGFIX</td>
<td>Checks for addressability through a certain seg register.</td>
</tr>
<tr>
<td>ONLY186</td>
<td>(186 Mode only) Identifies 186-only instructions.</td>
</tr>
<tr>
<td>RELB</td>
<td>Generates 1 or 2 bytes using the specified record template.</td>
</tr>
<tr>
<td>RELW</td>
<td>Generates 1-byte displacement.</td>
</tr>
<tr>
<td>RFIX</td>
<td>Generates a WAIT (9BH) followed by the first 5 bits of an ESC(0D8H).</td>
</tr>
<tr>
<td>RFIXM</td>
<td>Generates a WAIT (9BH) followed by a segment override byte (if needed) followed by the first 5 bits of an ESC (0D8H).</td>
</tr>
<tr>
<td>RNFIX</td>
<td>Generates an NOP (90H) followed by the first 5 bits of an ESC(0D8H).</td>
</tr>
<tr>
<td>RNFIXM</td>
<td>Generates a NOP (90H) followed by a segment override byte (if needed) followed by the first 5 bits of an ESC (0D8H).</td>
</tr>
<tr>
<td>RWFIX</td>
<td>Generates a WAIT (9BH).</td>
</tr>
<tr>
<td>SEGFIX</td>
<td>Generates segment-override byte if needed.</td>
</tr>
</tbody>
</table>
DB, DD, DW

Generates N-Bytes of Immediate Data

Syntax

DB absolute_numeric_expression
DB formal_name
DB formal_name.recordfield
DD absolute_numeric_expression
DD formal_name
DD formal_name.recordfield
DW absolute_numeric_expression
DW formal_name
DW formal_name.recordfield

Description

absolute_numeric_expression  An absolute numeric expression.

formal_name  A name that is a formal parameter to the codemacro.

formal_name.recordfield  A name that is a formal parameter to a codemacro but
                         shifted according to the recordfield.

The DB, DD, and DW directives are similar to their counterparts outside codemacros, but their legal operands are much more restricted.

Each consecutive appearance of a DB, DW, or DD directive within a codemacro generates one, two, or four bytes, respectively.

It is possible for a formal argument with specmod Dw to appear in a DB directive, where it will not fit, which will then cause an error at the time of codemacro reference.

A formal_name without a recordfield must be of specifier D for the DB directive and must be of specifier D, C, or X for the DW and DD directives. (A specifier is the first letter of a specmod listed beginning on page 223.)

A formal_name appearing with a recordfield must have specifier D.
MODRM
Generates ModRM Byte

Syntax

MODRM formal_name2,formal_name1

or

MODRM number,formal_name1

Description

formal_name1 An effective-address parameter. It must have a specifier of E, M, R, X, A, or S. (A specifier is the first letter of a specmod listed beginning on page 223.)

formal_name2 A parameter, usually a register. It must have a specifier of D, R, A, or S.

number An expression evaluating to an absolute number.

MODRM generates the ModRM byte, which can contain a wide variety of information: a register involved in the instruction, the base and index registers of an operand, the addressing mode (direct address, relative to the current location, immediate, register), a continuation of the opcode, etc.

as86 derives 5 bits of information from formal_name1, and 3 bits from the first parameter. If the first operand of MODRM is a number that is either a constant or a formal matching D, the low 3 bits are used in the generated byte. If the first operand is a register with a matching A, R, or S, the 3 bits to use are taken from the numeric values corresponding to registers as described in the section on Range Specification.
Chapter 11: Codemacros
Directives within Codemacros

NOSEGFIX
checks for Addressability

Syntax

NOSEGFIX  segreg,  formal_name

Description

segreg  One of the segment registers ES, CS, DS, SS.

formal_name  A formal argument name whose specifier is E, M, or X (a memory parameter).

NOSEGFIX ensures that a parameter is addressable through a specific segment register. It is used in the built-in instruction set for the string instructions MOV$S, STOS, CMPS, SCAS, INS[186], for which one operand must be addressable through ES.

NOSEGFIX checks the segment addressability attribute of the actual parameter corresponding to the formal_name to ensure that the parameter is addressable through the specified segment register. If the actual parameter is a register (matching E), it is considered addressable. If the attribute is a segment register, it must match the register on the NOSEGFIX. If the attribute is null, it is not addressable. If the attribute is a segment or group, as86 checks the assumed contents of the specified segment register through ASSUME, as it does for SEGFIX. NOSEGFIX never generates any code. It merely performs an error check. Note that this check is not performed at argument matching time. It is possible for the actual parameters to match the formal arguments of a codemacro that contains a NOSEGFIX directive and then get an error on the NOSEGFIX, even if another codemacro exists farther along in the codemacro chain that would not get this error. No codemacro in the built-in instruction set can do this.
Chapter 11: Codemacros
Directives within Codemacros

**ONLY186 (186 Mode Only)**

Identifies 186-Only Instructions

**Syntax**

`ONLY186`

**Description**  ONLY186 issues a warning message if the assembler is in 8086 mode. Generation of code proceeds normally. This protects you from accidentally writing a 186-only instruction which will not work when the target machine is an 8086.
Record Name Initialization

Syntax

recordname<expression>[,expression]...

Description

recordname The name of a previously-defined record.

expression One of the following:

• An expression evaluating to an absolute number
• A formal argument
• A formal argument plus a .recordfield
• Null
• PROCLEN

The record initialization directive lets you control bit fields in codemacro definitions.

Formal arguments in either construct (with or without a .recordfield) must be of specifier D, and the corresponding actual parameter cannot be relocatable or an error will be reported when codemacros are expanded.

Each expression must evaluate to an absolute number, and only the bits corresponding to the defined size of each .recordfield are used. Also, the least significant bits of the expression value are used, and more significant bits are discarded without any check. Null fields, as well as records outside codemacros, result in the use of the default value at the time the record was defined.
RELB, RELW

Generates N-byte Displacement

Syntax

RELB formal_name
RELW formal_name

Description

formal_name The name of a formal parameter to the codemacro with specmod type C.

The RELB and RELW directives generate a one- or two-byte displacement, respectively, denoting the distance from the location of the codemacro reference to a target which can only be a label. The displacement is measured from the location after the bytes generated by RELB or RELW. Specifically, if the target is the byte immediately following the generated displacement whether that is 1 or 2 bytes, the generated displacement will be 1. These directives take one operand, a formal argument that must be of specmod Cb or Cw. RELB and RELW do not concern themselves with segment addressability or the contents of CS.

During codemacro matching to Cb and Cw specmods, the assembler assumes that any RELB or RELW in the codemacro will follow exactly one generated byte and, as a result, the restriction of the displacement for Cb to -126 to + 129 occurs. This assumption is correct for all codemacros in the built-in instruction set. You can write codemacros for which this assumption does not hold. For example, you can write one equivalent to several predefined instructions, but if this is done, the wrong match can be made at codemacro reference-time.
RFIX, RFIXM, RNFIX, RNFIXM, RWFIX

Generates WAIT or NOP

Syntax

<table>
<thead>
<tr>
<th>Directive</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFIX</td>
<td>formal_or_number</td>
</tr>
<tr>
<td>RFIXM</td>
<td>formal_or_number, formal_name</td>
</tr>
<tr>
<td>RNFIX</td>
<td>formal_or_number</td>
</tr>
<tr>
<td>RNFIXM</td>
<td>formal_or_number, formal_name</td>
</tr>
<tr>
<td>RWFIX</td>
<td></td>
</tr>
</tbody>
</table>

Description

formal_or_number  A codemacro parameter with specifier type D or an absolute expression that evaluates to an absolute number.

formal_name       A codemacro parameter with specifier type E, M, or X.

These closely-related directives pertain to floating-point instructions. In all modes, they generate bytes as follows:

- **RFIX**
  - WAIT (9BH) followed by the first word of an ESC (0D8H)

- **RFIXM**
  - WAIT (9BH) followed by a segment override byte (if needed) followed by the first word of an ESC (0D8H)

- **RNFIX**
  - NOP (90H) followed by the first word of an ESC (0D8H)

- **RNFIXM**
  - NOP (90H) followed by a segment override byte (if needed) followed by the first word of an ESC (0D8H)

- **RWFIX**
  - WAIT (9BH)

RFIX and RNFIX have one operand; RFIXM and RNFIXM have two operands; RWFIX has no operands. The first operand of each, except RWFIX, is either a formal parameter with specifier D or an expression evaluating to an absolute number. The least significant 3 bits of this operand are taken as the last 3 bits of the generated ESC. If the corresponding actual parameter is relocatable, an error is reported when codemacros are referenced.
The second operand of RFIXM and RNFIXM is a formal argument of specifier E, M, or X representing a memory address. The segment override byte is issued or not, depending on this parameter; the algorithm is exactly the same as that described under SEGFIX.

The preceding descriptions assume that the object code will be used on an 8087 chip. These directives are designed for use within floating-point instructions. However, if the linker references the 8087 emulator library instead, the WAIT and NOP instructions described are changed into instructions to the emulator. The linker performs this function by resolving external references generated by the R?FIX? directives. This is why, for instance, a codemacro uses RWFIX instead of DB 9BH.

Intel provides two libraries, one of which is used as input to its linker for any given absolute object module. One library is used if the code is destined for an 8087, and the other is used if the 8087 is to be emulated.

This use of built-in external references, which typically will not be of concern to you, also means that any codemacro employing one of these directives displays an E flag (i.e. external reference) on the output listing when referenced. This includes all the floating-point instructions in the built-in instruction set.
SEGFIX

Generates Segment-Override Byte

**Syntax**

SEGFIX formal_name

**Description**

formal_name A codemacro parameter with specifier type E, M, or X.

The SEGFIX directive generates a segment-override byte, if needed (either 26H, 2EH, 36H, or 3EH). This instructs the hardware to use a different segment register for the following instruction.

SEGFIX has one parameter which must be a formal argument name. This argument represents a memory address and, therefore, must have one of the specifiers (1st letter of the specmod) E, M, or X. A register (matching E) never generates a segment override. An address expression has its segment addressability attribute checked as follows:

- If this attribute is null, an error is reported.
- If the attribute is a segment register, that register is used for addressing.
- If the attribute is a group, the assumed contents of the segment registers via ASSUME are checked to see if one of them contains the group.
- If the attribute is a segment, the assumed contents of the segment registers via ASSUME are checked to see if one of them contains the segment or a group containing the segment.

In the last two cases, the segment registers are examined in this order:

1. The register implied by the base and index attributes of the actual parameter (DS or SS).
2. The other registers are examined in the order ES, CS, SS, DS.

The first register for which the check succeeds is used for addressing. If the actual parameter cannot be addressed through any segment register, an error is issued. Otherwise, once as86 has determined which segment register to use for addressing, it determines whether that register is the default implied by the base and index attributes. If so, no override byte is generated; if not, a segment
override byte corresponding to the segment register used for addressing is generated.
Chapter 12: Macro String Preprocessor Introduction

Macro String Preprocessor Introduction

Introduction to the Macro String Preprocessor.
The Macro String Preprocessor (ap86) is a character string replacement program which performs pre-assembly processing of macros in assembly language source files. It searches the source code for macro calls, and then replaces those calls with the macro return values. The advantage of having the macro string preprocessor is to permit frequently-used segments of code to be used repeatedly by one or several users from a library, without having to re-write the code for each use. You can automatically insert a section of code into the source program by encoding a single line—the macro call.

At definition time, key constructs in the macro may be represented by formal parameters; actual parameters are later substituted for the formal ones. ap86 handles conditional assembly, assembly-time loops, and is also capable of recursion.

**Note**
The macro preprocessor is case sensitive by default. Upper and lower case characters are not equivalent to the preprocessor. The macro symbol MACSYM would not be the same as macSYM, MaCSYM, or macsym. Case sensitivity can, however, be turned off on the command line.

ap86 is implemented as a program separate from the assembler, thereby saving time for those who do not use macros. It is compatible with the Intel syntax for the 8086/186 macro languages. If you use macros in the source code, you must run the Macro Preprocessor to produce an output file for input to the assembler.

**Input Source Characteristics**
ap86 views its input file as a stream of characters instead of a sequence of statements. All processing is character-oriented. The ends of lines are treated as if they ended with a < line feed> . This character is called 'end-of-line’ or '< EOL> ' in text that follows.
The Metacharacter ‘%’ And The Call Pattern

The macro preprocessor searches the input source one character at a time, looking for a special character called the metacharacter. By default, this character is the percent sign (‘%’), but it can be dynamically changed. Until the metacharacter is found, characters are passed to the output file without change. When the metacharacter is found, the macro preprocessor reads and interprets the characters following it, isolating a call pattern. The call pattern is interpreted as instructions to the macro preprocessor and is not passed to the output file. However, the macro preprocessor produces an expansion of the call pattern that is written to the output file in place of the call pattern. The call pattern can contain other metacharacters followed by call patterns; these will also be expanded. Expansions are stacked, analogous to nested subroutines. When the current expansion is complete, the stack is popped, and the next higher expansion resumes where it left off. The expansion of a call pattern is always a string of characters which can be null (zero characters) in some cases, but most often it is one or more characters. When the outermost expansion is completed, the macro preprocessor goes back to copying characters while scanning for the metacharacter.

The source code below has statements that contain macros.

```
NOP
asymbl EQU 2
DB %LEN( %SUBSTR(5 DUP (0),1,1)) ;note blank before
; %SUBSTR
ADD AX,2
```

The example source code is treated by the macro preprocessor in this way:

1. Everything up to the first "%" is passed to the output unchanged. The text has no significance to the macro preprocessor.

2. The first "%" invokes the pre-defined macro function LEN, which counts the characters in its argument. (LEN, SUBSTR, and other pre-defined macro functions used in these examples are described in detail in the chapter called "Pre-defined Macro Functions.")

   Everything up to but not including the balancing right parenthesis (in this example, the last parenthesis) is the argument to LEN.

3. The argument to LEN contains a call to another pre-defined macro function, SUBSTR, which extracts a substring from its first argument according to parameters in the second and third arguments. The
expansion of the outer function LEN therefore pauses while SUBSTR is evaluated.

4 In this example, the result of SUBSTR is the single character '5'. After the evaluation, LEN resumes, in effect evaluating "%LEN(5)" (again, notice the space in front of the 5). This produces the string "02H," which is passed to the output.

The space between "%LEN(" and "%SUBSTR" is a significant part of the LEN argument, but is not part of the call to SUBSTR. Following "02H," ap86 puts out the <EOL>, which is the next character following the call pattern of LEN in the source file. Notice that <EOL> is not part of the call pattern. The assembler, therefore, sees the following line of text:

```
DB 02H<EOL>
```

### Metacharacter Syntax

The metacharacter can be followed by

- a symbol
- a left parenthesis ( 
- an apostrophe '
- a decimal digit
- an asterisk * (called the literal character), that in turn must be followed by a symbol.

No other characters are acceptable, particularly spaces and tabs. A symbol following the metacharacter (or the metacharacter-asterisk pair) must be one of three things:

- A pre-defined macro function.
- A call to a previously-defined user macro.
- A reference to a previously-defined macro-expansion-time symbol or, within a macro body, a formal argument or local symbol. The
metacharacter is recognized anywhere in the source text, including within character strings.

Getting a line such as

```
DB '20% inflation'
```

to pass through the macro preprocessor requires special handling. Getting these strings through the macro preprocessor is discussed in the "%n and % (Escape and Bracket Functions) in the chapter titled "Pre-defined Macro Functions."

---

**Literal Character * **

The literal character (*) specifies that metacharacters contained in the arguments to a function are not expanded. The literal character is placed between the metacharacter and the function or macro name, and spaces or other separators cannot precede or follow it. The literal character inhibits the expansion of all user macros, symbols, and pre-defined functions. It does not affect formal macro parameters, local symbols within macros, and the escape, comment and bracket functions. If one of the lines of code from the previous example were rewritten to contain the literal character before the LEN macro name,

```
DB %*LEN(%SUBSTR(5 DUP (0),1,1))
```

then the SUBSTR call is not expanded. Instead, LEN counts the length of the string "%SUBSTR(5 DUP (0),1,1)" and returns the string "16H." Output to the assembler would then be

```
DB 16H <EOL>
```

If the literal character preceded SUBSTR instead of LEN, it would have no effect in this example because the argument to SUBSTR does not contain any metacharacters. Misuse of the literal character causes the macro preprocessor to pass strings containing a metacharacter on to the assembler, where they will usually be flagged as errors. The literal character is prohibited all together with some functions; other functions accept it, but ignore it. The literal character should almost always be used when defining a user-macro.
Input Parsing

The macro preprocessor recognizes the operators listed on page 250. The macro preprocessor only understands symbols in specific constructs which are usually preceded by the metacharacter. Assembly-time user-defined symbols (labels, etc.), the location counter, and EQUs are all unknown to the macro preprocessor.

You must be careful that a macro call produces each \textless EOL\extrm{ } in the right place. Readable input to the macro preprocessor frequently results in a large number of output lines consisting only of blanks and end-of-lines. For user convenience and assembler speed, such lines are always omitted from the output. To create a blank line, deliberately use a blank comment line.

Output Buffering

The macro preprocessor buffers its output in an array that can hold 256 characters. When its buffer is full and another character (other than \textless EOL\extrm{ }) is received, ap86 breaks the output line into two pieces. The break occurs at the 256 character boundary and the remaining text is placed on the next line of output. This and all other lines created from the long input line will begin with a \text{"&\text{"} so the assembler can recognize the line as a continuation. Since the break is made at a fixed location, it is likely that the result will cause a syntax error in the assembler. Thus, it is best if line lengths are restricted to less than 256 characters.

Include Files

INCLUDE is an assembler control command, but the macro preprocessor will act on INCLUDE also. INCLUDE statements cause the macro preprocessor to temporarily stop reading source statements from the current file. It begins reading source statements from the file specified by the INCLUDE. It continues reading from the include file until it finds the end-of-file for the include file or it finds another INCLUDE. When the preprocessor resolves all
INCLUDEs and does find the end-of-file for the include file, it then returns to the file that contained the INCLUDE statement and again begins reading source statements immediately after the INCLUDE statement.

**Note**

The maximum depth that the macro preprocessor can handle nested INCLUDE controls is to a level of eight. The restriction on the assembler depends only upon the number of open files the operating system allows at one time.

The syntax for the INCLUDE statement:

```
$INCLUDE(filename)
```

The '$' must be in column 1 for the preprocessor to recognize it for processing.

The default directory for INCLUDE is always the current working directory.

Any INCLUDE starting in column 1 of a source statement, whether from a source file or an include file, is processed by the macro preprocessor when it is first read. An INCLUDE within a macro definition can be processed at assembly-time or at macro-expansion-time, depending on whether the '$' starts in column 1 in the definition. If an INCLUDE does have a '$' in column 1 in the definition, then it is expanded at definition time. Otherwise, INCLUDE is not processed at macro-expansion-time. Example:

```
%*DEFINE(MAC1) ($INCLUDE(filename)) ;assembly-time
%*DEFINE(MAC2) ( $INCLUDE(filename) ;macro-definition time
%*DEFINE(MAC3(PARM1)) ($INCLUDE(%PARM1)) ;assembly-time
%*DEFINE(MAC4(PARM1)) ( $INCLUDE(%PARM1) ;macro-definition time.
```

Since %PARM1 is an improper filename, this causes an error.

However, expansions of MAC4 will be the expected:

```
$INCLUDE(value-of-%parm1-at-expansion-time)
```

This is the same as MAC3, but MAC3 does not produce an error message.

Any $INCLUDE processed at macro-expansion-time causes the remainder of its source line to be lost. If an error is detected while processing an INCLUDE, the error message is placed in the output file as usual and the line containing the INCLUDE is handled as ordinary text. If INCLUDE is
Macro Expressions

Macro expressions appear in some of the pre-defined instructions and are particularly important to the %SET macro function.

Operators

Expressions consist of one or more operands, and zero or more operators. The recognized operator keywords and their relative precedence are in the following table: (Operators that appear on the same line in the table have the same relative precedence.)

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher</td>
<td>HIGH, LOW</td>
</tr>
<tr>
<td></td>
<td>* , /, MOD, SHR, SHL</td>
</tr>
<tr>
<td></td>
<td>↑</td>
</tr>
<tr>
<td></td>
<td>Unary and Binary + , -</td>
</tr>
<tr>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td>EQ, NE, LT, LE, GT, GE</td>
</tr>
<tr>
<td>Lower</td>
<td>NOT</td>
</tr>
<tr>
<td></td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>OR , XOR</td>
</tr>
</tbody>
</table>

Parentheses can be used to override the default precedence of these operators and are recommended for complex expressions.

See Also

Chapter 7, beginning on page 137, for definitions of the operators.
Numbers

Numbers are stored in 17-bit form with a range of -65535 to + 65535. Note that the sign bit is stored, therefore -1 is not the same as + 65535 for purposes of macro-time operations (although they can be the same to the assembler). Integer constants in bases other than decimal are defined by placing a coded descriptor after the integer. The descriptors are as follows:

- B - binary
- O - octal
- Q - octal
- D - decimal (default)
- H - hexadecimal

Symbols

Symbols must begin with a letter or one of two special characters: the question mark ('?'), or the underscore ('_').

Like assembler symbols, the second and following characters can be any letter, digit, question mark, or underscore. Only the first 31 characters of a symbol are used by the macro processor to define that symbol; any additional characters are only for documentation purposes.

By default, the macro preprocessor is case sensitive. That means that upper and lower case letters are not equivalent in macro symbols. "ASYMBOL," according to the default, is not equivalent to "asymbol" or "ASYmBOL." Case sensitivity, however, can be turned off on the command line.

A macro symbol must be preceded by the metacharacter ('%') or the macro preprocessor will treat it as ordinary text. The exception is a string argument to a specific macro function.

The macro preprocessor does not recognize forward references because it makes only one pass through the source. Any symbol must be defined before it is used. Keywords are stored separately from symbols. Symbol names can therefore duplicate operator keyword names without conflict.

Macro symbols always have a string as a value. If the string happens to represent a valid numeric constant (such as '01Q' or '2'), the symbol can be
used as the operand of an expression. Only macro-time symbols and 17-bit integer constants are valid macro expression operands. The macro preprocessor does not deal with relocatable numbers of any sort.

**See Also**

Page 56 for the set of characters supported by the assembler and macro preprocessor.

**Balanced Text String (baltex)**

A frequently-referenced concept is the balanced-text string (‘baltex’), which is a string of characters containing balanced parentheses. Formally, baltex either contains no parentheses, or one or more sets of balanced parenthesis, as in

‘baltex(baltex)baltex’

where each baltex is a balanced-text string (possibly null).
Chapter 13: Pre-Defined Macro Functions

Pre-Defined Macro Functions

A description of the pre-defined macro functions found in ap86.
Pre-defined macro functions are provided as building blocks so that you may create user-defined macros. It would be nearly impossible to duplicate many useful operations found in the pre-defined functions with equivalent user-defined macros.

**Note**
A user-defined macro may be re-defined in the source program at some point after the original user definition. Redefinition does not cause errors; it does cause the preceding macro definition to be lost. Pre-defined macro functions, however, may not be re-defined. It is an error to try to do so.

### Pre-Defined Macro Functions

The pre-defined macro functions listed below are recognized by the macro preprocessor.

**Table 13-1. ap86 Pre-Defined Macro Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%'</td>
<td>(comment function)</td>
</tr>
<tr>
<td>%n</td>
<td>(escape function)</td>
</tr>
<tr>
<td>%EQS</td>
<td>%DEFINE</td>
</tr>
<tr>
<td>%GTS</td>
<td>%GES</td>
</tr>
<tr>
<td>%LTS</td>
<td>%LES</td>
</tr>
<tr>
<td>%NES</td>
<td>%LES</td>
</tr>
<tr>
<td>%EVAL</td>
<td>%EXIT</td>
</tr>
<tr>
<td>%IF</td>
<td>%LEN</td>
</tr>
<tr>
<td>%MATCH</td>
<td>%METACHAR</td>
</tr>
<tr>
<td>%REPEAT</td>
<td>%SET</td>
</tr>
<tr>
<td>%SUBSTR</td>
<td>%WHILE</td>
</tr>
</tbody>
</table>

**Note**
The pre-defined macro functions %IN, %OUT, %CI and %CO are not supported by the ap86 macro preprocessor. These functions accept user input to macro functions. The pre-defined macro function %DEFINE does not appear in this chapter because it is discussed in detail in the "User-Defined Macros" chapter.
%’ (Comment Function)

Call Pattern:

%’ ...any text... ’ or end-of-line

Description: The comment function permits insertion of comments without being passed on to the assembler. Everything from the quote up to a matching closing quote or to an end-of-line is considered a comment. Metacharacters within the comment string are not expanded. In the output, the call pattern (including the closing end-of-line, if used) is replaced with the null string.

Example:

MOV AX,%ARG1 % ARG1 is the loop counter
MOV SI,0 % Initialize index register
JMP $-2 % Initialize: %SET(symbol,0H)
%SET(symbol,02H) % Initialize: %SET(symbol,03H)
DB %symbol

The second line in this example will result in an assembly-time error because the end-of-line terminating the comment is removed along with the comment, so the assembler sees the two instructions

MOV SI,0 JMP $-2

without an end-of-line between them. The fourth line shows that metacharacters inside a comment are not expanded; the last line expands to 'DB 02H' because the ' % SET' was not executed within the comment. The literal character (‘*’) cannot be used with the comment function.

%n and % (Escape and Bracket Functions)

Call Pattern:

escape function: %n[n-characters]
bracket function: %(baltex)

Escape Function

Description: n is a decimal (base-10) digit from 0 to 9 inclusive. The expanded value of the escape function pattern is the n-characters immediately following n itself. These will be passed to the assembler without being
Chapter 13: Pre-Defined Macro Functions
Pre-Defined Macro Functions

examined by the macro preprocessor. For example, '%1%' passes a '%' to the output. The pattern '%0' passes no characters.

Bracket Function

**Description:** The expanded value of the bracket function is the "baltex" that appears between the parenthesis. The bracket function inhibits the expansion of all macros and functions within its argument except the escape function, the comment function, and macro parameters. These are always expanded.

Escape and Bracket Functions (Generally)

**Description:** It is sometimes necessary to hide certain text from the macro preprocessor, such as when a percent sign (%) is desired in the output or when using strings involving unbalanced parentheses or commas as text. The escape and bracket functions serve this need.

The bracket function might be more flexible than the escape function, but it deals only with baltex, and the metacharacter is interpreted (although once a call pattern has been detected it cannot be expanded). Examples:

```
%(1,2,3)      ;1,2,3 is passed to the output (this might be used as the actual parameter to a macro to prevent the commas from being interpreted as delimiters)
%330%         ;30% is passed to the output;error – '%)' is not legal
%(30%)        ;%330% is evaluated, then used as an argument to %()
%(30%1%)      ;%30%1% is passed to the output
%(30)         ;%30% is passed to the output
DB '30%1%'    ;DB '30%' is passed to the output because quotes are ignored by preprocessor
```

The literal character ('*') is not accepted with the bracket or escape functions.

If the output of the macro is to include the escape character, you must double-escape the output. For example, if macro MAC1 needs to output "DB ' % '", you could define MAC1 as follows:

```
%DEFINE (MAC1) ( DB ' %3%1%' )
```

%EQS, %NES, %LTS, %LES, %GTS, %GES

Call Pattern:

```
%xxS(baltex1,baltex2)
```
In the above call pattern, \texttt{xx} represents the first two characters of any of the function names.

**Description:** The string relational functions all compare two strings, character by character, left to right, and expand to a *logical-valued string*: \texttt{–1H} for TRUE, and \texttt{00H} for FALSE.

The first string cannot contain a comma unless the comma is protected by parentheses, the escape function, or the bracket function.

Comparison is on the basis of ASCII character values. A blank character has the value \texttt{20H}, tab has the value \texttt{09H}, and \texttt{< EOL>} has the value \texttt{0AH} (\texttt{< line feed>}). The comparison is *true* if the first argument has the relationship to the second indicated by the function. (\texttt{EQS} is true if the two strings are equal. \texttt{GTS} is true if the first string is "greater" than the second string.)

If two strings are of different lengths, but are identical on all characters in the shorter string, the longer string is considered to be greater.

The literal character *\texttt{*} is allowed, but it has no effect. Metacharacters in the argument strings are always expanded. Example:

\begin{verbatim}
%EQS(0,00H) ;yields 00H (false), since comparison is of strings, not numeric values
%GTS(2,100H) ;yields -1H same reason as above
%GTS(c,CBA) ;yields -1H (true), since c>C (ASCII values), which ends comparison
\end{verbatim}

**%EVAL**

**Call Pattern:**

\texttt{%EVAL(expression)}

**Description:** \texttt{EVAL} is used to evaluate an expression and it expands to a string representing the numeric value of the expression. The expanded string represents the value in hexadecimal. The first character of the expanded string is always a digit 0-9, the last character is always 'H', and the characters between are the hexadecimal digits 0-F. The expression is evaluated using 17-bit arithmetic, as always, but the expanded value is at most 16-bits. Negative numbers are shown in two’s complement form. The expanded string can be 3, 4, 5 or 6 characters in length. Examples:

\begin{verbatim}
%EVAL(3+3) ;yields 06H
%EVAL(3-3) ;yields 00H
%EVAL(-2) ;yields OFFFEH
\end{verbatim}
Chapter 13: Pre-Defined Macro Functions

Pre-Defined Macro Functions

%SET(S1,44) ;null (decimal value)
%SET(S2,333Q) ;null (octal value)
%EVAL(%S1+%S2) ;yields 0107H

The call pattern %*EVAL is legal, but the literal character ('*') has no effect; metacharacters in the expression are always expanded.

%EXIT

Call Pattern:

%EXIT

Description: The EXIT function allows immediate exit from the most recently invoked %REPEAT, %WHILE, or a user-defined macro. The call pattern %EXIT has no argument; it ends with the character 'T'. Some common uses are to prevent a WHILE loop from going on forever and to allow multiple exit points from a user macro.

This macro illustrates the classic example of recursion, the factorial function:

%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI %EVAL((%X)*%FACTORIAL(%X-1)) )

The same result could also be accomplished by using %ELSE instead of %EXIT. In this simple case using an %ELSE might even be clearer, but in more complex examples the %IFs might be nested several levels deep, so %EXIT would be much easier.

The call pattern %*EXIT is legal, but the literal character ('*') has no effect.

%IF (Conditional Assembly Function)

Call Pattern:

%IF(expression) THEN (baltex1) [ELSE (baltex2)] FI

Description: The IF function enables a user to decide at macro-time whether to assemble certain code or not. Doing this at macro-time has the advantage that the assembler (which may require more execution time than the macro preprocessor) sees only that code that is to be assembled.
The expanded value of %IF is the expanded value of either baltex1 or baltex2 (if present), but not both. The call pattern %IF first evaluates the numeric expression. If the low bit of the 17-bit value is 1, then the expression is considered true. Baltex1 is passed to the output as the expanded value of %IF. If the low bit of the 17-bit value is 0, then the expression is considered false and baltex2 becomes the expanded value of %IF (if baltex2 is present). If it is not present, the expanded value of %IF is null.

Typically, the expression will contain comparison operators (EQ, and so forth) or string comparison macro functions (%EQS, and so forth). These always return –1 for true and 0 for false, so %IF does what you would expect. However, any numeric value is acceptable.

The baltex that is not selected is also not expanded. Any %SETs in it, for instance, will not be executed.

The keywords THEN, ELSE, and FI are not stored as symbols, and user symbols can duplicate these names. Since the arguments are all baltex with parentheses as delimiters, there is no problem with ambiguity.

Call patterns (%IFs) can be nested; each FI (and ELSE, if present) is considered to go with the most recently defined IF. Example:

```plaintext
%*DEFINE(MAC(symbol))  (
  %IF (%symbol LT 0)
  THEN (  %'goes with LT if'
          DB 00H
  )
  ELSE (  %'goes with LT if'
           %IF (%symbol GT 10)
           THEN (
                   %set(newsymbol,%symbol-10)
                   DB %newsymbol
           )
           ELSE (  %'goes with GT if'
                   %symbol
                   DB %symbol
           )
           FI   %'goes with GT if'
  )
  FI   %'goes with LT if'
)
```

The literal character (‘*’) is legal with %IF and has the effect of suppressing metacharacter expansion in whichever baltex is selected to become the output. Metacharacters in the expression are always expanded.

%LEN

Call Pattern:

%LEN(baltex)
**Chapter 13: Pre-Defined Macro Functions**

**Pre-Defined Macro Functions**

**Description:** The LEN function counts the characters in its argument and expands to a string representing the numeric value of the expression. The expanded string represents the value in hexadecimal. The first character of the expanded string is always a digit 0-9, the last character is always ‘H’, and the characters between are the hexadecimal digits 0-F. The expression is evaluated using 17-bit arithmetic, as always, but the expanded value is at most 16-bits. Negative numbers are shown in two's complement form. The expanded string can be 3, 4, 5 or 6 characters. The literal character (‘*’) is legal and prevents the expansion of metacharacters in the baltex string. Example:

```
%LEN(countme) ;yields 07H
%LEN(%EQS(ABC,abc)) ;depends on case sensitivity
%*LEN(%EQS(ABC,abc)) ;counts '%EQS(ABC,abc)'
and yields 0DH
%LEN() ;yields 00H
```

An < EOL> counts as one character (the line feed character). %LEN of a SET-symbol will produce a number between 3 and 7 inclusive. It is the number of characters of the internal string representation of the symbol value.

---

**Note**

The value is a full 17-bits, with a minus sign if needed (signed magnitude representation). Thus –2 is stored as ‘-02H’ and 65534 is stored as ‘0FFFEH’. This is the only time (within a %LEN) that the value of a SET-symbol is not really stored as a number.

---

**%MATCH**

**Call Pattern:**

```
%MATCH(name1 delimiter name2) (string)
```

**Note**

The spaces surrounding the delimiter in the syntax above are not a part of the call pattern; they are shown only for clarity. Spaces between the first and second pair of parentheses are acceptable. Spaces, tabs, or end-of-lines are skipped over if they appear there.

---

**Description:** Name1 and name2 are symbols (not necessarily previously defined) and delimiter is a single character separating them. It can be any character that is not valid in symbols. It could be a space, tab, comma, end-of-line, parenthesis, or others.
MATCH divides a string into two parts at the first occurrence of the delimiter, and assigns each part to a symbol. Its expansion is the null string. MATCH is most commonly used in connection with loops, as described below.

MATCH searches the (expanded) string for the first occurrence of the delimiter. When it is found, all characters in the string preceding the delimiter are assigned as the value of name1. All characters following the delimiter are assigned as the value of name2. Either value can be null. If the delimiter is not present in string, the entire string is assigned to name1 and name2 receives the null string as its value. Examples:

```
%MATCH(NAME1,NAME2) (A,B,C)       ;NAME1='A', NAME2='B,C'
%MATCH(NAME1 NAME2) (A,B,C)       ;NAME1='A,B,C', NAME2=null
%MATCH(NAME1 , NAME2) (A,B,C)     ;Error - illegal spaces
        ;around comma (delimiter in this example)
```

The literal character (`*`) is legal in conjunction with %MATCH and inhibits the expansion of any metacharacters in "string." Example:

```
%SET(sym,2)
%MATCH(VAR1,VAR2) (%sym,02H)  ;VAR1=02H, VAR2=02H
%*MATCH(VAR3,VAR4) (%SYM,02H)  ;VAR3=%SYM, VAR4=02H
%SET(SYM,3)
DB %VAR1                      ;yields DB 02H in the output
DB %VAR3                      ;yields DB 03H and %SYM is
        ;expanded at reference time
DB %*VAR3                     ;yields DB %SYM and causes an
        ;assembly-time error
```

The last example is case dependent and would not work if case sensitivity was enabled.

The MATCH function is often used to extract similar fields out of a string one at a time. Suppose a string consists of several numbers separated by spaces. Such a string might be the expected value of a formal argument, for instance. To generate a DB for each number:

```
%MATCH(TEMPVAR^JUNK) (%FORMALARG)
%WHILE( %LEN(%TEMPVAR) GT 0 )
(%MATCH(NEXTNUM TEMPVAR) (%TEMPVAR)
DB %NEXTNUM)
```

The first MATCH copies the formal argument to TEMPVAR, presuming there are no caret (^) in %FORMALARG (this is a trick to evade the fact that SET can assign only numeric values to a symbol; it cannot assign a string). The condition of the WHILE loop states that TEMPVAR must still be non-null. The MATCH inside the loop extracts the next number from TEMPVAR and stores the rest of the string back in TEMPVAR. The DB is then generated and we execute the WHILE test again.
Call Pattern:

%METACHAR (baltex)

Description: The METACHAR function changes the metacharacter (% by default) to a different, user-specified character. These are the acceptable alternative metacharacters:

\[ \emptyset / + - \# . \_ = [ ] > < ' " \& , = % \{ \} ~ \| \^ \]

The following characters cannot be used as a metacharacter:

- the letters (A-Z, a-z)
- the digits (0-9)
- the characters: _ ? * ( ) blank tab <EOL>

The new metacharacter is taken to be the first character of the expanded value of baltex, although baltex can be any number of characters long. The new metacharacter takes effect immediately at the first character following the right parenthesis delimiting the call pattern of METACHAR. The literal character ‘*’ is accepted on METACHAR, but it has no effect, as the argument of METACHAR is always expanded.

Changing the metacharacter can have unforeseen catastrophic effects. For example, any previously defined macros probably have the default metacharacter (‘%’) in the stored macro body. They will not expand correctly if the metacharacter changes. The expanded value of the METACHAR function is the null string.

Call Pattern:

%REPEAT (expression) (baltex)

Description: The REPEAT function is one way to program a loop. REPEAT evaluates the 17-bit numeric expression and then baltex is expanded that many times. Note that the expression is expanded only once. If baltex alters macro symbols that are involved in the expression, it does not affect loop control. If the expression evaluates to be less than or equal to zero, baltex
is expanded zero times (the expanded value of REPEAT is the null string).
Example:

\%REPEAT(5) (SHL AX, 1)

**Note**

The \texttt{< EOL>} within \texttt{baltex} is necessary for correct expansion. Without the \texttt{< EOL>} ; this REPEAT would produce

\texttt{SHL AX,1SHL AX,1SHL AX,1SHL AX,1SHL AX,1SHL AX,1}

\%*REPEAT is acceptable. The asterisk inhibits the expansion of metacharacters within baltex. Metacharacters in \texttt{`expression`} are always expanded.

**%SET**

**Call Pattern:**

\%SET(name, expression)

**Description:** SET defines the string "name" as a symbol, whether or not it was already defined, and gives it the value of "expression." Expression must result in a number, but the value of name is stored as a string (like all macro symbols). Generally, you can ignore this fact and treat name as if it were stored as a number. Multiple SET directives can reference the same name. The expanded value of the \%SET call pattern is the null string.

The literal character ("*") makes no sense with SET, since its first argument must be a symbol and its second argument must evaluate to a number. Neither argument can contain metacharacters after expansion. If the macro preprocessor attempts to expand \%*SET, it will report an error.

It is correct for the symbol-referencing construct to appear inside another SET for the same symbol. Example:

\%SET(username, %username+1)

This increments the value of `username` by one. However, the next example is incorrect:
Chapter 13: Pre-Defined Macro Functions
Pre-Defined Macro Functions

%SET(username, username+1)

This example generates a macro-time error because the character string "username" is not a legal expression operand. Symbol-referencing is discussed in the chapter titled "User-Defined Macros."

%SUBSTR

Call Pattern:

%SUBSTR(baltex, exp1, exp2)

**Description:** The SUBSTR function extracts a substring from its first argument based on its second and third arguments.

In this pattern, exp1 and exp2 are numeric expressions. The expanded value of the pattern is a substring of baltex. The substring begins at character number exp1 and contains exp2 characters. If exp1 is less than or equal to 0, or greater than the number of characters in baltex, then the expanded value is null. If exp2 is less than or equal to 0, then the expanded value is null. If exp1 is of such a size that the expansion value will not be null, but exp2 implies more characters than remain in baltex, then the expanded value is all characters from character exp1 to the end of baltex, inclusive. Examples:

```
%SUBSTR(12345678, 4, 2) ; yields 45
%SUBSTR(12345678, -1, 2) ; yields null
%SUBSTR(12345678, 10, 2) ; yields null
%SUBSTR(12345678, 2, -1) ; yields null
%SUBSTR(12345678, 2, 1000) ; yields 2345678
```

The literal character ("*") is accepted with SUBSTR, but is ignored. Metacharacters in any of the arguments are always expanded.

%WHILE

Call Pattern:

%WHILE (expression) (baltex)

**Description:** The WHILE function programs macro-time loops. It works similarly to the WHILE construct in high level languages.

WHILE evaluates the 17-bit numeric expression each time through the loop. If the least significant bit of the expression is 0, the expanded value of WHILE
is the null string. If the least significant bit of the expression is 1, then baltex is expanded and passed on as part of the expanded value of WHILE, and the expression is evaluated again. The loop continues until the expression evaluates to false (least significant bit is 0).

For the loop to terminate, baltex must modify the value of expression or an EXIT function must be used. Otherwise the loop will never exit. WHILE is often used in conjunction with either SET or MATCH, either of which will update a macro symbol on each pass through the loop (see the example under MATCH).

The call pattern %*WHILE is not accepted, since preventing the expansion of baltex would result in an infinite loop. An error will be reported if %*WHILE is found.

---

Example Problem

This example shows the effects of an incorrect factorial macro.

```
%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI
%EVAL(%X*%FACTORIAL(%X-1))
)
```

The only difference between this example and the one shown with the %EXIT function reference is that this one is missing the pair of parentheses around the second %X. They are necessary, because the arguments of macros are strings, not numbers. The incorrect version above called with the actual parameter 4 expands successively to the following:

```
4*FACTORIAL(4-1)
  4-1*FACTORIAL(4-1-1)
   4-1-1*FACTORIAL(4-1-1-1)
01H
   4-1-1*01H
02H
  4-1*02H
02H
```

---

Chapter 13: Pre-Defined Macro Functions

Example Problem
Chapter 13: Pre-Defined Macro Functions
Example Problem

4*02H
08H

The %FACTORIAL in the next lower calling level is evaluated before the %EVAL in the one that called it is executed. That is as it should be and the recursive property of this function is retained. The problem is that the normal rules of precedence govern within the enclosing parentheses of %EVAL. This means that the multiplication is done to just part of the intended value of %X, instead of the full value, at any level. The result is therefore less than it should be.

As a general guide, it is advisable to surround any macro-time symbol with either parentheses or %EVAL() if you expect to produce a numeric value. For this example, one fix is to put %EVAL() around %X-1 in the call to %FACTORIAL. This forces evaluation of the subtraction before the value is passed to the next lower calling level. Another fix is to put parentheses around the second %X—as has been discussed and was done in the example for %EXIT. This causes parentheses to be around the subtractions preceding the multiplication sign that then force the intended order of arithmetic evaluation. The corrected macro definition, using the %EVAL() fix, follows:

%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI
%EVAL(%X*%FACTORIAL(%EVAL(%X-1)))
)

The corrected macro definition called with the same parameter of 4 would expand as follows:

4*FACTORIAL(3)
   3*FACTORIAL(2)
      2*FACTORIAL(1)
         01H
         2*01H
         02H
      3*02H
         06H
   4*06H
      018H
User-Defined Macros

Information about defining macros, including the syntax for defining them, and how macros are referenced.
User-defined macros are created by using the `%DEFINE` macro function.

User-defined macros can be defined in terms of themselves which means they can invoke themselves within their own macro bodies. This ability is called recursion. Any macro that calls itself must include a terminating condition that causes the macro to "bottom out" eventually or the preprocessor can enter into an infinite loop.
%DEFINE

If you want to define a macro, you must use the DEFINE function.

Because the syntax for DEFINE is somewhat complicated, the following figure contains the syntax diagram for DEFINE.

Figure 14-1. Syntax for User-Defined Macros

Where:

% is the current metacharacter (which is usually %).

* is the optional literal character. This character should be used with most definitions. There are two reasons:

• It will inhibit the expansion of macro calls flagged by the current metacharacter (usually %) within the macro body at the time of macro definition. Instead, macro calls will be expanded at the time of macro reference.

• You must use the literal character with any macro that has formal parameters. Otherwise, the macro preprocessor will attempt to evaluate any references to the formal arguments within the macro body as symbols or other macro calls, which will result in errors.

Define is the pre-defined macro function for creating user-defined macros.
Name is the user-defined name to be associated with the macro. It cannot conflict with the predefined macro functions, but it can duplicate an earlier user-defined macro name or symbol. In the latter case, the previous meaning of the symbol is lost. The macro name should not be preceded by the current metacharacter (usually %).

Parameter is a formal parameter name. Formal parameters, if they exist, are replaced by actual parameters when the macro is invoked.

Note

Formal parameter names are not preceded by the metacharacter when they are being declared in the macroname argument list. To reference a formal parameter within the macro body, however, you must precede its name with the metacharacter (as in %ARGUMENT_NAME for the formal parameter ARGUMENT_NAME).

Parameter names must be distinct from one another within a macro, but they can duplicate other formal parameter names in other macros, since they have no existence outside the macro definition. They can also duplicate the names of other user macros or macro functions. If they do duplicate other macro function names, then the other macros or functions cannot be used within the macro body, since the duplicated name will refer instead to the parameter.

Local is the word that must precede the local parameter list.

Symbol is a local symbol name. Such symbols can be used only within the macro body. They are undefined outside of it.

The purpose of local symbols is to avoid multiply-defined symbols in the output of the macro processor. Each time the macro is referenced, each local symbol receives a unique two to five digit suffix. For example, if a local symbol LABEL were defined for use within a macro, then the first macro invocation might substitute LABEL00 and the second invocation might use LABEL01. This way, the assembler would not see a multiply-defined symbol. When locals are initially being declared following the LOCAL keyword, they must not be preceded by the metacharacter. However, when referencing local symbols in the macro body, they must be preceded by the metacharacter. The symbol LOCAL is not reserved; a user symbol or macro can have this name.

Body is a balanced-text string. It can contain references to formal arguments and local symbols, if any, as described above. It can also include references to
user-defined macros (including itself), to macro-expansion-time symbols (preceded by ‘%’), and to macro functions.

A macro should not redefine itself (%*DEFINE) within its body, however. The expanded value of DEFINE is the null string, but the macro body is stored internally for later use. A re-DEFINE in a macro body, then, is working at cross purposes.

Macro Reference

A macro is referenced by preceding its name with the metacharacter. If the macro was defined with formal arguments, the reference must include the same number of actual parameters, enclosed in parentheses and separated by commas. Actual parameters can be null, but the required delimiters must still be present between them. Each actual parameter is substituted for its corresponding formal parameter, wherever it appears in the macro body, on a string basis.

The literal character (‘*’) is acceptable in conjunction with references to user-defined macros. Normally, all metacharacters in the actual parameters are evaluated immediately when the macro reference is found and the resulting strings are stored. They are then substituted for the formal parameters as the macro body is copied. The literal character defers evaluation of actual parameters until they are found in the macro body, and they are re-evaluated each time they are found. It is possible, then, that the values of actual parameters might change between evaluations depending on what the macro body does.

Following are some sample macro definitions and references along with short discussions about each. Each new macro and discussion begins with the new %DEFINE, but an implied order of definition from first to last is understood in order that some of the discussions make sense. Some of the macros are intentionally incorrect.

%*DEFINE (MAC1) ( DB 2 )

MAC1 will have the string value "DB 2" when invoked.

%*DEFINE (MAC2(ARG1)) (DB %ARG1)
MAC2 is stored as "DB %ARG1". %ARG1 is to be evaluated at the time of macro reference because of literal character ("*") precedes DEFINE.

`%*DEFINE (ERR1 (ARG1)) (DB ARG1)`

ERR1 shows a common error. The '%' is omitted from the formal parameter in the macro body which means it will not be recognized. The assembler will be passed "DB ARG1" when the macro is invoked, which is not likely to be correct.

`%*DEFINE (MAC3 (ARG1)) (%MAC1 %MAC2 (%ARG1))`

MAC3 references the previously-defined macros MAC1 and MAC2. Since the evaluation of metacharacters in MAC3 is deferred (with *), this example would also work if the definitions of MAC1 or MAC2 followed that of MAC3 (as long as they are defined before MAC3 is invoked).

`%*DEFINE (ERR2 (ARG1)) (%MAC1 %MAC2 (%ARG1))`

ERR2 shows another common error—the literal character was omitted. The metacharacters in the macro body are expanded immediately (at macro-definition time). Since there is a reference to a formal parameter, this cannot be done—there is no actual parameter to substitute for it. The macro preprocessor actually attempts to expand %ARG1 as a macro symbol or user-macro. In some cases this might be possible, although it is not likely to be what is expected.

`%*DEFINE (ERR3 (ARG1)) (%MAC1 %MAC2 (%ARG1))`

ERR3 shows another frequent user-error, a missing `<EOL>`. Since the body of neither MAC1 nor MAC2 includes an `<EOL>`, ERR3 should include one between their invocations (as MAC3 does). The invocation `%ERR3(3)` will yield "DB 2 DB 3" and cause an assembler error. If MAC1 ended with an `<EOL>` or MAC2 began with an `<EOL>`, ERR3 would be correct.

`%DEFINE (MAC4) (%MAC1 %MAC2 (4))`
MAC4 shows an acceptable use of DEFINE without '*'. The stored body of MAC4 is shown in the following example, since the calls to MAC1 and MAC2 are evaluated immediately:

```
' DB 2
DB 4'
```

With the definitions of MAC1 and MAC2 shown above, %MAC4 is the same as %MAC3(4). But MAC1 and/or MAC2 might be redefined later on. In this case, MAC3 will reference the new values, while MAC4 will not.

```
%*DEFINE(MAC5(ARG1)) LOCAL LABEL (
%LABEL:        MOV AX,%ARG1[DI]
            INC DI
            LOOPNZ %LABEL)
```

MAC5 shows the use of a local symbol. Each invocation of MAC5 will create a unique assembler-time symbol from LABEL.

Note that the macro definitions above produce no output, since each DEFINE expands to the null string. Consider the macros as being defined sequentially without separating blank lines. The end-of-lines between the terminating right parenthesis of each macro body and the following metacharacter ('%') of the next macro result in blank lines that are not output. If the macro preprocessor did not remove blank lines, these examples would generate seven blank lines. This behavior is typical of readable macro code. All characters between the delimiting parentheses (including end-of-lines) are considered part of the macro body, which in turn is part of the syntax of DEFINE. Such characters are not considered for output.

---

**Referencing Macro-time Symbols**

Symbols are defined by the SET and MATCH functions. A symbol is referenced by preceding its name with the metacharacter, as in

```
%name
```

Without the metacharacter, the macro preprocessor treats "name" like any other character string. The call pattern of the symbol ends where the name
ends (there is no argument in parentheses). The expanded value of this
construct is the character string that had been assigned to it. (For instance:
'01H'; or 'STRINGVALUE'; or the null string.)

The literal character ('*') is proper with a macro-time symbol. It inhibits the
expansion of any metacharacters within the symbol value which otherwise
would be expanded. For example, suppose the value of a symbol SYM is
"%LEN(01H)." %SYM will expand to '03H', but %SYM will expand to
"%LEN(01H)". Generally, the literal character should be omitted.

The literal character similarly affects formal parameters of macros within the
macro body. A formal parameter is not recognized if preceded by the literal
character. This permits giving a formal parameter the same name as a macro
function while still being able to access the function within the macro body.
Example:

```assembly
%*DEFINE (MAC(LEN)) (DB %LEN
                                   DB %*LEN(%LEN))
```

The first %LEN is the formal argument LEN, as is the third. The second is
not recognized as an argument because of the literal character, so it reverts to
its normal meaning as a pre-defined function. The literal character has
meaning to this particular function, so the inner %LEN is not expanded.

The literal character cannot be used with local symbols within a macro body.
Assembler versions

Information about how this version of the software differs from previous versions.
Version 3.10

New Product Numbers

The old product number for this product was 64871. The new product number is B1449. The product numbers for some associated products, such as the C cross compiler, have also changed from 64xxx to B14xx.

New Assembler Controls

The GEN, GENONLY, OPTIMIZE, EXTERN_CHECK, and UNREFERENCED_EXTERNALS controls are new.

New Linker/Loader Controls

The ERROR, WARN, LISTMAP, and TYPEMERGE controls are new.

New Assembler Defaults

The defaults for the ERRORPRINT, SYMBOLS, and PRINT controls have changed.

New Location for Man Pages

The on-line manual pages have been moved to the $HP64000/man directory. Set your $MANPATH environment variable to include this directory.

New Linker Listing Format

The cross-reference table has been combined with the public and local symbol tables.
Version 3.00

Demo Directory Change

The directory that contains the example files, /usr/hp64000/demo/languages/hp64871, has changed to /usr/hp64000/demo/languages/as86. (This change does not apply to the HP 9000 Series 800 version.)

New Assembler Controls

The OPTIMIZE assembler control has been added. Use OPTIMIZE to reduce the number of NOPs generated for forward references.

The [NO]UNREFERENCED_EXTERNALS assembler control has been added. Unreferenced external symbols are removed by default. Use this control to cause all external symbols, including those that are unreferenced, to be placed into the generated object file.

The SYMBOLS assembler control now defaults to SYMBOLS instead of NOSYMBOLS. Thus assembler listings will now normally contain a symbol table.

New Assembler Operators

Four new operators have been added to the assembler. These operators are SEGSIZE, SEGOFFSET, GRPSIZE, and GRPOFFSET. These operators allow a program to access the size of segments or groups, the offset of the start of a segment from a paragraph address, and the offset of a segment from a group address.

New Linker Commands

The linker now supports two new commands: TYPEMERGE and NOTYPEMERGE. These commands are used to control the amount and number of type records within the generated output file. Problems can occur if the number of type records in input files exceeds 32k. If this occurs, an erroneous output file will result. The TYPEMERGE command allows the linker to merge like type records so the number of type records required in the
output file can be kept below this limit. Use of this command, however, will cause the linker to run for a longer amount of time. This is necessary, however, for very large programs.

Other Linker Changes

Modules in Incrementally-linked files will now be reported differently in the listing. Any module within the file will be reported with two dashes (--) before the module name. This is to indicate that the module is part of a larger overall module. Also, incremental linking now works correctly when used to create a final HP64000 absolute file.

One of the duplicate error 812 messages has been changed to become error 825.
Chapter 16: Converting HP 64853 Assembly Language Programs

Converting HP 64853 Assembly Language Programs

Changes that must be made to source files written for the HP 64853 assembler so that they can be assembled with the HP B1449 assembler.
Not everything that appears in the HP 64853 format source files can be translated into something that the HP B1449 assembler will recognize, but most can. Translation is done in two ways. Some translations must be done manually. Most translations, however, can be done by the acvt86 translation program described in this chapter.

**Note**

The program acvt86 automatically performs most of the transformations described here. acvt86 is an unsupported porting tool. acvt86 is not a part of the B1449 product and is distributed at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

The first section of this chapter discusses the acvt86 porting tool and issues that are caused by the differences between the two assemblers. The next section describes the manual translations to macros that must be done because the porting tool cannot perform some macro translations. The third section gives the command syntax for acvt86. The final section is an old and new list. This section is arranged alphabetically according to keywords in HP 64853 assembly language. It gives a side-by-side comparison between the old and new syntax and shows you how acvt86 transforms particular HP 64853 constructs.

**acvt86 Introduction**

This section describes the way that acvt86 approaches the conversion process, what it produces, and its limitations. It also describes the sequence you should follow to translate files that contain include files. This section will give you a better understanding of what you will have to do to complete the translation process.

This section is not a complete description. If you need to find out how acvt86 converts a particular construct, you should write a test program and examine the acvt86 output.
The first line of an HP 64853 program identifies the target processor. HP B1449 assembly language supports only the 8086/186 and 8088/188 processors. The HP B1449 assembler does not support the 70108, 70116, 8089, or 80286 microprocessors. Therefore, the following target processor identifiers are not recognized: "70108", "70116", "8089_86", "8089_88", and "80286".

**Assembler Differences**

The HP B1449 assembler is really two programs: the preprocessor, ap86; and the assembler, as86. The preprocessor implements the following features of the old HP 64853 assembler: SET directives, REPT directives, MACRO definitions and expansions, and IF/ELSE/ENDIF conditional assembly directives. The assembler then completes the process by assembling the file produced by the preprocessor. acvt86 translates these features in the following way.

**IF**

```
IF <expr>
lines
ELSE
lines
ENDIF
```

translates to

```
%IF((<expr>)NE 0)
THEN
(lines)
ELSE
(lines)
FI
```

**EQU**

```
id EQU <expr>
```

translates to
id EQU <expr>

If < expr> is a constant expression, acvt86 generates %SET(id,< expr> ) and acvt86 also stores id in its symbol table. Later, when id is referenced in preprocessor expressions, acvt86 recognizes it and translates it to %id.

MACRO

id MACRO &P1,&P2
lines
MEND

translates to

%*DEFINE(id(P1,P2))
(lines)

acvt86 also stores id in its symbol table. Later, when id is referenced, acvt86 recognizes it and translates it to %id.

REPT

REPT <expr>

next line

translates to

%REPEAT(<expr>)

(next line)

SET

id SET <expr>

translates to

%SET(id, <expr>)

acvt86 stores id in its symbol table. Later, when id is referenced, acvt86 recognizes it and translates it to %id.
Sometimes the constant expressions in IFs or REPTs cannot be translated. HP 64853 calculates its constant expressions using 32 bit numbers. ap86 uses only 17 bit numbers. HP 64853 allows constant expressions to be formed by subtracting two relocatable symbols. ap86 cannot do this because it has no knowledge of the value of relocatable symbols.

External Declarations

HP 64853 allows an external identifier to be associated with a segment register in the EXT directive. For example:

```assembly
EXT ES:X1 WORD
MOV AX,X1 ;references X1 using ES
```

If you use X1 in certain memory reference operands, the HP 64853 assembler will automatically generate an ES: segment override for the instruction.

The HP B1449 assembler does not have an equivalent capability. Instead, an external identifier can be associated with a segment by placing the EXTRN directive inside a SEGMENT/ENDS pair. The segment may then be associated with a segment register through an ASSUME directive.

Since it would be difficult to automatically perform this kind of rearrangement, acvt86 instead does the following:

- When an external declaration with an associated segment register is found, acvt86 stores the identifier and segment register in its symbol table.
- When the external identifier is referenced, acvt86 will generate (when appropriate) an explicit segment override. For example, the instructions shown above would be translated to the following.

```assembly
EXTRN X1:WORD %'ES:D:X1'
MOV AX,ES:X1 ;references X1 using ES
```

The preprocessor comment %'ES:D:X1' records the information in the original EXT directive. If, in a subsequent translation, acvt86 sees this comment when reading a translated include file, it can update its symbol table just as if it saw the original declaration.
You should not do SET definitions, constant EQU definitions, or "EXT segreg:idi" declarations using MACRO parameters. For example:

```assembly
CSWORD MACRO &P1
EXT CS:&P1 WORD
MEND
CSWORD X1
```

While this arrangement works perfectly well in the HP 64853 assembler, acvt86 cannot tell that the variable X1 will have the implied CS: override quality. It may not translate references to X1 correctly.

**Porting Procedure— Main Files with INCLUDE Files**

Here is a procedure for translating a main file and its INCLUDE files. This sequence gives acvt86 its most complete symbol table and allows it to do the most accurate translation.

1. Translate the include files first. Use the -c option to specify the main file as the context file. This allows definitions in the main program to be used when translating the include file. Furthermore, as each include file is translated, its definitions are available for translating subsequent include files.

2. Make manual corrections to translated include files. Typically, this means rewriting .IF, .GOTO, etc., directives in MACROs.

3. Translate the main file(s). Make corrections to the main file(s).

4. Assemble the main file(s) using the HP B1449 preprocessor/assembler. Correct preprocessor and assembly errors.

For example, here are three files: prog.S, inc1, and inc2

**Main File prog.S**

```assembly
" 8086 "
;prog.S
  EXT ES:X1 WORD
  INCLUDE inc1
  INCLUDE inc2
```
M2 ;defined in inc2
END

Include File inc1
;incl
DISP SET 6

Include File inc2
;inc2
M2 MACRO
  MOV AX,X1 ;X1 defined in prog.S
  ADD AX,DISP ;DISP defined in incl
MEND

First, translate inc1 as follows.
$ acvt86 -c prog.S incl > inc1.h

Second, translate inc2. Because of the "-c prog.S" option and because we have already created inc1.h, acvt86 will correctly translate the references to X1 and DISP.
$ acvt86 -c prog.S inc2 > inc2.h

Finally, translate prog.S. Because inc2.h exists, acvt86 will correctly translate the reference to macro M2.
$ acvt86 prog.S > prog.s

acvt86 Warnings, ap86 Errors, as86 Errors

To do a successful port, you must pay attention to messages from three sources.

acvt86

acvt86 issues warnings when it detects something that may need your attention. For example, it issues a warning when a MACRO call has more actual parameters than it has formal parameters in the MACRO definition. As previously explained, the two assemblers operate differently in this
sitting. Depending on how your MACRO is written, you may or may not need to change this statement.

**ap86**

After translating your files, you must understand and correct preprocessor errors. For example, errors may result from using constant expressions whose value is too large for the 17 bit preprocessor expression limit.

**as86**

Finally, you must understand and correct as86 errors. Assembler errors have numerous causes. For example, HP 64853 allowed user labels to duplicate instruction mnemonics (e.g. TEST). HP B1449 does not allow this and produces a syntax error. In this case, you should change the name of the offending label.

### Code Substitution

acvt86 has a feature that allows HP B1449 code to coexist with HP64853 code in an untranslated assembly source file. This feature is useful when, instead of doing a one-time port, you want to maintain a single, untranslated source file and then use acvt86 as necessary to obtain translated source.

acvt86 treats the comment ";sub64871;" in a special way. When acvt86 sees that comment, it does the following:

- Discards all the text before the ;sub64871; comment. Any warnings generated by this text are also discarded. Note that acvt86's symbol table is still updated normally if the discarded text contains certain directives.
- Writes any text following the ;sub64871; comment to standard output without any changes.

In the example below, we want to substitute legal HP B1449 code for the .IF and .NOP directives that acvt86 does not translate.

```
.IF &P1.GE.0 LAB ;sub64871;%IF(%P1 LT 0) THEN (  
  DW &P1
  LAB .NOP ;sub64871;) FI
```
acvt86 will produce the following output for the preceding text

```
%IF(%P1 LT 0) THEN (
  DW %P1
) FI
```

\[ \text{Note} \]

acvt86 only recognizes the substitution string `;sub64871;` at the beginning of the comment field. In the example below, acvt86 will not make a substitution because comment text precedes the `;sub64871;` string.

```
DW &P1 ;indexed parameter;sub64871;  DW %P1
```

---

**Byte ordering for BIN, DECIMAL, HEX, OCT**

These four HP 64853 directives generate data with bytes that are reversed from the normal 8086 convention. When translating, you must adjust the value of operands to these directives to compensate for this. This applies to any numeric format: binary, decimal, hex, or octal. Example:

**HEX**

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEX ABCD</td>
<td>DW 0CDABH</td>
</tr>
</tbody>
</table>

---

**Manual Macro Translations**

acvt86 automatically translates simple MACRO definitions (i.e. those without .IF, .SET, .GOTO, or .NOP directives and without indexed "&PNO" parameters).

More complicated structures must be translated manually. Generally, this can always be done except when .IF or .SET expressions use symbol values which cannot be calculated at preprocessor time.
The HP 64853 .IF, .GOTO, and .NOP conditional assembly directives must be manually translated into the HP B1449 %IF preprocessor directives. If the branches in your MACRO do not define a block structure, you must rearrange the MACRO to conform to the IF/THEN/ELSE structure of ap86.

Macros branches which do loops can be translated into %REPEAT or %WHILE structures.

The HP 64853 .IF directive performs either numeric or string comparisons depending on the operands being compared. Numeric comparisons must translate into ap86 numeric expressions; string comparisons must use the ap86 preprocessor string comparison functions.

HP 64853 allowed a null parameter either to be the null string ("") or to be omitted entirely (except for a comma placeholder). Here is how to test for an omitted or null macro parameter. Check for both of these possibilities in your translated .IF directive.

HP 64853 allows MACRO parameters to be referenced by number. HP B1449 has no equal facility. Two translation techniques can be used.

1 Use %*DEFINE to make a new identifier which has the value of the indexed parameter.

2 Sometimes a MACRO indexes an indefinite number of parameters. This can be handled with the %MATCH function. For example, the following MACRO defines one word for each actual parameter. It stops on the first null parameter or at the end of the list.

**Macro Calls**

Sometimes, a MACRO call specifies a different number of actual parameters than formal parameters in the MACRO definition. acvt86 records the number of formal parameters in a MACRO definition. It automatically handles the first two of three situations described below. The third situation usually requires a manual change.

3 If you specify fewer actual parameters than there are formal parameters, ap86 will error and not expand the macro. To prevent this, acvt86 automatically generates additional null parameters on the macro call.

4 If you specify actual parameters and no formal parameters were declared, ap86 does not consume the actual parameter list and they eventually cause a syntax error. To prevent this, acvt86 suppresses the actual parameter list.
If you specify more actual parameters than formal parameters, \texttt{ap86} acts as follows: the value of the last formal parameter is equal to the value of its corresponding actual parameter concatenated with all the additional actual parameters and comma delimiters. Any reference to the last formal parameter will generate a different value than it did in the HP 64853 assembler. \texttt{acvt86} issues a warning in this case. You should either eliminate the extra actual parameters or rewrite the macro to preserve its original function.
acvt86(1) Command Syntax

**Note**

The program acvt86 automatically performs most of the transformations described here. acvt86 is an unsupported porting tool. acvt86 is not a part of the B1449 product and is distributed at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

**Name**

acvt86 - converts 8086 assembly programs from HP 64853 format to HP B1449 format

**Synopsis**

```
/usr/contrib/bin/acvt86 [-dsw][-a align]
 [-c context] [-h suffix][file]
```

**Description**

acvt86 translates assembly source programs from one dialect to another. It assumes the input file is a legal 8086, 8088, 80186, or 80188 assembly program for the HP 64853 assembler. The output may be assembled with the HP B1449 assembler.

acvt86 does not translate "70108", "70116", "8089_86", "8089_88", or "80286" programs that were accepted by the HP 64853 assembler. Programs for these microprocessors are also not accepted by the HP B1449 assembler.

acvt86 reads from standard input or the named file. It writes the translated assembly to standard output. It writes warnings about functional differences between the input and output to standard error.

acvt86 supports a one-time porting of assembly programs from one product to another. The objective is to obtain the same (or functionally equal) bits from the HP B1449 assembler as from the HP 64853 assembler. acvt86 changes directives, delimiters, operators, and so on to achieve this goal. However, because of differences between the two assemblers, this porting process cannot be entirely automatic or trivial.

acvt86 makes two passes over its input file. The first pass builds a symbol table of certain identifiers (MACROS, externals, etc.) that will effect the translation; the second pass performs the translation.
acvt86 may look at other files to supplement its symbol table. The `-c contextfile` option incorporates the definitions from `contextfile` in the present translation. Typically, a `contextfile` is a main, untranslated assembly module while the present file is an INCLUDE file of `contextfile`. Whenever acvt86 encounters an INCLUDE directive (either in `contextfile` or the present input), it attempts to open the already translated include file and read its definitions. (See the `-h suffix option for include file naming conventions.)

acvt86 has a code substitution feature. It allows HP B1449 code to coexist with HP 64853 code in the same untranslated file. Refer to the section "Code Substitution" for more information.

acvt86 was implemented with lex(1) and yacc(1). The source code is available in `/usr/contrib/src/acvt86/`.

**Options**

- `-c context` Scan the context file (and translated INCLUDE files mentioned in it) for definitions to use when translating file. This option is useful when translating INCLUDE files. Specifying a context allows acvt86 to accurately translate references to certain identifiers (MACROS, externals, etc.) that were defined in the main "context" file or its (translated) INCLUDE files.

- `-a align` Align is one of the HP B1449 align-types of BYTE, WORD, PARA, PAGE, INPAGE. Specify the align-type used in segment directives for relocatable segments. The default align-type of BYTE duplicates the alignment behavior of the HP 64853 assembler. However, the HP B1449 assembler errors when an EVEN directive occurs within a BYTE aligned segment. If EVEN directives will be used, use the `-a WORD` option.

- `-d` (differences) acvt86 writes pairs of input/output lines only when they are different. This output is not suitable for subsequent assembly.

- `-h suffix` Specifies the suffix (default .h) which is added to file names in INCLUDE directives to form the name of the "translated" include file. If the file name in the
INCLUDE directive has a suffix (i.e. contains a period) then suffix replaces the original suffix. Otherwise, suffix is appended to the original file name.

For example, suppose an HP 64853 program contained the following directive.

```
INCLUDE file.H
```

acvt86 would translate this to the following HP B1449 control.

```
$INCLUDE(file.h)
```

It would also assume that file.H had already been translated into file.h and attempt to read file.h before continuing with the present translation.

- `s` (silent) Suppress warnings to standard error.
- `w` (warn) Include warning messages (as comment lines) in the standard output following the appropriate translated line.
Chapter 16: Converting HP 64853 Assembly Language Programs

acvt86(1) Command Syntax

Files

/usr/contrib/bin/acvt86

Executable file for assembly language porting tool.

/usr/contrib/src/acvt86/*

Source code files and make file for assembly language porting tool.

See Also

ap86(1), as86(1), asm(1).

Diagnostics

acvt86 returns non-zero if errors occur while performing I/O operations or while parsing the command line. Otherwise it returns zero.

Warning messages and the source lines which caused them are written to standard error.

Bugs

acvt86 performs a limited set of transformations. Errors may occur when assembling the output. The object code from the 64853 assembly may not be the same as from the B1449 assembly.

acvt86 may detect a syntax error reading a legal HP 64853 program. The syntax of the HP 64853 assembly language is irregular. Occasionally, a legal assembler statement will be unacceptable to the translator. acvt86 will issue a warning when it detects a syntax error. The offending statement must be translated manually.
Comparison of Keywords

This section provides a side-by-side comparison of some of the HP 64853 constructs with the HP B1449 constructs. acvt86 performs most of the conversions shown in this section.

ALIGN

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>label ALIGN</td>
<td>EVEN</td>
</tr>
<tr>
<td>label:</td>
<td></td>
</tr>
</tbody>
</table>

In HP B1449 assembly language, EVEN directives cause errors if they appear in segments with align-types of BYTE. Use an align-type of WORD if you want to use the EVEN directive. Any label may appear on the following line.

ASSUME

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSUME segreg:ORG</td>
<td>ASSUME segreg:abs_segname</td>
</tr>
</tbody>
</table>

Most ASSUME directives need not be changed when moving to the HP B1449 assembler. However, when referring to absolute (for instance, ORGed) segment, you must do things differently. Briefly, when translating the ORG directive, you must create a named absolute segment using the SEGMENT directive. The ASSUME directive should then refer to this segment name. (See ORG for more information.)

COMN

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>label COMN</td>
<td>&lt;prevproc&gt; END &lt;prevseg&gt; ENDS COMN SEGMENT BYTE COMMON</td>
</tr>
<tr>
<td>label:</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 16: Converting HP 64853 Assembly Language Programs
Comparison of Keywords

Issue an ENDP to end the previous PROC, if necessary. Issue an ENDS directive to end the previous segment, if necessary. Any label must appear on the line following the directive.

DATA
HP 64853 HP B1449

label DATA <prevproc> ENDP

<prevseg> ENDS

DATA SEGMENT BYTE PUBLIC
label:

Issue an ENDP to end the previous PROC, if necessary. Issue an ENDS directive to end the previous segment, if necessary. Any label must appear on the line following the directive.

< EOF>
HP 64853 HP B1449

<EOF> <prevproc> ENDP
<prevseg> ENDS
END
<EOF>

Add an END directive to the module if not already present. Also, issue ENDP and ENDS directives if necessary.

EQU
HP 64853 HP B1449
id EQU <expr> id EQU <expr>
%SET(id, <expr>)
If an EQU label is ever referenced in a preprocessor expression (IF, REPT, or SET), then you must define that label for the preprocessor using the %SET directive. References to id in preprocessor expressions must be changed to %id.

**EXPAND**

The EXPAND function cannot be translated.

**EXT**

- **HP 64853**
  - EXT id
  - EXT id type
  - EXT segreg:id type

- **HP B1449**
  - EXTRN id:Near
  - EXTRN id:type

The HP 64853 declaration "EXT segreg:id" causes an automatic segment override when id is used in a memory reference operand. The HP B1449 assembler does not have an equal feature. Two approaches can be used to obtain the same code. You can either find all the references to id and add an explicit segment override to the operand when appropriate, or, place all the EXTRN directives with a particular associated segment register inside a segment. In the second case, you then must make sure an ASSUME directive is in effect for the proper segment register when the external identifiers are used.

**Label Field**

- **HP 64853**
  - label: directive
  - label: instruction
  - label macro name operands

- **HP B1449**
  - label directive
  - label: instruction
  - label: %macroname(operands)

Colons following labels are now significant. With the HP 64853 assembler, a colon following a label was optional. HP B1449 assembler prohibits a colon on a label for an assembler directive. HPB1449 assembler requires a colon on a label for a blank line, an instruction, and a macro definition.

296
Chapter 16: Converting HP 64853 Assembly Language Programs
Comparison of Keywords

**LIST**

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST</td>
<td>$LIST</td>
</tr>
<tr>
<td>LIST n</td>
<td>$PAGELENGTH(n) LIST</td>
</tr>
</tbody>
</table>

**Note**

PAGELENGTH is a primary control. It, and other HP B1449 primary controls, must be placed at the beginning of the file before any executable statements.

**MASK**

The MASK function cannot be translated. You must find any ASC directives which are affected and change the operands.

**NAME**

The NAME function, which puts a comment in the relocatable object module, cannot be translated.

**NOWARN**

The NOWARN function cannot be translated.

**Operator Field**

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AN.</td>
<td>AND</td>
</tr>
<tr>
<td>.EQ.</td>
<td>EQ</td>
</tr>
<tr>
<td>.GE.</td>
<td>GE</td>
</tr>
<tr>
<td>.GT.</td>
<td>GT</td>
</tr>
</tbody>
</table>
Chapter 16: Converting HP 64853 Assembly Language Programs
Comparison of Keywords

.LE.       LE
.LT        LT
.NE.       NE
.NT.       NOT
.OR.       OR
.SL.       SHL
.SR.       SHR
#1234      1234

Remove the pound sign before literal operands.

Within a string, make the following translations.

- A quote (') becomes two quotes in series ("').
- To the macro preprocessor, the percent sign, left parenthesis, and right parenthesis are special characters. You should add a preprocessor escape sequence to percent and to unbalanced parentheses to avoid processor errors.
- HP B1449 string delimiters are different.

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;string&quot;</td>
<td>'string'</td>
</tr>
<tr>
<td>^string^</td>
<td>'string'</td>
</tr>
<tr>
<td>'string'</td>
<td>'string'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORG</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>label ORG</td>
<td>&lt;prevproc&gt; ENDP</td>
</tr>
<tr>
<td></td>
<td>&lt;prevseg&gt; ENDS</td>
</tr>
</tbody>
</table>
The HP 64853 ORG directive begins an absolute segment. Translate as follows.

- Issue an ENDP to end the previous PROC, if necessary.
- Issue an ENDS to end the previous segment, if necessary.
- The upper 16 bits of the ORG expression represents the segment value and the lower 16 bits represent the offset. You must extract the paragraph value and the offset manually because the HP B1449 does not do 32 bit arithmetic.
- Start an absolute segment, using the AT keyword, at the paragraph value.
- Set the offset using the ORG directive.
- Any label must follow the ORG to retain its original value. It is not necessary to create a new absolute segment for every ORG directive. Several ORGed sections (with the same segment values) may be combined. The HP B1449 ORG directive may be used to set the offset with the absolute segment.

**PROC**

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>label PROC type &lt;prevproc&gt; ENDP</td>
<td>label PROC type</td>
</tr>
<tr>
<td>PROC FAR</td>
<td>&lt;prevproc&gt; ENDP</td>
</tr>
<tr>
<td>dummy PROC FAR</td>
<td></td>
</tr>
</tbody>
</table>

Issue an ENDP to end the previous procedure if necessary.

An unlabeled PROC directive is only useful for its effect on subsequent RET instructions. If the unlabeled PROC has type FAR, create a dummy PROC to
retain the same behavior. This dummy procedure is unnecessary if the type of
the unlabeled PROC is NEAR because HP B1449, by default, creates NEAR
return instructions when RETs appear outside of any procedure.
PROG

HP 64853           HP B1449

label PROG         <prevseg> ENDS

PROG SEGMENT BYTE PUBLIC

label:

Issue an ENDS directive to end the previous segment if necessary. Any label must appear on the line following the directive.

REAL

The REAL directive cannot be translated. REAL is not useful because the byte order of its numbers is opposite the 8086/186 convention. Use DD, DQ, or DT to create useful real numbers.

Reserved Words

HP 64853           HP B1449

TEST EQU 0          TESTx EQU 0

HP B1449 assembler recognizes more reserved identifiers. HP 64853 assembly language allowed you to define labels that were spelled the same as either instruction mnemonics or assembler directives. HP B1449 assembler does not allow reserved word duplication. Change the spelling of identifiers that duplicate reserved words.

SPC

HP 64853           HP B1449

SPC

The SPC function can only be translated into an equal number of empty source lines.
WARN

The WARN function cannot be translated.

* (Comment)

<table>
<thead>
<tr>
<th>HP 64853</th>
<th>HP B1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>* comment</td>
<td>; comment</td>
</tr>
</tbody>
</table>

 instr operand comment  instr operand ;comment

HP 64853 sometimes allows comments to begin with an asterisk and sometimes does not require any delimiter. HP B1449 requires all comments to begin with semicolon.
Linking to 64853 Programs

The HP B1449 linker does not accept either HP 64000 relocatable files or HP 64000 link_sym files. ld86 accepts only binary OMF relocatable files.

The utility L_to_o86 will transfer global symbol definitions from an HP 64000 link_sym file to an HP B1449 relocatable file. Thus, one can reference symbols produced by the HP 64853 linker (or produced by ld86) from binary OMF modules. Using this method, only the symbol definitions are linked. The code from the HP 64853 program is contained in an HP 64000 absolute file and must be loaded separately.

You may reference HP B1449 symbols from HP 64853 modules. Simply link the link_sym file produced by ld86 with the HP 64853 linker.

L_to_o86 uses another utility called nm64. Both are described on the following pages.
L_to_o86(1)

Name

L_to_o86 - Transfer global symbol definitions from HP 64000 link_sym file(s) to HP B1449 relocatable file(s)

Synopsis

/usr/contrib/bin/L_to_o86 file ...

Note

This utility program is unsupported. It is not a part of any HP product and is provided at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

L_to_o86 takes the global symbol definitions from an HP 64000 link_sym file and puts them into a relocatable file. These absolute symbol values may then be used in a linking operation by the HP B1449 8086/186 linker, ld86.

L_to_o86 produces one relocatable file for each link_sym file. The output file name is formed from the input name by stripping any preceding path name or .L suffix and appending .o.

The conversion is done in three steps. First, nm64 converts the link_sym file to a printable listing. Second, awk rearranges the listing into a 8086 assembly source file. Third, as86 assembles the source into a relocatable.

See Also

nm64(1), as86(1), ld86(1).

Diagnostics

L_to_o86 returns 1 if there is a command line error or if any input file cannot be opened. Otherwise, it returns zero. If the input file is not an HP 64000 link_sym file, then as86 will generate an assembly error message on standard error. The relocatable file will be valid but will contain no symbol definitions.
nm64(1)

Name
nm64 - print symbolic information from HP 64000 asmb_sym and link_sym files

Synopsis
/usr/contrib/bin/nm64 [-t] [file] ...

HP-UX Compatibility

Level:
Contributed Software

Origin:
Hewlett Packard - Logic Systems Division

Note
This utility program is unsupported. It is not a part of any HP product and is provided at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

Description
Nm64 is similar to the HP-UX utility, nml(1). It prints (on standard output) the symbol names, values, and relocation counter names for symbols contained in HP 64000 asmb_sym (assembly symbol) and link_sym (linker symbol) files. For the latter, it also prints the other types of records present: processor configuration, name, and memory space.

Usually, asmb_sym files have a .A suffix and link_sym files have a .L suffix.

If no files are specified, nm64 attempts to read standard input.

If the -t option is specified, it will print lines prior to each section of the file telling what disk address the section starts at.

See Also
Diagnostics
Nm64 returns 1 if it cannot open the input file. Otherwise, it returns zero.

Bugs
Addresses for different target processors may be stored in either one-word or two-word quantities. Assembly symbol files have no indication of which is used so applies a simple heuristic test to figure it out. The test could possibly fail.

Nm64 may also attempt to interpret other types of files with unpredictable results.
Chapter 17: 8086/186 Instructions in Hexadecimal Order

8086/186 Instructions in Hexadecimal Order
<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00000000</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>BYTE ADD (REG) TO EA</td>
</tr>
<tr>
<td>01 00000001</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>WORD ADD (REG) TO EA</td>
</tr>
<tr>
<td>02 00000010</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>REG,EA</td>
<td>BYTE ADD (EA) TO REG</td>
</tr>
<tr>
<td>03 00000011</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>REG,EA</td>
<td>WORD ADD (EA) TO REG</td>
</tr>
<tr>
<td>04 00000010</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AL,DATA8</td>
<td>BYTE ADD DATA TO REG AL</td>
</tr>
<tr>
<td>05 00000101</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AX,DATA16</td>
<td>WORD ADD DATA TO REG AX</td>
</tr>
<tr>
<td>06 00000110</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>ES</td>
<td>PUSH (ES) ON STACK</td>
</tr>
<tr>
<td>07 00000111</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>ES</td>
<td>POP STACK TO REG ES</td>
</tr>
<tr>
<td>08 00001000</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>BYTE OR (REG) TO EA</td>
</tr>
<tr>
<td>09 00001001</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>WORD OR (REG) TO EA</td>
</tr>
<tr>
<td>0A 00001010</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>REG,EA</td>
<td>BYTE OR (EA) TO REG</td>
</tr>
<tr>
<td>0B 00001011</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>REG,EA</td>
<td>WORD OR (EA) TO REG</td>
</tr>
<tr>
<td>0C 00001100</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AL,DATA8</td>
<td>BYTE OR DATA TO REG AL</td>
</tr>
<tr>
<td>0D 00001101</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AX,DATA16</td>
<td>WORD OR DATA TO REG AX</td>
</tr>
<tr>
<td>0E 00001110</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>CS</td>
<td>PUSH (CS) ON STACK</td>
</tr>
<tr>
<td>0F 00001111</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>(not used)</td>
<td>POP STACK TO REG SS</td>
</tr>
<tr>
<td>10 00010000</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>BYTE ADD (REG) W/ CARRY TO EA</td>
</tr>
<tr>
<td>11 00010001</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>WORD ADD (REG) W/ CARRY TO EA</td>
</tr>
<tr>
<td>12 00010010</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>BYTE ADD (EA) W/ CARRY TO REG</td>
</tr>
<tr>
<td>13 00010011</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>EA,REG</td>
<td>WORD ADD (EA) W/ CARRY TO REG</td>
</tr>
<tr>
<td>14 00010100</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AL,DATA8</td>
<td>BYTE ADD DATA W/CARRY TO REG AL</td>
</tr>
<tr>
<td>15 00010101</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>AX,DATA16</td>
<td>WORD ADD DATA W/ CARRY TO REG AX</td>
</tr>
<tr>
<td>16 00010110</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>SS</td>
<td>PUSH (SS) ON STACK</td>
</tr>
<tr>
<td>17 00010111</td>
<td>MOD REG R/M</td>
<td>ADD</td>
<td>SS</td>
<td>POP STACK TO REG SS</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>18 00011000</td>
<td>MOD REG R/M</td>
<td>SBB</td>
<td>EA,REG</td>
<td>BYTE SUB (REG) W/ BORROW FROM EA</td>
</tr>
<tr>
<td>19 00011001</td>
<td>MOD REG R/M</td>
<td>SBB</td>
<td>EA,REG</td>
<td>WORD SUB (REG) W/ BORROW FROM EA</td>
</tr>
<tr>
<td>1A 00011010</td>
<td>MOD REG R/M</td>
<td>SBB</td>
<td>REG,EA</td>
<td>BYTE SUB (EA) W/ BORROW FROM REG</td>
</tr>
<tr>
<td>1B 00011011</td>
<td>MOD REG R/M</td>
<td>SBB</td>
<td>REG,EA</td>
<td>WORD SUB (EA) W/ BORROW FROM REG</td>
</tr>
<tr>
<td>1C 00011100</td>
<td></td>
<td>SBB</td>
<td>AL,DATA8</td>
<td>BYTE SUB DATA W/ BORROW FROM REG AL</td>
</tr>
<tr>
<td>1D 00011101</td>
<td></td>
<td>SBB</td>
<td>AX,DATA16</td>
<td>WORD SUB DATA W/ BORROW FROM REG AX</td>
</tr>
<tr>
<td>1E 00011110</td>
<td></td>
<td>PUSH</td>
<td>DS</td>
<td>PUSH (DS) ON STACK</td>
</tr>
<tr>
<td>1F 00011111</td>
<td></td>
<td>POP</td>
<td>DS</td>
<td>POP STACK TO REG DS</td>
</tr>
<tr>
<td>20 00100000</td>
<td>MOD REG R/M</td>
<td>AND</td>
<td>EA,REG</td>
<td>BYTE AND (REG) TO EA</td>
</tr>
<tr>
<td>21 00100001</td>
<td>MOD REG R/M</td>
<td>AND</td>
<td>EA,REG</td>
<td>WORD AND (REG) TO EA</td>
</tr>
<tr>
<td>22 00100010</td>
<td>MOD REG R/M</td>
<td>AND</td>
<td>REG,EA</td>
<td>BYTE AND (EA) TO REG</td>
</tr>
<tr>
<td>23 00100011</td>
<td>MOD REG R/M</td>
<td>AND</td>
<td>REG,EA</td>
<td>WORD AND (EA) TO REG</td>
</tr>
<tr>
<td>24 00100100</td>
<td>MOD REG R/M</td>
<td>AND</td>
<td>AL,DATA8</td>
<td>BYTE AND DATA TO REG AL</td>
</tr>
<tr>
<td>25 00100101</td>
<td></td>
<td>AND</td>
<td>AX,DATA16</td>
<td>WORD AND DATA TO REG AX</td>
</tr>
<tr>
<td>26 00100110</td>
<td></td>
<td>ES:</td>
<td></td>
<td>SEGMENT OVERIDE W/ SEGMENT REG ES</td>
</tr>
<tr>
<td>27 00100111</td>
<td></td>
<td>DAA</td>
<td></td>
<td>DECIMAL ADJUST FOR ADD</td>
</tr>
<tr>
<td>28 00101000</td>
<td>MOD REG R/M</td>
<td>SUB</td>
<td>EA,REG</td>
<td>BYTE SUBTRACT (REG) FROM EA</td>
</tr>
<tr>
<td>29 00101001</td>
<td>MOD REG R/M</td>
<td>SUB</td>
<td>EA,REG</td>
<td>WORD SUBTRACT (REG) FROM EA</td>
</tr>
<tr>
<td>2A 00101010</td>
<td>MOD REG R/M</td>
<td>SUB</td>
<td>REG,EA</td>
<td>BYTE SUBTRACT (EA) FROM REG</td>
</tr>
<tr>
<td>2B 00101011</td>
<td>MOD REG R/M</td>
<td>SUB</td>
<td>REG,EA</td>
<td>WORD SUBTRACT (EA) FROM REG</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>2C 00101100</td>
<td></td>
<td>SUB</td>
<td>AL,DATA8</td>
<td>BYTE SUBTRACT DATA FROM REG AL</td>
</tr>
<tr>
<td>2D 00101101</td>
<td></td>
<td>SUB</td>
<td>AX,DATA16</td>
<td>WORD SUBTRACT DATA FROM REG AX</td>
</tr>
<tr>
<td>2E 00101110</td>
<td></td>
<td>CS:</td>
<td></td>
<td>SEGMENT OVERIDE W/ SEGMENT REG CS</td>
</tr>
<tr>
<td>2F 00101111</td>
<td></td>
<td>DAS</td>
<td></td>
<td>DECIMAL ADJUST FOR SUBTRACT</td>
</tr>
<tr>
<td>30 00110000</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>EA,REG</td>
<td>BYTE XOR (REG) TO EA</td>
</tr>
<tr>
<td>31 00110001</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>EA,REG</td>
<td>WORD XOR (REG) TO EA</td>
</tr>
<tr>
<td>32 00110010</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>REG,EA</td>
<td>BYTE XOR (EA) TO REG</td>
</tr>
<tr>
<td>33 00110011</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>REG,EA</td>
<td>WORD XOR (EA) TO REG</td>
</tr>
<tr>
<td>34 00110100</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>AL,DATA8</td>
<td>BYTE XOR DATA TO REG AL</td>
</tr>
<tr>
<td>35 00110101</td>
<td>MOD REG R/M</td>
<td>XOR</td>
<td>AX,DATA16</td>
<td>WORD XOR DATA TO REG AX</td>
</tr>
<tr>
<td>36 00110110</td>
<td></td>
<td>SS:</td>
<td></td>
<td>SEGMENT OVERIDE W/ SEGMENT REG SS</td>
</tr>
<tr>
<td>37 00110111</td>
<td></td>
<td>AAA</td>
<td></td>
<td>ASCII ADJUST FOR ADD</td>
</tr>
<tr>
<td>38 00111000</td>
<td>MOD REG R/M</td>
<td>CMP</td>
<td>EA,REG</td>
<td>BYTE COMPARE (EA) WITH (REG)</td>
</tr>
<tr>
<td>39 00111001</td>
<td>MOD REG R/M</td>
<td>CMP</td>
<td>EA,REG</td>
<td>WORD COMPARE (EA) WITH (REG)</td>
</tr>
<tr>
<td>3A 00111010</td>
<td>MOD REG R/M</td>
<td>CMP</td>
<td>REG,EA</td>
<td>BYTE COMPARE (REG) WITH (EA)</td>
</tr>
<tr>
<td>3B 00111011</td>
<td>MOD REG R/M</td>
<td>CMP</td>
<td>REG,EA</td>
<td>WORD COMPARE (REG) WITH (EA)</td>
</tr>
<tr>
<td>3C 00111100</td>
<td></td>
<td>CMP</td>
<td>AL,DATA8</td>
<td>BYTE COMPARE DATA WITH (AL)</td>
</tr>
<tr>
<td>3D 00111101</td>
<td></td>
<td>CMP</td>
<td>AX,DATA16</td>
<td>WORD COMPARE DATA WITH (AX)</td>
</tr>
<tr>
<td>3E 00111110</td>
<td></td>
<td>DS:</td>
<td></td>
<td>SEGMENT OVERIDE W/ SEGMENT REG DS</td>
</tr>
<tr>
<td>3F 00111111</td>
<td></td>
<td>AAS</td>
<td></td>
<td>ASCII ADJUST FOR SUBTRACT</td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 01000000</td>
<td></td>
<td>INC</td>
<td>AX</td>
<td>INCREMENT (AX)</td>
</tr>
<tr>
<td>41 01000001</td>
<td></td>
<td>INC</td>
<td>CX</td>
<td>INCREMENT (CX)</td>
</tr>
<tr>
<td>42 01000010</td>
<td></td>
<td>INC</td>
<td>DX</td>
<td>INCREMENT (DX)</td>
</tr>
<tr>
<td>43 01000011</td>
<td></td>
<td>INC</td>
<td>BX</td>
<td>INCREMENT (BX)</td>
</tr>
<tr>
<td>44 01000100</td>
<td></td>
<td>INC</td>
<td>SP</td>
<td>INCREMENT (SP)</td>
</tr>
<tr>
<td>45 01000101</td>
<td></td>
<td>INC</td>
<td>BP</td>
<td>INCREMENT (BP)</td>
</tr>
<tr>
<td>46 01000110</td>
<td></td>
<td>INC</td>
<td>SI</td>
<td>INCREMENT (SI)</td>
</tr>
<tr>
<td>47 01000111</td>
<td></td>
<td>INC</td>
<td>DI</td>
<td>INCREMENT (DI)</td>
</tr>
<tr>
<td>48 01001000</td>
<td></td>
<td>DEC</td>
<td>AX</td>
<td>DECREMENT (AX)</td>
</tr>
<tr>
<td>49 01001001</td>
<td></td>
<td>DEC</td>
<td>CX</td>
<td>DECREMENT (CX)</td>
</tr>
<tr>
<td>4A 01001010</td>
<td></td>
<td>DEC</td>
<td>DX</td>
<td>DECREMENT (DX)</td>
</tr>
<tr>
<td>4B 01001011</td>
<td></td>
<td>DEC</td>
<td>BX</td>
<td>DECREMENT (BX)</td>
</tr>
<tr>
<td>4C 01001100</td>
<td></td>
<td>DEC</td>
<td>SP</td>
<td>DECREMENT (SP)</td>
</tr>
<tr>
<td>4D 01001101</td>
<td></td>
<td>DEC</td>
<td>BP</td>
<td>DECREMENT (BP)</td>
</tr>
<tr>
<td>4E 01001110</td>
<td></td>
<td>DEC</td>
<td>SI</td>
<td>DECREMENT (SI)</td>
</tr>
<tr>
<td>4F 01001111</td>
<td></td>
<td>DEC</td>
<td>DI</td>
<td>DECREMENT (DI)</td>
</tr>
<tr>
<td>50 01010000</td>
<td></td>
<td>PUSH</td>
<td>AX</td>
<td>PUSH (AX) ON STACK</td>
</tr>
<tr>
<td>51 01010001</td>
<td></td>
<td>PUSH</td>
<td>CX</td>
<td>PUSH (CX) ON STACK</td>
</tr>
<tr>
<td>52 01010010</td>
<td></td>
<td>PUSH</td>
<td>DX</td>
<td>PUSH (DX) ON STACK</td>
</tr>
<tr>
<td>53 01010011</td>
<td></td>
<td>PUSH</td>
<td>BX</td>
<td>PUSH (BX) ON STACK</td>
</tr>
<tr>
<td>54 01010100</td>
<td></td>
<td>PUSH</td>
<td>SP</td>
<td>PUSH (SP) ON STACK</td>
</tr>
<tr>
<td>55 01010101</td>
<td></td>
<td>PUSH</td>
<td>BP</td>
<td>PUSH (BP) ON STACK</td>
</tr>
<tr>
<td>56 01010110</td>
<td></td>
<td>PUSH</td>
<td>SI</td>
<td>PUSH (SI) ON STACK</td>
</tr>
<tr>
<td>57 01010111</td>
<td></td>
<td>PUSH</td>
<td>DI</td>
<td>PUSH (DI) ON STACK</td>
</tr>
<tr>
<td>58 01011000</td>
<td></td>
<td>POP</td>
<td>AX</td>
<td>POP STACK TO REG AX</td>
</tr>
<tr>
<td>59 01011001</td>
<td></td>
<td>POP</td>
<td>CX</td>
<td>POP STACK TO REG CX</td>
</tr>
<tr>
<td>5A 01011010</td>
<td></td>
<td>POP</td>
<td>DX</td>
<td>POP STACK TO REG DX</td>
</tr>
<tr>
<td>5B 01011101</td>
<td></td>
<td>POP</td>
<td>BX</td>
<td>POP STACK TO REG BX</td>
</tr>
<tr>
<td>5C 01011100</td>
<td></td>
<td>POP</td>
<td>SP</td>
<td>POP STACK TO REG SP</td>
</tr>
<tr>
<td>5D 01011101</td>
<td></td>
<td>POP</td>
<td>BP</td>
<td>POP STACK TO REG BP</td>
</tr>
<tr>
<td>5E 01011110</td>
<td></td>
<td>POP</td>
<td>SI</td>
<td>POP STACK TO REG SI</td>
</tr>
<tr>
<td>5F 01011111</td>
<td></td>
<td>POP</td>
<td>DI</td>
<td>POP STACK TO REG DI</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>60 01100000</td>
<td></td>
<td>PUSH A</td>
<td></td>
<td>PUSH ALL DATA</td>
</tr>
<tr>
<td>61 01100001</td>
<td></td>
<td>POP A</td>
<td></td>
<td>POP ALL DATA</td>
</tr>
<tr>
<td>62 01100010</td>
<td>MOD REG R/M</td>
<td>BOUND</td>
<td>REG.EA</td>
<td>CHECK INDEX IN REG AGAINST BOUNDS AT EA</td>
</tr>
<tr>
<td>63 01100011</td>
<td></td>
<td></td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>64 01100100</td>
<td></td>
<td></td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>65 01100101</td>
<td></td>
<td></td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>66 01100110</td>
<td></td>
<td></td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>67 01100111</td>
<td></td>
<td></td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>68 01101000</td>
<td></td>
<td>PUSH</td>
<td>DATA16</td>
<td>PUSH WORD DATA ON STACK</td>
</tr>
<tr>
<td>69 01101001</td>
<td>MOD REG R/M</td>
<td>IMUL</td>
<td>REG.EA,</td>
<td>MULTIPLY (EA) BY WORD DATA; SIGNED</td>
</tr>
<tr>
<td>6A 01101010</td>
<td></td>
<td>PUSH DATA8</td>
<td></td>
<td>PUSH BYTE DATA ON STACK; SIGN-EXTEND</td>
</tr>
<tr>
<td>6B 01101011</td>
<td>MOD REG R/M</td>
<td>IMUL</td>
<td>REG.EA,</td>
<td>MULTIPLY (EA) BY BYTE DATA; SIGNED</td>
</tr>
<tr>
<td>6C 01101100</td>
<td></td>
<td>INS</td>
<td>DST8</td>
<td>BYTE INPUT</td>
</tr>
<tr>
<td>6D 01101101</td>
<td></td>
<td>INS</td>
<td>DST16</td>
<td>WORD INPUT</td>
</tr>
<tr>
<td>6E 01101110</td>
<td></td>
<td>OUTS</td>
<td>DST8</td>
<td>BYTE OUTPUT</td>
</tr>
<tr>
<td>6F 01101111</td>
<td></td>
<td>OUTS</td>
<td>DST16</td>
<td>WORD OUTPUT</td>
</tr>
<tr>
<td>70 01110000</td>
<td></td>
<td>JO</td>
<td>DISP8</td>
<td>JUMP ON OVERFLOW</td>
</tr>
<tr>
<td>71 01110001</td>
<td></td>
<td>JNO</td>
<td>DISP8</td>
<td>JUMP ON NOT OVERFLOW</td>
</tr>
<tr>
<td>72 01110010</td>
<td></td>
<td>JC/JB/JNAE</td>
<td>DISP8</td>
<td>JUMP ON BELOW/NOT ABOVE OR EQUAL</td>
</tr>
<tr>
<td>73 01110011</td>
<td></td>
<td>JNC/JNB/JAE</td>
<td>DISP8</td>
<td>JUMP ON NOT BELOW/ABOVE OR EQUAL</td>
</tr>
<tr>
<td>74 01110100</td>
<td></td>
<td>JE/JZ</td>
<td>DISP8</td>
<td>JUMP ON EQUAL/ZERO</td>
</tr>
<tr>
<td>75 01110101</td>
<td></td>
<td>JNE/JNZ</td>
<td>DISP8</td>
<td>JUMP ON NOT EQUAL/NOT ZERO</td>
</tr>
<tr>
<td>76 01110110</td>
<td></td>
<td>JBE/JNA</td>
<td>DISP8</td>
<td>JUMP ON BELOW OR EQUAL/NOT ABOVE</td>
</tr>
<tr>
<td>77 01110111</td>
<td></td>
<td>JNBE/JA</td>
<td>DISP8</td>
<td>JUMP ON NOT BELOW OR EQUAL/ABOVE</td>
</tr>
</tbody>
</table>
## Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>78 01111000</td>
<td>MOD 000 R/M</td>
<td>ADD</td>
<td>EA,DATA8</td>
<td>BYTE ADD DATA TO EA</td>
</tr>
<tr>
<td>79 01111001</td>
<td>MOD 001 R/M</td>
<td>OR</td>
<td>EA,DATA8</td>
<td>BYTE OR DATA TO EA</td>
</tr>
<tr>
<td>7A 01111010</td>
<td>MOD 010 R/M</td>
<td>ADC</td>
<td>EA,DATA8</td>
<td>BYTE ADD DATA W/CARRY TO EA</td>
</tr>
<tr>
<td>7B 01111011</td>
<td>MOD 011 R/M</td>
<td>SBB</td>
<td>EA,DATA8</td>
<td>BYTE SUB DATA W/BORROW FROM EA</td>
</tr>
<tr>
<td>7C 01111100</td>
<td>MOD 100 R/M</td>
<td>AND</td>
<td>EA,DATA8</td>
<td>BYTE AND DATA TO EA</td>
</tr>
<tr>
<td>7D 01111101</td>
<td>MOD 101 R/M</td>
<td>SUB</td>
<td>EA,DATA8</td>
<td>BYTE SUBTRACT DATA FROM EA</td>
</tr>
<tr>
<td>7E 01111110</td>
<td>MOD 110 R/M</td>
<td>XOR</td>
<td>EA,DATA8</td>
<td>BYTE XOR DATA TO EA</td>
</tr>
<tr>
<td>7F 01111111</td>
<td>MOD 111 R/M</td>
<td>CMP</td>
<td>EA,DATA8</td>
<td>BYTE COMPARE DATA WITH (EA)</td>
</tr>
<tr>
<td>80 10000000</td>
<td>MOD 000 R/M</td>
<td>ADD</td>
<td>EA,DATA16</td>
<td>WORD ADD DATA TO EA</td>
</tr>
<tr>
<td>80 10000001</td>
<td>MOD 001 R/M</td>
<td>OR</td>
<td>EA,DATA16</td>
<td>WORD OR DATA TO EA</td>
</tr>
<tr>
<td>80 10000010</td>
<td>MOD 010 R/M</td>
<td>ADC</td>
<td>EA,DATA16</td>
<td>WORD ADD DATA W/CARRY TO EA</td>
</tr>
<tr>
<td>80 10000011</td>
<td>MOD 011 R/M</td>
<td>SBB</td>
<td>EA,DATA16</td>
<td>WORD SUB DATA W/BORROW FROM EA</td>
</tr>
<tr>
<td>81 10000000</td>
<td>MOD 100 R/M</td>
<td>AND</td>
<td>EA,DATA16</td>
<td>WORD AND DATA TO EA</td>
</tr>
<tr>
<td>81 10000001</td>
<td>MOD 101 R/M</td>
<td>SUB</td>
<td>EA,DATA16</td>
<td>WORD SUBTRACT DATA FROM EA</td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>81 10000001</td>
<td>MOD 110 R/M</td>
<td>XOR</td>
<td>EA,DATA16</td>
<td>WORD XOR DATA TO EA</td>
</tr>
<tr>
<td>81 10000001</td>
<td>MOD 111 R/M</td>
<td>CMP</td>
<td>EA,DATA16</td>
<td>WORD COMPARE DATA WITH (EA)</td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 000 R/M</td>
<td>ADD</td>
<td>EA,DATA8</td>
<td>BYTE ADD DATA TO EA</td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 001 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 010 R/M</td>
<td>ADC</td>
<td>EA,DATA8</td>
<td>BYTE ADD DATA W/ CARRY TO EA</td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 011 R/M</td>
<td>SBB</td>
<td>EA,DATA8</td>
<td>BYTE SUB DATA W/ BORROW FROM EA</td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 100 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 101 R/M</td>
<td>SUB</td>
<td>EA,DATA8</td>
<td>BYTE SUBTRACT DATA FROM EA</td>
</tr>
<tr>
<td>82 10000010</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 100 R/M</td>
<td>CMP</td>
<td>EA,DATA8</td>
<td>BYTE COMPARE DATA WITH (EA)</td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 001 R/M</td>
<td>ADD</td>
<td>EA,DATA8</td>
<td>WORD ADD DATA TO EA</td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 011 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 100 R/M</td>
<td>SUB</td>
<td>EA,DATA8</td>
<td>WORD ADD DATA W/ CARRY TO EA</td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 101 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 110 R/M</td>
<td>ADD</td>
<td>EA,DATA8</td>
<td>WORD SUB DATA W/ BORROW FROM EA</td>
</tr>
<tr>
<td>83 10000011</td>
<td>MOD 111 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84 10000100</td>
<td>MOD REG R/M</td>
<td>TEST</td>
<td>EA,REG</td>
<td>BYTE TEST (EA) WITH (REG)</td>
</tr>
<tr>
<td>85 10000101</td>
<td>MOD REG R/M</td>
<td>TEST</td>
<td>EA,REG</td>
<td>WORD TEST (EA) WITH (REG)</td>
</tr>
<tr>
<td>86 10000110</td>
<td>MOD REG R/M</td>
<td>XCHG</td>
<td>REG,EA</td>
<td>BYTE EXCHANGE (REG) WITH (EA)</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>87 10000111</td>
<td>MOD REG R/M</td>
<td>XCHG</td>
<td>REG,EA</td>
<td>WORD EXCHANGE (REG) WITH (EA)</td>
</tr>
<tr>
<td>88 10001000</td>
<td>MOD REG R/M</td>
<td>MOV</td>
<td>EA,REG</td>
<td>BYTE MOVE (REG) TO EA</td>
</tr>
<tr>
<td>89 10001001</td>
<td>MOD REG R/M</td>
<td>MOV</td>
<td>EA,REG</td>
<td>WORD MOVE (REG) TO EA</td>
</tr>
<tr>
<td>8A 10001010</td>
<td>MOD REG R/M</td>
<td>MOV</td>
<td>REG,EA</td>
<td>BYTE MOVE (EA) TO REG</td>
</tr>
<tr>
<td>8B 10001011</td>
<td>MOD REG R/M</td>
<td>MOV</td>
<td>REG,EA</td>
<td>WORD MOVE (EA) TO REG</td>
</tr>
<tr>
<td>8C 10001100</td>
<td>MOD 0SR R/M</td>
<td>MOV</td>
<td>EA,SR</td>
<td>WORD MOVE (SEGMENT REG SR) TO EA</td>
</tr>
<tr>
<td>8D 10001101</td>
<td>MOD REG R/M</td>
<td>LEA</td>
<td>REG,EA</td>
<td>LOAD EFFECTIVE ADDRESS OF EA TO REG</td>
</tr>
<tr>
<td>8E 10001110</td>
<td>MOD 0SR R/M</td>
<td>MOV</td>
<td>SR,EA</td>
<td>WORD MOVE (EA) TO SEGMENT REG SR</td>
</tr>
<tr>
<td>8F 10010000</td>
<td></td>
<td>XCHG</td>
<td>AX,AX</td>
<td>EXCHANGE (AX) WITH (AX)</td>
</tr>
<tr>
<td>8F 10010001</td>
<td></td>
<td>XCHG</td>
<td>AX,CX</td>
<td>EXCHANGE (AX) WITH (CX)</td>
</tr>
<tr>
<td>8F 10010010</td>
<td></td>
<td>XCHG</td>
<td>AX,DX</td>
<td>EXCHANGE (AX) WITH (DX)</td>
</tr>
<tr>
<td>8F 10010011</td>
<td></td>
<td>XCHG</td>
<td>AX,BX</td>
<td>EXCHANGE (AX) WITH (BX)</td>
</tr>
<tr>
<td>8F 10010100</td>
<td></td>
<td>XCHG</td>
<td>AX,SP</td>
<td>EXCHANGE (AX) WITH (SP)</td>
</tr>
<tr>
<td>8F 10010101</td>
<td></td>
<td>XCHG</td>
<td>AX,BP</td>
<td>EXCHANGE (AX) WITH (BP)</td>
</tr>
<tr>
<td>8F 10010110</td>
<td></td>
<td>XCHG</td>
<td>AX,SI</td>
<td>EXCHANGE (AX) WITH (SI)</td>
</tr>
<tr>
<td>8F 10010111</td>
<td></td>
<td>XCHG</td>
<td>AX,DI</td>
<td>EXCHANGE (AX) WITH (DI)</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>--------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>98 10011000</td>
<td></td>
<td>CBW</td>
<td></td>
<td>BYTE CONVERT (AL) TO WORD (AX)</td>
</tr>
<tr>
<td>99 10011001</td>
<td></td>
<td>CWD</td>
<td></td>
<td>WORD CONVERT (AX) TO DOUBLE WORD</td>
</tr>
<tr>
<td>9A 10011010</td>
<td></td>
<td>CALL</td>
<td>DISP16,SEG16</td>
<td>DIRECT INTER SEGMENT CALL</td>
</tr>
<tr>
<td>9B 10011011</td>
<td></td>
<td>WAIT</td>
<td></td>
<td>WAIT FOR TEST SIGNAL</td>
</tr>
<tr>
<td>9C 10011100</td>
<td></td>
<td>PUSHF</td>
<td></td>
<td>PUSH FLAGS ON STACK</td>
</tr>
<tr>
<td>9D 10011101</td>
<td></td>
<td>POPF</td>
<td></td>
<td>POP STACK TO FLAGS</td>
</tr>
<tr>
<td>9E 10011110</td>
<td></td>
<td>SAHF</td>
<td></td>
<td>STORE (AH) INTO FLAGS</td>
</tr>
<tr>
<td>9F 10011111</td>
<td></td>
<td>LAHF</td>
<td></td>
<td>LOAD REG AH WITH FLAGS</td>
</tr>
<tr>
<td>A0 10100000</td>
<td></td>
<td>MOV</td>
<td>AL,ADDR16</td>
<td>BYTE MOVE (ADDR) TO REG AL</td>
</tr>
<tr>
<td>A1 10100001</td>
<td></td>
<td>MOV</td>
<td>AX,ADDR16</td>
<td>WORD MOVE (ADDR) TO REG AX</td>
</tr>
<tr>
<td>A2 10100010</td>
<td></td>
<td>MOV</td>
<td>ADDR16,AL</td>
<td>BYTE MOVE (AL) TO ADDR</td>
</tr>
<tr>
<td>A3 10100011</td>
<td></td>
<td>MOV</td>
<td>ADDR16,AX</td>
<td>WORD MOVE (AX) TO ADDR</td>
</tr>
<tr>
<td>A4 10100100</td>
<td></td>
<td>MOV</td>
<td>DST8,SRC8</td>
<td>BYTE MOVE, STRING OP</td>
</tr>
<tr>
<td>A5 10100101</td>
<td></td>
<td>MOV</td>
<td>DST16,SRC16</td>
<td>WORD MOVE, STRING OP</td>
</tr>
<tr>
<td>A6 10100110</td>
<td></td>
<td>CMPS</td>
<td>SIPTR,DIPTR</td>
<td>COMPARE BYTE, STRING OP</td>
</tr>
<tr>
<td>A7 10100111</td>
<td></td>
<td>CMPS</td>
<td>SIPTR,DIPTR</td>
<td>COMPARE WORD, STRING OP</td>
</tr>
<tr>
<td>A8 10101000</td>
<td></td>
<td>TEST</td>
<td>AL,DATA8</td>
<td>BYTE TEST (AL) WITH DATA</td>
</tr>
<tr>
<td>A9 10101001</td>
<td></td>
<td>TEST</td>
<td>AX,DATA16</td>
<td>WORD TEST (AX) WITH DATA</td>
</tr>
<tr>
<td>AA 10101010</td>
<td></td>
<td>STOS</td>
<td>DST8</td>
<td>BYTE STORE, STRING OP</td>
</tr>
<tr>
<td>AB 10101011</td>
<td></td>
<td>STOS</td>
<td>DST16</td>
<td>WORD STORE, STRING OP</td>
</tr>
<tr>
<td>AC 10101100</td>
<td></td>
<td>LODS</td>
<td>SRC8</td>
<td>BYTE LOAD, STRING OP</td>
</tr>
<tr>
<td>AD 10101101</td>
<td></td>
<td>LODS</td>
<td>SRC16</td>
<td>WORD LOAD, STRING OP</td>
</tr>
<tr>
<td>AE 10101110</td>
<td></td>
<td>SCAS</td>
<td>DIPTR8</td>
<td>BYTE SCAN, STRING OP</td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF 10101111</td>
<td></td>
<td>SCAS</td>
<td>DIPTR16</td>
<td>WORD SCAN, STRING OP</td>
</tr>
<tr>
<td>B0 10110000</td>
<td></td>
<td>MOV</td>
<td>AL,DATA8</td>
<td>BYTE MOVE DATA TO REG AL</td>
</tr>
<tr>
<td>B1 10110001</td>
<td></td>
<td>MOV</td>
<td>CL,DATA8</td>
<td>BYTE MOVE DATA TO REG CL</td>
</tr>
<tr>
<td>B2 10110010</td>
<td></td>
<td>MOV</td>
<td>DL,DATA8</td>
<td>BYTE MOVE DATA TO REG DL</td>
</tr>
<tr>
<td>B3 10110011</td>
<td></td>
<td>MOV</td>
<td>BL,DATA8</td>
<td>BYTE MOVE DATA TO REG BL</td>
</tr>
<tr>
<td>B4 10110100</td>
<td></td>
<td>MOV</td>
<td>AH,DATA8</td>
<td>BYTE MOVE DATA TO REG AH</td>
</tr>
<tr>
<td>B5 10110101</td>
<td></td>
<td>MOV</td>
<td>CH,DATA8</td>
<td>BYTE MOVE DATA TO REG CH</td>
</tr>
<tr>
<td>B6 10110110</td>
<td></td>
<td>MOV</td>
<td>DH,DATA8</td>
<td>BYTE MOVE DATA TO REG DH</td>
</tr>
<tr>
<td>B7 10110111</td>
<td></td>
<td>MOV</td>
<td>BH,DATA8</td>
<td>BYTE MOVE DATA TO REG BH</td>
</tr>
<tr>
<td>B8 10111000</td>
<td></td>
<td>MOV</td>
<td>AX,DATA16</td>
<td>WORD MOVE DATA TO REG AX</td>
</tr>
<tr>
<td>B9 10111001</td>
<td></td>
<td>MOV</td>
<td>CX,DATA16</td>
<td>WORD MOVE DATA TO REG CX</td>
</tr>
<tr>
<td>BA 10111010</td>
<td></td>
<td>MOV</td>
<td>DX,DATA16</td>
<td>WORD MOVE DATA TO REG DX</td>
</tr>
<tr>
<td>BB 10111011</td>
<td></td>
<td>MOV</td>
<td>BX,DATA16</td>
<td>WORD MOVE DATA TO REG BX</td>
</tr>
<tr>
<td>BC 10111100</td>
<td></td>
<td>MOV</td>
<td>SP,DATA16</td>
<td>WORD MOVE DATA TO REG SP</td>
</tr>
<tr>
<td>BD 10111101</td>
<td></td>
<td>MOV</td>
<td>BP,DATA16</td>
<td>WORD MOVE DATA TO REG BP</td>
</tr>
<tr>
<td>BE 10111110</td>
<td></td>
<td>MOV</td>
<td>SI,DATA16</td>
<td>WORD MOVE DATA TO REG SI</td>
</tr>
<tr>
<td>BF 10111111</td>
<td></td>
<td>MOV</td>
<td>DI,DATA16</td>
<td>WORD MOVE DATA TO REG DI</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 000 R/M</td>
<td>ROL</td>
<td>EA,DATA8</td>
<td>BYTE ROTATE EA LEFT DATA8 BITS</td>
</tr>
</tbody>
</table>
# Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0 11000000</td>
<td>MOD 001 R/M</td>
<td>ROR</td>
<td>EA,DATA8</td>
<td>BYTE ROTATE EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 010 R/M</td>
<td>RCL</td>
<td>EA,DATA8</td>
<td>BYTE ROTATE EA LEFT THRU CARRY DATA8 BITS</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 011 R/M</td>
<td>RCR</td>
<td>EA,DATA8</td>
<td>BYTE ROTATE EA RIGHT THRU CARRY DATA8 BITS</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 100 R/M</td>
<td>SHL/SAL</td>
<td>EA,DATA8</td>
<td>BYTE SHIFT EA LEFT DATA8 BITS</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 101 R/M</td>
<td>SHR</td>
<td>EA,DATA8</td>
<td>BYTE SHIFT EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0 11000000</td>
<td>MOD 111 R/M</td>
<td>SAR</td>
<td>EA,DATA8</td>
<td>BYTE SHIFT SIGNED EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 000 R/M</td>
<td>ROL</td>
<td>EA,DATA8</td>
<td>WORD ROTATE EA LEFT DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 001 R/M</td>
<td>ROR</td>
<td>EA,DATA8</td>
<td>WORD ROTATE EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 010 R/M</td>
<td>RCL</td>
<td>EA,DATA8</td>
<td>WORD ROTATE EA LEFT THRU CARRY DATA8 BITSCARRY DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 011 R/M</td>
<td>RCR</td>
<td>EA,DATA8</td>
<td>WORD ROTATE EA RIGHT THRU CARRY DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 100 R/M</td>
<td>SHL/SAL</td>
<td>EA,DATA8</td>
<td>WORD SHIFT EA LEFT DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 101 R/M</td>
<td>SHR</td>
<td>EA,DATA8</td>
<td>WORD SHIFT EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1 11000001</td>
<td>MOD 111 R/M</td>
<td>SAR</td>
<td>EA,DATA8</td>
<td>WORD SHIFT SIGNED EA RIGHT DATA8 BITS</td>
</tr>
<tr>
<td>C2 11000010</td>
<td>MOD REG R/M</td>
<td>LES</td>
<td>ES,REG, EA</td>
<td>WORD LOAD REG AND SEGMENT REG ES</td>
</tr>
<tr>
<td>C3 11000011</td>
<td>MOD REG R/M</td>
<td>LDS</td>
<td>DS,REG,EA</td>
<td>WORD LOAD REG AND SEGMENT REG DS</td>
</tr>
</tbody>
</table>

---

318
## Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6 11000110</td>
<td>MOD 000 R/M</td>
<td>MOV</td>
<td>EA, DATA8</td>
<td>BYTE MOVE DATA TO EA</td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 001 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 010 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 011 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 100 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 101 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOD 111 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 000 R/M</td>
<td>MOV</td>
<td>EA, DATA16</td>
<td>WORD MOVE DATA TO EA</td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 001 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 010 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 011 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 100 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 101 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7 11000111</td>
<td>MOD 111 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8 11001000</td>
<td></td>
<td>ENTER</td>
<td>DATA16, DATA8</td>
<td>PERFORM ENTER SEQUENCE</td>
</tr>
<tr>
<td>C9 11001001</td>
<td></td>
<td>LEAVE</td>
<td></td>
<td>PERFORM LEAVE SEQUENCE</td>
</tr>
<tr>
<td>CA 11001010</td>
<td></td>
<td>RET</td>
<td>DATA16</td>
<td>INTER SEGMENT RETURN</td>
</tr>
<tr>
<td>CB 11001011</td>
<td></td>
<td>RET</td>
<td></td>
<td>INTER SEGMENT RETURN</td>
</tr>
<tr>
<td>CC 11001100</td>
<td></td>
<td>INT</td>
<td>3</td>
<td>TYPE 3 INTERRUPT</td>
</tr>
<tr>
<td>CD 11001101</td>
<td></td>
<td>INT</td>
<td>TYPE</td>
<td>TYPED INTERRUPT</td>
</tr>
<tr>
<td>CE 11001110</td>
<td></td>
<td>INTO</td>
<td></td>
<td>INTERRUPT ON OVERFLOW</td>
</tr>
<tr>
<td>CF 11001111</td>
<td></td>
<td>IRET</td>
<td></td>
<td>RETURN FROM INTERRUPT</td>
</tr>
<tr>
<td>D0 11010000</td>
<td>MOD 000 R/M</td>
<td>ROL</td>
<td>EA,1</td>
<td>BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D0 11010000</td>
<td>MOD 001 R/M</td>
<td>ROR</td>
<td>EA,1</td>
<td>BYTE ROTATE EA RIGHT 1 BIT</td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 11010000 MOD 010 R/M</td>
<td>RCL</td>
<td>EA,1</td>
<td>BYTE ROTATE EA LEFT THRU CARRY 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D0 11010000 MOD 011 R/M</td>
<td>RCR</td>
<td>EA,1</td>
<td>BYTE ROTATE EA RIGHT THRU CARRY 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D0 11010000 MOD 100 R/M</td>
<td>SHL</td>
<td>EA,1</td>
<td>BYTE SHIFT EA LEFT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D0 11010000 MOD 101 R/M</td>
<td>SHR</td>
<td>EA,1</td>
<td>BYTE SHIFT EA RIGHT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D0 11010000 MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0 11010000 MOD 111 R/M</td>
<td>SAR</td>
<td>EA,1</td>
<td>BYTE SHIFT SIGNED EA RIGHT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 000 R/M</td>
<td>ROL</td>
<td>EA,1</td>
<td>WORD ROTATE EA LEFT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 001 R/M</td>
<td>ROR</td>
<td>EA,1</td>
<td>WORD ROTATE EA RIGHT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 010 R/M</td>
<td>RCL</td>
<td>EA,1</td>
<td>WORD ROTATE EA LEFT THRU CARRY 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 011 R/M</td>
<td>RCR</td>
<td>EA,1</td>
<td>WORD ROTATE EA RIGHT THRU CARRY 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 100 R/M</td>
<td>SHL</td>
<td>EA,1</td>
<td>WORD SHIFT EA LEFT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 101 R/M</td>
<td>SHR</td>
<td>EA,1</td>
<td>WORD SHIFT EA RIGHT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1 11010001 MOD 111 R/M</td>
<td>SAR</td>
<td>EA,1</td>
<td>WORD SHIFT SIGNED EA RIGHT 1 BIT</td>
<td></td>
</tr>
<tr>
<td>D2 11010010 MOD 000 R/M</td>
<td>ROL</td>
<td>EA,CL</td>
<td>BYTE ROTATE EA LEFT (CL) BITS</td>
<td></td>
</tr>
<tr>
<td>D2 11010010 MOD 001 R/M</td>
<td>ROR</td>
<td>EA,CL</td>
<td>BYTE ROTATE EA RIGHT (CL) BITS</td>
<td></td>
</tr>
<tr>
<td>D2 11010010 MOD 010 R/M</td>
<td>RCL</td>
<td>EA,CL</td>
<td>BYTE ROTATE EA LEFT THRU CARRY (CL) BITS</td>
<td></td>
</tr>
<tr>
<td>D2 11010010 MOD 011 R/M</td>
<td>RCR</td>
<td>EA,CL</td>
<td>BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS</td>
<td></td>
</tr>
<tr>
<td>D2 11010010 MOD 100 R/M</td>
<td>SHL</td>
<td>EA,CL</td>
<td>BYTE SHIFT EA LEFT (CL) BITS</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2 11010010</td>
<td>MOD 101 R/M</td>
<td>SHR</td>
<td>EA,CL</td>
<td>BYTE SHIFT EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td>D2 11010010</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2 11010010</td>
<td>MOD 111 R/M</td>
<td>SAR</td>
<td>EA,CL</td>
<td>BYTE SHIFT SIGNED EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 000 R/M</td>
<td>ROL</td>
<td>EA,CL</td>
<td>WORD ROTATE EA LEFT (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 001 R/M</td>
<td>ROR</td>
<td>EA,CL</td>
<td>WORD ROTATE EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 010 R/M</td>
<td>RCL</td>
<td>EA,CL</td>
<td>WORD ROTATE EA LEFT THRU CARRY (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 011 R/M</td>
<td>RCR</td>
<td>EA,CL</td>
<td>WORD Rotate EA RIGHT THRU CARRY (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 100 R/M</td>
<td>SHL</td>
<td>EA,CL</td>
<td>WORD SHIFT EA LEFT (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 101 R/M</td>
<td>SHR</td>
<td>EA,CL</td>
<td>WORD SHIFT EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 110 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3 11010011</td>
<td>MOD 111 R/M</td>
<td>SAR</td>
<td>EA,CL</td>
<td>WORD SHIFT SIGNED EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td>D4 11010100</td>
<td>00001010</td>
<td>AAM</td>
<td></td>
<td>ASCII ADJUST FOR MULTIPLY</td>
</tr>
<tr>
<td>D5 11010101</td>
<td>00001010</td>
<td>AAD</td>
<td></td>
<td>ASCII ADJUST FOR DIVIDE</td>
</tr>
<tr>
<td>D6 11010110</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7 11010111</td>
<td>00001010</td>
<td>XLAT</td>
<td>TABLE</td>
<td>TRANSLATE USING (BX)</td>
</tr>
<tr>
<td>D8 11011---</td>
<td>MOD --- R/M</td>
<td>ESC</td>
<td>EA</td>
<td>ESCAPE TO EXTERNAL DEVICE</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 000 R/M</td>
<td>FADD</td>
<td>Short-real</td>
<td>ADD 4-BYTE EA TO ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 001 R/M</td>
<td>FMUL</td>
<td>Short-real</td>
<td>MULTIPLY ST BY 4-BYTE EA</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 010 R/M</td>
<td>FCOM</td>
<td>Short-real</td>
<td>COMPARE 4-BYTE EA WITH ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 011 R/M</td>
<td>FCOMP</td>
<td>Short-real</td>
<td>COMPARE 4-BYTE EA WITH ST AND POP</td>
</tr>
</tbody>
</table>
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 11011000</td>
<td>MOD 100 R/M</td>
<td>FADD</td>
<td>ST,ST(i)</td>
<td>ADD ELEMENT TO ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 110 R/M</td>
<td>FADD</td>
<td>ST,ST(i)</td>
<td>ADD ELEMENT TO ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>MOD 111 R/M</td>
<td>FDIV</td>
<td>ST,ST(i)</td>
<td>DIVIDE ST BY ELEMENT</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 000 (i)</td>
<td>FDIV</td>
<td>ST,ST(i)</td>
<td>DIVIDE ST BY ELEMENT</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 001 (i)</td>
<td>FMUL</td>
<td>ST,ST(i)</td>
<td>MULTIPLY ST BY ELEMENT</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 010 (i)</td>
<td>FCOM</td>
<td>ST(i)</td>
<td>COMPARE ST(i) WITH ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 011 (i)</td>
<td>FCOMP</td>
<td>ST(i)</td>
<td>COMPARE ST(i) WITH ST AND POP</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 100 (i)</td>
<td>FSUB</td>
<td>ST,ST(i)</td>
<td>SUBTRACT ELEMENT FROM ST</td>
</tr>
<tr>
<td>D8 11011000</td>
<td>1 1 101 (i)</td>
<td>FSUBR</td>
<td>ST,ST(i)</td>
<td>SUBTRACT ST FROM STACK ELEMENT</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 000 R/M</td>
<td>FLD</td>
<td>Short-real</td>
<td>PUSH 4-BYTE EA TO ST</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 001 R/M</td>
<td>FST</td>
<td>Short-real</td>
<td>STORE 4-BYTE REAL TO EA AND POP</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 010 R/M</td>
<td>FLDENV</td>
<td>14-BYTES</td>
<td>LOAD 8087 ENVIRONMENT FROM EA</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 011 R/M</td>
<td>FSTENV</td>
<td>14-BYTES</td>
<td>STORE 8087 ENVIRONMENT INTO EA</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 100 R/M</td>
<td>FLDCW</td>
<td>2-BYTES</td>
<td>LOAD CONTROL WORD FROM EA</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 110 R/M</td>
<td>FSTCW</td>
<td>2-BYTES</td>
<td>STORE CONTROL WORD INTO EA</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 101 R/M</td>
<td>FLD</td>
<td>ST(i)</td>
<td>PUSH ST(i) ONTO ST</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>MOD 111 R/M</td>
<td>FXCH</td>
<td>ST(i)</td>
<td>EXCHANGE ST AND ST(i)</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>1 1 000 (i)</td>
<td>FNOP</td>
<td></td>
<td>STORE ST IN ST</td>
</tr>
<tr>
<td>D9 11011001</td>
<td>1 1 001 (i)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001</td>
<td>1 1 010 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001</td>
<td>1 1 010 001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>D9 11011001 1 010 01-</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 010 1--</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 011 (i)</td>
<td>*(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 100 000</td>
<td>FCHS</td>
<td></td>
<td></td>
<td>CHANGE SIGN OF ST</td>
</tr>
<tr>
<td>D9 11011001 1 100 001</td>
<td>FABS</td>
<td></td>
<td></td>
<td>TAKE ABSOLUTE VALUE OF ST</td>
</tr>
<tr>
<td>D9 11011001 1 100 01-</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 100 100</td>
<td>FTST</td>
<td></td>
<td></td>
<td>TEST ST AGAINST 0.0</td>
</tr>
<tr>
<td>D9 11011001 1 100 101</td>
<td>FXAM</td>
<td></td>
<td></td>
<td>EXAMINE ST AND REPORT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CONDITION CODE</td>
</tr>
<tr>
<td>D9 11011001 1 100 11-</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 101 000</td>
<td>FLD1</td>
<td></td>
<td></td>
<td>PUSH + 1.0 TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 001</td>
<td>FLDL2T</td>
<td></td>
<td></td>
<td>PUSH log 2 10 TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 010</td>
<td>FLDL2E</td>
<td></td>
<td></td>
<td>PUSH log 2 e TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 011</td>
<td>FLDPI</td>
<td></td>
<td></td>
<td>PUSH Pi TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 100</td>
<td>FLDLG2</td>
<td></td>
<td></td>
<td>PUSH log 10 2 TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 101</td>
<td>FLDLN2</td>
<td></td>
<td></td>
<td>PUSH log e 2 TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 110</td>
<td>FLDZ</td>
<td></td>
<td></td>
<td>PUSH ZERO TO ST</td>
</tr>
<tr>
<td>D9 11011001 1 101 111</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 110 000</td>
<td>F2XM1</td>
<td></td>
<td></td>
<td>CALCULATE 2 x - 1</td>
</tr>
<tr>
<td>D9 11011001 1 110 001</td>
<td>FYL2X</td>
<td></td>
<td></td>
<td>CALCULATE FUNCTION Y*log 2 X</td>
</tr>
<tr>
<td>D9 11011001 1 110 010</td>
<td>FPTAN</td>
<td></td>
<td></td>
<td>CALCULATE TAN OF 0 AS A RATIO</td>
</tr>
<tr>
<td>D9 11011001 1 110 011</td>
<td>FPATAN</td>
<td></td>
<td></td>
<td>CALCULATE ARCTAN OF 0</td>
</tr>
<tr>
<td>D9 11011001 1 110 100</td>
<td>FXTRACT</td>
<td></td>
<td></td>
<td>EXTRACT EXPONENT AND SIGNIFICAND FROM ST VALUE</td>
</tr>
<tr>
<td>D9 11011001 1 110 101</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9 11011001 1 110 110</td>
<td>FDECSTP</td>
<td></td>
<td></td>
<td>DECREMENT STACK POINTER IN STATUS WORD</td>
</tr>
<tr>
<td>D9 11011001 1 110 111</td>
<td>FINCSTP</td>
<td></td>
<td></td>
<td>INCREMENT STACK POINTER IN STATUS WORD</td>
</tr>
</tbody>
</table>
### Hex Binary | MODRM Byte | Instruction | Parameters | Function
---|---|---|---|---
D9 11011001 1 111 000 |  | FPREM | | MODULO DIVISION OF ST BY ST(1)
D9 11011001 1 110 001 |  | FYL2XP | | CALCULATE VALUE OF Y*\log_2 (X + 1)
D9 11011001 1 111 010 |  | FSQRT | | CALCULATE SQUARE ROOT OF ST
D9 11011001 1 111 011 |  | (not used) | | ROUND ST TO INTEGER
D9 11011001 1 111 100 |  | FRNDINT | | ADD ST(1) TO EXponent OF ST
D9 11011001 1 111 101 |  | FSQALE | | ADD ST(1) TO EXPONENT OF ST
DA 11011010 MOD 000 R/M | RIADD | Short-integer | | ADD 4-BYTE INTEGER EA TO ST
DA 11011010 MOD 001 R/M | FIMUL | Short-integer | | MULTIPLY ST BY 4-BYTE INTEGER EA
DA 11011010 MOD 010 R/M | FICOM | Short-integer | | CONVERT 4-BYTE INTEGER EA, AND COMPARE WITH ST
DA 11011010 MOD 011 R/M | FICOMP | Short-integer | | CONVERT 4-BYTE INTEGER EA, COMPARE WITH ST, POP
DA 11011010 MOD 100 R/M | FISUB | Short-integer | | SUBTRACT 4-BYTE INTEGER EA FROM ST
DA 11011010 MOD 101 R/M | FISUBR | Short-integer | | SUBTRACT ST FROM 4-BYTE INTEGER EA
DA 11011010 MOD 110 R/M | FIDIV | Short-integer | | DIVIDE ST BY 4-BYTE INTEGER EA
DA 11011010 MOD 111 R/M | FIDIVR | Short-integer | | DIVIDE 4-BYTE INTEGER EA BY ST
DA 11011010 1 1 11 110- | (not used) | | |
DB 11011011 MOD 000 R/M | FILD | Short-integer | | |
DB 11011011 MOD 001 R/M | (not used) | | |
DB 11011011 MOD 010 R/M | FIST | Short-integer | | STORE ROUNDED ST IN 4-BYTE INTEGER EA
<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB 11011011 MOD 011 R/M</td>
<td>FISTP</td>
<td>Short-integer</td>
<td>STORE ROUNDED ST INTO 4-BYTE INTEGER EA, POP</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 MOD 100 R/M</td>
<td>(not used)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB 11011011 MOD 101 R/M</td>
<td>FLD</td>
<td>Temp-real</td>
<td>PUSH 10-BYTE EA ONTO ST</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 MOD 110 R/M</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB 11011011 MOD 111 R/M</td>
<td>FSTP</td>
<td>Temp-real</td>
<td>STORE ST INTO 10-BYTE EA, POP</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 0-- ---</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 100 000</td>
<td>FENI</td>
<td></td>
<td>ENABLE INTERRUPT</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 100 001</td>
<td>FDISI</td>
<td></td>
<td>DISABLE INTERRUPTS</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 100 010</td>
<td>FCLEX</td>
<td></td>
<td>CLEAR EXCEPTIONS</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 100 011</td>
<td>FINIT</td>
<td></td>
<td>INITIALIZE PROCESSOR</td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 100 1--</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 101 --</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB 11011011 1 1 11-- ---</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 000 R/M</td>
<td>FADD</td>
<td>Long-real</td>
<td>ADD 8-BYTE EA TO ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 001 R/M</td>
<td>FMUL</td>
<td>Long-real</td>
<td>MULTIPLY ST BY 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 010 R/M</td>
<td>FCOM</td>
<td>Long-real</td>
<td>COMPARE ST WITH 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 011 R/M</td>
<td>FCOMP</td>
<td>Long-real</td>
<td>COMPARE ST WITH 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 100 R/M</td>
<td>FSUB</td>
<td>Long-real</td>
<td>SUBTRACT 8-BYTE EA FROM ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 101 R/M</td>
<td>FSUBR</td>
<td>Long-real</td>
<td>SUBTRACT ST FROM 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 110 R/M</td>
<td>FDIV</td>
<td>Long-real</td>
<td>DIVIDE ST BY 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 MOD 111 R/M</td>
<td>FDIVR</td>
<td>Long-real</td>
<td>DIVIDE 8-BYTE EA BY ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 000 (i)</td>
<td>FADD</td>
<td>ST(i), ST</td>
<td>ADD ST TO ELEMENT</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 001 (i)</td>
<td>FMUL</td>
<td>ST(i), ST</td>
<td>MULTIPLY ELEMENT BY ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 010 (i)</td>
<td>*(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 011 (i)</td>
<td>*(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 100 (i)</td>
<td>FSUBR</td>
<td>ST(i), ST</td>
<td>SUBTRACT ST FROM ELEMENT</td>
<td></td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>DC 11011100 1 1 101 (i)</td>
<td>FSUB</td>
<td>ST(i), ST</td>
<td>SUBTRACT ELEMENT FROM ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 110 (i)</td>
<td>FDIVR</td>
<td>ST(i), ST</td>
<td>DIVIDE ST(i) BY ST</td>
<td></td>
</tr>
<tr>
<td>DC 11011100 1 1 111 (i)</td>
<td>FDIV</td>
<td>ST(i), ST</td>
<td>DIVIDE ST BY ST(i)</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 000 R/M</td>
<td>FLD</td>
<td>Long-real</td>
<td>PUSH 8-BYTE EA ONTO ST</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 001 R/M</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 010 R/M</td>
<td>FST</td>
<td>Long-real</td>
<td>STORE ST INTO 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 011 R/M</td>
<td>FSTP</td>
<td>Long-real</td>
<td>STORE ST INTO 8-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 100 R/M</td>
<td>FRSTOR</td>
<td>94-BYTES</td>
<td>RESTORE 8087 STATE FROM EA</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 101 R/M</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 110 R/M</td>
<td>FSAVE</td>
<td>94-BYTES</td>
<td>SAVE 8087 STATE TO EA</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 MOD 111 R/M</td>
<td>FSTSW</td>
<td>2-BYTES</td>
<td>STORE 8087 STATUS WORD TO 2-BYTE EA</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 1 1 000 (i)</td>
<td>FFREE</td>
<td>ST(i)</td>
<td>SET STACK TAG TO &quot;EMPTY&quot;</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 1 1 001 (i)</td>
<td>*(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD 11011101 1 1 010 (i)</td>
<td>FST</td>
<td>ST(i)</td>
<td>STORE ST INTO ST(i)</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 1 1 011 (i)</td>
<td>FSTP</td>
<td>ST(i)</td>
<td>STORE ST INTO ST(i), POP</td>
<td></td>
</tr>
<tr>
<td>DD 11011101 1 1 1-- ---</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 000 R/M</td>
<td>FIADD</td>
<td>Word-integer</td>
<td>ADD 2-BYTE INTEGER EA TO ST</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 001 R/M</td>
<td>FIMUL</td>
<td>Word-integer</td>
<td>MULTIPLY ST BY 2-BYTE INTEGER EA</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 010 R/M</td>
<td>FICOM</td>
<td>Word-integer</td>
<td>COMPARE 2-BYTE EA INTEGER WITH ST</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 011 R/M</td>
<td>FICOMP</td>
<td>Word-integer</td>
<td>COMPARE 2-BYTE INTEGER EA WITH ST, POP</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 100 R/M</td>
<td>FISUB</td>
<td>Word-integer</td>
<td>SUBTRACT 2-BYTE INTEGER EA FROM ST</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 101 R/M</td>
<td>FISUBR</td>
<td>Word-integer</td>
<td>SUBTRACT ST FROM 2-BYTE INTEGER EA</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 MOD 110 R/M</td>
<td>FIDIV</td>
<td>Word-integer</td>
<td>DIVIDE ST BY 2-BYTE INTEGER EA</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 17: 8086/186 Instructions in Hexadecimal Order
### Chapter 17: 8086/186 Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Binary</th>
<th>MODRM Byte</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE 11011110</td>
<td>MOD 111 R/M</td>
<td>FIDIVR</td>
<td>Word-integer</td>
<td>DIVIDE 2-BYTE INTEGER EA BY ST</td>
</tr>
<tr>
<td>DE 11011110 11000 (i)</td>
<td>MOD 000</td>
<td>FADDP</td>
<td>ST(i), ST</td>
<td>ADD ST TO ELEMENT</td>
</tr>
<tr>
<td>DE 11011110 11001 (i)</td>
<td>MOD 001</td>
<td>FMULP</td>
<td>ST(i), ST</td>
<td>MULTIPLY ST BY ELEMENT, POP</td>
</tr>
<tr>
<td>DE 11011110 11010 ---</td>
<td>MOD 010</td>
<td>*(5)</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 11011 000</td>
<td>MOD 011</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 11011 001</td>
<td>MOD 100</td>
<td>FCOMPP</td>
<td>ST(1), ST</td>
<td>COMPARE ST WITH ST(1), POP TWICE</td>
</tr>
<tr>
<td>DE 11011110 11011 01-</td>
<td>MOD 101</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DE 11011110 11100 (i)</td>
<td>MOD 000 R/M</td>
<td>FSUBRP</td>
<td>ST(i), ST</td>
<td>SUBTRACT ST FROM ELEMENT, POP</td>
</tr>
<tr>
<td>DE 11011110 11101 (i)</td>
<td>MOD 001 R/M</td>
<td>FSUBP</td>
<td>ST(i), ST</td>
<td>SUBTRACT ST(i) FROM ST, POP</td>
</tr>
<tr>
<td>DE 11011110 11110 (i)</td>
<td>MOD 010 R/M</td>
<td>FDIVRP</td>
<td>ST(i), ST</td>
<td>DIVIDE STACK ELEMENT BY ST, POP</td>
</tr>
<tr>
<td>DE 11011110 11111 (i)</td>
<td>MOD 011 R/M</td>
<td>FDIVP</td>
<td>ST(i), ST</td>
<td>DIVIDE ST BY STACK ELEMENT, POP</td>
</tr>
<tr>
<td>DF 11011111 MOD 000 R/M</td>
<td>MOD 001 R/M</td>
<td>FILD</td>
<td>Word-integer</td>
<td>CONVERT 2-BYTE EA AND PUSH ONTO STACK</td>
</tr>
<tr>
<td>DF 11011111 MOD 001 R/M</td>
<td>MOD 010 R/M</td>
<td>FIST</td>
<td>Word-integer</td>
<td>ROUND ST AND STORE IN 2-BYTE INTEGER EA</td>
</tr>
<tr>
<td>DF 11011111 MOD 010 R/M</td>
<td>MOD 011 R/M</td>
<td>FISTP</td>
<td>Word-integer</td>
<td>ROUND ST, STORE IN 2-BYTE INTEGER EA, POP</td>
</tr>
<tr>
<td>DF 11011111 MOD 100 R/M</td>
<td>MOD 010 R/M</td>
<td>FBLD</td>
<td>Packed decimal</td>
<td>LOAD BCD TO ST</td>
</tr>
<tr>
<td>DF 11011111 MOD 101 R/M</td>
<td>MOD 010 R/M</td>
<td>FILD</td>
<td>Long-integer</td>
<td>CONVERT 8-BYTE INTEGER EA AND PUSH ONTO STACK</td>
</tr>
<tr>
<td>DF 11011111 MOD 110 R/M</td>
<td>MOD 010 R/M</td>
<td>FBSTP</td>
<td>Packed decimal</td>
<td>CONVERT ST, STORE IN 10-BYTE BCD EA, POP</td>
</tr>
<tr>
<td>DF 11011111 MOD 111 R/M</td>
<td>MOD 010 R/M</td>
<td>FISTP</td>
<td>Long-integer</td>
<td>ROUND ST, STORE IN 8-BYTE INTEGER EA, POP</td>
</tr>
<tr>
<td>DF 11011111 11000 (i)</td>
<td>MOD 010 R/M</td>
<td>*(6)</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>DF 1101111</td>
<td>1 1 001 (i)</td>
<td>*(7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF 1101111</td>
<td>1 1 010 (i)</td>
<td>*(8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF 1101111</td>
<td>1 1 011 (i)</td>
<td>*(9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF 1101111</td>
<td>1 1 --- ---</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0 1111000</td>
<td>LOOPNZ/</td>
<td>DISP8</td>
<td></td>
<td>LOOP (CX) TIMES WHILE NOT ZERO/NOT EQUAL</td>
</tr>
<tr>
<td></td>
<td>LOOPNE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1 1110001</td>
<td>LOOPZ/</td>
<td>DISP8</td>
<td></td>
<td>LOOP (CX) TIMES WHILE ZERO/EQUAL</td>
</tr>
<tr>
<td></td>
<td>LOOPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2 1110010</td>
<td>LOOP</td>
<td>DISP8</td>
<td></td>
<td>LOOP (CX) TIMES</td>
</tr>
<tr>
<td>E3 1110011</td>
<td>J CXZ</td>
<td>DISP8</td>
<td></td>
<td>JUMP ON (CX)= 0</td>
</tr>
<tr>
<td>E4 1110010</td>
<td>IN</td>
<td>AL,PORT</td>
<td></td>
<td>BYTE INPUT FROM PORT TO REG AL</td>
</tr>
<tr>
<td>E5 1110010</td>
<td>IN</td>
<td>AX,PORT</td>
<td></td>
<td>WORD INPUT FROM PORT TO REG AX</td>
</tr>
<tr>
<td>E6 1110010</td>
<td>OUT</td>
<td>PORT,AL</td>
<td></td>
<td>BYTE OUTPUT (AL) TO PORT</td>
</tr>
<tr>
<td>E7 1110011</td>
<td>OUT</td>
<td>PORT,AX</td>
<td></td>
<td>WORD OUTPUT (AX) TO PORT</td>
</tr>
<tr>
<td>E8 1110100</td>
<td>CALL</td>
<td>DISP16</td>
<td></td>
<td>DIRECT INTRA SEGMENT CALL</td>
</tr>
<tr>
<td>E9 1110101</td>
<td>JMP</td>
<td>DISP16</td>
<td></td>
<td>DIRECT INTRA SEGMENT JUMP</td>
</tr>
<tr>
<td>EA 1110101</td>
<td>JMP</td>
<td>DISP16,SEG16</td>
<td></td>
<td>DIRECT INTER SEGMENT JUMP</td>
</tr>
<tr>
<td>EB 1110101</td>
<td>JMP</td>
<td>DISP8</td>
<td></td>
<td>DIRECT INTRA SEGMENT JUMP</td>
</tr>
<tr>
<td>EC 1110101</td>
<td>IN</td>
<td>AL,DX</td>
<td></td>
<td>BYTE INPUT FROM PORT (DX) TO REG AL</td>
</tr>
<tr>
<td>ED 1110101</td>
<td>IN</td>
<td>AX,DX</td>
<td></td>
<td>WORD INPUT FROM PORT (DX) TO REG AX</td>
</tr>
<tr>
<td>EE 1110101</td>
<td>OUT</td>
<td>DX,AL</td>
<td></td>
<td>BYTE OUTPUT (AL) TO PORT (DX)</td>
</tr>
<tr>
<td>EF 1110101</td>
<td>OUT</td>
<td>DX,AX</td>
<td></td>
<td>WORD OUTPUT (AX) TO PORT (DX)</td>
</tr>
<tr>
<td>F0 11110000</td>
<td>LOCK</td>
<td></td>
<td></td>
<td>BUS LOCK PREFIX</td>
</tr>
<tr>
<td>Hex Binary</td>
<td>MODRM Byte</td>
<td>Instruction</td>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>F1 11110001</td>
<td>(not used)</td>
<td>(not used)</td>
<td>(not used)</td>
<td>(not used)</td>
</tr>
<tr>
<td>F2 11110010</td>
<td>MOD 000 R/M</td>
<td>TEST</td>
<td>EA,DATA8</td>
<td>BYTE TEST (EA) WITH DATA</td>
</tr>
<tr>
<td>F3 11110011</td>
<td>MOD 010 R/M</td>
<td>NOT</td>
<td>EA</td>
<td>BYTE INVERT EA</td>
</tr>
<tr>
<td>F4 11110100</td>
<td>MOD 011 R/M</td>
<td>NEG</td>
<td>EA</td>
<td>BYTE NEGATE EA</td>
</tr>
<tr>
<td>F5 11110101</td>
<td>MOD 100 R/M</td>
<td>MUL</td>
<td>EA</td>
<td>BYTE MULTIPLY BY (EA), UNSIGNED</td>
</tr>
<tr>
<td>F6 11110110</td>
<td>MOD 101 R/M</td>
<td>IMUL</td>
<td>EA</td>
<td>BYTE MULTIPLY BY (EA), SIGNED</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 000 R/M</td>
<td>TEST</td>
<td>EA,DATA16</td>
<td>WORD TEST (EA) WITH DATA</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 001 R/M</td>
<td>(not used)</td>
<td>(not used)</td>
<td>(not used)</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 010 R/M</td>
<td>NOT</td>
<td>EA</td>
<td>WORD INVERT EA</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 011 R/M</td>
<td>NEG</td>
<td>EA</td>
<td>WORD NEGATE EA</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 100 R/M</td>
<td>MUL</td>
<td>EA</td>
<td>WORD MULTIPLY BY (EA), UNSIGNED</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 101 R/M</td>
<td>IMUL</td>
<td>EA</td>
<td>WORD MULTIPLY BY (EA), SIGNED</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 110 R/M</td>
<td>DIV</td>
<td>EA</td>
<td>WORD DIVIDE BY (EA), UNSIGNED</td>
</tr>
<tr>
<td>F7 11110111</td>
<td>MOD 111 R/M</td>
<td>IDIV</td>
<td>EA</td>
<td>WORD DIVIDE BY (EA), SIGNED</td>
</tr>
</tbody>
</table>
### Hex Binary | MODRM Byte | Instruction | Parameters | Function
--- | --- | --- | --- | ---
F8 11111000 |  | CLC |  | CLEAR CARRY FLAG
F9 11111001 |  | STC |  | SET CARRY FLAG
FA 11111010 |  | CLI |  | CLEAR INTERRUPT FLAG
FB 11111011 |  | STI |  | SET INTERRUPT FLAG
FC 11111100 |  | CLD |  | CLEAR DIRECTION FLAG
FD 11111101 |  | STD |  | SET DIRECTION FLAG
FE 11111110 | MOD 000 R/M | INC | EA | BYTE INCREMENT EA
FE 11111110 | MOD 001 R/M | DEC | EA | BYTE DECREMENT EA
FE 11111110 | MOD 010 R/M | (not used) |  | 
FE 11111110 | MOD 011 R/M | (not used) |  | 
FE 11111110 | MOD 100 R/M | (not used) |  | 
FE 11111110 | MOD 101 R/M | (not used) |  | 
FE 11111110 | MOD 110 R/M | (not used) |  | 
FE 11111110 | MOD 111 R/M | (not used) |  | 
FF 11111111 | MOD 000 R/M | INC | EA | WORD INCREMENT EA
FF 11111111 | MOD 001 R/M | DEC | EA | WORD DECREMENT EA
FF 11111111 | MOD 001 R/M | CALL | EA | INDIRECT INTRA SEGMENT CALL
FF 11111111 | MOD 011 R/M | CALL | EA | INDIRECT INTER SEGMENT CALL
FF 11111111 | MOD 100 R/M | JMP | EA | INDIRECT INTRA SEGMENT JUMP
FF 11111111 | MOD 101 R/M | JMP | EA | INDIRECT INTER SEGMENT JUMP
FF 11111111 | MOD 110 R/M | PUSH | EA | PUSH (EA) ON STACK
FF 11111111 | MOD 111 R/M | (not used) |  | 

Chapter 17: 8086/186 Instructions in Hexadecimal Order

---

330
FLAGS REGISTER CONTAINS:


*The marked encodings are not generated by the language translators. If however, the 8087 encounters one of these encodings in the instruction stream, it will execute it as follows:

1. FSTP ST(i)
2. FCOM ST(i)
3. FCOMP ST(i)
4. FXCH ST(i)
5. FCOMP ST(i)
6. FFREE ST(i) and pop stack
7. FXCH ST(i)
8. FSTP ST(i)
9. FSTP ST(i)

IAPX 86/88/186 Instruction Set Matrix

b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed.to accum.
ib = immediate byte
id = indirect
is = immed. byte sign ext.
iw = immediate word
l = long ie. intersegment
m = memory
r = register
r/m = EA is second byte
SI = short intrasegment
sr = segment register
t = to CPU reg
v = variable
w = word operation
z = zero
### REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE.

<table>
<thead>
<tr>
<th>16-BIT (W= 1)</th>
<th>8-BIT (W= 0)</th>
<th>SEGMENT REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 AX</td>
<td>000 AL</td>
<td>00 ES</td>
</tr>
<tr>
<td>001 CX</td>
<td>001 CL</td>
<td>01 CS</td>
</tr>
<tr>
<td>010 DX</td>
<td>010 DL</td>
<td>10 SS</td>
</tr>
<tr>
<td>011 BX</td>
<td>011 BL</td>
<td>11 DS</td>
</tr>
<tr>
<td>100 SP</td>
<td>100 AH</td>
<td></td>
</tr>
<tr>
<td>101 BP</td>
<td>101 CH</td>
<td></td>
</tr>
<tr>
<td>110 SI</td>
<td>110 DH</td>
<td></td>
</tr>
<tr>
<td>111 DI</td>
<td>111 BH</td>
<td></td>
</tr>
</tbody>
</table>

### EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)

<table>
<thead>
<tr>
<th>EA Expression</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 000 (BX) + (SI)</td>
<td>DS</td>
</tr>
<tr>
<td>00 001 (BX) + (DI)</td>
<td>DS</td>
</tr>
<tr>
<td>00 010 (BP) + (SI)</td>
<td>SS</td>
</tr>
<tr>
<td>00 011 (BP) + (DI)</td>
<td>SS</td>
</tr>
<tr>
<td>00 100 (SI)</td>
<td>DS</td>
</tr>
<tr>
<td>00 101 (DI)</td>
<td>DS</td>
</tr>
<tr>
<td>00 110 DISP16 (DIRECT ADDRESS)</td>
<td>DS</td>
</tr>
<tr>
<td>00 111 (BX)</td>
<td>DS</td>
</tr>
<tr>
<td>01 000 (BX) + (SI) + DISP8</td>
<td>DS</td>
</tr>
<tr>
<td>01 001 (BX) + (DI) + DISP8</td>
<td>DS</td>
</tr>
<tr>
<td>01 010 (BP) + (SI) + DISP8</td>
<td>SS</td>
</tr>
<tr>
<td>01 011 (BP) + (DI) + DISP8</td>
<td>SS</td>
</tr>
<tr>
<td>01 100 (SI) + DISP8</td>
<td>DS</td>
</tr>
<tr>
<td>01 101 (DI) + DISP8</td>
<td>DS</td>
</tr>
<tr>
<td>01 110 (BP) + DISP8</td>
<td>SS</td>
</tr>
<tr>
<td>01 111 (BX) + DISP8</td>
<td>DS</td>
</tr>
<tr>
<td>10 000 (BX) + (SI) + DISP16</td>
<td>DS</td>
</tr>
</tbody>
</table>
EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS) (Cont’d)

<table>
<thead>
<tr>
<th>Address</th>
<th>DS</th>
<th>SS</th>
<th>DS</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 001 (BX) + (DI) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 010 (BP) + (SI) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 011 (BP) + (DI) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 100 (SI) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 101 (DI) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 110 (BP) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 111 (BX) + DISP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 000 REG AX / AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 001 REG CX / CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 010 REG DX / DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 011 REG BX / BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 100 REG SP / AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 101 REG BP / CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 110 REG SI / DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 111 REG DI / BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# IAPX 86/186 Instruction Set Matrix

<table>
<thead>
<tr>
<th>Hi</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD b.f.r/m</td>
<td>ADD w.f.r/m</td>
<td>ADD b.t.r/m</td>
<td>ADD w.t.r/m</td>
<td>ADD b.ia</td>
<td>ADD w.ia</td>
<td>PUSH ES</td>
<td>POP ES</td>
</tr>
<tr>
<td>1</td>
<td>ADC b.f.r/m</td>
<td>ADC w.f.r/m</td>
<td>ADC b.t.r/m</td>
<td>ADC w.t.r/m</td>
<td>ADC b.ia</td>
<td>ADC w.ia</td>
<td>PUSH SS</td>
<td>POP SS</td>
</tr>
<tr>
<td>2</td>
<td>AND b.f.r/m</td>
<td>AND w.f.r/m</td>
<td>AND b.t.r/m</td>
<td>AND w.t.r/m</td>
<td>AND b.ia</td>
<td>AND w.ia</td>
<td>SEG ES</td>
<td>DAA</td>
</tr>
<tr>
<td>3</td>
<td>XOR b.f.r/m</td>
<td>XOR w.f.r/m</td>
<td>XOR b.t.r/m</td>
<td>XOR w.t.r/m</td>
<td>XOR b.ia</td>
<td>XOR w.ia</td>
<td>SEG SS</td>
<td>AAA</td>
</tr>
<tr>
<td>4</td>
<td>INC AX</td>
<td>INC CX</td>
<td>INC DX</td>
<td>INC BX</td>
<td>INC SP</td>
<td>INC BP</td>
<td>INC SI</td>
<td>INC DI</td>
</tr>
<tr>
<td>5</td>
<td>PUSH AX</td>
<td>PUSH CX</td>
<td>PUSH DX</td>
<td>PUSH BX</td>
<td>PUSH SP</td>
<td>PUSH BP</td>
<td>PUSH SI</td>
<td>PUSH DI</td>
</tr>
<tr>
<td>6</td>
<td>PUSHA</td>
<td>POPA</td>
<td>BOUND R.R/M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>JO</td>
<td>JNO</td>
<td>JB/ JNAE</td>
<td>JNB/ JAE</td>
<td>JE/ JZ</td>
<td>JNE/ JNZ</td>
<td>JBE/ JNA</td>
<td>JNBE/ JA</td>
</tr>
<tr>
<td>8</td>
<td>Immed b.r/m</td>
<td>Immed w.r/m</td>
<td>Immed b.r/m</td>
<td>Immed is.r/m</td>
<td>TEST b.r/m</td>
<td>TEST w.r/m</td>
<td>XCHG b.r/m</td>
<td>XCHG w.r/m</td>
</tr>
<tr>
<td>9</td>
<td>NOP</td>
<td>XCHG CX</td>
<td>XCHG DX</td>
<td>XCHG BX</td>
<td>XCHG SP</td>
<td>XCHG BP</td>
<td>XCHG SI</td>
<td>XCHG DI</td>
</tr>
<tr>
<td>A</td>
<td>MOV m → AL</td>
<td>MOV m → AX</td>
<td>MOV AL → m</td>
<td>MOV AX → m</td>
<td>MOVS b</td>
<td>MOVS w</td>
<td>CMPS b</td>
<td>CMPS w</td>
</tr>
<tr>
<td>B</td>
<td>MOV i → AL</td>
<td>MOV i → CL</td>
<td>MOV i → DL</td>
<td>MOV i → BL</td>
<td>MOV i → AH</td>
<td>MOV i → C</td>
<td>MOV i → DH</td>
<td>MOV i → BH</td>
</tr>
</tbody>
</table>
### IAPX 86/186 Instruction Set Matrix

#### LO

<table>
<thead>
<tr>
<th>Hi</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Shift b.r/m.i</td>
<td>Shift w.r/mi</td>
<td>IRET (i - SP)</td>
<td>IRET</td>
<td>LES</td>
<td>LDS</td>
<td>MOV b.i.r/m</td>
<td>MOV w.i.r/m</td>
</tr>
<tr>
<td>D</td>
<td>Shift b</td>
<td>Shift w</td>
<td>Shift b.v</td>
<td>Shift w.v</td>
<td>AAM</td>
<td>AAD</td>
<td>XLAT</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>LOOPNZ/LOOPNE</td>
<td>LOOPZ/LOOPE</td>
<td>LOOP</td>
<td>JCXZ</td>
<td>IN b</td>
<td>IN w</td>
<td>OUT b</td>
<td>OUT w</td>
</tr>
<tr>
<td>F</td>
<td>LOCK</td>
<td>REP</td>
<td>REPZ</td>
<td>HLT</td>
<td>CMC</td>
<td>Grp 1 b.r/m</td>
<td>Grp 1 w.r/m</td>
<td></td>
</tr>
</tbody>
</table>

#### LO

<table>
<thead>
<tr>
<th>Hi</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OR b.f.r/m</td>
<td>OR w.f.r/m</td>
<td>OR b.t.r/m</td>
<td>OR w.t.r/m</td>
<td>OR b.i.a</td>
<td>OR w.i.a</td>
<td>PUSH CS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SBB b.f.r/m</td>
<td>SBB w.f.r/m</td>
<td>SBB b.t.r/m</td>
<td>SBB w.t.r/m</td>
<td>SBB b.i.a</td>
<td>SBB w.i.a</td>
<td>PUSH DS</td>
<td>POP DS</td>
</tr>
<tr>
<td>2</td>
<td>SUB b.f.r/m</td>
<td>SUB w.f.r/m</td>
<td>SUB b.t.r/m</td>
<td>SUB w.t.r/m</td>
<td>SUB b.i.a</td>
<td>SUB w.i.a</td>
<td>SEG CS</td>
<td>DAS</td>
</tr>
<tr>
<td>3</td>
<td>CMP b.f.r/m</td>
<td>CMP w.f.r/m</td>
<td>CMP b.t.r/m</td>
<td>CMP w.t.r/m</td>
<td>CMP b.i.a</td>
<td>CMP w.i.a</td>
<td>SEG DS</td>
<td>AAS</td>
</tr>
<tr>
<td>4</td>
<td>DEC AX</td>
<td>DEC CX</td>
<td>DEC DX</td>
<td>DEC BX</td>
<td>DEC SP</td>
<td>DEC BP</td>
<td>DEC SI</td>
<td>DEC DI</td>
</tr>
</tbody>
</table>
## IAPX 86/186 Instruction Set Matrix

<table>
<thead>
<tr>
<th>LO</th>
<th>Hi</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
<td>POP</td>
</tr>
<tr>
<td></td>
<td>AX</td>
<td>CX</td>
<td>DX</td>
<td>BX</td>
<td>SP</td>
<td>BP</td>
<td>SI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PUSH</td>
<td>IMUL</td>
<td>PUSH</td>
<td>IMUL</td>
<td>INS</td>
<td>INS</td>
<td>OUTS</td>
<td>OUTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iw</td>
<td>r.iw.r/m</td>
<td>is</td>
<td>r.is.r/m</td>
<td>b</td>
<td>w</td>
<td>b</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>JS</td>
<td>JNS</td>
<td>JP/ JPE</td>
<td>JNP/ JPO</td>
<td>JL/ JNGE</td>
<td>JNL/ JGE</td>
<td>JLE/ JNG</td>
<td>JNLE/ JG</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>LEA</td>
<td>MOV</td>
<td>POP</td>
<td>r/m</td>
</tr>
<tr>
<td></td>
<td>b.f.r/m</td>
<td>w.f.r/m</td>
<td>b.t.r/m</td>
<td>w.t.r/m</td>
<td>sr.f.r/m</td>
<td>sr.t.r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CBW</td>
<td>CWD</td>
<td>CALL</td>
<td>WAIT</td>
<td>PUSHF</td>
<td>POPF</td>
<td>SAHF</td>
<td>LAHF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL</td>
<td>d</td>
<td>i.d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TEST</td>
<td>TEST</td>
<td>STOS</td>
<td>STOS</td>
<td>LODS</td>
<td>LODS</td>
<td>SCAS</td>
<td>SCAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b.i</td>
<td>w.i</td>
<td>b</td>
<td>w</td>
<td>b</td>
<td>w</td>
<td>b</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td></td>
<td>i → AX</td>
<td>i → CX</td>
<td>i → DX</td>
<td>i → BX</td>
<td>i → SP</td>
<td>i → BP</td>
<td>i → SI</td>
<td>i → DI</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ENTER</td>
<td>LEAVE</td>
<td>IRET</td>
<td>IRET</td>
<td>INT</td>
<td>Type 3</td>
<td>INT</td>
<td>INTO</td>
<td>IRET</td>
</tr>
<tr>
<td></td>
<td>i.w.ib</td>
<td></td>
<td>I. (i - SP)</td>
<td>I</td>
<td></td>
<td>(Any)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
<td>ESC</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CALL</td>
<td>JMP</td>
<td>JMP</td>
<td>JMP</td>
<td>IN</td>
<td>IN</td>
<td>OUT</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>d</td>
<td>i.d</td>
<td>si.d</td>
<td>v.b</td>
<td>v.w</td>
<td>v.d</td>
<td>v.w</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>CLC</td>
<td>STC</td>
<td>CLI</td>
<td>STI</td>
<td>CLD</td>
<td>STD</td>
<td>Grp 2</td>
<td>Grp 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w.r/m</td>
<td></td>
<td></td>
<td></td>
<td>b.r/m</td>
<td>w.r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Chapter 17: 8086/186 Instructions in Hexadecimal Order
## Where

<table>
<thead>
<tr>
<th>mod r/m</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immed</td>
<td>ADD</td>
<td>OR</td>
<td>ADC</td>
<td>SBB</td>
<td>AND</td>
<td>SUB</td>
<td>XOR</td>
<td>CMP</td>
</tr>
<tr>
<td>Shift</td>
<td>ROL</td>
<td>ROR</td>
<td>RCL</td>
<td>RCR</td>
<td>SHL/SAL</td>
<td>SHR</td>
<td>SHL/SAL</td>
<td>SAR</td>
</tr>
<tr>
<td>Grp 1</td>
<td>TEST</td>
<td>NOT</td>
<td>NEG</td>
<td>MUL</td>
<td>IMUL</td>
<td>DIV</td>
<td>IDIV</td>
<td></td>
</tr>
<tr>
<td>Grp 2</td>
<td>INC</td>
<td>DEC</td>
<td>CALL id</td>
<td>CALL id</td>
<td>JMP id</td>
<td>JMP id</td>
<td>PUSH</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 17: 8086/186 Instructions in Hexadecimal Order
Chapter 18: 8086/186 Instruction Set Summary

8086/186 Instruction Set Summary
### Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA TRANSFER</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVE = Move:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register/Memory</td>
<td>1000100w mod reg r/m</td>
<td>2/12</td>
<td></td>
</tr>
<tr>
<td>Register/Memory to register</td>
<td>1000101w mod reg r/m</td>
<td>2/9</td>
<td></td>
</tr>
<tr>
<td>Immediate to register/memory</td>
<td>1100011w mod 000 r/m</td>
<td>3-4</td>
<td>8/16-bit</td>
</tr>
<tr>
<td>Immediate to register</td>
<td>1011w reg data data if w= 1</td>
<td>12-13</td>
<td>8/16-bit</td>
</tr>
<tr>
<td>Memory to accumulator</td>
<td>1010000w addr-low addr-high</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Accumulator to memory</td>
<td>1010001w addr-low addr-high</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Register/memory to segment register</td>
<td>10001110 mod 0reg/r/m</td>
<td>2/9</td>
<td></td>
</tr>
<tr>
<td>Segment/register to register/memory</td>
<td>10001100 mod 0reg/r/m</td>
<td>2/11</td>
<td></td>
</tr>
</tbody>
</table>
### Function Formal Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSH = Push:</strong></td>
<td>11111111 mod 110</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>01010 reg</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Segment register</td>
<td>000 reg 110</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>*Immediate</td>
<td>011010s0 data data if ( s = 010 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*PUSHA = Push All</td>
<td>01100000</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td><strong>POP = Pop:</strong></td>
<td>10001111 mod 000</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>01011 reg</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Segment register</td>
<td>000 reg 111</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>*POPA = Pop All</td>
<td>01100001</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td><strong>XCHG = Exchange:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/memory</td>
<td>10000111w mod reg</td>
<td>4/17</td>
<td></td>
</tr>
<tr>
<td>with register</td>
<td>r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>10010 reg</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>with accumulator</td>
<td>1110010w port</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>IN = Input from:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed port</td>
<td>1110010w port</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable port</td>
<td>1110110w</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUT = Output to:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed port</td>
<td>1110011w port</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Variable port</td>
<td>1110111w</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>XLAT = translate byte to AL</td>
<td>11010111</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LEA = Load EA to register</td>
<td>10001101 mod reg r/m</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>LDS = Load pointer to DS</td>
<td>11000101 mod reg r/m (mod ≠ 11)</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>LES = Load pointer to ES</td>
<td>11000100 mod reg r/m (mod ≠ 11)</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>LAHF = Load AH with flags</td>
<td>10011111</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SAHF = Store AH into flags</td>
<td>10011110</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSHF = Push flags</td>
<td>10011100</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>POPF = Pop flags</td>
<td>10011101</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Formal</td>
<td>186 Clock Cycles</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>ARITHMETIC ADD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>= Add:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg/memory with</td>
<td>000000dw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register to</td>
<td>mod reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>either</td>
<td>r/m</td>
<td></td>
<td>3/10</td>
</tr>
<tr>
<td>Immediate to</td>
<td>10000sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register/memory</td>
<td>mod 000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data</td>
<td></td>
<td>4/16</td>
</tr>
<tr>
<td>Immediate to</td>
<td>000010w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>accumulator</td>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data if w - 1</td>
<td></td>
<td>3/4</td>
</tr>
<tr>
<td>ADC = ADD with</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>carry:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg/memory with</td>
<td>000100dw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register to</td>
<td>mod reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>either</td>
<td>r/m</td>
<td></td>
<td>3/10</td>
</tr>
<tr>
<td>Immediate to</td>
<td>10000sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register/memory</td>
<td>mod 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data</td>
<td></td>
<td>4/16</td>
</tr>
<tr>
<td>Immediate to</td>
<td>0001010w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>accumulator</td>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data if w - 1</td>
<td></td>
<td>3/4</td>
</tr>
<tr>
<td>INC = Increment:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/memory</td>
<td>111111w</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mod 000</td>
<td></td>
<td>3/15</td>
</tr>
<tr>
<td>Register</td>
<td>01000 reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUB = Subtract:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg/memory and register to either</td>
<td>001010dw mod reg r/m</td>
<td>3/10</td>
<td></td>
</tr>
<tr>
<td>Immediate from register/memory</td>
<td>100000sw mod 101 data r/m</td>
<td>4/16</td>
<td>data if s w 01</td>
</tr>
<tr>
<td>Immediate from accumulator</td>
<td>0010110w data</td>
<td>3/4</td>
<td>8/16-bit</td>
</tr>
<tr>
<td><strong>SBB = Subtract with borrow:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg/memory and register to either</td>
<td>000110dw mod reg r/m</td>
<td>3/10</td>
<td></td>
</tr>
<tr>
<td>Immediate from register/memory</td>
<td>100000sw mod 011 data r/m</td>
<td>4/16</td>
<td>data if s w - 01</td>
</tr>
<tr>
<td>Immediate from accumulator</td>
<td>0001110w data</td>
<td>3/4</td>
<td>8/16-bit</td>
</tr>
<tr>
<td><strong>DEC = Decrement:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/memory</td>
<td>1111111w mod 001 r/m</td>
<td>3/15</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>01001 reg</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
# Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP = Compare:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/memory</td>
<td>0011101w mod reg r/m</td>
<td>3/10</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>0011100w mod reg r/m</td>
<td>3/10</td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>100000sw mod 111 r/m</td>
<td>3/10</td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>0011110w data r/m</td>
<td>3/4 8/16-bit</td>
<td></td>
</tr>
<tr>
<td><strong>NEG = Change sign</strong></td>
<td>1111011w mod 011 r/m</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>AAA = ASCII adjust for add</strong></td>
<td>00110111</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>DAA = Decimal adjust for add</strong></td>
<td>00100111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>AAS = ASCII adjust for subtract</strong></td>
<td>00111111</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td><strong>DAS = Decimal adjust for subtract</strong></td>
<td>00101111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>MUL = Multiply (unsigned):</strong></td>
<td>1111011w mod 100 r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Byte</td>
<td>26-28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Word</td>
<td>35-37</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory-Byte</td>
<td></td>
<td>32-34</td>
<td></td>
</tr>
<tr>
<td>Memory-Word</td>
<td></td>
<td>41-43</td>
<td></td>
</tr>
<tr>
<td><strong>IMUL = Integer multiply (signed):</strong></td>
<td>1111011w mod 101 r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Byte</td>
<td></td>
<td>25-28</td>
<td></td>
</tr>
<tr>
<td>Register-Word</td>
<td></td>
<td>34-37</td>
<td></td>
</tr>
<tr>
<td>Memory-Byte</td>
<td></td>
<td>31-34</td>
<td></td>
</tr>
<tr>
<td>Memory-Word</td>
<td></td>
<td>40-43</td>
<td></td>
</tr>
<tr>
<td>*IMUL = Integer immediate multiply (signed)</td>
<td>011010s1 mod reg data data if s= 0 r/m</td>
<td>22-25/29-32</td>
<td></td>
</tr>
<tr>
<td><strong>DIV = Divide (unsigned):</strong></td>
<td>1111011w mod 110 r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Byte</td>
<td></td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Register-Word</td>
<td></td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>Memory-Byte</td>
<td></td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Memory-Word</td>
<td></td>
<td>44</td>
<td></td>
</tr>
<tr>
<td><strong>IDIV = Integer divide (signed):</strong></td>
<td>1111011w mod 111 r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Formal</td>
<td>186 Clock Cycles</td>
<td>Comments</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Register-Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAM = ASCII adjust for multiply</td>
<td>11010100 00001010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAD = ASCII adjust for divide</td>
<td>11010101 00001010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBW = Convert byte to word</td>
<td>10011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CWD = Convert word to double word</td>
<td>10011001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LOGIC**  
**Shift/Rotate Instructions:**

<table>
<thead>
<tr>
<th>Register/Memory by 1</th>
<th>1101000w mod TTT r/m</th>
<th>2/15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register/Memory by CL</td>
<td>1101001w mod TTT r/m</td>
<td>5+ m/17+ n</td>
</tr>
<tr>
<td>*Register/Memory by Count</td>
<td>1100000w mod TTT r/m count</td>
<td>5+ n/17+ n</td>
</tr>
<tr>
<td>Function</td>
<td>Formal</td>
<td>186 Clock Cycles</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>TTT Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 ROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 ROR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 RCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 RCR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 SHL/SAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 SHR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 SAR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AND = And:**

- **Reg/memory and register to either**: 001000dw mod reg r/m
- **Immediate to register/memory**: 100000w mod 100 r/m data data if w= 1
- **Immediate to accumulator**: 0010010w data data if w= 1

8/16-bit

**TEST = And function to flags, no result:**

- **Register/memory and register**: 1000010w mod reg r/m
- **Immediate data and register/memory**: 1111011w mod 000 r/m data data if w= 1
- **Immediate data and accumulator**: 1010100w data data if w= 1

8/16-bit
<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR = Or:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg/memory and register to either</td>
<td>000010dw mod reg r/m</td>
<td>3/10</td>
</tr>
<tr>
<td></td>
<td>Immediate to register/memory</td>
<td>1000000w mod 001 r/m data</td>
<td>4/16</td>
</tr>
<tr>
<td></td>
<td>Immediate to accumulator</td>
<td>0000110w data data if w= 1</td>
<td>3/4</td>
</tr>
<tr>
<td>XOR = Exclusive or:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg/memory and register to either</td>
<td>001100dw mod reg r/m</td>
<td>3/10</td>
</tr>
<tr>
<td></td>
<td>Immediate to register/memory</td>
<td>1000000w mod 110 r/m data</td>
<td>4/16</td>
</tr>
<tr>
<td></td>
<td>Immediate to accumulator</td>
<td>0011010w data data if w= 1</td>
<td>3/4</td>
</tr>
<tr>
<td></td>
<td>NOT = Invert register/memory</td>
<td>1111011w mod 010 r/m</td>
<td>3</td>
</tr>
<tr>
<td>STRING MANIPULATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV $\text{S} = \text{Move byte/word}$</td>
<td>1010010w</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>CMPS $\text{S} = \text{Compare byte/word}$</td>
<td>1010011w</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAS = Scan byte/word</td>
<td>1010111w</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>LODS = Load byte/word to AL/A</td>
<td>1010110w</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>STOS = Store byte/word from AL/A</td>
<td>1010101w</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>*INS = Input byte/word from DX port</td>
<td>0110110w</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>*OUTS = Output byte/word to DX port</td>
<td>0110111w</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Repeated by count in CX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVS - Move string</td>
<td>11110010 1010010w</td>
<td>8 + 8n</td>
<td></td>
</tr>
<tr>
<td>CMPS - Compare string</td>
<td>1111001z 11010011w</td>
<td>5 + 22n</td>
<td></td>
</tr>
<tr>
<td>SCAS - Scan string</td>
<td>1111001z 11010111w</td>
<td>5 + 15n</td>
<td></td>
</tr>
<tr>
<td>LODS - Load string</td>
<td>11110010 1010110w</td>
<td>6 + 11n</td>
<td></td>
</tr>
<tr>
<td>STOS - Store string</td>
<td>11110010 1010101w</td>
<td>6 + 9n</td>
<td></td>
</tr>
<tr>
<td>*INS = Input string</td>
<td>11110010 0110110w</td>
<td>8 + 8n</td>
<td></td>
</tr>
<tr>
<td>*OUTS = Output string</td>
<td>11110010 0110111w</td>
<td>8 + 8n</td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONTROL TRANSFER</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL = Call:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct within segment</td>
<td>11101000</td>
<td>disp-low</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>disp--high</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register memory</td>
<td>11111111</td>
<td>mod 010</td>
<td>13/19</td>
</tr>
<tr>
<td>indirect within sement</td>
<td>r/m 13/19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct intersegment</td>
<td>10011010</td>
<td>segment offset</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>segment selector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect intersegment</td>
<td>11111111</td>
<td>mod 011</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>(mod ≠ 11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r/m 11/17</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>JMP = Unconditional jump:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short long</td>
<td>11101011</td>
<td>disp-low</td>
<td>13</td>
</tr>
<tr>
<td>Direct within segment</td>
<td>11101001</td>
<td>disp-low</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>disp--high</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/memory</td>
<td>11111111</td>
<td>mod 100</td>
<td>11/17</td>
</tr>
<tr>
<td>indirect within segment</td>
<td>r/m 11/17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct intersegment</td>
<td>11101010</td>
<td>segment offset</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>segment selector</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect intersegment</td>
<td>11111111 mod 101 (r/m) ((\text{mod} \neq 11))</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>RET = Return from CALL:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Within segment</td>
<td>11000011</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Within seg adding immed to SP</td>
<td>11000010 data-low data-high</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersegment</td>
<td>11001011</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Intersegment adding immediate to SP</td>
<td>11001010 data-low data-high</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>JE/JZ = Jump on equal zero</td>
<td>01110100 disp</td>
<td>4/13</td>
<td>13 if JMP taken</td>
</tr>
<tr>
<td>JL/JNGE = Jump on less not greater or equal</td>
<td>01111100 disp</td>
<td>4/13</td>
<td>4 if JMP not taken</td>
</tr>
<tr>
<td>JLE/JNG = Jump on less or equal not greater</td>
<td>01111110 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JB/JNAE = Jump on below not above or equal</td>
<td>01110010 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JBE/JNA = Jump on below or equal not above</td>
<td>01110110 disp</td>
<td>4/13</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP/JPE = Jump on parity even</td>
<td>01111010 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JO = Jump on overflow</td>
<td>01110000 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JS = Jump on sign</td>
<td>01111000 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNE/JNZ = Jump on not equal not zero</td>
<td>01110101 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNL/JGE = Jump on not less greater or equal</td>
<td>01111011 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNLE/JG = Jump on not less or equal greater</td>
<td>01111111 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNB/JAE = Jump on not below above or equal</td>
<td>01110011 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNBE/JA = Jump on not below or equal above</td>
<td>01110111 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNP/JPO = Jump on not parity odd</td>
<td>01111011 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNO = Jump on not overflow</td>
<td>01110001 disp</td>
<td>4/13</td>
<td></td>
</tr>
<tr>
<td>JNS = Jump on not sign</td>
<td>01111001 disp</td>
<td>4/13</td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP = Loop CX times</td>
<td>11100010 disp</td>
<td>5/15</td>
<td></td>
</tr>
<tr>
<td>LOOPZ/LOOPE = Loop</td>
<td>11100011 disp</td>
<td>6/16</td>
<td></td>
</tr>
<tr>
<td>Loop while zero equal</td>
<td>11100001 disp</td>
<td>6/16</td>
<td></td>
</tr>
<tr>
<td>LOOPNZ/LOOPNE = Loop</td>
<td>11100000 disp</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Loop while not zero</td>
<td>11100000 disp</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JCPZ = Jump on CX</td>
<td>11100011 disp</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>zero</td>
<td>11100011 disp</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>*ENTER = Enter Procedure</td>
<td>11001000 data-low data-high L</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>L = 0</td>
<td>11001001</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>L= 1</td>
<td>11001001</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>L&gt; &gt; 1</td>
<td>11001001</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LEAVE = Leave Procedure</td>
<td>11001001</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>INT = Interrupt:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type specified</td>
<td>11001101 type</td>
<td>47</td>
<td>if INT.taken/</td>
</tr>
<tr>
<td>Type 3</td>
<td>11001100</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTO = Interrupt on overflow</td>
<td>11001110</td>
<td>48/4</td>
<td>if INT.not taken</td>
</tr>
<tr>
<td>IRET = Interrupt return</td>
<td>11001111</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>*BOUND = Detect value out of range</td>
<td>01100010 mod reg r/m</td>
<td>33-35</td>
<td></td>
</tr>
</tbody>
</table>

### PROCESSOR CONTROL

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC = Clear carry</td>
<td>11110000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CMC = Complement carry</td>
<td>11110101</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STC = Set carry</td>
<td>11111001</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLD = Clear direction</td>
<td>11111100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STD = Set direction</td>
<td>11111101</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLI = Clear interrupt</td>
<td>11111010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STI = Set interrupt</td>
<td>11111011</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HLT = Halt</td>
<td>11110100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>WAIT = Wait</td>
<td>10011011</td>
<td>6</td>
<td>if test = 0</td>
</tr>
<tr>
<td>LOCK = Bus lock prefix</td>
<td>11110000</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 18: 8086/186 Instruction Set Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Formal</th>
<th>186 Clock Cycles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC = Processor Extension Escape</td>
<td>10011TTT mod LLL</td>
<td>6</td>
<td>(TTT LLL are opcode to processor extension)</td>
</tr>
</tbody>
</table>
Footnotes
The effective Address (EA) of the memory operand is computed according to the mod and r/m fields:

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

SEGMENT OVERRIDE PREFIX

001 reg 110

reg is assigned according to the following:

<table>
<thead>
<tr>
<th>reg</th>
<th>Segment Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
</tr>
<tr>
<td>REG</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
</tr>
</tbody>
</table>

REG is assigned according to the following table:

<table>
<thead>
<tr>
<th>16-Bit (w = 1)</th>
<th>8-Bit (w = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 AX</td>
<td>000 AL</td>
</tr>
<tr>
<td>001 CX</td>
<td>001 CL</td>
</tr>
<tr>
<td>010 DX</td>
<td>010 DL</td>
</tr>
<tr>
<td>011 BX</td>
<td>011 BL</td>
</tr>
<tr>
<td>100 SP</td>
<td>100 AH</td>
</tr>
<tr>
<td>101 BP</td>
<td>101 CH</td>
</tr>
<tr>
<td>110 SI</td>
<td>110 DH</td>
</tr>
<tr>
<td>111 DI</td>
<td>111 BH</td>
</tr>
</tbody>
</table>

The physical addresses of all operands addressed by the BP register are computed using the SS segment register. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden.
Chapter 19: Linker/Loader Introduction

19

Linker/Loader Introduction

Introduction to the linking loader.
The linking loader can be used to combine several independently assembled relocatable object modules into a single absolute object module.

When an absolute load is performed, relocatable addresses are transformed into absolute addresses, external references between modules are resolved, and the final absolute address value is substituted for each external symbol reference. The loader allows you to specify the program segment addresses, external definitions, and assign the final load address and segment loading order. Absolute output can be produced in either HP-OMF 86 format absolute (default), Intel Hexadecimal Object file format absolute, or HP 64000 format absolute.

**Note**

HP-OMF 86 format is the HP implementation of Intel Binary OMF. It contains certain extensions to facilitate analysis and emulation. HP-OMF 86 has not been verified to be strictly compatible with Intel Binary OMF and may not work correctly in tools or systems designed to consume Intel Binary OMF.

The linking loader can also be used to combine relocatable object modules into a single relocatable object module suitable for later re-linking with other modules. This option is known as incremental linking. Output is produced in HP-OMF 86 format relocatable. Certain loader commands cannot be used with this option: RESADD, RESNUM, GROUP, INITDATA, START and ORDER.

**Linking And Loading From Libraries**

The linking loader provides the ability to load object modules from a library. The ar86 Librarian is used to create such a library. The loader will include only those modules from a library that are necessary to resolve external references.

**Linking to the 8087**

This section applies to the code generated in 8086 and 80186 modes that is intended to run on the 8087 coprocessor.
Many target systems use an 8087 coprocessor to execute floating point instructions. However, it is also possible to use software emulation for these same floating point instructions. In fact, Intel has an emulator package that includes an emulation library for the 8087 (named en87.lib). Most floating point instructions must be modified at link time if the emulation library code is to be accessed instead of the 8087. To do this, the as86 assembler generates certain external references to invoke the correct emulation function in the software library.

If you are using an 8087, you must still resolve the external references so that the code is not changed. Unfortunately, the external names that are used contain a colon (':'). Because the assembler does not accept this character in user-defined symbols, the public symbols needed to resolve these externals cannot be created in the assembler. The loader, however, does accept the colon in loader symbols. You must define these publics in the loader command file with the PUBLIC command. This command should look like this:

```
PUBLIC M:_WST=0,M:_WT=0,M:_NST=0
PUBLIC M:_WCS=0,M:_WDS=0,M:_WES=0,M:_WSS=0
PUBLIC M:_NCS=0,M:_NDS=0,M:_NES=0,M:_NSS=0
```

This command will cause the created floating point code to work as is with the 8087 coprocessor.

**M:_WST, M:_WT, M:_NST, and other Floating Point Externals**

A brief explanation of these externals will help you to understand why they exist. As can be seen from the above linker command, there are eleven externals that can be generated.

- The "M:_WST" external is generated whenever a floating point instruction is used that has a 9BH byte at the start of its object code.
- The "M:_NST" external is generated for those floating point instructions that start with a 90H byte.
- The "M:_WT" instruction is used for the FWAIT floating point instruction. (The WAIT instruction does NOT get modified through the mechanism described here, so it should not be used if software emulation is desired).
• The "M:_WCS" external is generated whenever a floating point instruction is used that has a 9BH byte at the start of the instruction and a CS segment override byte is required for the memory operand.

• The "M:_NCS" external is generated whenever a floating point instruction is used that has a 90H byte at the start of the instruction and a CS segment override byte is required for the memory operand.

• The "M:_WDS" external is generated whenever a floating point instruction is used that has a 9BH byte at the start of the instruction and a DS segment override byte is required for the memory operand.

• The "M:_NDS" external is generated whenever a floating point instruction is used that has a 90H byte at the start of the instruction and a DS segment override byte is required for the memory operand.

• The "M:_WES" external is generated whenever a floating point instruction is used that has a 9BH byte at the start of the instruction and a ES segment override byte is required for the memory operand.

• The "M:_NES" external is generated whenever a floating point instruction is used that has a 90H byte at the start of the instruction and a ES segment override byte is required for the memory operand.

• The "M:_WSS" external is generated whenever a floating point instruction is used that has a 9BH byte at the start of the instruction and a SS segment override byte is required for the memory operand.

• The "M:_NSS" external is generated whenever a floating point instruction is used that has a 90H byte at the start of the instruction and a SS segment override byte is required for the memory operand.

When a floating point instruction is used that meets one of the above criteria, a fix up is placed in the relocatable file so the floating point instruction can be converted into an equivalent instruction for the 8087 emulation libraries. The fixes convert the floating point instructions to become interrupt instructions that interpret floating point code. If the libraries are not being used, then the public must have a value of 0 so the fix up does not modify the code.

If you are linking with the Intel emulation library, there is no need to define these publics. Instead, the publics will be defined within the library such that the floating point instructions are modified to work with the library. Modify all programs that use the emulation libraries to call the INIT87 Intel library.
routine. It is equally valid to use the FINIT instruction if the emulation libraries are not being used.
Linker/Loader Operation

Description of loader operation.
Chapter 20: Linker/Loader Operation

Primary Functions

This chapter briefly describes the function of the loader, reviews some concepts necessary to understanding how the loader functions, and explains some aspects of the loader operation—for instance, how the loader goes about locating object modules in memory when a load occurs. A later section of this chapter explains the load procedure and touches on issues the loader must consider when loading modules that are very different in makeup. This information will help you to better control and optimize the load process.

Primary Functions

Many programs are too long to be conveniently assembled as a single module. To avoid long assembly times or to reduce the size of the assembler symbol table, long programs can be subdivided into smaller modules, assembled separately, and then linked together by the linking loader. After the separate program modules are linked and loaded, the output module functions as if it had been generated by a single assembly.

The primary functions of the linking loader are:

- Resolve external references between modules and check for undefined references (link).
- Adjust all relocatable addresses to the proper absolute addresses (load).
- Place debug information in proper format for analysis tools.
- Produce as output the final absolute object module.

Incremental Linking

A powerful and useful feature of the ld86 is the incremental linking ability. Incremental linking means that the loader can produce a single relocatable object module from assembled relocatable object modules, and resolve all external references between the modules loaded. Undefined external references to other modules can still exist in the output object module. These are reported on the load map.
Incremental linking is useful for two reasons:

- First, it enables groups of users to easily share relocatable object modules for joint development of code.
- Long lists of previously-debugged object modules do not have to be linked with those modules currently under development. Instead, one incrementally-linked module can be created and linked with the new modules; it is therefore unnecessary to know all the original module names that are in the incrementally-linked module.

The ORDER, GROUP, INITDATA, START, RESADD, and RESNUM loader commands cannot be used when incremental linking.

Segments and Load Addresses

To effectively use the as86 Assembler and ld86 Linking Loader, you must understand the various program segments and segment load addresses. The terms summarized below are described more completely elsewhere in this manual.

Logical Segment

A logical segment is a programmer-defined division of the assembly program that will assemble into a contiguous segment of related code that is no larger than 64K bytes. Logical divisions would be code, data, and stack segments. The word segment, when used alone, means logical segment or program segment.

Base Address

The base address is the lowest address of a physical segment. All offsets within a segment are counted from the base address.

Physical Segment

A physical segment is a contiguous block of 64K bytes of memory. Physical segments contain the code and/or data when the program is loaded into
memory. Physical segments are paragraph-aligned which means that their base addresses are divisible by 16 (least significant hex digit equal to 0H). The base address of a physical segment is the number you ASSUME (the reference is made through a segment name) for a segment register in order to be able to access memory within the segment. The loader does not check against ASSUME values. It is your responsibility to make sure they are correct.

**Absolute Segment**

An absolute segment is a segment whose base address is completely known at assembly-time. In the assembly source code, the AT keyword followed by an address produces an absolute segment. (The loader command SEG can also assign an absolute address to a relocatable segment.)

**Relocatable Segment**

A relocatable segment is one whose base address is not known at assembly-time. The base address is calculated by the loader during the load process.

**Paragraph (Segment) Number**

A paragraph is 16 bytes in length. A paragraph-aligned address starts at a 20-bit address that is divisible by 16 (lowest significant digit is 0H). The upper 16 bits of the 20-bit address is the paragraph number. A paragraph number lies between 0 and 0FFFFH, inclusive. The base address of a segment may be defined in terms of a paragraph number and offset. First, multiply the paragraph number by 16. Then, add an offset from 0 to 15 to that number. The result is a 20-bit address which can be a base address. A paragraph number may also be the start of a physical segment.

**Class**

A class is a collection of segments that have had the same symbol (class name) associated with them at assembly-time. The segments in a class are placed adjacent to each other in memory by the loader, unless you specify otherwise through loader commands. Note that adjacent does not necessarily mean contiguous. It only means that other segments will not be placed between them.
Group

A group is a collection of segments that are to be placed within the same physical segment. They can therefore be addressed from a single segment register. The segment register must contain the group base address, not the segment base address. Unlike segments in a class, the segments in a group are not necessarily adjacent in memory, but must lie within 64K of the base address of the group.

Note

If a group of a given name exists in more than one file and the lists of segments named as part of the group is not the same in each file, the loader will create a group of that name that has a segment list that is the union of the segment lists from the files. For instance, if file A had the following GROUP directive

```
datagrup GROUP data1, data2, data4
```

and file B has the same group, but the segment list is different, such as

```
datagrup GROUP data1, data3, data5
```

then the loader will merge this group. The result would be a group "datagrup" that contains data1, data2, data3, data4, and data5. Although data1 appears twice, it will not be duplicated in the group.

This is, in effect, a merging of the two group lists. The resulting group must still be contained within the 64K boundary. The loader does not report that it has merged groups.

This feature may be used to create groups when all member segments or externals are not known, or when it is inconvenient to create empty segments just to include them in a group list.

Group Base Address

If set by the loader, the base address of group is the base address of the "lowest" segment in the group (the segment base address that is less than or equal to all the other segment base addresses in the group.) It is a 20-bit address divisible by 16. It is either set at load time by the loader or specified explicitly with the GROUP loader command.
Module

A module is the relocatable object code resulting from a single assembly. It can contain pieces of one or more segments. (Each module contains at least the default segment ??SEG.)

Provided that the segment parts are not private (non-combinable), the loader can combine parts of a segment from different modules. When combined, these parts make up a contiguous block of memory as if they were from a contiguous piece of object code.

Note

The default combine-type is private (non-combinable).

In order for segment parts to be other than private, they must be explicitly declared as public, or some other combine-type, in the assembly source code.

Complete Name

The loader identifies a segment by its segment name and its class name together. The term complete name refers to the segment name/class name pair. If two segments from different assemblies have the same segment name but different class names (or if one has a class name and the other does not), these segments are considered to be different, unrelated entities. The loader may not recognize a segment as being valid if it must have a class name appended to it and the class name does not appear. This treatment is different from that of the assembler. Each segment has a unique complete name, but more than one segment can use the same segment name.

Segment Attributes

Each segment (or piece of a segment) has two attributes associated with it at assembly-time: a combine-type attribute and an align-type attribute. All segments have these attributes because if you do not specify explicit attributes,
the assembler defaults to a non-combinable combine-type and a paragraph-aligned align-type.

Parts of the same segment (as specified by a complete name) from different modules must have the same combine-type, or the loader issues an error. The align-types, however, do not have to be the same.

The align-type of an absolute segment is always paragraph because of the manner that it is defined in the assembly code.

**Combine-type Attributes**

The combine-type attribute specifies how different pieces of a segment will be united by the loader. A segment piece may have a combine-type attribute of non-combinable, public, common, stack, or memory. These attributes are described in the next sections.

**Non-combinable**

The assembler defaults to non-combinable if you do not specify a combine-type attribute. The term private is sometimes used to mean non-combinable. If different modules contain non-combinable segments of the same name, they are treated as separate segments and are not combined by the loader.

**Public**

Public means combinable. The loader concatenates pieces of a public segment from different modules to form a contiguous segment. The length of the resulting segment is the sum of lengths of the combined pieces.

**Common**

Pieces of the common segment from different modules are overlapped. The base address is the same for all such pieces. The length of the combined segment is the length of the largest of the pieces. Such segments are useful as a shared data area.

**Stack**

Pieces of the stack segment from different modules are concatenated such that they end at the same address (in high memory). The length of the combined
segment is the sum of the lengths of all the pieces. Such segments are generally used to hold the system stack.

**Memory**

Memory combines different segment pieces in the same manner as a common segment, but the memory segment is placed above all other segments in memory (unless specified otherwise by loader commands). The length of the combined memory segment is the length of the largest of those combined. If two segments with different complete names both have the memory combine-type attribute, only the first one encountered is treated as memory; any others are treated as common segments and an error is generated indicating the segment that has not been treated as memory.

**Align-type Attribute**

The align-type attribute specifies how segment base addresses will be aligned in memory, and, in some cases, how the parts of a segment are aligned within the segment.

A segment can have one of five align-type attributes.

**Page**

The base address of a page-aligned segment must be divisible by 256 (two least significant hexadecimal digits equal to 00H). Uninitialized bytes may be left between the pieces of a public segment to maintain align-type.

**Paragraph**

The base address of a paragraph-aligned segment must be divisible by 16 (least significant hexadecimal digit equal to 0H). Uninitialized bytes may be left between the pieces of a public segment to maintain align-type.

**Word**

The base address of a word-aligned segment must be even (divisible by two). Uninitialized bytes may be left between the pieces of a public segment to maintain alignment.
Byte

The pieces of a segment from different object modules will be placed immediately after each other, regardless of the base address, and there will be no memory wasted.

Inpage

The inpage segment must fit within a page (256 bytes). If the loader determines that such a program segment cannot fit within the current page, it begins the segment on the next page boundary. If the segment is greater than 256 bytes and will not fit within a page at all, a warning is issued. Within a page, segment pieces are byte-aligned.

Segment Alignment

The align-types for different segment parts from different modules do not have to be the same. For this reason, two questions must be addressed.

1. How is the segment base address affected by conflicting alignments among parts?
2. How is the alignment of the segment parts within the segment affected by conflicting alignments among parts?

The next two sections answer these questions. They will refer to the following example source code pieces.

Assembly source code for Module A:
```
DATASEG SEGMENT WORD PUBLIC ;word-aligned
 .
.
DATASEG ENDS
```

Assembly source code for Module B:
```
DATASEG SEGMENT PARA PUBLIC ;paragraph-aligned
 .
.
DATASEG ENDS
```

Alignment of Base Address

If align-types do conflict (and there is not an ALIGN command in the command stream), the loader will assign a base address with the most
restrictive alignment from any part of the segment. In Module A, DATASEG has a word alignment. In Module B, the same segment has a paragraph alignment. The loader will combine these two parts into a single segment at load time. If there is no ALIGN command to assign a different alignment, the loader will use the most restrictive alignment carried by any of the segment parts. The base address of the combined segment will be aligned according to that most restrictive alignment. Of the combine-types, page alignment is the most restrictive. The ranking, from higher to lower, is

```
PAGE
PARAgraph
WORD
BYTE
INPAGE
```

In this example, the loader would align DATASEG on a paragraph boundary.

The alignment assignment is the same regardless of the order that the modules were loaded or the combine-types (PUBLIC, STACK, and so on) of the segment parts.

However, if an ALIGN command appeared in the command stream, the alignment it specified would override any alignment carried by the segment parts. (ALIGN is, obviously, a loader command with which you may change the alignment of a segment.) This would occur even if the alignment set by the ALIGN command was less restrictive than the alignments carried by the segments. To continue the example, if the loader command

```
* byte-alignment specified
ALIGN  DATASEG=B
```

appeared in the command stream either before or after the modules containing the segments were loaded, the align-type for the DATASEG segment would be byte despite the fact that both parts carry alignments that are more restrictive (word and paragraph).

Another loader command that affects alignment is the SEG command. SEG is used to assign absolute addresses to relocatable segments. A SEG command will override both the alignment carried by a segment and the alignment specified by the ALIGN command.
Changing the alignment of a non-combinable segment can create problems. Incorrect code might be produced because the loader cannot correctly modify the offsets within non-combinable segments.

**Alignment Within a Combined Segment**

The combine-type of the segment parts *does* affect how the parts of the combined segment will be aligned within the segment.

**PUBLIC Combine-type.** Each part of a public segment retains the alignment it carried with it from the assembler. Within the combined segment, the loader aligns each segment part according to the alignment it carries, with the exception of the first segment part. The first segment might not retain its alignment because the loader might adjust it either to conform to the most restrictive alignment or in response to an ALIGN loader command. In the example given earlier, the loader would combine the segment and align the base address and parts in the following way (assume no ALIGN command will be used and the load order is Module A and then Module B):

The most restrictive alignment is paragraph. The entire combined segment will be aligned on a paragraph boundary. Note that since Module A is loaded first, it will be aligned not on a word boundary, but on a paragraph boundary.

Module B is the next segment part to be placed. Its alignment, within the segment, will be on a paragraph boundary. Suppose that the segment part from Module A is just 3 bytes in length. The loader will still move the segment part from Module B to the next paragraph boundary. That leaves 13 bytes unused between the first and second parts. Suppose the segment part from Module B is 10 bytes in length. The entire combined segment will be 26 bytes in length although 13 bytes are unused.

**STACK Combine-type.** For a stack segment, the align-type attribute applies to the base (low) address of the combined segment. The align-type of the pieces of the segment are ignored by the loader. They are concatenated contiguously in memory just as a stack should be. Again referring back to the example, if the combine-type was STACK instead of PUBLIC, the combined segment would begin on a paragraph boundary and its combined length would be the sum of the lengths of the parts.
COMMON Combine-type. Since pieces of a COMMON segment are overlaid, the align type of any part is only used to determine the most restrictive alignment in the absence of an ALIGN command.

MEMORY Combine-type. Since pieces of a MEMORY segment are overlaid, the align type of any part is only used to determine the most restrictive alignment in the absence of an ALIGN command.

NON-COMBINABLE Combine-type. Private segments are not combined. The alignment a private segment carries from assembly is used to align the base address unless it is changed with an ALIGN command.

Base Address Assignment

Segments are assigned space in memory in a user-controlled order. The order can be both implicit and explicit.

- Implicit order depends upon where the segments appeared in the original source code to the assembler. The order they appeared in the source code controls the order they appear in the object module. The order that they appear in the object module can control the order they appear in the output from the loader, if no other restrictions apply.

  The exception is the default segment ??SEG. It appears first in the object module unless the -h or -H command line options are used with the assembler. Then the order of the first three segments will depend upon the first code, data, and mixed segments. This ordering is used in the generation of HP64000 absolute files.

  Implicit order also depends on the order that the modules are given to the loader. Modules are given to the loader from the command line and from LOAD commands in the command file. The modules specified on the command line "precede" modules in files named in the command file. Among modules in the files named in the command files, the modules that the loader finds nearer to the beginning of the command file "precede" the modules found in files later in the command file.

- Explicit order comes from the loader command ORDER, from the ordering caused by the presence of segments made absolute either at
assembly time or load time, and from any classes that might appear in the load files.

If libraries are included in the load, library relocatable object modules that are not selected for inclusion in the absolute object module do not have their segment names examined by the loader. A description of the algorithm used to assign base addresses follows:

1. The loader reads all its commands and all files specified in LOAD commands, and determines what segments are present, the align-type, and the size of each. These are placed in an internal structure in the order in which the loader finds their names.

2. The loader blocks reserved memory areas and assigns base addresses for absolute segments. Memory reserving is done through the two loader commands RESADD and RESNUM. Absolute segments have their addresses specified either at assembly time with the AT keyword or at load time with the SEG command. The absolute segments are marked in the structure as having had their base addresses assigned.

A segment name can appear in a SEG command, but a SEG command may also refer to a class name. If a class name appears in a SEG command, the first segment in that class that does not yet have a base address (the first such segment whose name was encountered by the loader and was not an absolute segment or was not named in a SEG command) is assigned the specified base address and marked in the structure. Other segments in such a class are not assigned base addresses at this time, however.

3. All segments named in ORDER commands, and segments within a class named in ORDER commands, are assigned base addresses in the order in which they were named in the commands. The loader attempts to begin loading these segments at physical address 00000H, if possible.

There are several issues to consider in this step.

The areas of memory reserved with RESADD and RESNUM cannot be used.

If a class is named in an ORDER command, it may contain segments which have already been assigned base addresses because they were either absolute segments or they appeared first in the list following the order command. If, for instance, an ORDER command contains a segment name in the middle of the order specification and that segment was an absolute segment, the
loader must work around that absolute segment when placing the class in memory. Some segments will go before the absolute segment, some will follow it.

Since classes must be adjacent, they cannot be fitted around reserved memory areas.

Alignment must be maintained which means the loader must adjust base addresses so that they conform to the align-type for a segment and/or the alignment restrictions imposed by other segments in the same physical segment.

4 The first segment with a combine-type of MEMORY that does not yet have a base address is marked and saved, but is not assigned a base address at this time. (This segment would already have a base address only if it appeared in an ORDER command or SEG command or it was an absolute segment.)

5 Any segments remaining in the segment structure are assigned base addresses beginning just above the last segment assigned in step 3. The loader attempts to fill memory contiguously while taking into consideration the reserved memory, absolute segments, and alignment maintenance. The order that segments are assigned addresses is the order that they appear in the internal structure (the order in which the loader first found them.) Of course, if the loader finds a segment with a class name, then it loads all the segments in the class regardless of the order that they appear in the structure.

6 The memory segment marked and saved in step 4 is assigned a base address above all addresses used so far. If this cannot be done within a 20-bit address space (the base address needed does not exist in the address space, or the base address plus the size exceeds the address space), an error is reported.

7 The base addresses of any groups are assigned, unless they were user-specified in a GROUP loader command. If the loader sets the group base address, it is the 20-bit address of the segment in the group that is "lowest" in memory. (It is an address divisible by 16 and is less than or equal to the lowest base address of any segment in the group.) If the base address has been specified by the GROUP loader command, then the loader error-checks against the value in the GROUP command. Note that this algorithm does nothing to ensure that the segments in a group lie within 64K of the group base; that is your responsibility. However, the loader will report an error if this is not the case. One way to ensure
segments in a group will lie within the 64K limit is to assign all segments in a group the same class name; then they will be adjacent in memory unless you override with SEG commands.
Chapter 21: Loader Commands

Loader Commands

Description of the various loader commands.
Loader commands give you the ability to control the linking/loading process to a very high degree. These commands may be given to the loader in the interactive mode or they may appear in a load command file.

The descriptions include the syntax for the command, a short description of the purpose of the command, and possibly an example of the command in use.

---

**Loader Commands Introduction**

The ld86 Linking Loader reads a sequence of commands in batch mode from a command file or reads commands in the interactive mode from some other input device such as a terminal. One of the loader commands, the LOAD command, specifies the object modules to be loaded from files or other logical devices along with the loader commands. The loader generates an absolute load module suitable for loading into an actual microprocessor. The output module is written to the output device in HP-OMF 86 format absolute, unless the optional Intel Hexadecimal Object file format absolute or the HP 64000 format absolute is specified. The loader is also capable of producing HP-OMF 86 format relocatable from an option known as incremental linking. Incrementally-linked object modules can later be re-linked into absolute formats.

All commands must begin in column 1. Command arguments can begin in any column, but the arguments must be separated from the command by at least one separator. Generally, separators (blanks or tabs) are allowed anywhere, except within a symbol or a number. Exceptions are described under the individual commands. The loader command file may have comments placed in it. Comments are denoted by a preceding asterisk.

**Command Symbols**

Symbols that are command arguments follow the same rules as assembler symbols with the exception that a colon is an accepted character in loader symbols, although not as the first character. Command arguments that are numeric can be either decimal or hexadecimal. Hexadecimal constants are terminated by an upper or lower case H (for instance, IFH). Like the assembler, hex numbers that begin with the hex digits A-F must have a leading zero to avoid confusing them with symbols in certain commands.
Complete Name

Some of the loader commands take segment names as arguments. If a segment belongs to a class, you must use the segment name and class name together to refer to the segment. The proper form is segment name followed by a slash followed by the class name. The segment name/class name pair is known as the complete name. The loader will not recognize a segment as valid if its class name is missing. Any segment name without a specified class name is assumed to have a null class name. A null class name cannot be explicitly referenced, therefore SEGNAME/ without a following class name is not acceptable.

Note

The loader does not read all loader commands before it begins some processing. Therefore, it cannot always tell whether it has received a valid segment name with a command that accepts one. It is possible to enter a segment name that is invalid at the time it is entered, but that will be made valid by later actions. If the loader completes the load and the invalid segment name has not been resolved, it issues an error stating that it cannot find the segment in question. Unfortunately, a segment name that must have a class name, and does not, will cause this error. A segment name with a group name in the place of a class name will also cause this error. Both cases are not obvious as to why they caused errors so check that the complete name for the segment has been properly specified when this error appears.

Order of Commands

Commands may be entered in any order. The same command, with the exception of EXIT and END, may be used more than once. (EXIT and END both terminate the reading from the command file). If ORDER, START, NAME, LIST and NLIST appear more than once, only the last one will be in effect. Other commands may appear as often as required, and they will be executed each time.

Command Length

The maximum line length, for a command entered interactively or placed in a command file, is 254 characters.
### Loader Command Descriptions

In the command descriptions in this section, square brackets ("[" "]") are used to indicate optional arguments. Square brackets containing an ellipsis indicate that the preceding argument can be repeated zero or more times. The following summary lists the commands in the order of their occurrence.

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>Set Alignment for a Segment</td>
</tr>
<tr>
<td>Comment (*)</td>
<td>Specify Comment</td>
</tr>
<tr>
<td>END</td>
<td>End Command Stream and Finish Load</td>
</tr>
<tr>
<td>ERROR</td>
<td>Change Message Severity to ERROR</td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit Loader</td>
</tr>
<tr>
<td>FORMAT</td>
<td>Specifies Output Format</td>
</tr>
<tr>
<td>GROUP</td>
<td>Set Group Base Address</td>
</tr>
<tr>
<td>INITDATA</td>
<td>Specify Initialized Data in ROM</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Set Page Length</td>
</tr>
<tr>
<td>LIST</td>
<td>List Specified Elements</td>
</tr>
<tr>
<td>LISTABS</td>
<td>List Specified Elements</td>
</tr>
<tr>
<td>LISTMAP</td>
<td>Specifies Layout and Content of the Map</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load Specified Object Modules</td>
</tr>
<tr>
<td>NAME</td>
<td>Specify Output Module Name</td>
</tr>
<tr>
<td>NLIST</td>
<td>Do not List Specified Elements</td>
</tr>
<tr>
<td>NOERROR</td>
<td>Change Message Severity to NOERROR</td>
</tr>
<tr>
<td>NOTYPEMERGE</td>
<td>Do not Merge Type Information</td>
</tr>
<tr>
<td>ORDER</td>
<td>Specify Segment Order</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Specify Symbol Definitions</td>
</tr>
<tr>
<td>RESADD</td>
<td>Mark Memory as Reserved</td>
</tr>
<tr>
<td>RESNUM</td>
<td>Mark Memory as Reserved</td>
</tr>
<tr>
<td>SEG</td>
<td>Set Segment Base Address</td>
</tr>
<tr>
<td>SEGSIZE</td>
<td>Specify Segment Size</td>
</tr>
<tr>
<td>START</td>
<td>Specify Starting Output Module Address</td>
</tr>
<tr>
<td>TYPHEMEERGE</td>
<td>Merge Like Type Definitions</td>
</tr>
<tr>
<td>WARN</td>
<td>Change Message Severity to WARNING</td>
</tr>
<tr>
<td>WIDTH</td>
<td>Set Page Width</td>
</tr>
</tbody>
</table>
ALIGN

Syntax:

ALIGN segment=blank | B | P | I | G | W

The vertical bar between arguments means "or" and implies that only one of the six arguments may appear.

Where:

- **segment** is the name of a relocatable segment. Segment can be a segment name or a complete name (segment name/class name pair). An absolute segment is accepted syntactically, but ALIGN on an absolute segment has no effect. The equal sign must be included, even if a blank follows.

- **blank** keeps the align-type the same as specified in the assembler. This is the default. Because blank is a significant character in this location, separating blanks are not permitted between the equal sign and the alignment mnemonic.

- **B** specifies BYTE alignment.

- **P** specifies PAGE alignment.

- **I** specifies INPAGE alignment.

- **G** specifies PARAgraph alignment.

- **W** specifies WORD alignment.

Description:

Each segment from an assembler-generated module carries its align-type information. It is either the align-type specified in the assembly source code or the default align-type of PARA (paragraph alignment). Other possible align-types are BYTE, WORD, PAGE, and INPAGE.

At load time, you may accept the align-type the segment already has or you may override it without re-assembling the module. The ALIGN command allows you to do either.
Typically, you would use ALIGN to make all segments page-aligned to assist you with debugging and then, before the final load, use ALIGN to change the segments to byte-aligned to save memory space.

The ALIGN command can appear in the command stream either before or after the modules containing the segment or segment pieces are loaded and it will override the original alignment.

If an absolute segment (set at assembly-time with the AT keyword) appears in an ALIGN directive, ALIGN is ignored and the loader issues a warning. If a relocatable segment that has its base address set with a SEG command appears in an ALIGN directive, the SEG command overrides the ALIGN command and the alignment specified is ignored.

**Note**

If a non-combinable segment appears in an ALIGN command, incorrect code may be produced because the loader cannot modify the offsets within non-combinable segments.

Examples:

```plaintext
ALIGN SEG1=B
ALIGN SEG2/CLASS1=G
ALIGN SEG3=
  *blank is argument—align-type is that
  *which the segment carries from assembly
```
Comment (*)

Syntax:

* loader comment line

The asterisk is used to indicate a comment in the command stream. The asterisk must be entered in column 1. The loader ignores any text on the line until the end-of-line character is reached.

END

Syntax:

END

Description:

This command initiates the final steps in the load process. After an END command is found in the command file, the loader completes the load, produces an output object module, and returns to the host computer operating system. If the command file does not contain an END command, the loader stops reading commands when it detects an end-of-file and initiates the final steps at that point. However, using the END command promotes command file clarity and readability.

ERROR, WARN, NOERROR

Syntax:

ERROR condition{condition} ...
WARN condition{condition} ...
NOERROR condition{condition} ...
Where:

- **condition**: One of UNREF, UNRES, OVERLAP, or a number corresponding to the message number of the error or warning.

  - UNREF refers to the undefined external reference error. UNRES refers to the unreferenced external warning. OVERLAP refers to the memory overlap warning.

These commands change the way a message or group of messages is treated. ERROR causes the message to be treated as an error; WARN causes the message to be treated as a warning; NOERROR causes the message to be treated as a non-error (that is, the error condition is ignored).

The ERROR, WARN, and NOERROR commands affect all messages which are generated after the linker encounters the command. The change in message severity remains in effect until the linker has finished processing. The effect of these commands cannot be changed by subsequent ERROR, WARN, or NOERROR commands.

Fatal errors and messages generated by the ERROR, WARN, or NOERROR command cannot be overridden or modified.

---

**EXIT**

**Syntax:**

```
EXIT
```

**Description:**

EXIT terminates the loader execution without generation of a load map or output object module.

The EXIT command can be used in the interactive mode to exit the loader when an error occurs that requires leaving the loader to fix. In the interactive mode, most command errors are recoverable; however, errors in the LOAD command are generally not recoverable.
This command can also be used in a command file. In this case, the final load will not take place, but the commands up to and including the EXIT command will be read and checked for errors. The loader ignores any commands following the EXIT command in a command file.

FORMAT

Syntax:

```
FORMAT type
FORMAT modifier
FORMAT type modifier
FORMAT NOABS
```

Where:

type       One of the following:

- ASCII
- HP
- OMF86

modifier    One of the following:

- INCREMENTAL
- LIMITED
- LTL

Description:

The FORMAT command lets you specify the output object module format. The type option indicates which output format is to be generated by the linker. A list of acceptable formats follows:

- ASCII refers to the Intel Hexadecimal Object File Format
- HP refers to the HP 64000 Object Module Format (HP-OMF)
- OMF86 refers to Intel Binary OMF86 as extended by HP (HP-OMF86)
NOABS prevents an absolute file from being produced. If an unsupported type specifier is encountered, an error or warning will be generated and the default output format will be produced.

If NOABS is specified, no object file will be produced; however, internal processing will be carried out and a map file will be produced if requested. Only one format type may be specified. In addition, NOABS cannot be used with any of the modifier options. If either of these conditions occurs, an error or warning will be issued and the FORMAT command will be ignored.

The following modifiers may be used only with the OMF86 file type. If no type is specified, OMF86 will be assumed.

- The INCREMENTAL modifier to the FORMAT command indicates that incremental linking is to be performed.
- The LIMITED modifier to the FORMAT command limits the amount of usable segment base information contained in the OMF86 data records.
- The LTL modifier to the FORMAT command produces a Load-Time Locatable object module in OMF86 format.

FORMAT cannot be specified without any options. If such a situation is encountered, an error or warning will be issued and the command will be ignored.

The FORMAT command has a global effect. If multiple FORMAT commands are encountered, a warning message will be generated and the first FORMAT command will be used.

All FORMAT commands must appear before the first LOAD command in a command file. Any FORMAT commands appearing after the first LOAD command will be flagged with an error or warning and ignored.

---

**GROUP**

**Syntax:**

GROUP group=address
GROUP group=paragraph,offset
Where:

- **group** is the name of a group.
- **address** specifies an address where the group begins. The address must be divisible by 16, or an error is reported. The acceptable range, given the paragraph restriction, is from 0 to 0FFFF0H.
- **paragraph** specifies an actual value to be loaded into a segment register. The range for this value must be within 0 to 0FFFFH, inclusive, or an error is reported.
- **offset** specifies the offset from the given paragraph. The offset must be 0 or the loader reports an error.

Description:

This command specifies the absolute base address of a group. Such an address always lies on a paragraph boundary (a multiple of 16). The default group base, if there is no GROUP command, is calculated by the loader in the manner explained in the "Base Address Assignment" section beginning on page 378.

You can enter multiple GROUP commands specifying the same group name, but only the last one applies.

Note that the GROUP command does **not** assign a base address to any of the segments in the group. If you specify the location of a group with the GROUP command, you must ensure that all such segments lie entirely within the 64K limit imposed on physical segments. This is accomplished with the ORDER and SEG commands. The loader reports an error if this condition does not hold for any segment in the group. Examples:

```
GROUP DGROUP=100H
* DGroup starts at 100H
GROUP CGROUP=7,0
* CGroup starts at 70H
```
INITDATA

Syntax:

```
INITDATA segment [,segment [,...]] [,address]
```

Where:

- **segment** could be one of the following:
  - `segmentname`
  - `segmentname/classname`
  - `/classname`

- **address** could be one of the following:
  - An address value from 0 to 1048576 (0FFFFFH), inclusive, or
  - A paragraph,offset pair. The paragraph and offset may range from 0 to 65535 (0FFFFH), inclusive. With either form, leading zeros are required for hexadecimal values that start with the hex digits A-F.

Description:

The INITDATA command specifies those data segments or classes that will be initialized in memory at run time. The INITDATA command optionally can be used to specify the base address of the created logical segment(s).

Initialized data in the specified segments is placed in new segments at the specified address. These new segments are named ??DATAn??INIT, where `n` is the number of required segments depending upon the amount of data needed.

If an address is not specified, an address determined by the SEG command or the base address assignment algorithm is used. If you have compiler libraries from the Hewlett-Packard CC8086 C cross compiler, a routine in the compiler startup code copies all initialize data from the created segments into the original segments. Likewise, there is an assembly file, `/usr/hp64000/lib/8086/src/initdata.s`, which may be used to copy the initialize data.
Example:

INITDATA /data, 0fff0h

The example illustrates the use of the INITDATA command.

---

**LENGTH**

**Syntax:**

LENGTH n

**Description:** Specifies the page length of the output listing as "n" lines, where n is a number greater than 3. The default is 60 lines per page. Example:

LENGTH 55
LIST, NLIST

Syntax:

LIST [A, B, C, D, E, I, L, O, P, Q, S, T, U, V, W, X]

NLIST [A, B, C, D, E, I, L, O, P, Q, S, T, U, V, W, X] @NCW = Note

The LIST and NLIST commands are being replaced by the FORMAT, LISTABS, and LISTMAP commands.

Note

LIST A, LIST B, and LIST I are mutually exclusive. If more than one appears in a command stream, the first one is used to specify the output format and the others generate warnings and are ignored. Similarly, LIST Q can only be used with LIST B, since LIST Q makes the Intel Binary OMF produced more compatible with older Intel tools that accept the binary format.

Where:

A
LIST - Creates an Intel Hexadecimal Object file format absolute output file.

NLIST - Accepted syntactically, but has no effect. Will not toggle LIST option for this flag.

B
LIST - Creates an absolute file in the HP-OMF 86 format absolute. (default)

NLIST - Accepted syntactically, but has no effect. Will not toggle LIST option for this flag.

C
LIST - Creates a cross reference listing. Use of this option could slow program execution.

NLIST - Inhibits production of a cross reference listing. (default)
D  LIST - Places public definition symbols in the output object module. This option causes the Intel Hexadecimal Object format absolute to be unusable in Intel tools. (default)

NLIST - Inhibits placement of public definition symbols in the output object module.

E  LIST - Causes warning messages to be generated for any remaining undefined external symbols during an incremental link.

NLIST - Inhibits warning messages for any remaining undefined external symbols during an incremental link. (default)

I  LIST - Produces relocatable output object modules in the HP-OMF 86 format relocatable that can then be incrementally linked. (default is HP-OMF 86 absolute output)

NLIST - Accepted syntactically, but has no effect. Will not toggle LIST option for this flag.

L  LIST - Causes warning messages to be printed for any unreferenced, unresolved, external references.

NLIST - Inhibits warning messages for any unreferenced, unresolved, external references. (default)

O  LIST - Specifies that an output object module is to be produced. (default)

NLIST - Inhibits production of an output module. This is useful when checking for errors.

P  LIST - Places local symbols present in the input modules in the loader symbol table. (default)
NLIST - Inhibits placement of local symbols from the input object modules into the loader symbol table. Useful when many modules are being loaded and the loader is executing more slowly due to the large number of symbols.

Q LIST - Causes the loader to produce a "limited" form of Intel binary OMF that is strictly compatible with the Intel Binary OMF document.

NLIST - Causes the loader to produce HP-OMF 86 absolute, the HP extension of the Intel Binary OMF used in high level analysis. (default)

S LIST - Writes local symbol information to the object module. This feature is useful for debugging. Local symbols are those placed into the object module by the assembler that are not external definitions. This option causes the Intel Hexadecimal Object format absolute to be unusable in Intel tools. (default)

NLIST - Inhibits writing of the local symbol table to the object module.

Note

In the relocatable and absolute HP-OMF 86 output modes, line numbers and procedure definitions present in the input files are preserved and stored in the output file. In the Intel Hexadecimal Object file format absolute output mode, external definitions and debug symbols are written to the output file. Since the set of external definition symbols cannot be distinct from the debug symbols, duplicate symbol definitions can occur in the ASCII hexadecimal output file.

T LIST - Prints the local symbol table on the output listing.

NLIST - Inhibits printing the local symbol table on the list output device. (default)
U LIST - Disables case-sensitivity for matching public and external symbols. Converts all symbols in the file to upper case (which may affect debugging).

NLIST - Enables case-sensitivity for matching public and external symbols. (default)

V LIST - Produces an expanded segment summary in the load map that lists the modules where the segment parts were found.

NLIST - Inhibits production of an expanded segment summary in the load map that lists the modules where the segment parts were found. (default)

W LIST - Enables display of warning messages to the output listing and to the terminal. (default)

NLIST - Inhibits display of warning messages to the output listing and to the terminal.

X LIST - Prints the public definition symbol table on the output listing.

NLIST - Inhibits listing the public definition symbol table on the list output device. (default)

Description: The LIST and NLIST commands are used to generate or suppress listings of the elements specified.

LIST T,X
*prints local and external definition
*symbol tables in the output listing
NLIST O
*suppresses production of
*an output object module
LISTABS

Syntax:

The LISTABS command controls the output of certain items to the output object module. Multiple LISTABS commands can be specified and have an accumulative effect.

- **INTERNALS** causes local symbols to be written to the output file. This is equivalent to the LIST S command. (default: INTERNALS)
- **PUBLICS** causes globally defined symbols to be written to the output file. This is equivalent to the LIST D command. (default: PUBLICS)

The LISTABS command will eventually replace the LIST/NLIST D and LIST/NLIST S commands.

LISTMAP

Syntax:

```
LISTMAP option[,option]...
```

Where:

<table>
<thead>
<tr>
<th>option</th>
<th>One of the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NO]CROSSREF</td>
<td></td>
</tr>
<tr>
<td>[NO]INTERNALS/BY_NAME/NAME</td>
<td></td>
</tr>
<tr>
<td>LENGTH number</td>
<td></td>
</tr>
<tr>
<td>[NO]MODULE</td>
<td></td>
</tr>
<tr>
<td>[NO]PUBLICS/BY_ADDR/ADDR/BY_NAME/NAME</td>
<td></td>
</tr>
<tr>
<td>[NO]SEGMENT</td>
<td></td>
</tr>
<tr>
<td>[NO]VERBOSE</td>
<td></td>
</tr>
<tr>
<td>[NO]WARNINGS</td>
<td></td>
</tr>
<tr>
<td>WIDTH number</td>
<td></td>
</tr>
</tbody>
</table>
The LISTMAP command controls the output of certain items to the linker’s map file.

Each of the functions of the LISTMAP command are described below:

- **CROSSREF** causes a cross-reference listing to be output to the map file. **NOCROSSREF** suppresses the generation of this cross-reference listing. (default = NOCROSSREF)

- **INTERNALS** causes a listing of the non-public (local) symbol table to be output to the map file. **NOINTERNALS** suppresses the output of the non-public symbol table. If /BY_NAME or /NAME is specified, the symbol table is listed in ASCII order. (default = NOINTERNALS)

- **LENGTH** specifies the map file page length to a number between 5 and 255. (default = 255)

- **MODULE** controls the output of the module summary to the map file. (default = MODULE)

- **PUBLICS** causes a listing of the public symbol table to be output to the map file. **NOPUBLICS** suppresses the output of the public symbol table. If /BY_NAME, /NAME, or nothing is specified, the public symbol table is listed in ASCII order. If /BY_ADDR or /ADDR is specified, the table is listed in address order. (default = NOPUBLICS)

- **SEGMENT** controls the output of the segment summary to the map file. (default = SEGMENT)

- **VERBOSE** controls the output of additional information to the segment summary in the map file. This option has no effect if LISTMAP NOSEGMENT is specified.

- **WARNINGS** controls the output of warnings to the map file. (default = WARNINGS)

- **WIDTH** specifies the page width as a number between 20 and 255. (default = WIDTH 80)

**Note**
LISTMAP CROSSREF was formerly known as LIST C. LISTMAP INTERNALS was formerly known as LIST T, and LISTMAP PUBLICS was formerly known as LIST X.
Chapter 21: Loader Commands
LOAD

LOAD

Syntax:
LOAD [-]module[, ...]

Where:
module names a file in which the object module or library resides. Any module or library preceded by a minus sign will have its object modules read until an EOF is detected. Without the minus sign present, the loader would load only those modules from the library that were necessary to resolve external references. The minus sign preceding a library forces all modules in the library to be loaded.

Description:
The LOAD command is used to specify one or more input object modules to be loaded. The command operand is the name of the file containing the object module. Input object modules can consist of relocatable modules from the assembly process, relocatable modules from incremental linking, or libraries.

If any file name is preceded by a minus sign, it indicates that all object modules should be read from the file. In order that external references are handled correctly, the following order for loading libraries, along with other kinds of object modules, should be observed.

- Libraries should be loaded after all non-libraries. From libraries, the loader will load only those modules that are necessary to resolve undefined external references (EXTRNs), unless the library file name is preceded by a minus sign.

- Backward external references within a library are resolved correctly. However, external references to a library from a file loaded after the library has been loaded are generally not resolved correctly. Therefore, libraries should be loaded last.

- When two libraries make external references to each other, it is generally necessary to LOAD one of them twice (for example, LOAD LIBA,LIBB,LIBA) in order to pick up all the necessary modules.

Object modules may or may not be read until the EOF. The object modules are loaded in the order specified, with each piece of each segment being
loaded into memory at a higher address than all preceding pieces of the same segment. Any number of LOAD commands can be used. Example:

LOAD ONE,-EACH.o

In the example, suppose that EACH.o contains two modules. This load command will cause three modules to be loaded: the first from the file named ONE.o, and the next two from the file EACH.o.

---

**NAME**

**Syntax:**

```latex
NAME name
```

**Where:**

*name* is a symbol that specifies the new name for the object module.

**Description:**

The name command is used to give a new name to the output object module. In the load map listing, the new name (or current name) is found next to the heading "OUTPUT MODULE NAME:"

---

**ORDER**

**Syntax:**

```latex
ORDER element[,...]
```

**Where:**

*element* could be one of the following:
ORDER

- `segmentname`
- `segmentname/classname`
- `/classname` Equivalent to specifying all segment names with that class attribute in the order their names were encountered by the loader. All such segments are placed as adjacent as possible in memory (allowing for SEG commands, absolute segments, and reserved areas).

- `classname—segname1—segname2—...—segnameN` (Notice the hyphen separating each name.) The specifically named segments are moved to the beginning of the class and ordered the way they appear in the command. Any segments remaining in the class are assigned memory immediately after the specified ones.

Description:
The ORDER command is used to override the loader’s default order of assigning base addresses to segments. It is useful in forcing collections of segments addressed from the same segment register (for instance, a group) to lie close to each other in memory.

All segments specified in the ORDER command are assigned base addresses as follows: the first one specified begins at the lowest address possible, and subsequent segments begin immediately after the preceding one. The loader does not assign addresses that conflict with absolute segments, areas specified in the RESADD/RESNUM commands, or segments specified in a SEG command. The ORDER command does not override the base address of an absolute segment or one assigned with SEG. If any such segment appears in the ORDER command, any segment following it in the ORDER command is assigned space in memory above the absolute segment.

Continuation Line. If more than one line of ordering information is needed, use an ampersand (`&`) where the linker is expecting a comma or a hyphen. If breaking at a comma, leave the comma on the first line. If breaking at a hyphen, place the hyphen on the second line.

Examples:

ORDER SEG1,SEG2,SEG3
*orders segments

ORDER SEG1/CLASS1,/CLASS2-SEG2-SEG3
*(the remaining segments in CLASS2 follow, if they exist)
ORDER SEG1,CLASS1-SEG2-SEG3, &
SEG4,CLASS24 -SEG5
*note continuation line

See Also
The "Base Address Assignment" section beginning on page 378.

PUBLIC

Syntax:
PUBLIC symbol=address [, ...]
PUBLIC symbol=paragraph,offset [, ...]

Where:
symbol is a user-defined public symbol

address is the new 20-bit address of the symbol. The address has a range of
0 to 1048575 (0FFFFFFH), inclusive. The symbol's paragraph value is equal to
the address shifted right by 4. The offset of the symbol is the address modulo
16.

paragraph is a paragraph boundary number. Paragraph is multiplied by 16
and then the offset is added to it. Paragraph may range from 0 to 65535
(0FFFFH).

offset is a number in the range of 0 to 65535 (0FFFFH). It is added to the
multiplied paragraph number to form a 20-bit address.

Description:
This command is used to define and/or change the address of a public
definition. If a symbol specified by this command is already a public
definition (from an input object module where the symbol was an argument
to the assembler PUBLIC directive), the address of the symbol is changed to
the user-specified value. If the symbol is not already defined, it is entered into
the loader symbol table along with the specified address. It will then be
available to satisfy external references from object modules. This command
allows you to specify the address of some public symbols at load-time and possibly to avoid a reassembly. All symbols used with this command are considered absolute rather than relative to either a segment or a group.

Example:

PUBLIC INPUT=2FH, OUTPUT=200H

### RESADD, RESNUM

**Syntax:**

```
RESADD lowaddress, highaddress
RESNUM lowaddress, number
```

**Where:**

- **lowaddress** is the lowest address of the reserved memory space.
- **highaddress** is the highest address of the reserved memory space. Highaddress must be greater than or equal to lowaddress.
- **number** is the number of bytes, beginning at and including the low address, to reserve. If the number is 0, no area is reserved.

**Description:**

The RESADD and RESNUM commands allow you to declare certain areas of memory as off limits to the loader; no relocatable code is placed in these areas. You might wish to use these commands to avoid overwriting an operating system in low memory, for example.

If a reserved area conflicts with a previously reserved area, an absolute segment, or a segment name in a SEG directive, the loader issues a warning message and loading continues. If the warning is caused by the RESNUM or RESADD command, any non-overlapping space is reserved.

If the highaddress of the reserved area (either specified directly or computed as lowaddress+ number-1) is greater than 1048575 (0FFFFFFH), all memory from lowaddress to this limit is marked reserved. Examples:
RESADD 0,1FFH
"this and the following are equivalent"
RESNUM 0,200H

---

**SEG**

**Syntax:**

- SEG  segment=address
- SEG  segment=paragraph,offset
- SEG  /class=address
- SEG  /class=paragraph,offset

**Where:**

- **segment**  is the name of a relocatable segment. It can have a classname attached with a slash as in segname/classname.
- **class**  is the name of a class
- **address**  specifies that the segment will begin at the given address. The range of the address must be from 0 to 1048575 (0FFFFFFH), inclusive, or an error occurs.
- **paragraph**  will be a paragraph number ranging from 0 to 65535 (0FFFFH), inclusive, or an error occurs.
- **offset**  is a number ranging from 0 to 65535 (0FFFH), inclusive, or an error occurs. Base address of the segment will be 16 times the paragraph number plus the offset.

**Description:**

The SEG command specifies the base address of a logical segment.

In most cases, when you use a SEG command, you should also specify an ORDER command to control the placement of other segments that did not appear in the SEG command.

A class name, preceded by slash, can appear in place of a segment name. In this case, the first segment whose class attribute matches the class name is
assigned the base address. Exceptions apply to absolute segments and segments that appear explicitly in a SEG command. They are not then eligible to be assigned a base address with this classname construct. Other segments with the same class attribute are not assigned base addresses at this time.

However, the loader algorithm for assigning base addresses eventually causes these segments to lie immediately above the first segment in the class, unless you have entered an ORDER command.

If you enter a classname and no segments with that class attribute are ever found, a warning is issued following the END command, and loading continues.

The address specification in this command has two variations: it can use either one numeric argument or two numeric arguments separated by a comma. The first form indicates a 20-bit address, which becomes the base address of the segment. The second form indicates a 16-bit paragraph number followed by a 16-bit offset; the base address of the segment is 16*(paragraph number)+ offset. For example, SEG name=4440H and SEG name=444H,0 specify the same address. So does SEG name=440H,40H and other combinations.

A base address specified by the SEG command is never rounded up or down to conform with the alignment attribute carried from the assembly or reset by an ALIGN command. Instead, the loader uses the base address that you specified with SEG and issues a warning if an alignment conflict occurs.

If an absolute segment appears as an argument to SEG, an error is reported, though it might not be reported until the absolute segment is read from an object module. In this conflicting address case, the loader uses the address first found.

---

**Note**

Do not use SEG to place a non-combinable segment on anything other than a paragraph boundary. Doing so can cause incorrect output code to be created because the loader cannot properly modify the offsets within a non-combinable segment.

---

Multiple SEG commands specifying the same segment name or classname can occur, and the loader does not issue an error. The last command, for a given segment, applies.
The "Base Address Assignment" section beginning on page 378 describes the algorithm used to calculate load addresses when they are not explicitly provided.

SEGSIZE

Syntax:

SEGSIZE segment=length
SEGSIZE /class=length

Where:

segment is the name of a relocatable segment. It can have a classname attached with a slash as in segname/classname.

class is the name of a class.

length specifies the segment length in bytes.

Description:

The SEGSIZE command is used to specify the length of a segment in bytes. If you do not use the SEGSIZE command, the length of each of the segments in the output object module defaults to the length appropriate for the combine-type of the segment.

Use SEGSIZE only for STACK and COMMON segments. SEGSIZE is typically used to set the size of a stack segment.
START

Syntax:

START CS-value,IP-value
START address

Where:

CS-value used to initialize the CS (code) segment register. The value must be in the range 0 through 65535 (0FFFFH).

IP-value used to initialize IP (instruction pointer). The value must be in the range 0 through 65535 (0FFFFH).

Address Used to initialize CS and IP. CS will be assigned the value of address divided by 16 and IP will be assigned address modulo 16. The address must be in the range 0 through 1048575 (0FFFFFFH).

Description:

This command is used to provide the starting values for CS and IP in the terminator record of the object module. If START is not used, then the CS:IP value comes from the END directive initialization in the main program module. If the END directive has a value and START is also used, then the value specified with START overrides the value from the END directive. If no START is used and no main module is present, then the start value defaults to zero. Example:

START 0,100H

Note

If the output is to be HP 64000 format absolute, the loader only allows CS:IP pair to be loaded by using an initialization value with the END directive for the main module. The DS and SS registers values may also have had initialization values specified with the END directive, but the loader ignores these values if the output is to be HP 64000 format absolute. You must have assembly code in the program that explicitly loads DS and SS when the target format is to be HP absolute. Do not expect the START command to allow you to get around this restriction.
TYPEMERGE

Syntax:  

TYPEMERGE [ ALL | SIMPLE ]
NOTYPEMERGE

Description:  

The HP-OMF 86 file format is only able to store up to 32k type definitions. If many modules are being linked, many type definitions might exist within each of the modules. In a large executable, the 32k type limit may be exceeded, even though the code size is small. If this limit is exceeded, the loader will stop processing the type information from that point on. Since this information is useful for debugging the executable, it would be best if this information were not lost.

To get around this problem, it is possible to have the loader merge some of the redundant type information so that the total number of types stored in the resulting executable is minimal.

If the SIMPLE form of the TYPEMERGE is used, only the basic type definitions used in assembly code will be merged. While this, by itself, may greatly reduce the number of types in the resulting executable, it may not be enough.

The ALL form of the TYPEMERGE command will cause all redundant type information to be removed. This form will create the smallest amount of type information possible for the resulting executable and will create less than 32k types in all but the most exotic cases. This is the default form of the command, when there are no modifiers to the basic command.

Either form of the TYPEMERGE command will cause extra processing time during the loader’s execution. This is due to the extra overhead caused by the checking for redundant types. This overhead may be necessary, however, if the 32k limit is being exceeded during normal linking.

If you do not want to spend the extra processing time for removing redundant types, the NOTYPEMERGE command may be used. Since this is also the default operating mode of the loader, you do not need to specify NOTYPEMERGE to get this behavior. An explicit command is necessary only if you want to remove some or all redundant type information.
Either command must be used before any input executables are loaded by the loader. If input has already been read in, a warning is generated and the command is ignored.

Examples:

```
typemerge simple       ; remove only redundant simple types
typemerge all          ; remove all redundant types
notypemerge            ; don’t merge any type information
```

---

**WIDTH**

**Syntax:**

```
WIDTH n
```

**Description:**

The WIDTH command specifies the listing page width in number of characters, where n is a number less than 254. Characters outside this page width range are not printed. The default is 80 characters. Note that WIDTH can only appear in a load command file. It does not take effect until the loader finds it. Any output that may have occurred before the WIDTH command will be the default width. Since the WIDTH command cannot appear on the command line, the echo of the command line and the HP header line will always appear at their full width.

Example:

```
WIDTH 60
```
Chapter 22: Linker/Loader Listing Description

22

Linker/Loader Listing Description

Examples of loader operation.
This chapter demonstrates the operation of the loader. It contains a load command file and a load map listing produced by a load using a command file. For reference, this chapter also includes the assembly source listings for the modules that are loaded.

---

**Two-Pass Load**

The loader uses a two-pass process. During the first pass, the loader commands and object modules are checked for errors. After the loader finds an END command, a symbol table is formed.

Errors detected during the first pass of processing will be displayed on the listing. If the loader is executed in batch mode, fatal errors cause the loader to terminate with the message "LOAD NOT COMPLETED."

If the loader is executed in the interactive mode, many errors are not fatal and the loader command processing will continue. The loader will report the errors it encounters with a message immediately following the line in error, and the load will end with the message "LOAD COMPLETED."

During pass two of processing, the final absolute object module is produced, along with a module summary and a segment summary. If there are any groups present, they will also appear in the segment summary. A local symbol table, public symbol table, and cross reference table are listed in the load map if you use the options specifying their output. The load map also indicates the starting address of the load, as well as the output module name and format.

---

**Object Module Format**

The output object module can be produced in HP-OMF 86 format absolute, Intel Hexadecimal Object file format absolute, or HP 64000 format absolute. Optionally, an incremental format, HP-OMF 86 format relocatable, can be produced instead of the absolute formats. These relocatable format output modules can then be re-linked to form absolute output modules.
Loader Command File

The following figure shows the loader command file "load.k." It contains several of the loader commands described in earlier chapters.

* TEST PROGRAM FOR 8086/8087/80186 LINKING LOADER
  *
  * Note that object modules are read from the files ld86a.o,
  * ld86b.o and ld86c.o.
  *
  list t,s,x,d,c,u
  seg /code=500h
  seg /data=80000h
  order /code,comseg,/stack,/data
  resadd 5A0h,5A2h
  nlist p
  public extraneous=1000
  load ld86a.o
  list p
  load ld86b.o
  load ld86c.o
  end

Figure 22-1. The "load.k" Loader Command File
Starting the Loader

The following command is used to start the ld86 linking loader with the command file "load.k."

```
$ ld86 -c load.k -o load.x -L > load.lis
```

- The dash c option tells ld86 to use the command file "load.k."
- The dash o option tells ld86 to output the object file as "load.x."
- The dash L option tells ld86 to output a load map listing to standard output.
- The greater than sign redirects standard output to the listing file "load.lis."

The load map file later in this chapter is produced by this command line entry.

Loader Listings

The following pages show a sample loader listing. Note the following points when examining the sample loader listing.

- The first page of the sample listing shows the loader command file, the output module name and format, and warnings or errors that occur.
  - For this example, the absolute object module is produced in the default HP-OMF 86 format absolute. Object modules can also be produced in Intel Hexadecimal Object file format absolute, HP-OMF 86 format relocatable (incremental links), or HP 64000 format absolute.

- The load map file also begins on the first page. Within each summary, the width of each field expands to fit the largest number of characters needed. The line wraps if it is longer than the WIDTH setting.

- The MODULE SUMMARY information contains a listing of all modules, the name of each segment in the modules, the class of each
segment, the segment start address and end address, and a complete filename (including search path if appropriate).

- The SEGMENT SUMMARY shows the segments, the class and/or group of each segment, the segment start and end address, segment length, segment align-type, and segment combine-type. An extra three bytes is generated by the loader for processing the INITDATA command, because you have no initialized data. This is shown in the segment summary as ??DATA1/??INIT.

- The LOCAL SYMBOL TABLE lists the local symbols and the modules (and function if applicable) where they reside. The table also shows the segment where the symbol is found, the class of that segment, the absolute address of the symbol, and modules where the symbol is referenced.

- The PUBLIC SYMBOL TABLE lists the public symbols and the modules where they reside. The table also shows the segment where the symbol is found, the class of that segment, the absolute address of the symbol, and modules where the symbol is referenced.

The loader listing file follows in the next figure. Following that are the three sample assembler listing files.
Chapter 22: Linker/Loader Listing Description
Load Map Listing

Load Map Listing

Hewlett-Packard ld86  Thu Apr  1 14:51:02 1993
HPB1449-19302 A.03.10 24Mar93 Un
released Copr. HP 1988
Command line: ld86 -c load.k -o load.x -L

*  TEST PROGRAM FOR 8086/8087/80188 LINKING LOADER
*  Note that object modules are read from the files ld86a.o,
*  ld86b.o and ld86c.o.
*  list t,s,x,d,c,u
  seg /code=500h
  seg /data=80000h
  order /code,comseg,/stack,/data
  resadd 5A0h,5A2h
  nlist p
  public extraneous= 1000
  load ld86a.o
  list p
  load ld86b.o
  load ld86c.o
  end

OUTPUT MODULE NAME:  load
OUTPUT MODULE FORMAT: OMF-86

START ADDRESS:  00050:00000 -> 00500
** ERROR (308): Undefined external(s):
SYMBOL          REFERENCES
SCAN            MAIN
SEGMEMENT SUMMARY
---------------
SEGMENT/CLASS GROUP START END LENGTH ALIGN COMBINE
ASEG1/            00000 00025 00026 Abs seg Private
??DATA1/??INIT   00026 00028 00003 Byte Common
??SEG/            00030 00030 00000 Para Public
CSEG1/CODE CODEGRP 00500 00567 00048 Byte Public
CSEG2/CODE CODEGRP 00548 00591 0004A Byte Public
COMSEG/          00592 00594 00003 Byte Common
(Reserved Area)  005A0 005A2 00003

Figure 22-2. The "load.lis" Load Map File
### Module Summary

<table>
<thead>
<tr>
<th>Module</th>
<th>Segment/Class</th>
<th>Start</th>
<th>End</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>/8086/asm/lkref-list/ld86a.o</td>
<td>00500</td>
<td>00547</td>
<td>00048</td>
</tr>
<tr>
<td></td>
<td>CSEG1/CODE</td>
<td>00500</td>
<td>00547</td>
<td>00048</td>
</tr>
<tr>
<td></td>
<td>SSEG1/STACK</td>
<td>005A3</td>
<td>005B6</td>
<td>00014</td>
</tr>
<tr>
<td></td>
<td>COMSEG/</td>
<td>00592</td>
<td>00592</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td>DSEG1/DATA</td>
<td>80000</td>
<td>8004F</td>
<td>00050</td>
</tr>
<tr>
<td>ABSCODE</td>
<td>/8086/asm/lkref-list/ld86b.o</td>
<td>00592</td>
<td>00594</td>
<td>00003</td>
</tr>
<tr>
<td></td>
<td>DSEG2/DATA</td>
<td>80100</td>
<td>8010D</td>
<td>0000E</td>
</tr>
<tr>
<td></td>
<td>ASEG1/</td>
<td>00000</td>
<td>00025</td>
<td>00026</td>
</tr>
<tr>
<td>READMOD</td>
<td>/8086/asm/lkref-list/ld86c.o</td>
<td>00548</td>
<td>00591</td>
<td>0004A</td>
</tr>
<tr>
<td></td>
<td>CSEG2/CODE</td>
<td>00592</td>
<td>00592</td>
<td>00001</td>
</tr>
</tbody>
</table>

### Public Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Segment/Class</th>
<th>Address/Value</th>
<th>Module</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRLF</td>
<td>CSEG1/CODE</td>
<td>0050:003D</td>
<td>MAIN</td>
<td>READMOD</td>
</tr>
<tr>
<td>ECHO</td>
<td>COMSEG/</td>
<td>0059:0002</td>
<td>MAIN</td>
<td></td>
</tr>
<tr>
<td>EXTRANEOUS</td>
<td></td>
<td>003E:0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBUFEND</td>
<td>DSEG1/DATA</td>
<td>8000:0050</td>
<td>MAIN</td>
<td>READMOD</td>
</tr>
<tr>
<td>IN8</td>
<td>CSEG1/CODE</td>
<td>0050:0025</td>
<td>MAIN</td>
<td>READMOD</td>
</tr>
<tr>
<td>INBUF</td>
<td>DSEG1/DATA</td>
<td>8000:0000</td>
<td>MAIN</td>
<td>READMOD</td>
</tr>
<tr>
<td>MAINF</td>
<td>CSEG1/CODE</td>
<td>0050:0000</td>
<td>MAIN</td>
<td>ABSCODE</td>
</tr>
<tr>
<td>OUT8</td>
<td>CSEG1/CODE</td>
<td>0050:0032</td>
<td>MAIN</td>
<td>READMOD</td>
</tr>
<tr>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0008</td>
<td>READMOD</td>
<td>MAIN</td>
</tr>
<tr>
<td>TABLE1</td>
<td>DSEG2/DATA</td>
<td>8010:0000</td>
<td>ABSCODE</td>
<td></td>
</tr>
</tbody>
</table>

### Local Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
<th>Segment/Class</th>
<th>Address/Value</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MODULE</td>
<td>ABSCODE</td>
<td>0000:0000</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td></td>
<td>TABLE1</td>
<td>ABSCODE</td>
<td>DSEG2/DATA</td>
<td>8010:0000</td>
</tr>
<tr>
<td></td>
<td>TABLE2</td>
<td>ABSCODE</td>
<td>DSEG2/DATA</td>
<td>8010:0004</td>
</tr>
<tr>
<td></td>
<td>FIN</td>
<td>ABSCODE</td>
<td>ASEG1/</td>
<td>0000:0025</td>
</tr>
<tr>
<td></td>
<td>DT1</td>
<td>ABSCODE</td>
<td>COMSEG/</td>
<td>0059:0002</td>
</tr>
</tbody>
</table>

Figure 22-2. The "load.lis" Load Map File (Cont'd)
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FUNCTION</th>
<th>SEGMENT/CLASS</th>
<th>ADDRESS/VALUE</th>
<th>ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0000</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ10</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0008</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ20</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:001E</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ30</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0024</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ40</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:002A</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ50</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0033</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ60</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:003B</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ70</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:003F</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>READ80</td>
<td>READ</td>
<td>CSEG2/CODE</td>
<td>0054:0045</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>ECHO</td>
<td>READ</td>
<td>CSEG2/COMSEG</td>
<td>0059:0002</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>ASCR</td>
<td>READ</td>
<td>COMSEG/</td>
<td>0000:000D</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>BSPA</td>
<td>READ</td>
<td></td>
<td>0000:0008</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>BLNK</td>
<td>READ</td>
<td></td>
<td>0000:0020</td>
<td>ABS ADDRESS</td>
</tr>
<tr>
<td>TAB</td>
<td>READ</td>
<td></td>
<td>0000:0009</td>
<td>ABS ADDRESS</td>
</tr>
</tbody>
</table>

Link completed

Figure 22-2. The "load.lis" Load Map File (Cont'd)
Figure 22-3. The "ld86a.lis" Assembly Listing
Chapter 22: Linker/Loader Listing Description
First Assembler Listing

```assembly
36 0000 ; CODE STARTS HERE
37 0000
38 0000 ;
39 0000 CSEG1 SEGMENT BYTE PUBLIC 'CODE'
40 0000 MAINF LABEL FAR
41 0000 B8 00 00 R MAIN: MOV AX,DSEG1 ; SET DS, ES and SS
SEGMENT REGISTERS
42 0003 8E D8 MOV DS,AX ; AND BX AND SP AS POINTERS WITHIN SEGMENT
43 0005 BB 00 00 R MOV BX,OFFSET INBUF
44 0008 B8 00 00 R MOV AX,SSEG1
45 000B 8E D0 MOV SS,AX
46 000D BC 14 00 R MOV SP,OFFSET STAKTOP
47 0010 B8 00 00 R MOV AX,COMSEG
48 0013 8E C0 MOV ES,AX

Figure 22-3. The "ld86a.lis" Assembly Listing (Cont'd)
```

422
### NAME - OUT8

This routine is used to output a character to the terminal.

**Entry Parameters**
- DL - Character to output

**Exit Parameters**
- None

**Registers Used**
- AL, BL, DL

```assembly
OUT8   PROC
  E4 00  IN      AL, USTAT
  24 01  AND     AL, TRDY
  FA 74  JZ      OUT8
  C2 8A  MOV     AL, DL
  E6 00  OUT     UDATOUT, AL
  C3  RET
OUT8   ENDP
```

### NAME - CRLF

This routine outputs a carriage return and line feed.

**Entry Parameters**
- None

**Exit Parameters**
- None

**Registers Used**
- AL, BL, DL

```assembly
CRLF   PROC
  B2 0D  MOV     DL, ASCR
  F0 0B  CALL    OUT8
  0A 8B  MOV     DL, ASLF
  C3 04  CALL    OUT8
  C3 04  RET
CRLF   ENDP
```

**Cross Reference**

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Value</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>??SEG</td>
<td>SEGM</td>
<td>SIZE=0000 PUBLIC PARA</td>
<td></td>
</tr>
<tr>
<td>ASCR</td>
<td>EQU</td>
<td>000D</td>
<td>-34 110</td>
</tr>
<tr>
<td>ASLF</td>
<td>EQU</td>
<td>000A</td>
<td>-33 112</td>
</tr>
</tbody>
</table>

Figure 22-3. The "ld86a.lis" Assembly Listing (Cont’d)
Chapter 22: Linker/Loader Listing Description
First Assembler Listing

BINK   EQU   0020   -35  51
CODE   CLASS
CODEGRP GROUP CSEGI -6  7
COMSEG SEG M SIZE=0001 COMMON BYTE  7 -19 21 47
CRLF  PROC CSEGI:003D NEAR  -3 -109 -115
CSEGI SEG M SIZE=0048 PUBLIC BYTE CLASS 'CODE'  6 -39 116
DATA   CLASS
DSEG1 SEG M SIZE=0050 PUBLIC BYTE CLASS 'DATA'  7 -23 26 41
ECHO   PUBLIC COMSEG:0000 BYTE  -3 -20
IBUFEND PUBLIC DSEGI:0050 BYTE  -3 -25
IN8    PROC CSEGI:0025 NEAR  -3 -72 75 -80
INBUF  PUBLIC DSEGI:0000 BYTE  -3 -24 43
MAIN   LABEL CSEGI:0000 NEAR  -41 56
MAIN10 LABEL CSEGI:0018 NEAR  -50 53
OUT8   PROC CSEGI:0032 NEAR  -3 -95 98 -102 111 113
READ   EXTERN NEAR  -5 49
RRDY   EQU   0002  -32  74
SCAN   EXTERN NEAR  -5 54
SSEG1 SEG M SIZE=0014 STACK BYTE CLASS 'STACK'  7 -13 17 44
STAKTOP LOCAL SSEG1:0014 BYTE  -16 46
TRDY   EQU   0001  -31  97
UDATIN EQU   0000  -30  76
UDATOUT EQU  0000  -29 100
USTAT  EQU   0000  -28  73  96

NO ASSEMBLY ERRORS
NO ASSEMBLY WARNINGS

Figure 22-3. The '1d86a.lis' Assembly Listing (Cont’d)
Second Assembler Listing

Figure 22-4. The 'ld86b.lis" Assembly Listing

Chapter 22: Linker/Loader Listing Description
Chapter 22: Linker/Loader Listing Description

Second Assembler Listing

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>??SEG</td>
<td>SEGM</td>
<td>SIZE=0000 PUBLIC PARA</td>
</tr>
<tr>
<td>ASEG1</td>
<td>SEGM</td>
<td>SIZE=0026 ABSOLUTE AT 0000</td>
</tr>
<tr>
<td>ASEG2</td>
<td>SEGM</td>
<td>SIZE=0005 ABSOLUTE AT FFFF</td>
</tr>
<tr>
<td>COMSEG</td>
<td>SEGM</td>
<td>SIZE=0003 COMMON BYTE</td>
</tr>
<tr>
<td>DATA</td>
<td>CLASS</td>
<td></td>
</tr>
<tr>
<td>DSEG2</td>
<td>SEGM</td>
<td>SIZE=000E PAGE CLASS 'DATA'</td>
</tr>
<tr>
<td>DT1</td>
<td>LOCAL</td>
<td>COMSEG:0000 BYTE</td>
</tr>
<tr>
<td>DT2</td>
<td>LOCAL</td>
<td>COMSEG:0001 WORD</td>
</tr>
<tr>
<td>FIN</td>
<td>LABEL</td>
<td>ASEG1:0025 NEAR</td>
</tr>
<tr>
<td>MAINF</td>
<td>EXTERN</td>
<td>FAR</td>
</tr>
<tr>
<td>TABLE1</td>
<td>PUBLIC</td>
<td>DSEG2:0000 BYTE</td>
</tr>
<tr>
<td>TABLE2</td>
<td>LOCAL</td>
<td>DSEG2:0004 WORD</td>
</tr>
</tbody>
</table>

NO ASSEMBLY ERRORS
NO ASSEMBLY WARNINGS

Figure 22-4. The "ld86b.lis" Assembly Listing (Cont’d)
Chapter 22: Linker/Loader Listing Description
Third Assembler Listing

Third Assembler Listing

Hewlett Packard AS86 HPB1449-19302 A.03.10 24Mar93 Copr. HP 1988 Page 1 Thu
Apr 1 14:51:01 1993
READMOD      HPB1449-19302 A.03.10 24Mar93 Copr. HP 1988
Cmdline - as86 -o ld86c.o -L ld86c.s
Line Offset Object-Bytes
1  0000                     $XREF DEBUG
2  0000                                      NAME    READMOD
3  0000                                      PUBLIC  READ
4  0000                                      EXTRN   CRLF:NEAR
5  0000                                      EXTRN   IN8:NEAR,OUT8:NEAR
6  0000                                      CODEGRP GROUP  CSEG2
7  0000                                      DSEG1  SEGMENT BYTE PUBLIC ‘DATA’ ; PLACE
8  0000                                      DSEG1  ENDS
9  0000                                      ASSUME CS:CODEGRP,DS:DSEG1,ES:COMSEG
10 0000                                      ; DEFINE DATA AREA AND EQU’S
11 0000                                      ;
12 0000                                      ;  COMSEG SEGMENT BYTE COMMON
13 0000                                      ECHO   DB      1 DUP(?)
14 0000                                      COMSEG ENDS
15 0000                                      ;
16 0000                                      ASCR   EQU     13
17 0000                                      BSPA   EQU     8
18 0000                                      BLNK   EQU     20H
19 0000                                      TAB    EQU     09H
20 0000                                      ;
21 0000                                      ;  TERMINAL AND
22 0000                                      FOLLOWING ARR
23 0000                                      ;  SPECIAL CHARACTERS.
24 0000                                      ;   CR    - END OF CURRENT LINE
25 0000                                      ;   CONTROL-X - DELETE CURRENT LINE
26 0000                                      ;   DEL   - DELETE CHARACTER
27 0000                                      ;   ALL DISPLAYABLE CHARACTERS BETWEEN BLANK
28 0000                                      ;
29 0000                                      ;
30 0000                                      ;
31 0000                                      ;
32 0000                                      ;

Figure 22-5. The "ld86c.lis" Assembly Listing
Chapter 22: Linker/Loader Listing Description
Third Assembler Listing

AND Z AND THE
33 0000 ; ABOVE SPECIAL CHARACTERS ARE RECOGNIZED BY
THIS ROUTINE AS
34 0000 ; WELL AS THE TAB. ALL OTHER CHARACTERS ARE
IGNORED. AN
35 0000 ; ATTEMPT TO INPUT MORE CHARACTERS THAN IS
ALLOWED IN THE
36 0000 ; INPUT BUFFER WILL BE INDICATED BY A
BACKSPACE.
37 0000
38 0000 ; ENTRY PARAMETERS
39 0000 ; ECHO - ECHO FLAG, 0=NO ECHO
40 0000
41 0000
42 0000 ; EXIT PARAMETERS
43 0000 ; INBUF - CONTAINS INPUT LINE
44 0000
45 0000 ; REGISTERS USED
46 0000 ;
47 0000
48 0000
49 0000 BB 00 00 E MOV BX,OFFSET INBUF
50 0003 B9 00 00 MOV CX,0

Figure 22-5. The "ld86c.lis" Assembly Listing (Cont’d)
### Chapter 22: Linker/Loader Listing Description
#### Third Assembler Listing

```assembly
78  003D  26 80 3E 00 00 00 R       READ80: CMP ECHO,0
79  0043  74 C1                        JZ READ10
80  0045  E8 00 00                      CALL OUT8
81  0048  EB BC                        JMP READ10
82  004A  READ ENDP
83  004A  CSEG2 ENDS
84  004E  0000 ;
85  0000  END
```

Hewlett Packard AS86 HPB1449-19302 A.03.10 24Mar93 Copr. HP 1988 Page 3 Thu Apr 1 14:51:01 1993

**Cross Reference**

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Value</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>??SEG</td>
<td>SEGM</td>
<td>SIZE=0000 PUBLIC PARA</td>
<td>-19 56 59</td>
</tr>
<tr>
<td>ASCR</td>
<td>EQU</td>
<td>000D</td>
<td>-19 56 59</td>
</tr>
<tr>
<td>BLNK</td>
<td>EQU</td>
<td>0020</td>
<td>-21 71</td>
</tr>
<tr>
<td>BSPA</td>
<td>EQU</td>
<td>0008</td>
<td>-20 66</td>
</tr>
<tr>
<td>CODE</td>
<td>CLASS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODEGRP</td>
<td>GROUP</td>
<td>CSEG2</td>
<td>-6 11</td>
</tr>
<tr>
<td>COMSEG</td>
<td>SEGM</td>
<td>SIZE=0001 COMMON BYTE</td>
<td>11 -15 17</td>
</tr>
<tr>
<td>CRLF</td>
<td>EXTERN</td>
<td>NEAR</td>
<td>-4 54</td>
</tr>
<tr>
<td>CSEG2</td>
<td>SEGM</td>
<td>SIZE=004A PUBLIC BYTE CLASS 'CODE'</td>
<td>6 -47 83</td>
</tr>
<tr>
<td>DATA</td>
<td>CLASS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSEG1</td>
<td>SEGM</td>
<td>SIZE=0000 PUBLIC BYTE CLASS 'DATA'</td>
<td>-7 9 11</td>
</tr>
<tr>
<td>ECHO</td>
<td>LOCAL</td>
<td>COMSEG:0000 BYTE</td>
<td>-16 78</td>
</tr>
<tr>
<td>IBUFEND</td>
<td>EXTERN</td>
<td>DSEG1: BYTE</td>
<td>-8 76</td>
</tr>
<tr>
<td>IN8</td>
<td>EXTERN</td>
<td>NEAR</td>
<td>-5 51</td>
</tr>
<tr>
<td>INBUF</td>
<td>EXTERN</td>
<td>DSEG1: BYTE</td>
<td>-8 49</td>
</tr>
<tr>
<td>OUT8</td>
<td>EXTERN</td>
<td>NEAR</td>
<td>-5 67 80</td>
</tr>
<tr>
<td>READ</td>
<td>PROC</td>
<td>CSEG2:0000 NEAR</td>
<td>-3 -48 55 -82</td>
</tr>
<tr>
<td>READ10</td>
<td>LABEL</td>
<td>CSEG2:0006 NEAR</td>
<td>-51 58 63 79 81</td>
</tr>
<tr>
<td>READ20</td>
<td>LABEL</td>
<td>CSEG2:0012 NEAR</td>
<td>53 -56</td>
</tr>
<tr>
<td>READ30</td>
<td>LABEL</td>
<td>CSEG2:001C NEAR</td>
<td>57 -61</td>
</tr>
<tr>
<td>READ40</td>
<td>LABEL</td>
<td>CSEG2:0022 NEAR</td>
<td>-64 77</td>
</tr>
<tr>
<td>READ50</td>
<td>LABEL</td>
<td>CSEG2:0028 NEAR</td>
<td>62 -69</td>
</tr>
<tr>
<td>READ60</td>
<td>LABEL</td>
<td>CSEG2:0033 NEAR</td>
<td>70 -73</td>
</tr>
<tr>
<td>READ70</td>
<td>LABEL</td>
<td>CSEG2:0037 NEAR</td>
<td>68 72 -76</td>
</tr>
<tr>
<td>READ80</td>
<td>LABEL</td>
<td>CSEG2:003D NEAR</td>
<td>-78</td>
</tr>
<tr>
<td>TAB</td>
<td>EQU</td>
<td>0009</td>
<td>-22 69</td>
</tr>
</tbody>
</table>

**NO ASSEMBLY ERRORS**

**NO ASSEMBLY WARNINGS**

---

Figure 22-5. The "ld86c.lis" Assembly Listing (Cont’d)
Chapter 22: Linker/Loader Listing Description
Third Assembler Listing
Chapter 23: Librarian Introduction

Librarian Introduction

General operation of the librarian.
Librarian Introduction

The ar86 Librarian is used to build program libraries, or collections of relocatable object modules, that reside in a single file. These libraries are the best place to store frequently-used object modules that the linking loader can then efficiently access and combine with newly developed assembly programs. Efficiency is realized through reducing the number of files that must be opened by the loader.

The word "module," when used in discussing the librarian, refers to a relocatable object module that results from assembling a source program with the as86 cross assembler.

Starting the Librarian

There are three ways to start the ar86 Librarian.

Command Line

You may enter librarian commands on the command line. Only certain library commands can be used on the command line. They are -a, -d, -r, -e, and -L. (They are equivalent to ADDMOD, DELETE, REPLACE, EXTRACT and LIST, respectively.) All command line commands require a list as an argument and a library file name argument. These librarian commands can be entered in any order on the command line, but the librarian processes the commands in a fixed order of -a, -d, -r, -e, and -L.

Command File

You may place librarian commands in a command file to be read in batch mode. Any error that occurs during command file processing is considered fatal. The command that generated the error is skipped and processing of any remaining commands continues. These remaining commands are checked for errors—and executed, if possible—but a library file, if specified, is not generated if an error was found. The librarian processes commands in the command file in the exact order in which they are specified.
Interactive Operation

The third method enables you to enter librarian commands interactively from the terminal. In interactive mode, most librarian command errors are not fatal. When an illegal command is entered, the librarian displays an error message and provides an opportunity to re-enter the command.

Librarian Function

When writing modular programs, communication among the various modules is established through use of PUBLIC and EXTERNAL symbols. Public and external symbols can be seen as a way to pass information to the functions and receive information from the functions contained in the library modules. In addition, the functions contain entry points in the form of PUBLIC labels that can be used in CALL and JMP instructions. While it is necessary to know the entry points and parameter passing mechanisms, it is not necessary to know the name of the object module that contains the function. For instance, a library file could contain a dozen or more functions in a single module or a dozen or so functions in a dozen or so modules. As long as you know the function entry points, it does not matter to you how the modules are organized in the library.

The following example is a more practical illustration of using the library.

Suppose a programmer writes a series of program modules consisting of a number of mathematical routines including a few modules that calculate transcendental functions. These modules are then gathered into a library file through use of the ar86 Librarian.

Sometime later, a programmer, either the one who wrote the mathematical routines or someone else, has a requirement to calculate an arc-tangent function within a program being written. The programmer is aware of the fact that there is an arc-tangent function in a library file, knows the name of the entry point of the routine, knows how to pass parameters to the arc-tangent function, and knows how to accept the result of the calculation.

The programmer must do only two things:
• CALL the arc-tangent function from the program being developed, placing the public name of the entry point into the argument field of the CALL or JMP instruction, and

• Place the public entry point name of the arc-tangent function in the argument field of an external reference pseudo-op in the program being written.

Even though the programmer does not know the name of the relocatable object module that contains the arc-tangent function, the linking loader includes the relocatable module containing the correct module by informing the loader to use the required library file(s).

You do not have to specify which module contains the arc-tangent function. The loader automatically searches the named library, looking for the entry point name coded as the argument of the CALL statement. When the entry point name has been found, the loader identifies the module in which it resides, and then includes the module containing the entry point name in the current load.

The loader determines which of the library modules to load by examining the internal list of unresolved external references accumulated during the load process. It then accesses the library file to determine if there is a match between unresolved external references, and a label or name that has been declared public in the library file modules. The loader then identifies which modules contain the matching public symbols, and loads those modules just as if the programmer had explicitly directed the loader to load the proper modules.

When the inclusion of a module in the library adds an undefined reference to the list of undefined references, the loader will access the library again until all external references have been satisfied or until no more matches are possible. All public symbols within a library must have unique names.
The librarian is always case sensitive with respect to symbol names. Two symbols with the same characters are only identical if the cases of the characters match as well. This means that "SYMBOL" is not equivalent to "symbol" or "SYMbol." The librarian will treat these three symbols as unique. Trouble might arise if the loader is started with case sensitivity turned off and then asked to load a library containing, for instance, these three symbols. The first occurrence of that symbol, regardless of case, will satisfy the external reference the loader is trying to resolve. However, the resolution may be to the wrong symbol because the loader, unlike the librarian, did not consider case. The result may not be what you intended.
Chapter 23: Librarian Introduction

Librarian Function
Chapter 24: Librarian Commands

Librarian Commands

Descriptions of the librarian commands.
Chapter 24: Librarian Commands

The librarian reads a sequence of commands from the command input device in interactive or batch mode. The command sequence must be terminated by the END command. Relocatable object modules are read as input and collected in organized libraries as specified in the command input file.

Command Syntax

The librarian recognizes six special characters:

*  -  asterisk
;  -  semicolon
,  -  comma
(  -  left parenthesis
)  -  right parenthesis
+  -  plus

Use of Special Characters

The use of special characters in the command syntax is described in this section.

The asterisk ("*") and the semicolon (";"), when used on a command line, cause the librarian to ignore the rest of the line. These characters can be used to place comments in a command sequence. The librarian does not process comments; they are passed to the output file.

The comma ("," ) separates members of a list of similar elements. The list can contain module names or module filenames.

The left and right parentheses ("()"), used in pairs, denote a list of similar elements in a command. Parentheses can be used to group module names that are members of a library only.

The plus sign ("+" ) followed by a carriage return allows you to continue a list on subsequent line(s). Care should be exercised when using line continuation. Do not break up or interrupt a complete syntactical unit (for instance, do not
try to continue a filename, a module name, or a command). If the
continuation character is used immediately after a command, it must be
separated from the command by at least one blank or the librarian cannot
recognize the command. Except as noted above, the line continuation
can appear anywhere in a command line.

Also, blanks can be used freely within commands (between
syntactically-identifiable units). Example:

DELETE MOD1, MOD2 ; is the same as
DELETE MOD1, MOD2

Command File Comments

Comments can be included in a command file to document the processing.
These are included by use of the semicolon (;) or asterisk (*). Example:

; this is a complete line of comment
addmod modulea.o
; this is a command line comment
addmod moduleb
* this is another comment

File Names

File names appear in commands as arguments. A file name might be the name
of an existing library file, a library file being created, or an object file
containing one or more modules to be archived. If an open fails on a file name
that does not have a suffix, the librarian will append a "a" for a library file—or
a "o" for an object file—and again try to open the file. Similarly, if the
librarian is writing a library to a file name that does not have a suffix, it will
append a "a" to the file before writing it.

The SAVE Command

The SAVE command causes the librarian to save a library with any changes
you may have made. At the same time, the SAVE command does the
equivalent of the CLEAR command and clears the librarian. You may then
open or create another file without affecting or corrupting the previous work.
Return Codes

The librarian provides operating-system specific return codes. The librarian either completes without encountering an error, displays a message or warning, or terminates with an error.
## Commands Summary

The following commands are described in this chapter in the order shown:

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDLIB</td>
<td>Add Module(s) from Another Library</td>
</tr>
<tr>
<td>ADDMOD</td>
<td>Add Object Module(s) to Current Library</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Clear Library Session Since Last SAVE</td>
</tr>
<tr>
<td>CREATE</td>
<td>Define New Library</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete Module(s) from Current Library</td>
</tr>
<tr>
<td>DIRECTORY</td>
<td>List Library Modules</td>
</tr>
<tr>
<td>END</td>
<td>Terminate Librarian Execution</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>Copy Library Module to a File</td>
</tr>
<tr>
<td>FULLDIR</td>
<td>Display Library or Library Module Contents</td>
</tr>
<tr>
<td>HELP</td>
<td>Display Context-sensitive Command Syntax</td>
</tr>
<tr>
<td>LIST</td>
<td>Display Library or Library Module Contents (Same as FULLDIR)</td>
</tr>
<tr>
<td>OPEN</td>
<td>Open an Existing Library</td>
</tr>
<tr>
<td>QUIT</td>
<td>Terminate Librarian Executions</td>
</tr>
<tr>
<td>REPLACE</td>
<td>Replace Library Module</td>
</tr>
<tr>
<td>SAVE</td>
<td>Save Contents of Current Library</td>
</tr>
</tbody>
</table>
Shorthand Names

The librarian allows shortened forms of the above commands. The following list is the minimum characters that may be entered for the command to be recognizable. However, the librarian will accept anything from the minimum number of characters to the full command name as correct. That means that CR, CRE, CREA, CREAT, and CREATE are all acceptable for the create command. (The command is in uppercase here for clarity. Commands can be in either uppercase or lowercase.) The shorthand forms of the commands are as follows:

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>SHORTHAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDLIB</td>
<td>ADDL</td>
</tr>
<tr>
<td>ADDMOD</td>
<td>ADDM</td>
</tr>
<tr>
<td>CLEAR</td>
<td>CL</td>
</tr>
<tr>
<td>CREATE</td>
<td>CR</td>
</tr>
<tr>
<td>DELETE</td>
<td>DE</td>
</tr>
<tr>
<td>DIRECTORY</td>
<td>DI</td>
</tr>
<tr>
<td>END</td>
<td>EN</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>EX</td>
</tr>
<tr>
<td>FULLDIR</td>
<td>FUL</td>
</tr>
<tr>
<td>HELP</td>
<td>H</td>
</tr>
<tr>
<td>LIST</td>
<td>L</td>
</tr>
<tr>
<td>OPEN</td>
<td>OP</td>
</tr>
<tr>
<td>QUIT</td>
<td>Q</td>
</tr>
<tr>
<td>REPLACE</td>
<td>R</td>
</tr>
<tr>
<td>SAVE</td>
<td>S</td>
</tr>
</tbody>
</table>
The ar86 librarian archives modules into library files. The librarian references modules by module names. Modules are contained in object files created by the as86 assembler, but a module name may not be the same as the object file name because a module may be explicitly named within the assembly code using the NAME assembler directive. It may be entirely different than the file name. If a module is not explicitly named, then the module name defaults to the assembly source file name stripped of its leading path name and trailing suffix (including the period) if the suffix exists.

Module names are of no concern once the library has been built, but each module name must be unique. Therefore, when trying to add modules to a library, it is possible to have module name conflicts. Since the librarian will not allow duplicate module names within a library, it may be necessary to re-assemble the module to change its name. Merely changing the object file name will not change the module name because that information is coded into the object file.

To the librarian, module names are always case sensitive, regardless of how the assembler was started.

In the references for the library commands, square brackets (‘[]’) indicate optional arguments. Square brackets containing an ellipsis denote that the preceding argument can be repeated zero or more times.
Chapter 24: Librarian Commands

ADDLIB

ADDLIB

Syntax:

ADDLIB library_filename[(module_name[,...])]}

Where:

library_filename is the name of the library where the modules reside.

module_name is the name(s) of the relocatable object module(s) to be added to the library currently being created or modified.

Description:

The ADDLIB command is used to add one or more object modules from one library to the library currently being created or modified.

The OPEN or CREATE command must precede the ADDLIB command, and name the library to which the object modules will be added. Example:

OPEN library1.a
ADDLIB math.a (square,sqroot) ;math.a contains modules
;to be added to library1.a

ADDMOD

Syntax:

ADDMOD filename [,...]

Where:

filename is the file to be added to the library currently being created or modified.

Description:

The ADDMOD command adds a non-library file containing one or more relocatable object modules to the library named in the OPEN or CREATE command. The OPEN or CREATE command must precede the ADDMOD command. Example:
OPEN    library2.a
ADDMOD  math.mbr

---

### CLEAR

**Syntax:**

```plaintext
CLEAR
```

**Description:**

Use the CLEAR command to clear the current library session since the last SAVE (or since entering the librarian if no SAVE has been entered since librarian startup). Using CLEAR is equivalent to re-starting the librarian.

---

### CREATE

**Syntax:**

```plaintext
CREATE  library_name
```

**Where:**

`library_name` is the name of the library file being created. If the file name already exists, an error occurs.

**Description:**

Use the CREATE command to define a new library. You can create only one library at a time. A newly-created library must be saved before a second one is created.

In the interactive mode, if the library file name already exists, a warning is displayed. In the command line mode, if the library file already exists, the librarian issues an error message. No library is created. Example:

```plaintext
CREATE math.a
```
Chapter 24: Librarian Commands

DELETE

Syntax:

```
DELETE module_name [,..]
```

Where:

- `module_name` is the name of the module to be removed from the library currently being created or modified.

Description:

The DELETE command removes one or more relocatable object modules from the library named in the OPEN or CREATE command. Object module names are case-sensitive. An OPEN or CREATE command must precede DELETE.

DIRECTORY

Syntax:

```
DIRECTORY library_name
    [(module_name [,..]]) [list_filename]
```

Where:

- `library_name` is the name of the library whose module names and sizes are to be listed.
- `module_name` is the name of a specific module in the library file whose size is to be listed.
- `list_filename` is the file where the directory information should be written. If the listing output file is not specified, the output defaults the standard list device (usually the terminal).
The DIRECTORY command lists module names and sizes of the modules in the specified library. The sizes listed are the number of bytes required to store the modules on the host computer system. If you enter just the library_name, all modules are listed; if you enter specific module_names, directory information is displayed for the named modules only. You can include full file specification (including pathname) for the desired library directory. Object module names are case-sensitive. The directory displays on the standard output device, or it can be directed to a file.

Syntax:

```
DIRECTORY
```

END

**Syntax:**

```
END
QUIT
```

**Description:**

The END and QUIT commands terminate librarian command processing. No library file is implicitly saved.

**Note**

Because END and QUIT do not implicitly save the library file, you must issue a SAVE command before issuing an END command or the library you are working on will be lost. END will not issue a warning that will tell you to save your library before ending.

EXTRACT

**Syntax:**

```
EXTRACT module_name [, ...]
```

```
Chapter 24: Librarian Commands

FULLDIR
LIST

Where:

module_name is the module to be copied from the library currently being created or modified.

Description:
The EXTRACT command copies a library module to a file outside the library. The file name will be the module name with a ".o" (dot o) appended. The file can then be added to another library. An OPEN or CREATE command must precede the EXTRACT command.

FULLDIR
LIST

Syntax:

FULLDIR library_name
   [ (module_name[,...]) ] [ list_filename ]

LIST library_name
   [ (module_name[,...]) ] [ list_filename ]

Where:

library_name is the library file whose contents are to be listed.

module_name is the name of a specific module whose contents will be listed.

list_filename is the output listing filename. If you do not enter the filename, the output defaults to the standard list device (usually the terminal).

Description:
The FULLDIR and LIST commands are used to request a full directory display of a library's contents including module names, their sizes, and all public symbol definitions and external references. The sizes listed are the number of bytes required to store the modules on the host computer system.

If you enter just the library_name, the contents of all modules are listed; if you enter specific module_names, information is displayed for the named modules only. Both commands perform the same operation.
Chapter 24: Librarian Commands

HELP

Syntax:

HELP

Description: The HELP command is used to obtain a list of commands with the correct invocation syntax. HELP is context-sensitive. The commands displayed are only those that can be legally entered at the time you type HELP.

OPEN

Syntax:

OPEN library_name

Where: library_name is the name of the library file to be opened.

Description: The OPEN command enables an existing library to be referenced in conjunction with succeeding commands that add modules, delete modules, or replace modules. Only one library can be opened at a time.

If the librarian commands require creation of a new version of the library, the old version will be overwritten when the SAVE command is issued.

If the library cannot be located or opened for input, an error is reported. In batch mode or command-line entry, execution is terminated.
Chapter 24: Librarian Commands

REPLACE

Syntax:

REPLACE file_name [, ...]

Where:

file_name  the file containing one or more modules that will replace the module of the same name in the library currently being created or modified.

Description:

The REPLACE command is used to replace one or more library modules with one or more non-library object modules with the same name. The replacement object modules (there may be more than one in the object file) must have the same names as the library modules they replace. REPLACE must be preceded by an OPEN or CREATE command.

SAVE

Syntax:

SAVE

Description:

The SAVE command saves the contents of the library being created or modified. When a SAVE command is issued, all the librarian commands issued since the library was CREATED or OPENed are executed, the library is modified accordingly, and it is written to the library file. If an old version of the library already exists, it will be overwritten by the new version at this time. No backup of the old library file is made. Until a SAVE command is issued, librarian commands are only checked for form, content, and syntax.

Note

You must use the SAVE command to explicitly save the library file. If you END the command session without a SAVE, the library file is not modified and all the changes that you specified will be lost.
Chapter 25: Librarian Listing Description

Librarian Listing Description

Example librarian command files.
This chapter shows example librarian command files and librarian listings to illustrate the input command file invocation and the information that can be produced about the library being created or modified. The assembly listings for the object modules that are being archived do not appear in this chapter.

Unless a LIST command or the -L appears in the command stream, only error messages and commands are echoed to standard output. If a LIST command is used in command file batch mode or in interactive mode, a listing that gives more information about the library can be produced. The -L option in command line mode also produces such a listing.

The listing shows each module name, the public and external definitions for each module, the size of each module in bytes, and a count of the public and external definitions for each module. After all modules are listed, the listing gives the number of modules in the library and may report any errors that might have occurred. The listing may also show any loader commands from interactive or command file batch mode execution.
Librarian Sample 1

In librarian sample 1, a new library, libcmd1.a, is created. Three modules (modu1.o, modu2.o, and modu3.o) are added to it. The contents of the library are then listed. The librarian command file libcmd1 is shown in the following figure.

    cr libcmd1.a
    addmod modu1.o
    addmod modu2.o
    addmod modu3.o
    list libcmd1.a

Figure 25-1. The "libcmd1" Librarian Command File

The librarian is started in batch mode with a command file in the following way:

    $ ar86 < libcmd1 > libcmd1.lis

ar86 is started interactively, receives input redirected from the command file, and produces a listing in libcmd1.lis by redirecting standard output. The libcmd1.lis listing file appears in the next figure.
Chapter 25: Librarian Listing Description
Librarian Sample 1

Hewlett-Packard AR86 Wed Nov 2 11:05:44 1988

```
kr libcmd1.a 
addmod modu1.o 
addmod modu2.o 
addmod modu3.o 
list libcmd1.a 
```

Hewlett-Packard AR86 Wed Nov 2 11:05:45 1988

Library being built libcmd1.a

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODU1</td>
<td>424</td>
</tr>
</tbody>
</table>

****** PUBLIC DEFINITIONS ******
MODU1TEN MODU1SIX
MODU1NINE MODU1FIVE

****** EXTERNAL REFERENCES ******
MODU1ONE MODU1TWO
MODU1THREE MODU1FOUR
MODU1SEVEN MODU1EIGHT

Public Count = 4
External Count = 6

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODU2</td>
<td>428</td>
</tr>
</tbody>
</table>

****** PUBLIC DEFINITIONS ******
MODU2SIX MODU2NINE
MODU2FIVE MODU2TEN

****** EXTERNAL REFERENCES ******
MODU2ONE MODU2TWO
MODU2THREE MODU2FOUR
MODU2SEVEN MODU2EIGHT

Figure 25-2. The "libcmd1.lis" Librarian Listing
Public Count = 4
External Count = 6

Module Size
MODULE3 ... 436

***** PUBLIC DEFINITIONS *****

MODU3NINE MODU3FIVE
MODU3TEN MODU3SIX

***** EXTERNAL REFERENCES *****

MODU3ONE MODU3TWO
MODU3THREE MODU3FOUR
MODU3SEVEN MODU3EIGHT

Public Count = 4
External Count = 6

Module Total = 3

save
end

Figure 25-2. The 'libcmd1.lis' Library Listing (Cont'd)
In librarian sample 2, a new library, `libcmd2.a`, is created. Four modules (modu1.o, modu2.o, modu3.o, and modu4.o) are added to it. However, one of the modules, modu4.o, does not exist. Two things occur because it does not. It causes an error to be generated that says it does not exist and the library is not created because the error occurred in batch mode. A listing is still produced. It lists the modules that did exist and the information about them just as it appears in the first sample, but it also contains a message that informs you the library was not created.

For this sample, the librarian is started in command line batch mode in the following way:

```
ar86 -L -a modu1.o,modu2.o,modu3.o,modu4.o libcmd2.a  > libcmd2.lis
```

- The dash L option specifies a listing.
- The dash a option directs the loader to add the file list that follows the -a.
- `libcmd2.a` is the library to be created.
- The greater than sign redirects the listing to the file `libcmd2.lis`.

In addition to the errors reported in the file, a duplicate set of errors are reported to the terminal. The listing appears in the next figure.
Figure 25-3. The 'libcmd2.lis' Library Listing
Chapter 25: Librarian Listing Description
Librarian Sample 2

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULE3 ...</td>
<td>436</td>
</tr>
</tbody>
</table>

***** PUBLIC DEFINITIONS *****

MODU3NINE MODU3FIVE
MODU3TEN MODU3SIX

***** EXTERNAL REFERENCES *****

MODU3ONE MODU3TWO
MODU3THREE MODU3FOUR
MODU3SEVEN MODU3EIGHT

Public Count = 4
External Count = 6

Module Total = 3

(253) Library libc2d.a not written.

Warnings = 1
Errors = 1

Figure 25-3. The "libc2d.lis" Library Listing (Cont’d)
Part 4

Error Messages Reference
Chapter 26: Error Message Formats

Error Message Formats
There are three classes of errors that may occur during assembler, macro preprocessor, linker, or librarian execution:

**Warning**

Warnings announce something that might be a problem in the output file. This may or may not indicate a problem with the program.

After a warning, the output files are written normally.

After a warning, as86, ap86, ld86, and ar86 return a code indicating "success" so that command files and "make" operations continue normally.

**Error**

Errors announce something that is wrong in the output file. For example, a reference to an unresolved symbol will cause problems at run-time.

After an error, the output files are written normally. The output files are complete and may be useful in subsequent operations.

After an error, an "error" code is returned so that command files and "make" operations stop.

**Fatal Error**

A fatal error announces a condition that causes processing to be discontinued. After a fatal error, the output files are incomplete and corrupt. They are not useful for subsequent operations.

After a fatal error, an "error" code is returned so that command files and "make" operations stop.

---

**Interactive and Non-Interactive Conditions**

Some conditions produce either warnings or errors, depending on whether the tool is run in interactive or batch mode. In interactive mode, a particular condition causes a warning because the user has a chance to reissue the command correctly. In batch mode, the same condition causes an error.
For example, suppose the file `tt2.o` does not exist and that `lib.a` does exist. If we invoked the librarian in batch mode as follows:

```bash
$ ar86 -a "tt2.o" lib.a
```

We would see an error.

```
< ar86 >
   (101) unable to open file tt2.o.
   ERROR:  (104) file tt2.o not included.
            (253) Library lib.a not written.
```

`ar86` would terminate and return you to the system prompt.

However, in the interactive mode, if you type the following command:

```
ar86> addmod tt2.o
```

You would see a warning.

```
(101) unable to open file tt2.o.
   WARNING: (104) file tt2.o not included.
```

`ar86` would then again display its command prompt and allow you to continue.
Chapter 26: Error Message Formats
Interactive and Non-Interactive Conditions
Chapter 27: Assembler Error Messages

Assembler Error Messages
When the assembler encounters a syntax error, it does not generate code for the instruction or directive on the line and any of its continuation lines where the error occurs. The error message is printed on the line below the error, with a caret (^) pointing to the offending syntax.

In some cases, the assembler issues a general syntax error that indicates there is something wrong at the place the caret points, but the specific nature of the error is not determined.

In the event of a syntax error, the assembler does not generate code, but continues processing with the next statement.

### Syntax Errors

**500**  
**Expecting an expression.**  
The assembler expected an expression, but found something different at the location pointed to by the caret.

**501**  
**Expecting an OR-level expression.**  
The assembler expected an OR-level expression, but found something different at the location pointed to by the caret.

OR-level expressions include all the AND-level expressions plus the OR and XOR operators.

**502**  
**OR or XOR expected.**  
The assembler expected an OR or XOR operator, but found something different at the location pointed to by the caret.

**503**  
**Expecting an AND-level expression.**  
The assembler expected an AND-level expression, but found something different at the location pointed to by the caret.

AND-level expressions include all the NOT-level expressions, and the AND operator.

**504**  
**AND expected.**
Chapter 27: Assembler Error Messages
Syntax Errors

The assembler expected an AND operator, but found something different at the location pointed to by the caret.

505  **Expecting a NOT-level expression.**

The assembler expected a NOT-level expression, but found something different at the location pointed to by the caret.

506  **Expecting a relational operator-level expression.**

The assembler expected a relational-level expression, but found something different at the location pointed to by the caret.

Relational-level expressions include all binary addition-level expressions plus the EQ, NE, LT, LE, GT, and GE operators.

507  **Expecting a relational operator.**

The assembler expected a relational operator, but found something different at the location pointed to by the caret.

The relational operators are: EQ, NE, LT, LE, GT, and GE.

508  **ENDS or constant definition directive expected.**

The assembler expected to find an ENDS directive but found something different at the location pointed to by the caret.

509  **Expecting an addition operator.**

The assembler expected an addition operator, but found something different at the location pointed to by the caret.

The addition operators are plus (+) and minus (—).

510  **Expecting a multiplication-level expression.**

The assembler expected a multiplication-level expression, but found something different at the location pointed to by the caret.

Multiplication-level expressions include all

- byte-level expressions
- MOD, SHR, SHL
- multiplication and division operators
Chapter 27: Assembler Error Messages
Syntax Errors

- base registers (BX, BP) and index registers (SI, DI)

511 Expecting a multiplication operator.
The assembler expected a multiplication operator, but found something different at the location pointed to by the caret.
The multiplication operators are MOD, SHR, SHL, and multiplication (*) and division (/).

512 Expecting a valid argument to NAME.
The assembler expected a valid module name argument to the NAME directive, but found something different at the location pointed to by the caret.
Byte-level expressions include all secondary-level expressions plus the HIGH and LOW operators.

513 Expecting a secondary-level expression.
The assembler expected to find a secondary-level instruction but found something different at the location pointed to by the caret. Secondary-level expressions include all primary-level expressions, the segment override (colon), the PTR, OFFSET, SEG, and TYPE operators.

514 Expecting a primary-level expression.
The assembler expected a primary-level expression, but found something different at the location pointed to by the caret.
Primary-level expressions include all expression primitives as well as the MASK, WIDTH, SIZE, and LENGTH operators, and the dot operator for structures.

516 Expecting a symbolic name.
The assembler expected a symbolic name, but found something different at the location pointed to by the caret.

517 Expecting an integer constant.
The assembler expected an integer constant, but found something different at the location pointed to by the caret.

520 Expecting a register.
Chapter 27: Assembler Error Messages
Syntax Errors

The assembler expected a register (such as AX, BX, BP, SI, and others) but found something different at the location pointed to by the caret.

521  
Segment register expected.

The assembler expected a segment register (CS, DS, ES, or SS) but found something different at the location pointed to by the caret.

522  
NOTHING or segment register expected.

The assembler expected the keyword NOTHING or a segment register (CS, DS, ES, or SS) but found something different at the location pointed to by the caret.

523  
Expecting an identifier or integer constant.

The assembler expected an identifier or integer constant, but found something different at the location pointed to by the caret.

524  
Expecting identifier, directive, or colon.

The assembler expected an identifier, directive, or colon, but found something different at the location pointed to by the caret.

525  
Expecting an identifier or constant definition directive.

The assembler expected an identifier or constant definition directive (such as DB, DW, DD, and others) but found something different at the location pointed to by the caret.

526  
Expecting an identifier or type.

The assembler expected an identifier or type, but found something different at the location pointed to by the caret.

527  
SEGMENT expected.

The assembler expected a segment, but found something different at the location pointed to by the caret.

528  
PTR expected.

The assembler expected a PTR operator, but found something different at the location pointed to by the caret.

529  
DUP expected.
The assembler expected DUP, but found something different at the location pointed to by the caret.

530  **Expecting a comma.**

The assembler expected a comma, but found something different at the location pointed to by the caret.

531  **Expecting a colon.**

The assembler expected a colon, but found something different at the location pointed to by the caret.

532  **Expecting a period, left bracket, or left angle bracket.**

The assembler expected a period (.), left bracket ([), or left angle bracket (\), but found something different at the location pointed to by the caret.

533  **Expecting right bracket.**

The assembler expected a right bracket, but found something different at the location pointed to by the caret.

534  **Expecting a left parenthesis.**

The assembler expected a left parenthesis, but found something different at the location pointed to by the caret.

535  **Dollar sign expected.**

The assembler expected a dollar sign '$', but found something different at the location pointed to by the caret.

536  **Expecting comma or right angle bracket.**

The assembler expected a comma or right angle bracket, but found something different at the location pointed to by the caret.

537  **Expecting comma or right parenthesis.**

The assembler expected a comma or right parenthesis, but found something different at the location pointed to by the caret.

538  **Expecting a left bracket.**

The assembler expected a left bracket, but found something different at the location pointed to by the caret.
Chapter 27: Assembler Error Messages
Syntax Errors

539  **Expecting a right parenthesis.**
The assembler expected a right parenthesis, but found something different at the location pointed to by the caret.

540  **Expecting a label or a statement.**
The assembler expected a label or a statement, but found something different at the location pointed to by the caret.

541  **Expecting an instruction mnemonic.**
The assembler expected an instruction mnemonic, but found something different at the location pointed to by the caret.

543  **Assembler general control expected.**
The assembler expected a general control, but found something different at the location pointed to by the caret.

544  **Expecting an assembler control.**
The assembler expected an assembler control, but found something different at the location pointed to by the caret.

545  **Constant definition directive expected.**
The assembler expected a constant definition directive such as DB, DW, DD, and others, but found something different at the location pointed to by the caret.

546  **Unexpected control or directive name, or missing END directive.**
An illegal primary control or directive was found at the location pointed to by the caret or an END directive was not found before the end of the source file.

547  **Expecting a string.**
The assembler expected a string, but found something different at the location pointed to by the caret.

548  **Expecting parenthesized text.**
The assembler expected a valid attribute to the SEGMENT directive, but found something different at the location pointed to by the caret.

549  **Expecting valid attribute to the SEGMENT directive.**
Chapter 27: Assembler Error Messages
Syntax Errors

The assembler expected to find an alignment type such as BYTE, PARA, INPAGE, and others, but found something different at the location pointed to by the caret.

550

**Expecting a combine type.**

The assembler expected a combine type (PUBLIC, STACK, COMMON, and others) but found something different at the location pointed to by the caret.

551

**Continuation line found where initial line was expected.**

The assembler found the continuation character (ampersand ['&']) as the first character on a line that it was expecting to begin rather than to continue with an assembly statement.

552

**Logical end of program already encountered.**

Assembler statements, directives, or controls were found in a source file AFTER an END directive was encountered. The only legal input after an END directive are comment lines or blank lines.

554

**Structure or record initialization expected.**

The assembler expected to encounter a left angle bracket, but found something different at the location pointed to by the caret.

555

**Record field initialization expected.**

The assembler expected to encounter an equal sign, but found something different at the location pointed to by the caret.

556

**Expecting a valid member of a GROUP.**

The assembler expected a valid member of a GROUP (such as a segment name), but found something different at the location pointed to by the caret.

557

**Expecting an item which can be purged.**

The assembler expected an item that can be purged (such as symbolic names, instructions, and others), but found something different at the location pointed to by the caret.

558

**Expecting a valid END initialization element.**

The assembler expected a valid END initialization element, but found something different at the location pointed to by the caret.
Chapter 27: Assembler Error Messages
Syntax Errors

559 Expecting a valid ASSUME element.
The assembler expected a valid ASSUME element, but found something different at the location pointed to by the caret.

561 Expecting valid CODEMACRO parameter information.
The assembler expected to find valid CODEMACRO parameter information but found something different at the location pointed to by the caret.

562 Expecting a codemacro parameter specifier.
The assembler expected to find a codemacro parameter specifier but found something different at the location pointed to by the caret.

563 This statement is not valid in a codemacro definition.
The caret points to a statement that is not legal in the body of a codemacro definition.

564 Expecting a type.
The assembler expected a Type (such as BYTE, WORD, DWORD, and others) but found something different at the location pointed to by the caret.

565 Unbalanced string delimiters.
A string that was opened with an apostrophe or quotation mark does not have a closing apostrophe or quotation mark. Usually this is caused by failing to double occurrences of apostrophes or quotation marks that are contained in the text of the string.

566 Syntax error.
In some cases, the assembler can determine that there is a syntax error, but can’t determine exactly what the error is. In these cases, this general message is generated, with the caret indicating the point of the error.

567 Syntax error in command line options.
Control options on the command line may only be delimited with spaces, tabs, or commas. Also, any arguments to controls must be delimited with parentheses.

568 Unbalanced parentheses.
Chapter 27: Assembler Error Messages
Syntax Errors

The number of right parentheses in the line does not match the number of left parenthesis. In complicated continued expressions, this could be due to the following line not having its continuation character in the first column.

569

**Illegal operand for unary MINUS or NOT.**

Neither the unary minus nor the NOT operator can have a relocatable operand. The operand pointed to by the caret is relocatable.

570

**Expecting a unary addition-level expression.**

The assembler expected to find a unary addition-level expression, but found something different at the location pointed to by the caret. Unary addition-level expressions include all of the multiplication level expressions as well as unary plus and minus.

571

**Additional information encountered beyond end of statement.**

After reaching what it thought was the logical end of a statement, the assembler found additional text at the location pointed to by the caret.

572

**Expecting decimal or hexadecimal floating-point constant.**

The assembler expected to find a decimal or hexadecimal floating-point constant at the location pointed to by the caret.

573

**Expecting a signed integer constant.**

The assembler expected to find an integer constant with or without a leading unary plus or minus, but found something different at the location pointed to by the caret.

574

**Expecting a SHORT-level expression.**

The assembler expected to find a SHORT-level expression but found something different at the location pointed to by the caret. SHORT-level expressions include all of the OR-level expressions as well as the SHORT operator.

575

**Expecting an argument to an instruction or codemacro.**

The assembler expected to find an argument to an instruction or codemacro, but found something different at the location pointed to by the caret.

600

**Illegal or mismatched argument.**
Chapter 27: Assembler Error Messages
Syntax Errors

The caret points to the place where the operand type is incorrect for the instruction, or where the type doesn’t match up correctly with another of the operands in the instruction.

601 Anonymous memory type.

The size of the operand pointed to by the caret cannot be determined from the operand’s expression, or from the content of other operands in the instruction.

602 Illegal type of expression.

The expression pointed to by the caret is either not allowed in the directive or in the instruction in which it is specified, or the expression is not a valid expression.

603 Illegal type of argument in expression.

The operator that precedes or follows the sub-expression being pointed to by the caret does not allow this type of sub-expression or one of its operands. Certain operators (such as * or /) allow only sub-expressions that resolve to an absolute number as an operand. Other operators only allow non-absolute expressions when certain conditions exist (see the description of ‘-‘ and relational operators).

604 Illegal or duplicate memory argument.

Only one argument that references a memory location is allowed in any given instruction.

605 This instruction requires at least one operand.

More than one operand had been supplied to this instruction, when only one operand is allowed.

606 This instruction requires at least two operands.

Less than two operands (or more than two) have been supplied to this instruction; two are required.

607 This instruction requires three operands.

Less than three operands have been supplied to this instruction; three are required.

608 Duplicate declaration of symbolic name.
Chapter 27: Assembler Error Messages
Syntax Errors

The symbolic name, pointed to by the caret, has already been declared in a previous statement.

609  **Duplicate specification of module name.**
This message occurs when more than one NAME directive appears in the source program.

610  **Duplicate occurrence of base register in register expression.**
Only one base register (BX or BP) may be used in any given register expression.

611  **Duplicate occurrence of index register in register expression.**
Only one index register (SI or DI) may be used in any given register expression.

612  **This symbol is not defined as a label.**
The caret points to a symbol, in a directive or expression, that must be a label. The symbol pointed to by the caret is not a label.

613  **This symbol is not defined as a segment or group.**
The caret points to a symbol, in a directive or expression, that must be a segment name or group name. The symbol pointed to by the caret is not a segment or group name.

614  **This symbol is not defined as a variable.**
The caret points to a symbol, in a directive or expression, that must be a variable. The symbol pointed to by the caret is not a variable.

615  **This symbol is not defined as a structure.**
The caret points to a symbol, in a directive or expression, that must be a structure. The symbol pointed to by the caret is not a structure.

616  **This symbol is not defined as a structure field.**
The caret points to a symbol, in a directive or expression, that must be a structure field. The symbol pointed to by the caret is not a structure field.

617  **This symbol is not defined as a structure or record.**
The caret points to a symbol, in a directive or expression, that must be a structure or record. The symbol pointed to by the caret is not a structure or record.

618  This symbol is not defined as a record field.

The caret points to a symbol in a directive or expression that is required to be a record field in order to be valid. The symbol pointed to by the caret is not of this kind.

619  This symbol is not defined as a segment.

The caret points to a symbol, in a directive or expression, that must be a segment. The symbol pointed to by the caret is not a segment.

620  Alignment type inconsistent.

The alignment type specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

621  Combine type inconsistent.

The combine type specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

623  Illegal or premature termination of segment.

This error indicates improper nesting of segments or a misspelling of the segment name in either the SEGMENT or ENDS directives.

624  Segment nesting level exceeded.

Segments can be nested to a level of 16 only.

625  Missing SEGMENT directive or previous segment nesting error.

This ENDS directive has no associated SEGMENT directive, either due to omission or to a nesting error on its associated SEGMENT directive.

626  Expecting alignment type, combine type, or classname.

The assembler expected an alignment type, combine type, or classname, but found something different at the location pointed to by the caret.

627  Classname inconsistent.
Chapter 27: Assembler Error Messages
Syntax Errors

The classname specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

628  Illegal type of symbol in this ASSUME.
This error occurs when a symbol other than a segment or group is used in an ASSUME directive without being preceded by the SEG operator.

629  Initialization nest level exceeded.
When using the DUP construct in conjunction with a data directive (DB, DW, DD, DQ, or DT), the maximum nesting level for DUPs is eight.

630  This symbol does not have a defined segment value, or segment not addressable.
The symbolic name pointed to by the caret does not have a segment attribute in the list of legal attributes.

631  This argument does not have a defined offset value.
The symbolic name pointed to by the caret does not have an offset attribute in the list of accepted attributes.

632  This argument does not have a defined type value.
The symbolic name pointed to by the caret does not have a type attribute in the list of accepted attributes.

633  This argument does not have a defined length value.
The symbolic name pointed to by the caret does not have a length attribute in the list of accepted attributes.

634  This argument does not have a defined size value.
The symbolic name pointed to by the caret does not have a size attribute in the list of accepted attributes.

635  This argument does not have a defined field width value.
The symbolic name pointed to by the caret does not have a field width attribute in the list of accepted attributes.

636  This argument does not have a defined mask value.
Chapter 27: Assembler Error Messages
Syntax Errors

The symbolic name pointed to by the caret does not have a mask attribute in the list of accepted attributes.

637 Immediate value overflow.
The immediate value is not within the proper range for its context. Specifically, it is not within the range 0 to 0FFH for DB, 0 to 0FFFFH for DW or an instruction, and 0 to 0FFFFFFFFH for all others.

638 This expression must be absolute.
The expression must resolve to an absolute number to be permissible in this context.

639 Item cannot be addressed by segment registers.
The segment associated with the variable pointed to by the caret is not currently ASSUMEd into any of the segment registers, nor has an explicit segment override been used.

641 Invalid floating point constant
The floating point constant pointed to by the caret is not a valid floating point constant. No valid floating-point value can be stored for this constant.

642 Illegal operand in this register expression.
Register expressions may contain a base register (BX or BP), an index register (SI or DI), and any expression that evaluates to an absolute value. Expressions or symbols with relocatable results are not permitted.

643 Division by zero attempted.
The divisor portion of this expression involving the division operator is itself an expression that evaluates to an absolute number with a value of 0.

645 This relational operator has an invalid operand or operands.
See the description of relational operators for what operands are valid.

648 Hexadecimal real constants are invalid in this context.
Hexadecimal real constants are allowed only in data definition statements or EQU definitions.

649 Illegal floating-point stack register (0-7 allowed).
Chapter 27: Assembler Error Messages
Syntax Errors

A mnemonic representing an 8087 floating-point stack register was not in the legal list of mnemonics (ST,ST(0),ST(1),...,ST(7)).

650 **Value too large for one-byte displacement.**
The number (or expression that evaluates to an absolute number) is pointed to by the caret is either less than -128 or greater than 255, and thus cannot be represented in just one byte.

651 **Hex real constant size does not match with data directive.**
Hex real constants must be eight significant hex digits for the DD directive, sixteen significant digits for the DQ directive, and twenty significant digits for the DT directive.

653 **This symbol cannot be purged.**
The following kinds of symbols cannot be purged:
- keywords
- segment names (including ??SEG)
- group names
- any user-defined symbol that has appeared in a PUBLIC statement

654 **Symbol cannot be declared PUBLIC.**
PUBLIC symbols must be variables, labels or 17-bit constants; any other types will generate an error.

655 **This symbol cannot be a member of a group.**
Only segments, externals, or variables may be used in a GROUP directive. Only a segment may be forward referenced.

656 **Illegal statement in this context.**
This error is generated if a PROCLEN directive appears outside of a CODEMACRO definition, a STRUC statement appears within a structure definition, or if a structure initialization occurs within another structure initialization.

658 **Illegal or premature termination of procedure.**
This error indicates improper nesting of procedure or a misspelling of the procedure name in either the PROC or ENDP directives.

659

**Procedure nesting level exceeded.**

Procedures can be nested to a level of 16 only.

660

**Illegal type in this context.**

This error is generated if a type other than NEAR or FAR appears in a PROC directive, or if a type other than a standard type (e.g. a structure or record name) appears as the argument to the THIS operator.

661

**Illegal termination of structure.**

This error indicates a misspelling of the structure name in either the STRUC or ENDS directives.

662

**Null initialization is not allowed in this context.**

Null (or default) initialization is permitted only in structure or record initialization, not in structure or record definition or data definition directives.

663

**Invalid record field size.**

A given field within a record can be no larger than 16 bits, or no smaller than 1 bit.

664

**Maximum record size exceeded.**

The size of a record is limited to 16 bits.

666

**This variable is not defined as a record.**

The caret points to a symbol, in a directive or expression, that must be a record. The symbol pointed to by the caret is not a record.

667

**Include file nesting limit exceeded.**

The limit for nested include files has been exceeded. This limit is operating system specific.

668

**Cannot open include file.**
Chapter 27: Assembler Error Messages
Syntax Errors

The filename specified in the preceding include control is misspelled, the associated file is not in the current directory, or the associated file cannot be opened.

669  **Illegal type of EQU in this context.**

An example of this error is an EQU to an 8086 instruction mnemonic as the expression portion of a data definition directive, such as DB. Many other similar conditions exist that will generate this error.

670  **Too many arguments specified for this instruction.**

The particular instruction pointed to by the caret does not allow as many arguments as are specified. INC AX,BX, for example, has one too many arguments.

671  **This type of segment override is illegal in this context.**

Certain types of expressions are not permitted to have a segment override operator (colon operator) as part of the expression. The expression pointed to by the caret is one such expression.

672  **Illegal value for PAGELENGTH control.**

The minimum value in the PAGELENGTH control is 20 lines.

673  **Illegal value for PAGEWIDTH control.**

The legal values for the PAGEWIDTH control fall in the range of 41 to 255 columns, inclusive.

674  **Illegal value for TITLE control.**

The string for a TITLE control is limited to a length of 40 characters.

675  **More than 64 levels of control saves.**

The $SAVE control cannot be nested to a depth greater than 64.

676  **More than 64 levels of control restores.**

The $RESTORE control cannot be nested to a depth greater than 64.

677  **This symbol is not a parameter to this codemacro.**

The symbol pointed to by the caret, which is contained within a codemacro definition, is not present in the CODEMACRO statement for the current
codemacro. Therefore, the symbol cannot be a parameter to the current codemacro.

678

This symbol is not defined as a codemacro parameter.

The caret points to a symbol in a directive or expression that is required to be a codemacro parameter in order for it to be valid. The symbol pointed to by the caret is not a codemacro parameter.

679

This codemacro parameter’s specifier is invalid in this context.

Certain directives within a codemacro definition allow only parameters that have specific types of codemacro specifiers. The codemacro parameter pointed to by the caret is not of the specific type needed for the directive in which it is used.

680

Illegal range expression in codemacro parameter definition.

Either the range expression pointed to by the caret does not evaluate to an absolute number, or it is out of range according to the codemacro specifier with which it is associated.

681

This symbol is not a valid codemacro specifier.

The symbol pointed to by the caret is not one of the valid codemacro specmod fields listed on page 223.

682

Duplicate definition of codemacro parameter.

The symbol pointed to by the caret has appeared more than once in the same codemacro directive and is a duplicate definition.

683

This expression is illegal within a codemacro definition.

Null initialization expressions, DUP expressions, and dot operator expressions that don’t use a record field as their right operand are illegal within a codemacro definition.

684

This statement is not allowed in a codemacro definition.

Only a limited number of types of statements is allowed in a codemacro definition. For a complete list, see the chapter titled Codemacros.

685

This instruction or codemacro has too many operands.

as86 limits the number of operands to 3 in an instruction and to 255 in a codemacro.
Chapter 27: Assembler Error Messages

Syntax Errors

686  Duplicate use of NOSEGFIX directive in codemacro definition.
Only one NOSEGFIX directive can be used in any given codemacro definition.

687  Duplicate use of SEGFIX directive in codemacro definition.
Only one SEGFIX directive can be used in any given codemacro definition.

688  PREFIX and non-PREFIX codemacros cannot have the same name.
The codemacro symbol being pointed to by the caret has been defined in codemacro directives both with and without the PREFIX keyword. The last definition of the codemacro is the one that will be in effect.

689  Missing PROC directive or previous procedure nesting error.
This ENDP directive has no matching PROC directive due to an omission or a nesting error involving its associated PROC directive.

690  This symbol has not been defined.
During Pass 1, the assembler assumes that an undefined symbol is a forward reference. This message occurs when the symbol is still not defined in Pass 2. The assembler generates NOPs and continues assembly. You should modify the code to define the symbol, or the symbol will have no value.

691  CS cannot be destination register.
CS can only be changed by using an ASSUME directive, a JMP or CALL instruction to a FAR location, and a MOV or POP has been used to load the CS register.

692  Pass 1 estimate of instruction bytes insufficient.
The number of bytes reserved for an instruction as a result of a forward reference in Pass 1 did not leave enough code space for the instruction in Pass 2. There are two possible remedies:

Specify the sizes of forward-referenced variables using the PTR operator.

Use the $OPTIMIZE control.

693  This symbol is not defined as a group.
The symbol before the GRPOFFSET operator or following the GRPSIZE operator must be a group name. If it is not, then this error is generated.
694  **Shift values greater than 31.**

A value for one of the shift or rotate instructions evaluated to a value that was greater than 31. Adjust the shift value and reassemble.

695  **ES cannot be overridden in this string instruction.**

Certain types of string instructions (e.g. MOVS) require that their second operand use the ES:DI combination for their reference. In such instances, the ES register cannot be overridden. Modify the program to do such operations through the ES register, and reassemble.

697  **Illegal character in numeric constant.**

An illegal character for a numeric constant was found in the constant pointed to by the caret. Remove the illegal character and reassemble.

698  **Illegal DUP value.**

A negative or zero repeat count value for a DUP initialization was found at the location pointed to by the caret. Only positive repeat values are allowed. Correct and reassemble.

699  **No forward references allowed in EQU expressions.**

The expression pointed to by the caret contains an as-yet undefined symbol. Since this expression is being defined as an EQU symbol, such forward references are not allowed. Eliminate the forward reference by moving the definition of the as-yet undefined symbol in front of this EQU definition, and reassemble.

701  **This construct is invalid in the current assembly mode.**

Certain constructs that are accepted only by a given assembly mode (MOD086, MOD186, MODV20) that aren’t accepted in the current assembly mode will cause this error to be generated.

702  **No module name specified.**

No NAME directive was found in the source program. The default name, which is the basename of the source file, will be used.

703  **This symbol was previously declared public.**

The symbol pointed to by the caret previously appeared in this or another PUBLIC directive.
Chapter 27: Assembler Error Messages
Syntax Errors

704  Too many initializations specified: remainder ignored.
When re-initializing a structure or record at allocation time, this message is
generated if more initialization values were specified than there were fields in
the structure or record.

705  This field cannot be re-initialized: value not changed.
Structure fields with many values or a DUP expression cannot be re-initialized
at allocation time.

706  Illegal initialization value: not re-initialized.
An attempt was made to initialize a structure or record field with an invalid
value.

707  Location counter overflow.
Addition of the current instruction or data definition directive causes the
current segment's location counter to exceed the value 0FFFFH, i.e. the 64K
limit of a segment. The location counter is set to the value MOD 65536.

Note  This may cause previous code or data to be overwritten if this is ignored.

708  This EQU cannot be made public.
Certain types of EQU symbols, such as those representing instructions or
address expressions, are not permitted to be declared PUBLIC.

709  Floating point overflow: set to infinity.
The number of bytes in a floating point value exceeds the limit of a DD (32
bytes), DT (80 bytes) or DQ (64 bytes) directive. Assembly continues; adjust
the value to fit within the limit of the Data Directive used.

710  Floating point underflow: set to zero.
The number of bytes in a floating point value is under the limit of a DD (32
bytes), DT (80 bytes) or DQ (64 bytes) directive. Assembly continues; adjust
the value to fit within the limit of the Data Directive used.

711  BCD value exceeds 18 decimal digits.
A packed decimal value (DT) can take 18 digits only; anything over 18 is
truncated. Assembly continues; adjust the value to fit within the 18 digit limit.
**Chapter 27: Assembler Error Messages**

**Syntax Errors**

712  **Integer value exceeds 64-bit limit.**

This warning occurs when an integer constant used in a DQ directive has a value outside the range 0 to FFFFFFFFFFFFFFFFH. Correct the value and reassemble.

716  **This and future preprocessor statements will be ignored.**

Meta characters have not been preprocessed; assembly continues. The assembler does not process any lines with meta characters. Execute the macro string preprocessor before assembling.

717  **Segment limit exceeded for this segment.**

The specified segment contains instructions and/or data that take up more than the maximum allowable 64K bytes of space. Break the segment into multiple segments or shrink the size of the segment, and reassemble.

718  **Procedure not closed within this segment.**

A procedure (or procedures) whose PROC directive was defined in the segment having an ENDS directive which is currently being processed has not yet been closed. The procedure should be closed by inserting an ENDP directive at some point before the ENDS directive.

719  **Segment not closed by end of module.**

One or more segments were open at the point where the assembler found the END directive. The segments should be closed at the appropriate point within the source file.

720  **Procedure closed in segment other than the one it was defined in.**

The ENDP directive, which closes a procedure, appears in a different segment than the one in which the matching PROC directive appears. Make sure that the PROC and ENDP directives reside within the same segment.

722  **String truncated to 2 characters before integer conversion.**

A string that appears anywhere other than in a DB directive must be either 1 or 2 characters long. If such a string is longer than 2 characters, it will be truncated to 2 characters and converted to an integer.

724  **Record field overflow: 'value' modulo 'field width' used.**
If a record field initialization or reinitialization expression evaluates to a value that won’t fit the specified record field, the appropriate modulo operation is performed in order to force the value to fit.

726  **Illegal assembly mode.**

The instruction pointed to by the caret is not valid in this assembler.

727  **Overriding string too large for field.**

If a string field in a structure is reinitialized and the string is too long for the specified field, the string is truncated and this warning message is displayed.

728  **Source path names for debug have been truncated to 255 characters.**

If the assembly module was produced by the AxLS C compiler and the full path name for the source file or any include file is longer than 255 characters, the assembler will truncate the path name from the left, adding an ellipsis to the name to create a total length of 255 characters, and emit this message.

729  **High-level block nesting limit exceeded: some variable scoping lost.**

Nesting of high-level procedure or code blocks is allowed up to a depth of 15. Any nesting beyond this depth will result in the loss of information about which block symbols belong to.

800  **EVEN directive cannot be in a BYTE aligned segment.**

You cannot use the EVEN directive within a segment whose alignment attribute is BYTE. In such a segment, there is no need to force the alignment to be on a word boundary as it will not be any more effective by doing so. Comment out or remove the unnecessary EVEN directive and reassemble.

801  **CS-IP initialization required for main module.**

Some register initializations were provided on the END directive; however, this error message indicates that no initialization for the CS:IP registers was provided. If any register initializations are provided, an initialization for CS:IP must be provided as well. Add the appropriate initialization and reassemble.

802  **Illegal initialization of SS register.**

It is illegal to initialize the SS register to anything other than a segment base. In particular, group bases are not allowed. Correct the initialization on the END directive and reassemble.
Circular chain of equates.

EQU symbols in a list with a length of at least one were defined as other EQU symbols in such a way that the last symbol in the list was defined as the first symbol in the list. Usually, such a construct results from symbol spelling errors, or in larger programs, widely scattered EQU definitions. Correct the erroneous EQU definition and reassemble.

Illegal to use relocatables in DB, DQ, or DT.

If a relocatable value appears in an expression for a DB, DQ, or DT directive, this error is generated. Remove the relocatable value and reassemble.

Variables or Labels cannot be in DB, DQ, or DT.

An expression that contains a variable or label is not allowed in a DB, DQ, or DT directive.

Illegal to use multiple INCLUDE controls on line.

Only one INCLUDE control is allowed on any given line containing assembler controls. Split the control line into as many lines as necessary to obtain control lines with only one INCLUDE control per line, and reassemble.

Inconsistent AT value given for segment.

A segment was specified in a previous SEGMENT directive with a different absolute paragraph number than is specified in the current SEGMENT directive. The paragraph values should be the same.

This codemacro specifier cannot have a range.

The codemacro specmod field being pointed to by the caret is not permitted to have an associated range. Only codemacro parameters with specifiers A, D, R, or S can have range values.

Duplicate specification of alignment type.

A segment directive can only have a single alignment type as an option. This error is generated if more than one alignment type is detected in the segment directive.

Duplicate specification of combine type.
Chapter 27: Assembler Error Messages
Syntax Errors

A segment directive can only have a single combine type as an option. This error is generated if more than one combine type is detected in the segment directive.

811  Duplicate specification of class name.
A segment was specified in a previous SEGMENT directive with a different class name than is specified in the current SEGMENT directive. Both SEGMENT directives should use the same class name.

812  Maximum source line length exceeded.
An input source line exceeded 1024 characters in length. The assembler will not accept lines longer than this length.

813  Maximum string length exceeded.
A string was defined that exceeded 1024 characters in length. The assembler will not accept strings longer than this length.

820  Relocatable numbers not allowed in DD.
A relocatable value was used in a DD directive, which is not allowed. Only relocatable full addresses, segment, or group names may be used in a DD directive.

825  Codemacro argument cannot be addressed by the required segment register.
The codemacro requires that one of its arguments be addressable through a specific segment register. The current ASSUME contents for that register does not allow that argument to be reached, so this error is generated.

826  Iterated Data record offset is too large for a fixup.
Fixups to object code can only occur within the first 1024 bytes of a record. In this instance, an iterated data record is being created that is larger than 1024 bytes and requires a fixup beyond that point. This cannot be represented in HP-OMF86 so this error is generated.

827  OMF record length exceeds maximum value.
An HP-OMF86 record can only be 64K in size. Any attempt to generate more than 64K of text in a single HP-OMF86 record will result in this error message.

828  Codemacro instruction length exceeds 247 bytes.
A single codemacro instruction can only generate up to 247 bytes of object code. Any instruction that generates more than that number of bytes will result in this error message.

996 Internal error.

997 Fatal Error.

998 *** Fatal Internal Error: Unimplemented Semantics ***.

999 ********** FATAL INTERNAL ERROR **********.
The Macro Preprocessor produces numbered error messages. This chapter explains the meaning of the numeric codes. More than one message may appear for a given source line. Each message is printed immediately upon detection of the error (because the macro processor is character-oriented, not line-oriented). The usual effect is for a message to appear before any output from the source line that caused the error. Macro error messages appear as assembler comments in the output source file, like this:

; ***** ERROR 301

---

## Error Codes and Messages

### 301

**Undefined macro name.**

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

### 302

**Illegal call to %EXIT.**

%EXIT is outside any user macros, WHILEs, or REPEATs. The call is ignored, %EXIT is not passed to the output file, and processing continues.

### 303

**Illegal expression.**

A numeric expression was expected. There could be a missing % from a macro-time symbol or a syntax error, among others. This message is produced when ap86 is trying to evaluate an expression within EVAL, IF, WHILE, SUBSTR or REPEAT. The function call is aborted (any output from it is lost) and processing continues following the call pattern of the function. This message is also reported when an illegal character is detected in a string being compared with %EQS (or other string comparison functions).

### 304

**Logical Expression Error**

### 305

**Missing 'FI'.**

Self-explanatory. This has no effect except to produce the message. However, the search for FI is character-by-character, so that if FE was present when FI
was expected, the F would be removed from the output file. The E and subsequent characters would be passed on normally.

306 Missing 'THEN'.

Self-explanatory. The call to IF is aborted and processing continues following the first character which failed to match. Thus the THEN and ELSE clauses, and the ELSE and FI keywords, will be treated as normal text and expanded normally. As with FI, the search for THEN is character-by-character.

307 Illegal attempt to redefine macro.

A built-in function cannot be re-defined at any time. It is not possible to re-define a macro formal parameter within the macro body or a macro name within its own body.

309 Missing balanced string.

In a call to a built-in function, a required balanced-text string delimited by parentheses is not present. This error can also be generated when the leading left parenthesis is not found where expected. The function call is aborted and scanning continues from the point at which the error was detected.

310 Missing list item.

A list item (delimited by commas) is missing. The function or macro call is aborted and scanning continues from the point where the error was detected.

311 Missing delimiter.

A delimiter required when scanning of a user-defined macro or built-in function (a comma, usually) is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

312 Premature EOF.

The end of the input file occurred while the call to the macro was being scanned. This usually occurs when a right parenthesis is omitted, causing the Macro Preprocessor to scan to the end of the file searching for it. Note that even if the closing parenthesis of a macro call is given, this error may occur if any preceding commas are missing, since the Macro Preprocessor searches for delimiters one by one.

313 Macro stack overflow.
Chapter 28: Macro String Preprocessor Error Messages
Error Codes and Messages

The macro context stack MSTAK has overflowed. This stack is 64 deep and contains an entry for each symbol preceded by the metacharacter. The cause of this error is excessive recursion in macro calls or expansions; a likely source is a user-programmed infinite loop. When this error is encountered, the stack is emptied and all pending output destroyed; scanning continues at the next character in the input file. This message can also be produced to indicate that INCLUDEs were nested too deeply.

314 Nested macro error.

315 String buffer overflow.

The string buffer used in conjunction with the macro stack to save intermediate results from nested macro calls has overflowed.

318 Illegal metacharacter.

Self-explanatory. The current metacharacter remains unchanged.

319 Unbalanced right parenthesis.

During the scan of a call to a user-defined macro, an unmatched right parenthesis was encountered. This is frequently because of a missing argument (the right parenthesis terminating the macro call was found when a comma was expected). The call is aborted and scanning continues from the point at which the error was detected.

338 Invalid symbol.

A symbol (not preceded by the metacharacter) is required in certain contexts, such as the MATCH, DEFINE and SET functions. This symbol was not valid.

340 Literal character on SET or WHILE.

The constructs %SET and %WHILE make no sense and produce this message. The * is ignored, and the Macro Preprocessor attempts to expand SET or WHILE normally.

401 Bad or missing parameter.

The parameter to a control is not correctly formed, or a control that requires a parameter does not have one. Typographical errors often lead to this message.

414 Unable to open include file.
Self-explanatory.

901  
**Scan stack overflow.**

This error indicates that the stack used for evaluating complex expressions has overflowed. This will not occur for any expression likely to be useful in practice. Break the expression into smaller ones.

906  
**Macro symbol table exhausted.**

The macro-time symbol table is full. This table contains symbol names plus the string values of SET and MATCH symbols.
List of the loader error messages.
Warning Messages

400 Repeated segment name in ORDER command: [SEGMENT__NAME].

The loader displays this message when there is an implicit or explicit reference to the same segment name. If the reference is implicit, the duplicate segment name is embedded in a class specified by a classname in the ORDER command. If the reference is explicit, the duplicate segment name occurs at least twice in the current ORDER command. The duplicate occurrences of the segment name are ignored and loading continues.

401 SEG, ALIGN, or SEGSIZE command used on absolute segment.

The loader encountered a user-specified base address for an absolute segment. The user-specified base address is ignored and loading continues.

402 ORDER command cannot be obeyed.

The loader generates this message because the placement of absolute segments, SEG commands, and/or reserved areas prohibits placement of the segments in memory in the order specified. This message applies only to segments explicitly named in the ORDER command. Segments implicitly named (such as by specifying a classname with an ORDER command) do not have an implied order. The order of segments will not be as specified in the ORDER command and loading continues.

403 This external symbol is undefined: [SYMBOL__NAME].

The loader displays this message when it finds an external symbol in the symbol section of a module that is undefined by the user or undefined in any module. The undefined external symbol may or may not actually be referenced. Loading continues.

404 Group is larger than 65536 bytes: [GROUP__NAME].

According to the base address assignment, all members of the group specified by GROUP__NAME do not fit within 64K bytes of one another; thus the segments as currently loaded cannot be addressed by a single segment register (which is the purpose of having groups). By judicious use of classnames in the assembler, or SEG and ORDER commands in the loader, it may be possible to get all group elements within 64K bytes of each other. Relocation errors may result from any references to the group parts outside the 65536 byte group base. Loading continues.
405 Group contains undefined or absolute external.

One of the input modules contains a GROUP element defined by means of an external name. The external name is either undefined or defined as absolute; therefore its segment base cannot be determined. Relocation errors will result from references to the external name, whether or not an error is reported. Loading continues.

406 Memory segment is not at the top of memory: [SEGMENT__NAME].

The loader displays this message whenever more than one segment has the memory attribute, or whenever a SEG or ORDER command causes the single memory segment to be misplaced. Everything is loaded where assigned and loading continues.

407 Memory overlap by segment [SEGMENT__NAME].

The loader generates this warning when an area reserved by a RESNUM or RESADD command, an absolute segment, or a segment specified in a SEG directive conflicts with a previously-reserved area. The previously-reserved area could have been reserved by the RESNUM, RESADD, or SEG directive or by an absolute segment. Any additionally relocatable segments will not overlap. Any additional previously-unreserved space is reserved and loading continues.

408 No segments with classname [CLASS__NAME].

The classname displayed in the message appears in a SEG or ORDER command, but no segments with that classname exist. The classname is ignored and loading continues.

409 SEGSIZE used with public or private segment: [SEGMENT__NAME].

The SEGMENT__NAME displayed in the message is the name of the segment altered by the SEGSIZE command. The loader issues this message if the segment does not have a combine type of STACK or COMMON.

SEGSIZE is typically used to set the size of a stack segment but you can use it to set the length of any segment. The user-specified length is used in the segment. There is a possibility of overlapped data. No further warning will be given and loading continues.

410 User-specified base address does not match alignment type for segment [SEGMENT__NAME].
Chapter 29: Loader Error Messages

The loader displays this message when the base address specified in the SEG command does not match either the alignment carried by the segment or with the alignment specified by an ALIGN command. The loader ignores the alignment attribute and loads the segment at the user-specified base address. Loading continues.

411 Inpage alignment cannot be performed for segment [SEGMENT__NAME].
The named segment is too large to fit in a page (256 bytes). The segment is loaded at the next page boundary. Loading continues.

412 Respecification of output object module format.
In any given invocation of the linking loader, you can specify only one output format among the following three: HP-OMF 86 format absolute, Intel Hexadecimal Object file format absolute, or HP 64000 format absolute. You can override the default format; however, if you explicitly request either HP 64000 absolute output or Intel Hexadecimal output on the command line or in a LIST command, or any combination of these, the above warning message is generated. Loading continues; the output format is left as it was first set.

413 The following command/option is not allowed after a LOAD command: [OPTION__NAME].
The output format options in the LIST and NLIST commands (B, I, or H) cannot be specified after a LOAD command is issued. This is because the loader’s internal data structures have been set to accommodate only the output format in force at the time of the first LOAD command. The option specified in the OPTION__NAME field of the error message is ignored and loading continues.

414 SEGSIZE value too small for stack or common segment: [SEGMENT__NAME].
The length chosen for the stack or common segment named in the error message was too small to accommodate the length requested by a portion of the segment. Depending upon the actual amount of stack or common space used by the program, the length may be adequate. To alleviate this condition, eliminate the offending SEGSIZE directive, or increase its length parameter. The user-specified length for the segment is used. The possibility of data overlap exists and no further warning will be given. Loading continues.

415 OMF buffer flushed: NAME command ignored; output module name is: [MODULE__NAME].
Chapter 29: Loader Error Messages

This message occurs if an internal table containing class and segment group names is about to overflow the buffer. It has to be flushed to the output object module before any NAME command is encountered. At that time, the module_name will default to the name of the loader command file, or if there is no command file, to the first file loaded. Module_name becomes the default module name. Loading continues.

416 Undefined external referenced in module.

This message is generated only when the LIST E option is selected during an incremental link. This option specifies this warning message be generated during an incremental link for any undefined external symbols that are referenced. The module name displayed in the message makes a reference to an external symbol that was neither defined by you nor defined in another module. The name of the referenced external is listed along with the module name. Linking continues. Note that this warning usually does not indicate a problem as undefined external symbols are permitted in relocatable object modules. However, this warning alerts you to any unresolved external symbols that you may have thought were already resolved.

417 Duplicate ORDER command: previous order commands ignored.

The linker found a second or subsequent ORDER command. Any information contained in the previous ORDER command(s) is now lost, even if the newly-found ORDER command contains errors. If the newly-found ORDER command contains errors, no ORDER command information is saved unless another valid ORDER command is found later in the command file. Linking continues.

418 Null GRPDEF record referenced during relocation.

A group containing no member segments or externals was used as the base of a relocation operation. The base of the item being relocated rather than the non-existent base of the group was used to perform the relocation. Linking continues.

419 No object file entries for user-defined segment.

A segment name or class name has been specified in an ALIGN, INITDATA, ORDER, SEG, or SEGSIZE command that was not found anywhere in the object modules that were loaded by the linker. This error message either means that your command file has an unnecessary segment or class name reference or you are not loading one or more of your object modules. Both of these situations can be remedied in the command file by deleting references
Chapter 29: Loader Error Messages

to the displayed segment or class or by including LOAD commands for the object modules containing the displayed segment or class. Linking continues.

420

No object file entries for user-defined group.

A group name has been specified in a GROUP command that was not found anywhere in the object modules that were loaded by the linker. This either means that your command file has an unnecessary group name reference or you are not loading one or more of your object modules. Both of these situations can be remedied in the command file by deleting references to the displayed group or by including LOAD commands for the object modules containing the displayed group. Linking continues.

421

Nonstandard debug information encountered and removed.

Debug information that is not part of the standard HP-OMF86 or Intel file format was stripped from the file.

422

External defined in different segment than specified in reference: [SYMBOL]

An external symbol was defined to exist within a specific segment but, after resolving the external with a public symbol, the base segment for the external is different. In this case, the linker gives a warning so the user is aware of this incompatibility. The best solution is to place the definition of the external symbol within the correct segment. If this segment is not known, the external symbol should not be defined within any segments.

423

TYPDEF record limit exceeded - types set to NULL.

The HP-OMF86 file format is only able to represent 32k types. If this limit is exceeded in the linker, this warning is generated. All type information after the first 32k types will be lost. All of the type information can probably be saved through the appropriate use of the 'TYPEMERGE' command in the linker command file.

424

The following pathname was truncated in the OMF output: [FILENAME]

The maximum length for pathnames in OMF is 255 characters. If the length of the absolute path to the object file is greater than this limit, then the pathname must be truncated.

425

Multiple Register Initialization for: [REGISTER]

Three registers in the 8086/186 processor may be initialized at runtime. These registers are CS:IP, DS, and SS. Normally, only a single input module contains
the register initialization information. If more than one input module contains this information, then this error is generated and the last initialization value is used.
**Chapter 29: Loader Error Messages**

### Error Messages

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| 300  | **Invalid name [INVALID__TEXT].**  
The loader displays this message when it encounters illegal characters or too many characters in a command. The maximum length of PUBLICs is 255 characters; the maximum length of SEGs, CLASSES, and GROUPs is 40 characters. The command that uses the name is not processed. Loading continues. |
| 301  | **Invalid hexadecimal value: [HEX_VAL]**  
The linker expected to find a valid hexadecimal constant; instead, the text displayed in the HEX_VAL field of the message was found. The text should be corrected to conform to the syntax for hexadecimal constants: digits 0 to 9, letters a through f or A through F, preceded by a leading zero if a letter is used as the most significant digit, and followed by a trailing h or H. |
| 302  | **Invalid command operand.**  
An operand specified for a command contains invalid characters, does not exist, or is too large. Loading continues; the command in which the error occurs is ignored. |
| 303  | **Error occurred when closing a file.**  
Loading continues; however, the file in question may become corrupt. |
| 304  | **Undefined external encountered as a member of group.**  
In the assembler, it is possible to define a group in terms of segments, segments of variables, or segments of external symbols. Since external symbols may be defined outside of any segments, the assembler does not have enough information to decide what segment the external belongs to, so it must wait until link time before this is decided. And if the external symbol is not defined in any input modules, the linker is also unable to decide which segment belongs in the group. Under these circumstances, this error message is generated and no segment is placed in the group. |
| 305  | **Invalid loader command: [INVALID__COMMAND].**  
The command displayed in this message is not a valid loader command. Loading continues; the erroneous command is ignored. |
Chapter 29: Loader Error Messages

306 Invalid command syntax at or before: [COMMAND__TEXT].
Loading continues; the command in which the error occurs is ignored.

307 Duplicate public symbol: [DUPLICATE SYMBOL__NAME].
The symbol name displayed in this message was defined previously in another module. Loading continues; the first definition of this public symbol is the one the loader uses for symbol recognition.

308 Undefined external referenced in module [MODULE__NAME:EXTERNAL__NAME].
The module name displayed in this message makes reference to an external symbol that was neither defined by the user nor defined in another module. The name of the referenced external is listed along with the module name. Loading continues; zeros are substituted for the value of the external.

309 Relocation error at [ERROR__ADDRESS].
A relocated value is too large to fit in the number of bytes allocated for it. For example, this message occurs if a self-relative jump to a NEAR label is outside the boundary of a 16-bit displacement. Loading continues; however, the specific relocation is not performed and the absolute object module is not likely to be useful.

310 Segment mismatch on combine type for segment [SEGMENT__NAME].
The loader displays this message when different modules that contain parts of the same segment have different combine type attributes. Loading continues; the specified segment is NOT COMBINED.

311 Combined segment is larger than 65536 bytes.
During initial loading, a (combined) segment length exceeds the 64K segment size. Loading continues; however, the absolute file produced is most likely useless.

312 Unexpected character in symbol [SYMBOL__NAME].
The loader displays the name of the symbol in which an invalid character occurs. Loading continues; the symbol containing the invalid character is not processed.

313 Segment(s) located beyond 0FFFFFFH boundary.
Chapter 29: Loader Error Messages

The linked module exceeds memory requirements for the 8086/186 microprocessor. Loading continues; however, the absolute file produced is most likely useless.

314  **Procedure/block nesting limit exceeded.**

The loader displays this message if the source program contains procedures and functions nested to a level deeper than 15. Loading continues; however, the loader cannot produce the proper procedure and scoping information.

315  **Continuation line error.**

The loader displays this message when it encounters incorrect syntax for specifying a continuation line. Loading continues; however, the loader ignores continued information until it encounters a line of code beginning with the correct syntax.

316  **Specified group base is not divisible by 16 [GROUP__NAME].**

A group base value specified in a GROUP command for the GROUP__NAME displayed in the message is not divisible by 16, and hence is not on a paragraph boundary. In order to fix this problem, a value divisible by 16 should be substituted in the GROUP command. Loading continues; no group addresses are assigned.

317  **RESNUM or RESADD command overlaps previously-reserved memory.**

The loader generates this warning when the first part of an area previously reserved by a RESNUM or RESADD directive, an absolute segment, or a segment specified in a SEG directive conflicts with a reserved area. Relocatable segments will not overlap. Loading continues; any additional non-overlapping space is reserved.

318  **This command not allowed with relocatable output: ignored.**

The INITDATA command should only be used when you are performing a final (not incremental) link. Loading continues; the INITDATA command is ignored.

319  **Numeric value out of range: [BAD_VALUE]**

A numeric value found in the command file was outside the range (0 ≤ value ≤ 0FFFFH) for 16-bit values or outside the range (0 ≤ value ≤ 0FFFFFFH) for 20-bit values. This can be fixed by correcting the erroneous value in the command file. Linking continues. The command containing the incorrect value is ignored.
Invalid BLKDEF record for block: [BLOCK_NAME]
Invalid unnamed BLKDEF record

An illegal BLKDEF record was seen by the linker. This record either has no name or has an illegal attribute. This error probably indicates that the file is either corrupt or was generated by a nonsupported tool.
Chapter 29: Loader Error Messages

Fatal Error Messages

330  Bad Intel object record.
The byte that was read in as the first byte of an Intel OMF record was not a valid record marker as indicated in the Intel 8086 Relocatable Object Module Format Specification. The input file has probably been corrupted.

331  Invalid checksum in object record.
The object record has a checksum error; the input object file has probably been corrupted.

332  Load terminated by user.
This error is generated when the EXIT command is encountered.

333  Unable to open file [FILENAME].
The file named in this message could not be opened. Check for correct spelling and verify the file’s existence.

334  Linker Internal error # [ERROR__NUMBER].
Should this error display, contact Hewlett-Packard Customer Support.

335  First record is not a valid relocatable header record.
A header record is not the first record in the object module.

336  Unexpected end of file encountered.
The loader encountered a physical end-of-file before it read records to indicate end-of-module.

337  Illegal object record length [LENGTH].
The object record exceeds 64K bytes. LENGTH is the length of the object record in bytes.

338  Segment, group, or external index out of range.
An external reference is made to an external symbol that does not exist in the object module.
Unable to open temporary file.

Disk file output error -- program aborted.
While the linker was attempting to write to a disk file, an error occurred. Usually, this is caused by a lack of disk space. Freeing up some disk space may help.

Disk file input error -- program aborted.
While the linker was attempting to read from a disk file, an error occurred.

Disk file seek error -- program aborted.
While the linker was attempting to seek a position in a disk file, an error occurred.

The command line exceeds the command line length limit:
The maximum command line length is 1024 characters. If this limit is exceeded, then the command line cannot be processed.

File Overwrite condition: [TEXT].
It is possible for the creation of an output file to destroy one of the input files or another output file. For this reason, the linker checks to see if the creation of the output file will overwrite one of these other files. If this is the case, then this error is generated and the linker does not proceed any further.

BLKEND record has no matching BLKDEF, or nesting limit previously exceeded.
If the end of a procedure block (in the debug information within a module) does not have a matching beginning, then this error is generated. The input object file must be recreated by the assembler without any detected errors.
Chapter 30: Librarian Error Messages

Librarian Error Messages

List of the librarian error messages.
Chapter 30: Librarian Error Messages

Librarian Error Messages

100 Could not close file [FILENAME] to open another file.

In order to reduce processing overhead, the librarian keeps OPENed files open. This message is displayed if too many files are open and the librarian unsuccessfully attempts to close a file in order to open a new one. To remedy this situation, reduce the number of files you are working with during a given session.

101 Unable to open file [FILENAME].

The librarian could not open the named file when executing an ADDMOD, REPLACE, or OPEN command. This error could be caused by either an invalid filename specification or when the specified file does not exist. The librarian ignores the command that generates this error.

102 Unable to close file [FILENAME].

The librarian generates this error when it encounters an operating system error, and cannot close the named file. This message typically is accompanied by another error message that provides a more specific reason for not closing the named file.

104 File [FILENAME] not included.

The librarian issues this message when it cannot execute the ADDMOD command because the named file is corrupted or does not exist. This message has a companion message that specifically states why the named file is not included in the library.

106 File [LIBRARY__NAME] exists already.

The librarian generates this message when you have used the CREATE command, and the named library currently exists. The librarian displays a warning in batch and interactive modes.

107 File [FILENAME] does not exist.

The librarian generates this message when you have issued an OPEN command, and the named file does not exist. If you are creating a new library file, you may ignore this message.

108 Library file [LIBRARY__NAME] not opened.
Chapter 30: Librarian Error Messages

201  Module [MODULE__NAME] not found.

The librarian could not locate the named module in the library to execute a DELETE, REPLACE, or EXTRACT command.

203  Module [MODULE__NAME] already exists in current library

The librarian cannot execute an ADDMOD or ADDLIB command because the module named in the message exists in the current library. If you wish to replace a module in the library, use the REPLACE command. The librarian ignores the ADDMOD or ADDLIB command that contains a duplicate module name.

204  [FILENAME] is a library file.

The librarian generates this error message when it attempts to execute an ADDMOD or OPEN command, and the associated filename is not an object module. The command containing the erroneous file is ignored.

205  [FILENAME] is not a library file.

The librarian issues this error message when it attempts to execute an ADDLIB or OPEN command, and the associated filename is an object module. The librarian ignores the command containing the erroneous file.

206  Module [MODULE__NAME] is not included in the library.

The librarian issues this message with a companion message that gives the specific reason for not including the named module in the library. This message describes the result: the named module is not included in the library.

207  Bad object record.

Either the object module has been corrupted or it is not a legal relocatable object file. The librarian issues this message with a companion message that names the file with the bad object record. The command associated with the bad object record file will be ignored.

208  Bad library header record.

The library has a bad header record. The librarian issues this message with a companion message that names the file with the bad header record. The command associated with the bad library header record will be ignored.

209  Duplicate symbol [SYMBOL__NAME].
A module named in an ADDLIB, ADDMOD, or REPLACE command has the same public definition symbol that occurs in another module. The librarian issues this message with a companion message that provides information about what action it takes. The librarian considers symbols to be case-sensitive.

**Bad object record in file [FILENAME].**

The named library or module file may have been corrupted.

**Out of memory.**

The librarian issues this message when there is not enough system memory to execute commands issued since the last CREATE or OPEN command.

**Failed writing library [REASON].**

The librarian generates this message when it attempts to execute a SAVE command and cannot. The message provides the reason for inability to create a library. The librarian abandons the current session affected by the SAVE command that caused the error.

**Library [LIBRARY__NAME] not written.**

The librarian issues this message when an error occurs earlier in the session that prevents the library from being saved. This message is typically accompanied by another message that contains the reason the named library was not created.

**Failed writing module [MODULE__NAME] to file [FILENAME].**

When attempting to execute an EXTRACT command, the librarian cannot write the named module from an existing library to the (new) file, which is external to the library. A companion error message describes the reason that the module cannot be extracted. If an error is encountered in batch mode, all commands following the EXTRACT command will not be executed; however, they will still be checked for syntactic validity.

**Replacement not done.**

The librarian issues this message when it cannot execute the REPLACE command for the reason specified in the companion message.

**Extraction failed.**

The module named in the EXTRACT command is not extracted.
Chapter 30: Librarian Error Messages

257  Illegal command.
The librarian generates this message when it encounters either an incorrect command sequence or an incorrect command syntax.

259  Quote Not terminated.
A string is missing a closing quote.

262  There is no library to be saved.
The SAVE command does not have a library to save.
Index

% escape character, 256
_ (underscore character), 24

A
.A suffix, 23–24
absolute expression, 129
absolute segment
  loader, 370
acvt86 translation tool, 290–293
adding base and index register
  in expression, 134
addition operator
  binary, 138
  unary, 137
ADDLIB librarian command, 444
ADDMOD librarian command, 444
ALIGN loader command, 387–388
align-type attribute
  loader, 374
alignment, 37
allocating record storage, 115
allocating structure storage, 124
AND operator, 144
anonymous reference, 163
  with expression, 135
ap86, 30–31
ar86, 45–48
archiver, 45–48
arithmetic operator, 137
as86, 23–29
ASCII codes, 58
asmb_sym file, 23
assembler, 23–29
  control general syntax, 195
  cross reference format, 213–216
directive, 79–126
error messages, 465–492
assembler (continued)
general controls, 195
listing, 211–216
operation, 52
primary controls, 195
statement syntax, 68
symbol table format, 213–216
assembler controls
CASE, 197
DATE, 197
DEBUG, 198
EJECT, 206
ERRORPRINT, 198
GEN, 206
general, 206–208
GENONLY, 206
INCLUDE, 207
INCLUDE with macro preprocessor, 248
LIST, 207
MACRO, 200
MOD086, 200
MOD186, 201
MODV20, 201
OBJECT, 202
OPTIMIZE, 202
PAGELENGTH, 202
PAGEWIDTH, 203
primary, 197
PRINT, 203
RESTORE, 207
SAVE, 208
SYMBOLS, 204
TITLE, 208
TYPE, 204
UNREFERENCED_EXTERNALS, 204
WORKFILES, 205
XREF, 205
assembler syntax
blank line, 70
comment, 69
continuation line, 70
assembler syntax (continued)
  keyword, 69
  label, 69
  operand, 69
  prefix, 69
  symbol, 59
assembling program modules, 9–14
assembly source translation
  acvt86 tool, 290–293
assembly source translation
  HP 64853 to HP B1449, 279–306
ASSUME directive, 86–87
assumed, 163
  * operator, 141, 389
attribute
  BASE, 74
  CS ADDRESSABILITY, 77
  INDEX, 74
  OFFSET, 73
  RELOCATION TYPE, 75
  SEGMENT, 75
  SEGMENT ADDRESSABILITY, 76
  SEGMENT RELOCATION, 75
  TYPE, 73

B
  balanced text string, 252
  baltex, 252
base address
  loader, 369
base address assignment
  loader, 378–382
  BASE attribute, 74
base register
  in expression, 134
  binary minus, 138
  binary plus, 138
  blank line in syntax, 70
  bracket macro function, 255
byte align-type
  loader, 375
Index

caret, 466
CASE assembler control, 197
case sensitivity, 399
   assembler controls, 197
   macro preprocessor, 244, 251
case-sensitivity, 25, 31, 35, 41, 56
changes to the assembler, 275–278
character constant, 67
character set, 56–58
characters, 24
class
   loader, 370
class name
   loader, 372
CLEAR librarian command, 445
clearing flags, 24
code translation
   acvt86 tool, 290–293
   HP 64853 to HP B1449, 279–306
colon
   with label, 62
combine-type attribute
   loader, 373
command file
   loader listing, 416
command files, 18, 33
command line length, 511
command syntax, 21–48
commands
   arguments to loader commands, 384
   length of with loader, 385
   librarian, 441
   order in loader, 385
comment in syntax, 69
comment macro function, 255
comments
   librarian, 439
   loader, 389
comments, linker, 35
common segment
   loader, 373
complete name, 385
  loader, 372
constant, 64
  character, 67
  integer, 65
  real, 66
continuation line in syntax, 70
controls, 25
controls, assembler
  CASE, 197
  DATE, 197
  DEBUG, 198
  EJECT, 206
  ERRORPRINT, 198
  GEN, 206
  general, 195, 206–208
  GENONLY, 206
  INCLUDE, 207
  LIST, 207
  MACRO, 200
  MOD086, 200
  MOD186, 201
  MODV20, 201
  OBJECT, 202
  OPTIMIZE, 202
  PAGELENGTH, 202
  PAGEWIDTH, 203
  primary, 195, 197
  PRINT, 203
  RESTORE, 207
  SAVE, 208
  SYMBOLS, 204
  TITLE, 208
  TYPE, 204
  UNREFERENCED_EXTERNALS, 204
  WORKFILES, 205
  XREF, 205
CREATE librarian command, 445
creating macros, 269
cross reference format, 213–216
Index

cross reference table, 28
CS ADDRESSABILITY attribute, 77

D

data definition directive, 83
data object, 83
DATE assembler control, 197
DB directive, 88–93
with string, 92
DD directive, 88–93
DEBUG assembler control, 198
debug information, 25
default
PROC directive, 109
segment, 82
segment register, 87
segments for memory addressing, 171
DEFINE macro function, 269
defining macros, 269
definitions
external, 6
PUBLIC, 6–7
storage locations, 6
DELETE librarian command, 446
differences between processor modes, 209–210
directive
assembler, 79–126
ASSUME, 86–87
data definition, 83
DB, 88–93
DB with string, 92
DD, 88–93
DQ, 88–93
DT, 88–93
DW, 88–93
DW, DD, DQ, DT with string, 92
END, 94
ENDP, 109–110
ENDS (segments), 118–122
ENDS (structures), 123–126
EQU, 96–98
EXTRN, 100–102
GROUP, 103–104
directive (continued)
   LABEL, 105–106
   NAME, 107
   ORG, 108
   PROC, 109–110
   program linkage, 84
   PUBLIC, 111
   PURGE, 112–113
   RECORD, 114–117
   SEGMENT, 118–122
   segmentation, 81
   STRUC, 123–126
   DIRECTORY librarian command, 446
   division operator, 141
   DQ directive, 88–93
   DT directive, 88–93
   DW directive, 88–93
   DW, DD, DQ, DT directive
      with string, 92

E  8086 processor mode, 209
   EBCDIC codes, 58
   EJECT assembler control, 206
   eject page, 25
   END directive, 94
   END librarian command, 447
   END loader command, 389
   ENDP directive, 109–110
   ENDS directive, 118–122
   ENDS directive (structures), 123–126
   EQ operator, 145
   EQS macro function, 257
   EQU directive, 96–98
   EQU symbols defined, 64
   error messages, assembler, 465–492
   ERROR loader command, 389
   error messages
      formats, 461–464
      interactive vs. non-interactive, 462–464
      librarian, 513–518
      loader, 499–512
      macro preprocessor, 493–498
Index

error messages, suppressing, 25
ERRORPRINT assembler control, 198
escape macro function, 255
EVAL macro function, 257
example program
  assembling program modules, 9–14
  description of, 3–8
  linking relocatable object files, 17–20
  objectives of, 2
EXIT loader command, 390
EXIT macro function, 258
expression
  absolute, 129
  anonymous, 135
  base register in, 134
  with EQU directive, 136
  external, 130
  generally, 128
  group name operand, 133
  index register in, 134
  label name operand, 133
  in macro preprocessor, 250–252
  numeric operand, 131
  operand, 131
  operands, 162–169
  operator, 137
  operator, arithmetic, 137
  operator, logical, 144
  operator, record, 154
  record field operand, 132
  record operand, 132
  register indirect, 134
  relocatable, 130
  segment name operand, 133
  string operand, 132
  structure field operand, 134
  variable name operand, 133
external definitions, 6
external expression, 130
external references, 6
checking, 25
EXTRACT librarian command, 447
EXTRN directive, 100–102

file format, 18, 23–24, 29, 32–35, 40, 44
file names
  assembler output, 24
  object, 17, 23
  output, 33
  source, 23
  symbol file, 24
flags
  assembler, 24–25
  unsetting, 24
FORMAT loader command, 391
formats for error messages, 461–464
FULLDIR librarian command, 448
function
  % ( (bracket) macro, 255
  bracket macro, 255
  % ' (comment) macro, 255
  comment macro, 255
  DEFINE macro, 269
  EQS macro, 257
  %n (escape) macro, 255
  escape macro, 255
  EVAL macro, 257
  EXIT macro, 258
  GES macro, 257
  GTS macro, 257
  IF macro, 258
  LEN, 245
  LEN macro, 260
  LES macro, 257
  LTS macro, 257
  MATCH macro, 260
  METACHAR macro, 262
  NES macro, 257
  REPEAT macro, 262
  SET macro, 263
  SUBSTR, 245
  SUBSTR macro, 264
  WHILE macro, 264
Index

G  GE operator, 145
    GEN assembler control, 206
    general assembler controls, 195, 206–208
    general controls, 25
    general syntax, 68
    GENONLY assembler control, 206
    GES macro function, 257

    group, 103–104
      loader, 371
        mismatched groups merged, 371
        OFFSET operator with, 104
        override operator, 148

    group base address
      loader, 371
    GROUP directive, 103–104
    GROUP loader command, 392–393

    group name
      defined, 64
      as expression operand, 133

    groups, 26, 35–36
    GT operator, 145
    GTS macro function, 257

H  HELP librarian command, 449
    HIGH operator, 143
    HP 64000 format
      See file format
    HP 64853 programs, linking to, 303–304
    HP 64853 to HP B1449 translation, 279–306

I  identifiers, 24
    IF macro function, 258
    immediate, 164
    immediate value
      See also numeric value
    INCLUDE assembler control, 207
      with macro preprocessor, 248
    include file, 26
    incorrect macro example, 265
    incremental linking, 32, 368
    INDEX attribute, 74
index register
  in expression, 134
INITDATA loader command, 394
initialization
  record, 115
  segment register, 95
  structure, 124
initialize data, 394
initialized memory, 38
inpage align-type
  loader, 375
instruction mnemonic defined, 62
instruction set, 52
  8086/186, 339
  8086/186 in hexadecimal order, 307–338
  assembler, 174
instruction sets, 26
integer constant, 65
K
  keyword defined, 60
  keyword in syntax, 69
L
  L_to_o86 porting tool, 304
label, 83
  in syntax, 69
LABEL directive, 105–106
label name
  defined, 62
  as expression operand, 133
ld86, 32–44
LE operator, 145
LEN function, 245
LEN macro function, 260
LENGTH loader command, 395
LENGTH operator, 151
LES macro function, 257
library maintainer (ar86), 45–48
librarian
  command syntax, 438
  commands, 441
  comments, 439
  error messages, 513–518
Index

librarian (continued)
  features, v
  introduction, 432
  sample program, 453, 456
  special characters, 438
  use of, 433

librarian command
  ADDLIB, 444
  ADDMOD, 444
  CLEAR, 445
  CREATE, 445
  DELETE, 446
  DIRECTORY, 446
  END, 447
  EXTRACT, 447
  FULLDIR, 448
  HELP, 449
  OPEN, 449
  REPLACE, 450
  SAVE, 450

library files, creating, 15–16
linker, 17
linker/loader, 32–44
linking loader
  introduction, 361–366
  linking to 64853 programs, 303–304
  linking, definition of, 17–20
  LIST assembler control, 207
  LIST loader command, 396–399
  LISTABS loader commands, 400
  listing, assembler, 212
  listings, 9, 15, 18, 23
  LISTMAP loader commands, 400–401
  literal (*) character, 247
  literal character, 256
  LOAD loader command, 402
  load map, 18
loader, 17, 361–366
absolute segment, 370
align-type attribute, 374
base address, 369
base address assignment, 378–382
byte align-type, 375
class, 370
class name, 372
combine-type attribute, 373
command file listing, 416
common segment, 373
complete name, 372
error messages, 499–512
fatal error messages, 510
features, iv
group, 371
group base address, 371
incremental linking, 368
inpage align-type, 375
introduction, 361–366
linking/loading from libraries, 362
logical segment, 369
memory segment, 374
merging mismatched groups, 371
module, 372
non-combinable segment, 373
non-fatal error messages, 506
page align-type, 374
paragraph number, 370
physical segment, 369
primary functions, 368
public segment, 373
relocatable segment, 370
segment number, 370
segments and load addresses, 369
stack segment, 373
symbol with, 384
warning messages, 500
word align-type, 374
Index

loader command
ALIGN, 387–388
command argument, 384
command length, 385
command order, 385
* (comment), 389
descriptions, 386
END, 389
EXIT, 390
FORMAT, 391
GROUP, 392–393
INITDATA, 394
LENGTH, 395
LIST, 396–399
LOAD, 402
NAME, 403
NLIST, 396–399
NOTYPEMERGE, 411
ORDER, 403–404
PUBLIC, 405
RESADD, 406
RESNUM, 406
SEG, 407–408
SEGSIZE, 409
START, 410
TYPEMERGE, 411
WIDTH, 412

loader commands
ERROR, 389
LISTABS, 400
LISTMAP, 400–401
NOERROR, 389
WARN, 389

logical operator, 144
logical segment, 82
loader, 369
LOW operator, 143
LT operator, 145
LTS macro function, 257
MACRO assembler control, 200
macro example (incorrect), 265
macro function
  bracket, 255
  comment, 255
  DEFINE, 269
  EQS, 257
  escape, 255
  EVAL, 257
  EXIT, 258
  GES, 257
  GTS, 257
  IF, 258
  LEN, 245, 260
  LES, 257
  LTS, 257
  MATCH, 260
  METACHAR, 262
  NES, 257
  REPEAT, 262
  SET, 263
  string relational, 257
  SUBSTR, 245, 264
  WHILE, 264
macro preprocessor, 30–31, 243–252
  balanced text string (baltex), 252
  error messages, 493–498
  INCLUDE file, 248
  input parsing, 248
  input source characteristics, 244
  literal character, 247
  metacharacter (%), 245
  output buffering, 248
  starting, 9
  symbol in, 251
  with expressions, 250–252
  with operators, 250–252
man pages, 22
MASK operator, 154
MATCH macro function, 260
memory addressing, 168–169
<table>
<thead>
<tr>
<th>Memory Segment Loader</th>
<th>374</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Severity</td>
<td>389</td>
</tr>
<tr>
<td>METACHAR Macro Function</td>
<td>262</td>
</tr>
<tr>
<td>Microprocessors</td>
<td>52</td>
</tr>
<tr>
<td>Binary</td>
<td>138</td>
</tr>
<tr>
<td>Unary</td>
<td>137</td>
</tr>
<tr>
<td>With Base and Index Register</td>
<td>134</td>
</tr>
<tr>
<td>Mismatched Groups, Merging</td>
<td>371</td>
</tr>
<tr>
<td>MOD Operator</td>
<td>141</td>
</tr>
<tr>
<td>MOD086 Assembler Control</td>
<td>200</td>
</tr>
<tr>
<td>MOD186 Assembler Control</td>
<td>201</td>
</tr>
<tr>
<td>Modifier (Codemacro Specmod)</td>
<td>220</td>
</tr>
<tr>
<td>MODRM Codemacro Directive</td>
<td>234</td>
</tr>
<tr>
<td>Description of MODRM Byte</td>
<td>169</td>
</tr>
<tr>
<td>Values for MODRM Byte</td>
<td>358</td>
</tr>
<tr>
<td>MODRM Byte</td>
<td>169</td>
</tr>
<tr>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>Loader</td>
<td>372</td>
</tr>
<tr>
<td>Size</td>
<td>411</td>
</tr>
<tr>
<td>MODV20 Assembler Control</td>
<td>201</td>
</tr>
<tr>
<td>Multiple Register Initialization</td>
<td>505</td>
</tr>
<tr>
<td>Multiple Segment Definition</td>
<td>120</td>
</tr>
</tbody>
</table>

**N**

| NAME Directive         | 107 |
| NAME Loader Command    | 403 |
| NE Operator            | 145 |
| NES Macro Function     | 257 |
| Nesting Segments       | 121 |
| NLIST Loader Command   | 396–399 |
| Nm64 Porting Tool      | 305 |
| NOERROR Loader Command | 389 |
| Non-Combinable Segment Loader | 373 |
| NOPs, Removing         | 27  |
| NOT Operator           | 145 |
| NOTYPEMERGE Loader Command | 411 |
| Number                 |     |
| 17-bit                 | 130 |
numeric constant
other bases, 65
numeric value
character constant, 67
constant, 64
as expression operand, 131
immediate value, 164
integer constant, 65
real constant, 66

.O suffix, 23, 27
OBJECT assembler control, 202
OFFSET attribute, 73
OFFSET operator, 149
    with group, 104
OMF format
See file format
OPEN librarian command, 449
operand
    in syntax, 69
    positioning, 164
    required typing, 162
operating notices, 275–278
operation differences, processor modes, 209–210
operation of assembler, 52
operator
    AND, 144
    /, 141
    EQ, 145
    GE, 145
    GT, 145
    HIGH, 143
    LE, 145
    LENGTH, 151
    logical, 144
    LOW, 143
    LT, 145
    macro preprocessor, 250–252
    MASK, 154
    —, unary, 137–138
    MOD, 141
    *, 141
Index

operator (continued)
  NE, 145
  NOT, 145
  OFFSET, 149
  OR, 144
  +, unary, 137–138, 145
  PTR, 147
  record, 154
  SEG, 149
  SHL, 142
  SHORT, 146
  SHR, 142
  SIZE, 152
  THIS, 146
  TYPE, 150
  WIDTH, 155
  XOR, 144
operator precedence, 159
operators, 137
OPTIMIZE assembler control, 202
optimizing, 27
OR operator, 144
ORDER loader command, 403–404
order of input files, 33
ORG directive, 108
override
  group, 148
  segment, 148, 172
  segment override checked against ASSUME, 173

P
  % (metacharacter), 245, 255
page align-type loader, 374
page eject, 25
page length, 27
page width, 27
PAGELENGTH assembler control, 202
PAGEWIDTH assembler control, 203
paragraph number loader, 370
physical segment, 81
  loader, 369
+ 145
  binary, 138
  unary, 137
  with base & index register, 134
porting tool
  L_to_o86, 304
  nm64, 305
position of operand, 164
pre-defined macro function, 253–266
precedence
  of operators, 159
prefix in syntax, 69
preprocessor, 30–31
primary assembler controls, 195, 197
primary controls, 25
primary functions
  loader, 368
PRINT assembler control, 203
PROC directive, 109–110
  default, 109
processor mode
  80186, 209
  8086, 209
  differences, 209–210
  V20, 209
program linkage, 84
program linkage directive, 84
program segmentation, 81
PTR operator, 147
PUBLIC directive, 111
PUBLIC loader command, 405
public segment
  loader, 373
PURGE directive, 112–113
Q
  quoted string
    as expression operand, 132
R
  real constant, 66
  record
    differences from structure, 84
    as expression operand, 132
record (continued)
  initialization, 115
  name defined, 63
  similarities to structure, 83
RECORD directive, 114–117
record field
  as expression operand, 132
  name defined, 63
record operator, 154
register
  16-bit, 167
  8-bit, 167
  8087, 168
  assumed type, 163
  base, 167
  floating point, 168
  index, 167
  segment, 81, 167–168
register indirect expression, 134
relocatable expression, 130
relocatable segment
  loader, 370
RELOCATION TYPE attribute, 75
REPEAT macro function, 262
REPLACE librarian command, 450
RESADD loader command, 406
RESNUM loader command, 406
RESTORE assembler control, 207
ROM, 18
S
  17-bit number, 130
SAVE assembler control, 208
SAVE librarian command, 450
saving and restoring settings, 28
SEG loader command, 407–408
SEG operator, 149
segment
  addressability, 170
  default, 82
  logical, 82
  maximum number, 121
  nesting, 121
segment (continued)
  override operator, 148
  register, 81
SEGMENT ADDRESSABILITY attribute, 76
SEGMENT attribute, 75
SEGMENT directive, 118–122
segment name
defined, 64
  as expression operand, 133
segment number, loader, 370
segment override, 172
  checked against ASSUME, 173
segment register
default value, 87
  initialization, 95
SEGMENT RELOCATION attribute, 75
segmentation
directive, 81
  multiple segment definition, 120
  of program, 81
segments and load addresses
  loader, 369
SEGSIZE loader command, 409
SET macro function, 263
severity, message, 389
sharing code between files, 30
SHL operator, 142
SHORT operator, 146
SHR operator, 142
SIZE operator, 152
/ operator, 141
specifier (codemacro specmod), 220
specmod, 219
stack segment
  loader, 373
* (comment) loader command, 389
START loader command, 410
string
  as expression operand, 132
  with DB directive, 92
  with DW, DD, DQ, DT directive, 92
Index

string relational macro function, 257
STRUC directive, 123–126
structure
differences from record, 84
initialization, 124
name defined, 63
similarities to record, 83
structure field
as expression operand, 134
name defined, 63
SUBSTR macro function, 245, 264
subtraction operator
binary, 138
unary, 137
suffixes, 23–24, 29, 33, 45–46
supported instruction set, 52
supported microprocessors, 52
symbol
EQU symbols, 64
group name, 64
instruction mnemonic, 62
keyword, 60
label, 62
label with colon, 62
macro preprocessor, 251
record field name, 63
record name, 63
segment name, 64
structure field name, 63
structure name, 63
variable, 62
symbol in syntax, 59
symbol information, 26
symbol table format, 213–216
symbol with loader, 384
SYMBOLS assembler control, 204
syntax
blank line, 70
comment, 69
continuation line, 70
keyword, 69
syntax (continued)
  label, 69
  operand, 69
  prefix, 69
  symbol, 59

T  THIS operator, 146
  TITLE assembler control, 208
  translation
    acvt86 tool, 290–293
    HP 64853 to HP B1449, 279–306
  TYPE assembler control, 204
  TYPE attribute, 73
  type limit, 411
  TYPE operator, 150
  TYPEMERGE loader command, 411

U  unary minus, 137
  unary plus, 137
  unreferenced externals, 28
  UNREFERENCED_EXTERNALS assembler control, 204
  unsetting flags, 24
  upper case
    See case-sensitivity
  user-defined macro, 269
  user-defined macros, 267–274

V  V20 processor mode, 209
  V20/V30 mnemonics, 27
  variable, 83
  variable name
    defined, 62
    as expression operand, 133
  version number, 275–278

W  WARN loader command, 389
  warnings, suppressing, 28
  WHILE macro function, 264
  WIDTH loader command, 412
  WIDTH operator, 155
  word align-type
    loader, 374
  WORKFILES assembler control, 205
Index

X XOR operator, 144
XREF assembler control, 205
Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau’s calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer’s facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer’s facility only upon HP’s prior agreement and Buyer shall pay HP’s round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.
For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

**Limitation of Warranty**

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

**Exclusive Remedies**

The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.