Operating Systems and Languages Library

MS-MACRO ASSEMBLER under XENIX V

User Guide and Reference Manual

olivetti

RELATED PUBLICATIONS

MS-Macro Assembler Under XENIX V Reference Manual (Code 4033800 B)

XENIX V User Guide (Code 4022940 Y)

XENIX V User and System Administrator Reference Manual (Code 4022320 Y)

XENIX V Installation and System Administration User Guide (Code 4022960 S)

XENIX V System and Application Software Development Tools User Guide (Code 4022990 B)

XENIX V System and Application Software Development Tools Reference Manual (Code 4031710 V)

C-Language Under XENIX V User Guide (Code 4022970 T)

C-Language Under XENIX V Reference Manual (Code 4033810 Y)

X/OPEN Portability Guide (Code 4024080 B)

XENIX V Text Processing User Guide (Code 4022980 C)

XENIX V Text Processing Reference Manual (Code 4031720)

DISTRIBUTION: General (G)

FIRST EDITION: July 1986

SECOND EDITION: October 1986

Copyright © Microsoft Corporation 1980/1985

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C. S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

PREFACE

This manual describes how to create and debug assembly language programs using the XENIX V Macro Assembler, MASM. It should be used in conjunction with the "MS-Macro Assembler Under XENIX V Reference Manual". The "XENIX V System and Application Software Development Tools Reference Manual" contains formal definitions of MASM(CP) and LD(CP), and the supporting utilities such as CREF(CP).

MASM is a powerful two-pass assembler for the Intel 8086/80186/80286 family of microprocessors, and the 8087 and 80287 floating point coprocessors. Strong typing is available for memory operands, and conditional directives allow the exclusion of parts of the source code from assembly. In addition, a range of error detection facilities is provided.

The following syntax notations are used:

[ ] Square brackets indicate that the enclosed entry is optional.

{ } Braces indicate that there is a choice between two or more entries. At least one of the entries enclosed in the braces must be chosen, unless the entries are also enclosed in square brackets.

| A vertical stroke refers to the logical OR operation.

... Ellipses indicate that an entry may be repeated as many times as needed.

CAPS Capital letters indicate portions of statements or commands that must be entered exactly as shown.

_ The underscore joins multiple-name parameters.

All other punctuation, such as commas, colons, slash marks and equals signs, must be entered exactly as shown.

SUMMARY

This manual consists of three main sections. Chapter 1 is a brief introduction to starting up MASM. Chapter 2 descibes how to invoke MASM, and how to read the assembly code listings generated by MASM. For a complete description of the Macro Assembler elements, see "MS-Macro Assembler Under XENIX V Reference Manual". Appendix A lists and explains the error messages that may be encountered during use of MASM and the XENIX linker, ld.

TRADEMARK NOTICE

- OLIVETTI is a trademark of Ing. C. Olivetti & C., S.p.A.
- OLITERM is a trademark of Ing. C. Olivetti & C., S.p.A.
- SORTP is a trademark of Ing. C. Olivetti & C., S.p.A.
- ASM-86 is a trademark of Digital Research
- CB-86 is a trademark of Digital Research
- CLEO is a trademark of Phone 1 Inc.
- ETHERNET is a trademark of Xeros Corp.
- GW is a trademark of Microsoft Corp.
- IBM is registered trademark of International Business Machines Corp.
- MICROSOFT is a registered trademark of Microsoft Corp.
- MS is a trademark of Microsoft Corp.
- OMNINET is a trademark of Corvus System Inc.
- p-System is a trademark of Softech Microsystems, Inc.
- PC-DOS is a trademark of International Business Machines Corp.
- PEACHPACK is a trademark of Peachtree Software International Ltd.
- SID-86 is a trademark of Digital Research
- TerminALL is a trademark of TOPIC System, Inc.
- TOPIC is a trademark of TOPIC System, Inc.
- UNIX is a trademark of Bell Laboratories
- XENIX V is a trademark of Microsoft Corporation
- Z80 is a registered trademark of Zilog Inc.
- Z8000 is a registered trademark of Zilog Inc.

CONTENTS

1. INTRODUCTION 1-1

   WHAT YOU NEED 1-1

   HOW TO BEGIN 1-1

2. MASM: A MACRO ASSEMBLER 2-1

   INTRODUCTION 2-1

   STARTING AND USING MASM 2-1

   ASSEMBLING A SOURCE FILE 2-1

   USING MASM OPTIONS 2-2

   OUTPUTTING SEGMENTS IN ALPHABETICAL ORDER 2-2

   SYMBOL DEFINITION 2-3

   CREATING A PASS 1 LISTING 2-3

   CREATING CODE FOR A FLOATING POINT EMULATOR 2-4

   INCLUDE FILE PATHNAMES 2-4

   PRODUCE LISTING FILE 2-4

   PRESERVING LOWER CASE NAMES 2-5

   CONVERTING NAMES TO UPPERCASE 2-5

   PRESERVING LOWERCASE IN PUBLIC AND EXTERNAL NAMES 2-5

   SUPPRESS SYMBOL TABLE INFORMATION 2-6

   OUTPUT OBJECT CODE 2-6

   CHECKING FOR IMPURE MEMORY REFERENCES 2-7

   CREATING CODE FOR A FLOATING POINT PROCESSOR 2-7

   OUTPUT ASSEMBLER STATISTICS 2-8

   LISTING FALSE CONDITIONALS 2-8

   OUTPUT ERROR MESSAGES 2-8

   READING THE ASSEMBLY LISTING 2-9

   READING PROGRAM CODE 2-9

   READING A MACRO TABLE 2-10

   READING A STRUCTURE AND RECORD TABLE 2-11

READING A STRUCTURE AND RECORD TABLE 2-11

READING A SEGMENT AND GROUP TABLE 2-12

READING A SYMBOL TABLE 2-13

READING A PASS 1 LISTING 2-14

1. INTRODUCTION

ABOUT THIS CHAPTER

This chapter is a brief introduction on how to start up MASM, and what is needed to create and debug programs.

CONTENTS

WHAT YOU NEED 1-1

HOW TO BEGIN 1-1

INTRODUCTION

This manual is an introduction to the MASM Macro Assembler. It describes how to create and debug an assembler program, and how to interpret the messages returned by MASM and the XENIX linker, ld.

WHAT YOU NEED

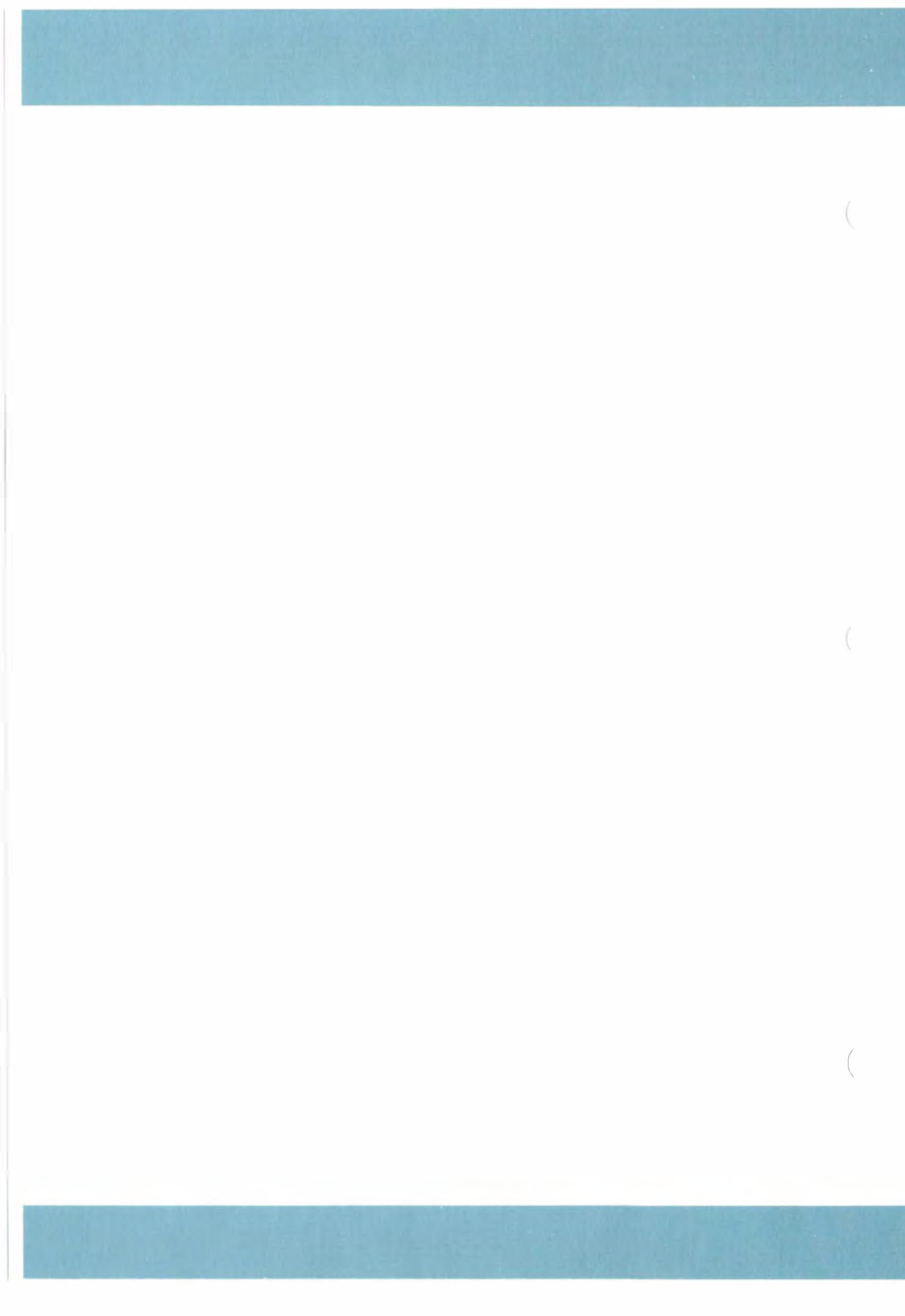
To make an assembly language program, you need a text editor and you need to know the correct syntax and format of assembly language source files. In addition, you need to be familiar with the function and operation of the instruction sets for the 8086/80186 (186)/80286 (286) family of microprocessors.

The MASM Macro Assembler supports these instruction sets and creates programs that can be executed within the 8086/186/286 family. (This family includes the 8086, 8088, 186, and 286 microprocessors and the 8087 and 287 coprocessors.) MASM provides a logical program syntax ideally suited for the segmented architecture of the 8086. This syntax is fully explained in the "MS-Macro Assembler Under XENIX V Reference Manual", which describes the assembly language directives, operands and expressions.

HOW TO BEGIN

You begin by creating an assembly language source file with a text editor. Any of the editors supported by XENIX will work. Then you assemble the source file using MASM.

Once you have tested the program, you can invoke it from the command line at any time. The assembly programs that you create, like all other programs, can accept command parameters, be copied to other systems, and be invoked with shell scripts. The techniques are fully described in this manual.



2. MASM: A MACRO ASSEMBLER

ABOUT THIS CHAPTER

This chapter describes how to invoke MASM, and how to read the listings produced during the various assembly stages.

CONTENTS

INTRODUCTION	2-1
STARTING AND USING MASM	2-1
ASSEMBLING A SOURCE FILE	2-1
USING MASM OPTIONS	2-2
OUTPUTTING SEGMENTS IN ALPHABETICAL ORDER	2-2
SYMBOL DEFINITION	2-3
CREATING A PASS 1 LISTING	2-3
CREATING CODE FOR A FLOATING POINT EMULATOR	2-4
INCLUDE FILE PATHNAMES	2-4
PRODUCE LISTING FILE	2-4
PRESERVING LOWER CASE NAMES	2-5
CONVERTING NAMES TO UPPERCASE	2-5
PRESERVING LOWERCASE IN PUBLIC AND EXTERNAL NAMES	2-5
SUPPRESS SYMBOL TABLE INFORMATION	2-6
OUTPUT OBJECT CODE	2-6
CHECKING FOR IMPURE MEMORY REFERENCES	2-7
CREATING CODE FOR A FLOATING POINT PROCESSOR	2-7
OUTPUT ASSEMBLER STATISTICS	2-8
LISTING FALSE CONDITIONALS	2-8
OUTPUT ERROR MESSAGES	2-8
READING THE ASSEMBLY LISTING	2-9
READING PROGRAM CODE	2-9
READING A MACRO TABLE	2-10

READING A SEGMENT AND GROUP TABLE 2-12

READING A SYMBOL TABLE 2-13

READING A PASS 1 LISTING 2-14

A. INTRODUCTION A-1

MACRO ASSEMBLER MESSAGES A-1

LINKER MESSAGES A-7

INTRODUCTION

The XENIX Macro Assembler, MASM, assembles 8086, 186, and 286 assembly language source files and creates relocatable object files that can be linked and executed under the XENIX operating system. This chapter explains how to invoke MASM and describes the format of assembly listings generated by MASM. For a complete description of the syntax of assembly language source files, see the "MS-Macro Assembler Under XENIX V Reference Manual".

STARTING AND USING MASM

This section explains how to start and use MASM to assemble your program source files.

Assembling a Source File

You can assemble a program source file by typing the MASM command name and the names of the files you wish to process. The command line has the form:

```
MASM [ options ] filename
```

The options given can be any combination of MASM options. These options are described in the section entitled "Using MASM Options", below. Options can be placed anywhere on the command line.

The filename given must be the name of the source file to be assembled.

Unless otherwise specified MASM uses a default filename for the relocatable object code. The default filename is the same as the source file, except that the filename extension is replaced with ".o".

MASM also uses a default filename for the listing file (if requested). The assembly listing lists the assembled code for each source statement and the names and types of symbols defined in the program. If you do not request a listing by using the appropriate option, MASM does not create an assembly listing. The default filename is the same as the source file, except that the filename extension is replaced by ".lst".

For a full specification of the MASM options and command syntax, see masm(CP) in the "XENIX V System and Application Software Development Tools Reference Manual".

USING MASM OPTIONS

The MASM options control the operation of the assembler and the format of the output files it generates.

MASM has the following options:

-a	Alphabetical ordering for segments
-Dsym	Define symbol, "sym".
-d	Output a pass 1 listing
-e	Emulated floating point instructions
-Ipath	Include file pathname, "path".
-l[file]	Output listing to "file", not default
-Ml	Preserve case sensitivity in names
-Mu	Map all symbols to uppercase
-Mx	Preserve case sensitivity in externals
-n	Supress symbol table listing
-o[file]	Output object code to "file", not default
-p	Check for impure memory references
-r	Real floating point instructions
-v	Output verbose assembler statistics
-X	False conditional listing toggle
-x	List errors to standard error channel

You can place options anywhere on a MASM command line. An option affects all relevant files in the command line even if the option appears at the end of the line.

Outputting Segments in Alphabetical Order

The -a option directs MASM to place the assembled segments in alphabetical order before copying them to the object file. If this option is not given, MASM copies the segments in the order encountered in the source file.

Example

```
masm -a file.s
```

This example creates an object file, "file.o", whose segments are arranged in alphabetical order. Thus, if the source file "file.s" contains definitions for the segments DATA, CODE and MEMORY, the assembled segments in the object file have the order CODE, DATA, and MEMORY.

Symbol Definition

The -D option directs MASM to define the symbol appended to the -D option as a text macro with a null value. (See the EQU directive in the "MS-Macro Assembler Under XENIX V Reference Manual" for a discussion of text macros.) The symbol will be defined with the case in effect at that point in the command line. Any number of -D options can be used. The defined symbol can be tested with the IFDEF and IFNDEF directives during the assembly.

Example

```
masm -DSymbol file.s
```

This example directs MASM to define the symbol "Symbol" as a null text macro. The default conversion to uppercase will occur in this example.

Creating a Pass 1 Listing

The -d option directs MASM to add a pass 1 listing to the assembly listing file, making the assembly listing show the results of both assembler passes. A pass 1 listing is typically used to locate and understand program phase errors. Phase errors occur when MASM makes assumptions about the program in pass 1 that are not valid in pass 2.

The -d option does not create a pass 1 listing unless you also direct MASM to create an assembly listing. It does direct MASM to display error messages for both pass 1 and pass 2 of the assembly, even if no assembly listing is created.

Example

```
masm -d -l file.s
```

This example directs MASM to create a pass 1 listing for the source file "file.s". The listing is placed in the file "file.lst".

Creating Code For a Floating Point Emulator

The -e option directs MASM to generate floating point instruction codes that can be fixed up at link time to software interrupts.

This is the default option under XENIX V. If a 287 is present, the XENIX V system changes the software interrupts into real 287 instructions. If the chip is not present, a software emulator in the XENIX V system is used to process the software interrupts as if the 287 chip were actually present. The emulator does not handle all valid 287 instructions. Unemulated instructions will give a SIGILL signal.

Example

```
masm -e file.s
```

This example directs MASM to create emulation code for any floating point instructions it finds in the program.

Include File Pathnames

The -I option directs MASM to use the specified pathname as a prefix to the filenames given in the INCLUDE directives in an assembly program. Up to ten -I options can be specified on the command line. To force searching of the current directory in a specific order, -I can be used.

Example

```
masm -I /usr/include -I. file.s
```

This example forces the INCLUDE directives to search /usr/include, then the current directory, for the given filename.

Produce Listing File

The -l option directs MASM to generate a listing file to the standard output file, which is usually to the console device. If the -l option has a filename appended to it (of the form "-llistfile"), then the listing is written to the file "listfile" rather than the default listing file whose name is the same as that of the first input file except that it has the extension ".lst".

Examples

```
masm -l file.s
```

This example directs MASM to generate a listing in the file "file.lst".

```
masm -llist file.s
```

This example directs MASM to generate a listing in the file "list".

Preserving Lowercase Names

The -Ml option directs MASM to preserve lowercase letters in label, variable, and symbol names. This means names that have the same spelling but use different case letters are considered unique. For example, with the -Ml option, "DATA" and "data" are unique. Under XENIX V, this is the default case mapping option.

THE -Ml option is typically used when a source file is to be linked with object modules created by a case-sensitive compiler.

Example

```
masm -Ml file.s
```

This example directs MASM to preserve lowercase letters in any names defined in the source file "file.s".

Converting Names To Uppercase

The -Mu option directs MASM to convert all letters in all symbols to uppercase.

Example

```
masm -Mu file.s
```

This example directs MASM to convert lowercase letters in any names defined in the source file "file.s".

Preserving Lowercase in Public and External Names

The -Mx option directs MASM to preserve lowercase letters in public and external names only when copying these names to the object file. For all other purposes, MASM converts the lowercase letters to uppercase.

Public and external names are any label, variable, or symbol names that have been defined using the EXTRN or PUBLIC directives. Since MASM converts the letters to uppercase for assembly, these names must have unique spellings. That is, the names "DATA" and "data" are not unique.

The -Mx option is used to ensure that the names of routines or variables copied to the object module have the correct spelling. The option is used with any source file that is to be linked with object modules created by a case-sensitive compiler, and is particularly useful for transporting assembler files from MS-DOS to XENIX V when working with C.

Example

```
masm -Mx file.s
```

This example directs MASM to preserve lowercase letters in any public or external names defined in the source file "file.s".

Suppress Symbol Table Information

The -n option directs MASM to suppress information about the symbols used in the assembled program. For this option to take effect, the -l option must also be used.

Example

```
masm -l -n file.s
```

This example directs MASM to generate a listing file without any symbol information in the file "file.lst".

Output Object Code

The -o option directs MASM to generate an object code file. If the -o option has a filename appended to it (of the form "-oobjfile"), then the object code is written to the file "objfile" rather than the default file whose name is the same as that of the first input file except that it has the extension ".o".

The -o option without a filename suppresses the generation of an object file.

Example

```
masm -oObj file.s
```

This example directs MASM to generate object code in the file "Obj".

Checking for Impure Memory References

The -p option directs MASM to check for impure memory references. This option ensures that you don't do any explicit stores into memory via the CS:override. If you want your code to run in protected mode 286, you can use the -p option to avoid errors due to impure memory references. For example, a typical violation might look like this:

```
    .286p
code            segment
                assume      cs:code
                .
                .
                .
codewrd dw                  ?
                .
                .
                .
                move        cw:codewrd, <data>
```

If the above example were assembled with -p, it would generate an error message.

Example

```
masm -p file.s
```

This example directs MASM to check the source file, "file.s", for any impure memory references.

Creating Code For a Floating Point Processor

The -r option directs MASM to generate floating point instruction code that can be executed by an 8087 or 287 coprocessor. Programs created using the -r option can only run on máchines having an 8087 or 287 coprocessor.

Example

```
masm -r file.s
```

This example directs MASM to assemble the source file, "file.s", and create actual 8087 or 287 instruction code for floating point instructions.

Output Assembler Statistics

The -v option directs MASM to print the number of source lines, the number of lines assembled, the number of symbols in addition to the standard statistics of bytes of symbol space available, and the number of warning and severe errors.

Example

```
masm -vfile.s
```

This example directs MASM to give additional assembly statistics.

Listing False Conditionals

The -X option directs MASM to copy to the assembly listing all statements forming the body of an IF directive whose expression (or condition) evaluates to false. If you do not give the -X option in the command line, MASM suppresses all such statements. The -X option lets you display conditionals that do not generate code. This option applies to all IF directives: IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN, and IFDIF.

The -X option behaves like an initial .TFCOND directive in a source file. The .SFCOND and .LFCOND directives supercede the -X option and .TFCOND directive. See the "MS-Macro Assembler Under XENIX V Reference Manual" for a complete discussion of the .TFCOND, .SFCOND, .LFCOND directives.

The -X option does not affect the assembly listing unless you direct MASM to create an assembly listing file.

Example

```
masm -X -l file.s
```

If the source file, "file.s", does not contain a .TFCOND directive, this example directs MASM to list all false conditionals it finds in the source file.

Output Error Messages

The -x option directs MASM to print error messages on the standard error channel, in addition to the messages generated in the listing file, without displaying the the source line in error. If -l is given, then the -x option has no effect. By using this option the assembler will assemble faster. Error messages can be completely suppressed by using the -x option, which makes assemblies silent, i.e. sending no output to STDERR.

Example

```
masm -x file.s
```

This example directs MASM to print copies of error messages only, written to the standard error file.

READING THE ASSEMBLY LISTING

MASM creates an assembly listing of your source file whenever you give an assembly listing filename on the MASM command line. The assembly listing contains a list of the statements in your program and the object code generated for each statement. The listing also lists the names and values of all labels, variables, and symbols in your source file. MASM creates one or more tables for macros, structures, records, segments, groups, and other symbols and places these tables at the end of the assembly listing.

MASM lists symbols only if it encounters any in the program. If there are no symbols in your program for a particular table, the given table is omitted. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

The assembly listing will also contain error messages if any errors occur during assembly. MASM places the messages below the statements that caused the errors. At the end of the listing, MASM displays the number of error and warning messages it issued.

The following sections explain the format of the assembly listing and the meaning of special symbols used in the listing.

Reading Program Code

MASM lists the program code generated from the statements of a source file. Each line has the form:

[line-number] offset code statement

The "linenumber" is from the first statement in the assembly listing. The line numbers are given only if a cross reference file is also being created. The "offset" is the offset from the beginning of the current segment to the code. The "code" is the actual instruction code or data generated by MASM for the statement. MASM gives the actual numeric value of the code if possible. Otherwise, it indicates what action needs to be taken to compute the value. The "statement" is the source statement shown exactly as it appears in the source file, or after processing by a MACRO, IRP, or IRPC directive.

If any errors occur during assembly, the error message will be printed directly below the statement where the error occurred, displaying the

source file and line number in addition to the error number and error message.

MASM uses the following special characters to indicate addresses that need to be resolved by the linker or values that were generated in a special way:

Character	Meaning
R	Relocatable address; linker must resolve
E	External address; linker must resolve
----	Segment/group address; linker must resolve
=	EQU or = directive
nn:	Segment override in statement
nn/	REP or LOCK prefix instruction
nn [xx]	DUP expression; nn copies of the value xx
+	Macro expansion
C	Included line from INCLUDE file

Example

```
                            extrn go:near

0000                        data    segment public 'DATA'
                                    assume es:data
0000 0002                   s2      dw      2
0002                        data    ends

0000                        code    segment public 'CODE'
                                    assume cs:code
0000                        start:
0000 E8 0000 E                      call    go
0003 36:A1 0000 R                   mov     ax, s2
0007 B4 4C                          mov     ah, 4ch
0009 CD 21                          int     21h
000B                                code    ends

                                    end
```

Reading a Macro Table

MASM lists the names and sizes of all macros defined in a source file. The list has two columns: Name and Length.

The Name column lists the names of all macros. The names are listed in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Length column lists the size of the macro in terms of non-blank lines. This size is in hexadecimal.

Example

Name	Length
BIOSCALL	0002
DISPLAY	0005
DOSCALL	0002
KEYBOARD	0003
LOCATE	0003
SCROLL	0004

Reading a Structure and Record Table

MASM lists the names and dimensions of all structures and records in a source file. The table contains two sets of overlapping columns. The Width and # Fields list information about the structure or record. The Shift, Width, Mask, and Initial columns list information about the structure or record members.

The Name column lists the names of all structures and records. The names are listed in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

For a structure, the Width column lists the size (in bytes) of the structure. The # Fields column lists the number of fields in the structure. Both values are in hexadecimal.

For fields of structures, the Shift column lists the offset (in bytes) from the beginning of the structure to the field. This value is in hexadecimal. The other columns are not used.

Example

Name	Width Shift	# Fields Width	Mask	Initial
PARMLIST	001C	0004		
BUFSIZE	0000			
NAMESIZE	0001			
NAMETEXT	0002			
TERMINATOR		001B		

For a record, the Width column lists the size (in bits) of the record. The # Fields column lists the number of fields in the record.

For fields in a record, the Shift count lists the offset (in bits) from the lower order bit of the record to the first bit in the field. The Width column lists the number of bits in the field. The Mask column lists the maximum value of the field, expressed in hexadecimal. The Initial column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed

in the record and all other fields were set to 0.

Example

Name		Width Shift	# Fields Width	Mask	Initial
RECO		0008	0003		
	FLD1	0006	0002	00C0	0040
	FLD2	0003	0003	0038	0000
	FLD3	0000	0003	0007	0003
REC1		000B	0002		
	FLD1	0003	0008	07F8	0400
	FLD2	0000	0003	0007	0002

Reading a Segment and Group Table

MASM lists the names, sizes, and attributes of all segments and groups in a source file. The list has five columns: Name, Size, Align, Combine, and Class.

The Name column lists the names of all segments and groups. The names in the list are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name. Names are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Size column lists the size (in bytes) of each segment. Since a group has no size, only the word GROUP is shown. The size, if given, is in hexadecimal.

The Align column lists the alignment type of the segment. The types can be any of the following:

BYTE
WORD
PARA
PAGE

If the segment is defined with no explicit alignment type, MASM lists the default alignment for that segment.

The Combine column lists the combine type of the segment. The types can be any one of the following:

NONE
PUBLIC
STACK
MEMORY
COMBINE

NONE is given if no explicit combine type is defined for the segment. NONE represents the private combine type.

The Class column lists the class name of the segment. The name is spelled exactly as given in the source file. If no name is given, none is shown.

Example

Name	Size	Align	Combine	Class
AAAXQQ	0000	WORD	NONE	'CODE'
DGROUP	GROUP			
DATA	0024	WORD	PUBLIC	'DATA'
STACK	0014	WORD	STACK	'STACK'
CONST	0000	WORD	PUBLIC	'CONST'
HEAP	0000	WORD	PUBLIC	'MEMORY'
MEMORY	0000	WORD	PUBLIC	'MEMORY'
ENTXCM	0037	WORD	NONE	'CODE'
MAIN_STARTUP	007E	PARA	NONE	'MEMORY'

Reading a Symbol Table

MASM lists the names, types, values, and attributes of all symbols in the source file. The table has four columns: Name, Type, Value, and Attr.

The Name column lists the names of all symbols. The names in the list are given in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Type column lists each symbol's type. A type is given as one of the following:

L NEAR	A near label
L FAR	A far label
N PROC	A near procedure label
F PROC	A far procedure label
Number	An absolute label
Alias	An alias for another symbol
Opcode	An instruction opcode
Text	A memory operand, string, or other value

If Type is Number, Opcode, Alias, or Text, the symbol is defined by an EQU directive or an = directive. The Type column also lists the symbol's length if it is known. A length is given as one of the following:

BYTE	One byte (8-bits)
WORD	One word (16-bits)
DWORD	Doubleword (2 words)
QWORD	Quadword (4 words)
TBYTE	Ten-bytes (5 words)

A length can also be given as a number. In this case, the symbol is a structure, and the number defines the length (in bytes) of the structure. For example, the type:

```
L 0031
```

identifies a label to a structure that is 31 bytes long.

The Value column shows the numeric value of the symbol. For absolute symbols, the value represents an absolute number. For labels and variable names, the value represents that item's offset from the beginning of the segment in which it is defined. If Type is Number, Opcode, Alias, or Text, the Value column shows the symbol's value, even if the value is simple text. Number shows a constant numeric value. Opcode shows a blank (the symbol is an alias for an instruction mnemonic). Alias shows the name of another symbol. Text shows the text the symbol represents. Text is any operand that does not fit one of the other three categories.

The Attr column lists the attributes of the symbol. The attributes include the name of the segment in which the symbol is defined, if any, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the EXTRN or PUBLIC directives. The scope can be External or Global. The code length is given only for procedures.

Example

```
Symbols:

Name      Type      Value       Attr

SYM       Number    0005
SYM1      Text      1.234
SYM2      Number    0008
SYM3      Alias     SYM4
SYM4      Text      5[BP][DI]
SYM5      Opcode
SYM6      L BYTE    0002        DATA
SYM7      L WORD    0012        DATA Global
SYM8      L DWORD   0022        DATA
SYM9      L QWORD   0000        External
LAB0      L FAR     0000        External
LAB1      L NEAR    0010        CODE
```

Reading a Pass 1 Listing

When you specify the -d option in the MASM command line, MASM adds a pass 1 listing to the assembly listing file, making the listing file show the results of both assembler passes. The listing is intended to help locate the source of phase errors.

The following examples illustrate the pass 1 listing for a source file that assembled without error. Although an error was produced on pass 1, MASM corrected the error on pass 2 and completed assembly correctly.

During pass 1, a JLE instruction to a forward reference produces an error message:

```
0017    7E 00               JLE     SMLSTK
file(line) : error 9: Symbol not defined SMLSTK
0019    BB 1000                     MOV     BX,4096
001C                SMLSTK:
```

MASM displays this error since it has not yet encountered the definition for the symbol SMLSTK.

By pass 2, SMLSTK has been defined and MASM can fix the instruction so no error occurs:

```
0017    7E 03               JLE     SMLSTK
0019    BB 1000                     MOV     BX,4096
001C                SMLSTK:
```

The JLE instruction's code now contains 03 instead of 00. This is a jump of 3 bytes.

Since MASM generated the same amount of code for both passes, there was no phase error. If a phase error had occurred, MASM would have displayed an error message.

In the following program fragment, a mistyped label creates a phase error. In pass 1, the label "go" is used in a forward reference and creates a "Symbol not defined" error. MASM assumes that the symbol will be defined later and generates three bytes of code, reserving two bytes for the symbol's actual value.

```
0000                        code    segment
0000 E9 0000    U                   jmp      go
file(line) : error 9: Symbol not defined go
0003            go          label   byte
0003 BB 0001                mov     ax, 1

0006                        code    ends
```

In pass 2, the label "go" is known to be a label of BYTE type which is an illegal type for the JMP instruction. As a result, MASM produces only two bytes of code in pass 2, one less than in pass 1. The result is a phase error.

```
0000                    code    segment
0003 R                  jmp     go
file(line) : error 57: Illegal size for item
0003            go      label   byte
file(line) : error 6: Phase error between passes
0003 B8 0001            mov     ax, 1
0006                    code    ends
```

A. INSTRUCTION SUMMARY

ABOUT THIS APPENDIX

This appendix lists all the microprocessor and coprocessor instruction sets.

CONTENTS

INTRODUCTION	A-1
8086 INSTRUCTIONS	A-1
8087 INSTRUCTIONS	A-5
186 INSTRUCTIONS	A-7
286 NON-PROTECTED INSTRUCTIONS	A-8
286 PROTECTED INSTRUCTIONS	A-8
287 INSTRUCTIONS	A-9

INTRODUCTION

This appendix lists and explains the error messages that can be generated by the Macro Assembler, MASM, and the Linker, LD.

MACRO ASSEMBLER MESSAGES

This section lists and explains the messages displayed by the Macro Assembler, MASM. MASM displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message contains a count of errors and warning messages it displayed during the assembly. The message has the form:

```
n   Bytes of symbol space free
n   Warning Errors
n   Severe Errors
```

This message is also copied to the source listing.

The Assembler Errors are as follows:

0: Block nesting error
Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is closing an outer level of nesting with inner level(s) still open.

1: Extra characters on line
This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

2: Register already defined
This will only occur if the assembler has internal logic errors.

3: Unknown symbol type
Symbol statement has something in the type field that is unrecognizable.

4: Redefinition of symbol
This error occurs on pass 2 and succeeding definitions of a symbol.

5: Symbol is multi-defined
This error occurs on a symbol that is later redefined.

6: Phase error between passes
The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required.

There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the -D option to produce a listing to aid in resolving phase errors between passes. See Chapter 2, "MASM: A Macro Assembler."

7: Already had ELSE clause
Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

8: Not in conditional block
An ENDIF or ELSE is specified without a previous conditional assembly directive active.

9: Symbol not defined
A symbol is used that has no definition.

10: Syntax error
The syntax of the statement does not match any recognizable syntax.

11: Type illegal in context
The type specified is of an unacceptable size.

12: Should have been group name
Expecting a group name but something other than this was given.

13: Must be declared in pass 1
An item was referenced before it was defined in pass 1. For example, "IF DEBUG" is illegal if DEBUG is not previously defined.

14: Symbol type usage illegal
Illegal use of a PUBLIC symbol.

15: Symbol already different kind
Attempt to define a symbol differently from a previous definition.

16: Symbol is reserved word
Attempt to use an assembler reserved word illegally. For example, to declare MOV as a variable.

17: Forward reference is illegal
Attempt to reference something before it is defined in pass 1.

18: Must be register
Register expected as operand but you furnished a symbol that was not a register.

19: Wrong type of register
Directive or instruction expected one type of register, but another was specified. For example, INC CS.

20: Must be segment or group
Expecting segment or group and something else was specified.

21: Symbol has no segment
Trying to use a variable with SEG, and the variable has no known

segment.

22: Must be symbol type
Must be WORD, DW, QW, BYTE, or TB but received something else.

23: Already defined locally
Tried to define a symbol as EXTERNAL that had already been defined locally.

24: Segment parameters are changed
List of arguments to SEGMENT were not identical to the first time this segment was used.

25: Not proper align/combine type
SEGMENT parameters are incorrect.

26: Reference to mult defined
The instruction references something that has been multi-defined.

27: Operand was expected
Assembler is expecting an operand but an operator was received.

28: Operator was expected
Assembler was expecting an operator but an operand was received.

29: Division by 0 or overflow
An expression is given that results in a divide by 0 or a number larger then can be represented.

30: Shift count is negative
A shift expression is generated that results in a negative shift count.

31: Operand types must match
Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

32: Illegal use of external
Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

33: Must be record field name
Expecting a record field name but received something else.

34: Must be record or field name
Expecting a record name or field name and received something else.

35: Operand must have size
Expected operand to have a size, but it did not.

36: Must be var, label or constant
Expecting a variable, label, or constant but received something else.

37: Must be structure field name
Expecting a structure field name but received something else.

38: Left operand must have segment
Used something in right operand that required a segment in the left operand. (For example, ":.")

39: One operand must be const
This is an illegal use of the addition operator.

40: Operands must be same or 1 abs
Illegal use of the subtraction operator.

41: Normal type operand expected
Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

42: Constant was expected
Expecting a constant and received an item that does not evaluate to a constant. For example, a variable name or external.

43: Operand must have segment
Illegal use of SEG directive.

44: Must be associated with data
Use of code related item where data related item was expected. For example, MOV AX,<code-label>.

45: Must be associated with code
Use of data related item where code item was expected.

46: Already have base register
Trying to double base register.

47: Already have index register
Trying to double index address.

48: Must be index or base register
Instruction requires a base or index register and some other register was specified in square brackets, [ ].

49: Illegal use of register
Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

50: Value is out of range
Value is too large for expected use. For example, MOV AL,5000.

51: Operand not in IP segment
Access of operand is impossible because it is not in the current IP segment.

52: Improper operand type
Use of an operand such that the opcode cannot be generated.

53: Relative jump out of range
Relative jumps must be within the range -128 to +127 of the current instruction, and the specific jump is beyond this range.

54: Index displ. must be constant
Illegal use of index display.

55: Illegal register value
The register value specified does not fit into the "reg" field (the value is greater than 7).

56: No immediate mode
Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

57: Illegal size for item
Size of referenced item is illegal. For example, shift of a double word.

58: Byte register is illegal
Use of one of the byte registers in context where it is illegal. For example, "PUSH AL," is illegal.

59: CS register illegal usage
Trying to use the CS register illegally. For example, "XCHG CS,AX," is illegal.

60: Must be AX or AL
Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

61: Improper use of segment reg
Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

62: No or unreachable CS
Trying to jump to a label that is unreachable.

63: Operand combination illegal
Specification of a two-operand instruction where the combination specified is illegal.

64: Near JMP/CALL to different CS
Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

65: Label can't have seg. override
Illegal use of segment override.

66: Must have opcode after prefix
Use of a REPE, REPNE, REPZ, or REPNZ instructions without specifying any opcode after it.

67: Can't override ES segment
Trying to override the ES segment in an instruction where this

override is not legal. For example, "STOS DS:TARGET" is illegal.

68: Can't reach with segment reg
There is no ASSUME that makes the variable reachable.

69: Must be in segment block
Attempt to generate code when not in a segment.

70: Can't use EVEN on BYTE segment
Segment was declared to be byte segment and attempt to use EVEN was made.

72: Illegal value for DUP count
DUP counts must be a constant that is not 0 or negative.

73: Symbol already external
Attempt to define a symbol as local that is already external.

74: DUP is too large for linker
Nesting of DUPs was such that too large a record was created for the linker.

75: Usage of ? (indeterminate) bad
Improper use of the "?". For example, ?+5.

76: More values than defined with
Too many initial values given when defining a variable using a REC or STRUC type.

77: Only initialize list legal
Attempt to use STRUC name without angle brackets, < >.

78: Directive illegal in STRUC
All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

79: Override with DUP is illegal
In a STRUC initialization statement, you tried to use DUP in an override.

80: Field cannot be overridden
In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

81: Override is of wrong type
In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

82: Register can't be forward ref

83: Circular chain of EQU aliases
An alias EQU eventually points to itself.

84: 8087 opcode can't be emulated
Either the 8087 opcode or the operands you used with it produce an

instruction that the emulator cannot support.

85: End of file, no END directive
You forgot an end statement or there is a nesting error.

86: Data omitted with no segment

98: Override value is wrong length
There is an improper sized value in a RECORD or STRUC field.

99: Line too long expanding <symbol>
The line became too long for one of the assembler's internal buffers.

100: Impure memory reference
You attempted to explicitly store into memory via the CS register.

Linker Messages

This section lists the error messages that can occur when linking programs. The messages are in alphabetical order.

-A and -F are mutually exclusive
The -A and -F switches are mutually exclusive.

-u seen before -n <num>
An undefined symbol has been given before the maximum name-length switch. Reverse the order of the two switches.

<switch> ignored
You have given an unrecognized switch.

Address missing
The -A switch is missing a following number.

Array element size mismatch
A far communal array has been declared with two or more different array element sizes (e.g., declared once as an array of characters and once as an array of reals). NOTE: At the present time, communal arrays are not available in MASM.

Attempt to access data outside segment bounds
An LEDATA or an LIDATA record specifies bytes that occupy offsets beyond the end of the segment as defined by the corresponding SEGDEF record. It is hard for you to cause this message; usually, it indicates a bug in the compiler or assembler.

Attempt to put segment 'name' in more than one group in file 'filename'
A segment was declared to be a member of two different groups. Correct the source and recreate the object files.

Cannot create list file
The linker was unable to create the list (map) file. Possibly you do not have permission, or the disk is full.

Cannot find file
  You are specifying an object module or library file which the linker cannot find or is not able to open for reading.

Cannot open run file
  The directory or disk is full, or you don't have the right permissions. Make space on the disk or in the directory, or change permissions.

Cannot open temporary file
  The directory or disk is full. Make space on the disk or in the directory.

Common area longer than 65536 bytes
  Your program has more than 64K of communal variables. NOTE: At the present time, only Microsoft C programs can possibly cause this message to be displayed.

Data record too large
  LEDATA record (in an object module) contains more than 1024 bytes of data. This is a translator error. Note the translator (compiler or assembler) that produced the incorrect object module and the circumstances under which it was produced, and report the information to Microsoft.

Dup record too large
  LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a STRUC definition that is very complex, or a series of deeply nested DUP statements (e.g. ARRAY db 10 dup(11 dup (12 dup (13 dup (...))))). Simplify and reassemble.

Error accessing library
  The linker was unable to open a specified library. Make sure the file exists and has the proper permissions.

Fixup overflow near 'num' in segment 'name' in 'filename(name)' offset 'num'
  Some possible causes are: 1) A group is larger than 64K bytes, 2) your program contains an intersegment short jump or intersegment short call, 3) your have a data item whose name conflicts with that of a subroutine in a library included in the link, and 4) you have an EXTRN declaration inside the body of a segment, for example:

```
CODE    segment public 'code'
extrn   main:far
start   proc    far
        call    main
        ret
start   endp
CODE    ends
```

  The following construction is preferred:

```
extrn   main:far
```

```
CODE     segment public 'code'
start    proc    far
         call    main
         ret
start    endp
CODE     ends
```

Revise the source and recreate the object file.

Group <name> larger than 64Kbytes
A group has been defined that is larger than the maximum allowed for a physical segment. Reduce the size of the group.

Invalid object module
One of the object modules is invalid. Try recompiling. If the error persists, contact Microsoft.

List file name missing
The -m switch is missing a following string.

Multiple code segments -- should be medium model
There is more than one code segment and the -Mm, -Ml or -Me switch was not given. Make sure all modules have the same memory model or link with the -Me switch. Not fatal.

Multiple data segments -- should be large model
There is more than one code segment and the -Mm, -Ml or -Me switch was not given. Make sure all modules have the same memory model or link with the -Me switch. Not fatal.

Name length missing
The -n switch is missing a following number.

NEAR/HUGE conflict
Conflicting near and huge definitions for a communal variable. NOTE: At the present time, communal variables are not available in MASM.

No scratch file
Internal or system error. Do a file system check, and if the error persists notify Microsoft.

Number missing
The -S switch is missing a following number.

Object not found
See "Cannot find file".

Out of space on list file
Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file
Disk on which executable file is being written is full. Free more space on the disk and try again.

Run file name missing
The -o switch is missing a following string.

Segment limit too high
There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with /SEGMENTS or the default: 128). Either try the link again using /SEGMENTS to select a smaller number of segments (e.g. 64, if the default were used previously) or free some memory.

Segment size exceeds 64K
You have a small model program with more than 64K bytes of code, or you have a middle model program with more than 64K bytes of data. Try compiling and linking middle or large model.

Symbol already defined: <symbol>
A public symbol has been defined more than once.

Symbol missing
The -u switch is missing a following string.

Symbol table overflow
Your program has greater than 256K of symbolic information (publics, extrns, segments, groups, classes, files, etc). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user
You entered an interrupt.

Too many external symbols in one module
Your object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module
Your program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups
Your program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module
Ld encountered more than 9 GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries
You tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many segments
Your program has too many segments. Relink using the -S switch with an appropriate number of segments specified.

Too many segments in one module
Your object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs
An object module contains to many TYPDEF records. These records are emitted by a compiler to describe communal variables. NOTE: At the present time, communal variables are not available in MASM.

Unexpected end-of-file on scratch file
The temporary scratch file has probably been removed. Restart linker.

Unknown model specifier <-M?>
The -M switch was given with a character following that was not equal to s, m, l, or e.

Unrecognized XENIX version number
The version number following the -v switch must currently be either 2 or 3.

Use -i switch
There is more than one segment and the program is being linked impure, i.e. without the -i switch. Impure executables can only have one segment.

Version number missing
You gave the -v switch without a version number following.

Warning: _.SYMDEF out of date in <library name>
A library archive (.a) file has been modified since the last time its dictionary was updated with ranlib. Re-ranlib the file.

If you install new libraries with the cp command you may see this error message when in fact the library is up-to-date. This is because cp makes the modification time of the new library equal to the time the library was copied and the dictionary time is left unchanged. To avoid this situation, always copy libraries with "copy -m" or extract them from tar-format archive files.

Warning: Groups <name1> and <name2> overlap
Input to the linker has established a segment ordering such that the first segment of one group is ordered before the last segment of the previous group. Change the segment ordering or redefine appropriate class names.

Warning: model mismatch
You are linking object modules with different memory models. Recompile so all modules have the same model, or else use the -Me switch.

Warning: too many public symbols
You have asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

Operating Systems and Languages Library

MS-MACRO ASSEMBLER under XENIX V

Reference Manual

olivetti

RELATED PUBLICATIONS

MS-Macro Assembler Under XENIX V User Guide (Code 4022950 Z)

XENIX V User Guide (Code 4022940 Y)

XENIX V User and System Administrator Reference Manual (Code 4022320 Y)

XENIX V Installation and System Administration User Guide (Code 4022960 S)

XENIX V System and Application Software Development Tools User Guide (Code 4022990 B)

XENIX V System and Application Software Development Tools Reference Manual (Code 4031710 V)

C-Language Under XENIX V User Guide (Code 4022970 T)

C-Language Under XENIX V Reference Manual (Code 4033810 Y)

X/OPEN Portability Guide (Code 4024080 B)

XENIX V Text Processing User Guide (Code 4022980 C)

XENIX V Text Processing Reference Manual (Code 4031720)

DISTRIBUTION: General (G)

FIRST EDITION: July 1986

SECOND EDITION: October 1986

Copyright © Microsoft Corporation 1980/1985

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C. S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

PREFACE

This manual is a technical reference to the XENIX Macro-Assembler, MASM, and is intended for experienced assembly language programmers. It should be used in conjunction with the *MS-Macro Assembler Under XENIX V User Guide*.

MASM is a powerful two-pass assembler for the Intel 8086/80186/80286 family of microprocessors, and the 8087 and 80287 floating point coprocessors. Strong typing is available for memory operands, and conditional directives allow the exclusion of portions of the source file from assembly. In addition, a range of error detection facilities is provided. These features are all described.

The following syntax notations are used:

[ ] Square brackets indicate that the enclosed entry is optional.

{ } Braces indicate that there is a choice between two or more entries. At least one of the entries enclosed in the braces must be chosen, unless the entries are also enclosed in square brackets.

| A vertical stroke refers to the logical OR operation.

... Ellipses indicate that an entry may be repeated as many times as needed.

CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

_ The underscore joins multiple-name parameters.

All other punctuation, such as commas, colons, slash marks and equals signs must be entered exactly as shown.

SUMMARY

This manual is divided into seven chapters and three appendices. Chapter 1 is an overview of the Macro Assembler and its features. Chapter 2 defines the MASM character set and describes the syntax of constants, identifiers, statements and comments. Chapter 3 describes the meaning and syntax of the operands and expressions used in MASM statements and directives. Chapter 4 describes the Instruction Set Directives and the Memory Directives, and supplies an example of MASM source code. Chapter 5 describes the Conditional Assembly and Conditional Error Directives. Chapter 6 is a full description of how MASM macros are defined and called, and the functions of the Macro Directives. Chapter 7 describes the listing facilities supplied by MASM, and defines the File Control and Listing Control Directives.

Appendix A is a summary of the various microprocessor and coprocessor instruction sets. Appendix B lists the complete range of MASM directives and operators. Appendix C describes the naming conventions used to form assembly language source files compatible with the object modules produced by high-level programming languages.

TRADEMARK NOTICE

. OLIVETTI is a trademark of Ing. C. Olivetti & C., S.p.A.
. OLITERM is a trademark of Ing. C. Olivetti & C., S.p.A.
. SORTP is a trademark of Ing. C. Olivetti & C., S.p.A.
. ASM-86 is a trademark of Digital Research
. CB-86 is a trademark of Digital Research
. CLEO is a trademark of Phone 1 Inc.
. ETHERNET is a trademark of Xeros Corp.
. GW is a trademark of Microsoft Corp.
. IBM is registered trademark of International Business Machines Corp.
. MICROSOFT is a registered trademark of Microsoft Corp.
. MS is a trademark of Microsoft Corp.
. OMNINET is a trademark of Corvus System Inc.
. p-System is a trademark of Softech Microsystems, Inc.
. PC-DOS is a trademark of International Business Machines Corp.
. PEACHPACK is a trademark of Peachtree Software International Ltd.
. SID-86 is a trademark of Digital Research
. TerminALL is a trademark of TOPIC System, Inc.
. TOPIC is a trademark of TOPIC System, Inc.
. UNIX is a trademark of Bell Laboratories
. XENIX V is a trademark of Microsoft Corporation
. Z80 is a registered trademark of Zilog Inc.
. Z8000 is a registered trademark of Zilog Inc.

CONTENTS

1. INTRODUCTION 1-1
   OVERVIEW 1-1
   THE MACRO ASSEMBLER 1-1
   ASSEMBLER LISTINGS 1-2
   FEATURES OF THE MACRO ASSEMBLER 1-4
   RELOCATABLE OBJECT CODE 1-4
   PROGRAM MODULES 1-5
   MACRO CALLS 1-6
   INSTRUCTIONS AND DIRECTIVES 1-7
   INSTRUCTIONS 1-9
   SINGLE TYPE OPERAND INSTRUCTIONS 1-9
   MULTIPLE TYPE OPERAND INSTRUCTIONS 1-9
   DIRECTIVES 1-10
2. ELEMENTS OF THE ASSEMBLER 2-1
   INTRODUCTION 2-1
   CHARACTER SET 2-1
   CONSTANTS 2-1
   INTEGERS 2-1
   RANGE AND PRECISION OF INTEGERS 2-2
   REAL NUMBERS 2-3
   RANGE AND PRECISION OF REALS 2-4
   IMPLEMENTATION OF FORMATS 2-5
   ENCODED REAL NUMBERS 2-6
   RULES FOR FORMATION 2-6
   PACKED DECIMAL NUMBERS 2-7
   CHARACTER AND STRING CONSTANTS 2-8
   IDENTIFIERS 2-8
   NAMES 2-9

RESERVED NAMES 2-9
STATEMENTS AND COMMENTS 2-10
STATEMENTS 2-10
COMMENTS 2-11
COMMENT DIRECTIVE 2-12
3. OPERANDS, OPERATORS AND EXPRESSIONS 3-1
INTRODUCTION 3-1
OPERANDS 3-1
CONSTANT OPERANDS 3-1
DIRECT MEMORY OPERANDS 3-2
RELOCATABLE OPERANDS 3-3
LOCATION COUNTER OPERAND 3-4
REGISTER OPERANDS 3-4
FLAG REGISTER 3-5
BASED OPERANDS 3-6
INDEXED OPERANDS 3-7
BASED-INDEX OPERANDS 3-9
STRUCTURE OPERANDS 3-10
RECORD OPERANDS 3-11
RECORD-FIELD OPERANDS 3-12
OPERATORS AND EXPRESSIONS 3-13
ARITHMETIC OPERATORS 3-13
RELATIONAL OPERATORS 3-15
LOGICAL OPERATORS 3-16
EXPRESSION EVALUATION 3-17
PRECEDENCE OF OPERATORS 3-17
INDEXED MEMORY OPERANDS 3-18
ATTRIBUTE OPERATORS 3-19

CONTENTS

OVERRIDE OPERATORS 3-19

: (COLON) (SEGMENT OVERRIDE) 3-19

PTR (POINTER) 3-21

SHORT 3-23

THIS 3-23

HIGH, LOW 3-24

VALUE RETURNING OPERATORS 3-25

SEG 3-25

OFFSET 3-26

TYPE 3-27

.TYPE 3-27

LENGTH 3-28

SIZE 3-29

RECORD SPECIFIC OPERATORS 3-30

SHIFT_COUNT 3-31

WIDTH 3-32

MASK 3-33

EXPRESSION EVALUATION AND PRECEDENCE 3-34

OPERATOR PRECEDENCE .3-34

FORWARD REFERENCES 3-35

STRONG TYPING FOR MEMORY OPERANDS 3-37

4. INSTRUCTION SET AND MEMORY DIRECTIVES 4-1

SOURCE FILES 4-1

INSTRUCTION SET DIRECTIVES 4-2

.8086 4-3

.186 4-3

.286 4-4

.287 4-4

MEMORY DIRECTIVES	4-5
ASSUME	4-5
DB, DW, DD, DQ, DT (DEFINE)	4-6
DUP	4-8
END	4-9
EQU	4-10
EQUALS SIGN	4-11
EVEN	4-12
EXTRN	4-12
GROUP	4-14
LABEL	4-15
ORG	4-17
PROC AND ENDP	4-17
PUBLIC	4-19
RECORD	4-20
SEGMENT AND ENDS	4-22
STRUC	4-26
5. CONDITIONAL DIRECTIVES	5-1
INTRODUCTION	5-1
IF	5-2
IFE	5-3
IF1	5-4
IF2	5-5
IFDEF	5-6
IFNDEF	5-7
IFB	5-8
IFNB	5-9
IFIDN	5-10

CONTENTS

IFDIF 5-11

CONDITIONAL ERROR DIRECTIVES 5-12

.ERR, .ERR1 AND .ERR2 5-13

.ERRE AND .ERRNZ 5-14

.ERRDEF AND .ERRNDEF 5-14

.ERRB AND .ERRNB 5-15

.ERRIDN AND .ERRDIF 5-16

6. MACRO DIRECTIVES 6-1

INTRODUCTION 6-1

MACRO DEFINITION 6-1

CALLING A MACRO 6-3

ENDM (END MACRO) 6-4

EXITM (EXIT MACRO) 6-5

LOCAL 6-6

PURGE 6-7

REPEAT DIRECTIVES 6-8

REPEAT 6-8

IRP (INDEFINITE REPEAT) 6-9

IRPC (INDEFINITE REPEAT CHARACTER) 6-10

SPECIAL MACRO OPERATORS 6-11

& 6-11

<TEXT> 6-12

;; 6-13

! 6-13

% 6-14

7. LISTING DIRECTIVES 7-1

INTRODUCTION 7-1

INCLUDE 7-1

.RADIX	7-2
%OUT	7-3
NAME	7-3
TITLE	7-4
SUBTITLE	7-5
PAGE	7-6
.LIST AND .XLIST	7-7
.LFCOND	7-8
.SFCOND	7-8
.TFCOND	7-9
.LALL	7-9
.SALL	7-10
.XALL	7-10
.CREF AND .XCREF	7-11
A. INSTRUCTION SUMMARY	A-1
INTRODUCTION	A-1
8086 INSTRUCTIONS	A-1
8087 INSTRUCTIONS	A-5
186 INSTRUCTIONS	A-7
286 NONPROTECTED INSTRUCTIONS	A-8
286 PROTECTED INSTRUCTIONS	A-8
287 INSTRUCTIONS	A-9
B. DIRECTIVE SUMMARY	B-1
INTRODUCTION	B-1
DIRECTIVES	B-1
OPERATORS	B-4
C. SEGMENT NAMES FOR HIGH-LEVEL LANGUAGES	C-1
INTRODUCTION	C-1

CONTENTS

TEXT SEGMENTS	C-1
SMALL MODEL PROGRAMS	C-2
MIDDLE AND LARGE MODEL PROGRAMS	C-2
DATA SEGMENTS - NEAR	C-3
DATA SEGMENTS - FAR	C-4
BSS SEGMENTS	C-5
CONSTANT SEGMENTS	C-6

1. INTRODUCTION

ABOUT THIS CHAPTER

This chapter provides an overview of the Macro Assembler, its features and functions. It describes the assembly process, and the use of macro instructions, normal instructions and directives. Figures are provided to illustrate these descriptions.

CONTENTS

OVERVIEW	1-1
THE MACRO ASSEMBLER	1-1
ASSEMBLER LISTINGS	1-2
FEATURES OF THE MACRO ASSEMBLER	1-4
RELOCATABLE OBJECT CODE	1-4
PROGRAM MODULES	1-5
MACRO CALLS	1-6
INSTRUCTIONS AND DIRECTIVES	1-7
INSTRUCTIONS	1-9
SINGLE TYPE OPERAND INSTRUCTIONS	1-9
MULTIPLE TYPE OPERAND INSTRUCTIONS	1-9
DIRECTIVES	1-10

INTRODUCTION

OVERVIEW

This reference manual describes the syntax and structure of the XENIX Macro Assembler, MASM. MASM is an assembler for the Intel 8086/80186/80286 family of microprocessors. It can assemble instructions for the 8086, 8088, 80186, and 80286 microprocessors, and the 8087 and 80287 floating-point coprocessors. It has a powerful set of assembly-language directives that gives you complete control of the segmented architecture of the 8086, 80186 and 80286 microprocessors. MASM instruction syntax allows a wide variety of operand data types, including integers, strings, packed decimals, floating-point numbers, structures, and records.

The assembler produces 8086, 8088, 80186 or 80286 relocatable object modules from assembly-language source files. The relocatable object modules can be linked, using ld, the XENIX V link editor, to create executable programs for the XENIX V operating system. Details of ld and other XENIX utilities can be found in the XENIX V Reference Manuals.

MASM is a macro assembler. It has a full set of macro directives that let a programmer create and use macros in a source file. The directives instruct MASM to repeat common blocks of statements, or replace macro names with the block of statements they represent. MASM also has conditional directives that let the programmer exclude portions of a source file from assembly, or include additional program statements by simply defining a symbol.

MASM carries out strict syntax checking of all instruction statements, including strong typing for memory operands and detects questionable operand usage that can lead to errors or unwanted results.

MASM produces object modules that are compatible with object modules created by high-level language compilers. Thus, you can make complete programs by combining MASM object modules with object modules created by cc, the XENIX C language compiler, or other language compilers.

THE MACRO ASSEMBLER

The Macro Assembler is a two-pass assembler. This means that the source file is assembled twice, but slightly different actions occur during each pass. See the figure entitled "Pass 1 and Pass 2", which illustrates the following description.

During the first pass, the assembler:

- evaluates the statements and expands macro call statements
- calculates the amount of code it will generate
- builds a symbol table where all symbols, variables, labels, and macros are assigned values

During the second pass, the assembler:

- fills in the symbol, variable, label, and expression values from the symbol table

- expands macro call statements

- emits the relocatable object code into a file with the default filename extension .o.

Assembler Listings

MASM creates, on optional command, two types of listing file:

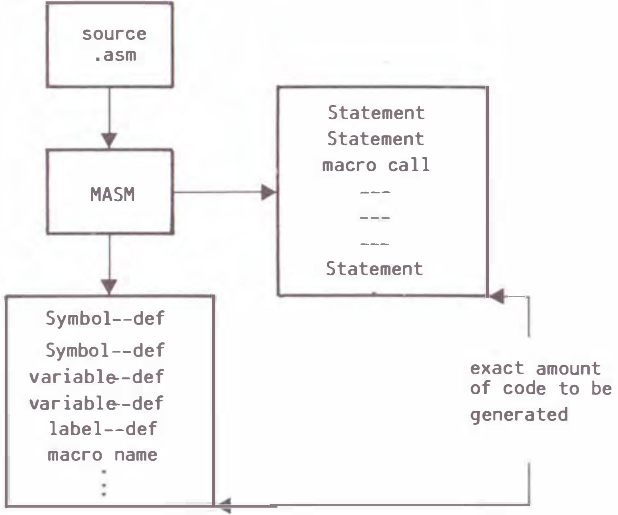
. A normal listing file

. A cross-reference file.

See the figure entitled "Files That The Macro Assembler Produces", which illustrates the following description.

The normal listing file contains the beginning relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. The listing also contains a symbol table which shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file receives the default filename extension .lst.

PASS 1



PASS 2

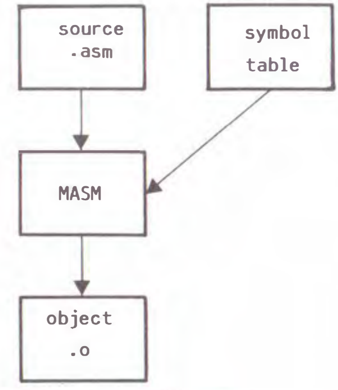


Fig. 1-1 Pass 1 and Pass 2

The cross-reference file is generated using the MASM -c option, and contains a compact representation of variables, labels and symbols. The cross-reference file receives the default filename extension .crf. When this cross-reference file is processed by cref, the XENIX cross reference utility, the file is expanded into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order; followed by the line number in the source program where each is defined; followed by the line numbers where each is used in the program.

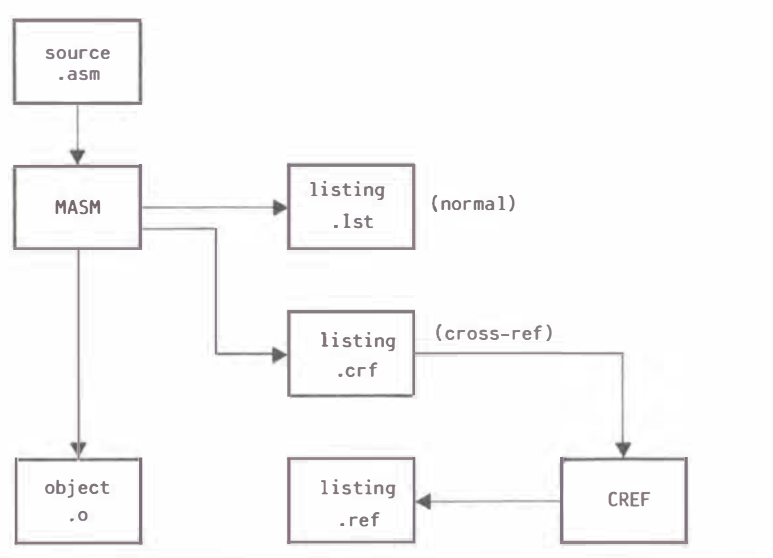


Fig. 1-2 Files That The Macro Assembler Produces

FEATURES OF THE MACRO ASSEMBLER

The XENIX Macro Assembler is a very powerful assembler and incorporates many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from the microprocessor.

RELOCATABLE OBJECT CODE

The Macro Assembler produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked using the ld utility to produce relocatable, executable object code. Relocatable code can be loaded anywhere in memory. Thus, the program can execute where it is most efficient, instead of in some fixed range of memory addresses.

PROGRAM MODULES

In addition, relocatable code means that programs can be created in modules, each of which can be assembled, tested, and perfected individually. This saves recoding time because testing and assembly are performed on smaller pieces of program code. Also, all modules can be error-free before being linked together into larger modules or into the whole program.

See the figure entitled "The Assembly Process", which illustrates the above description.

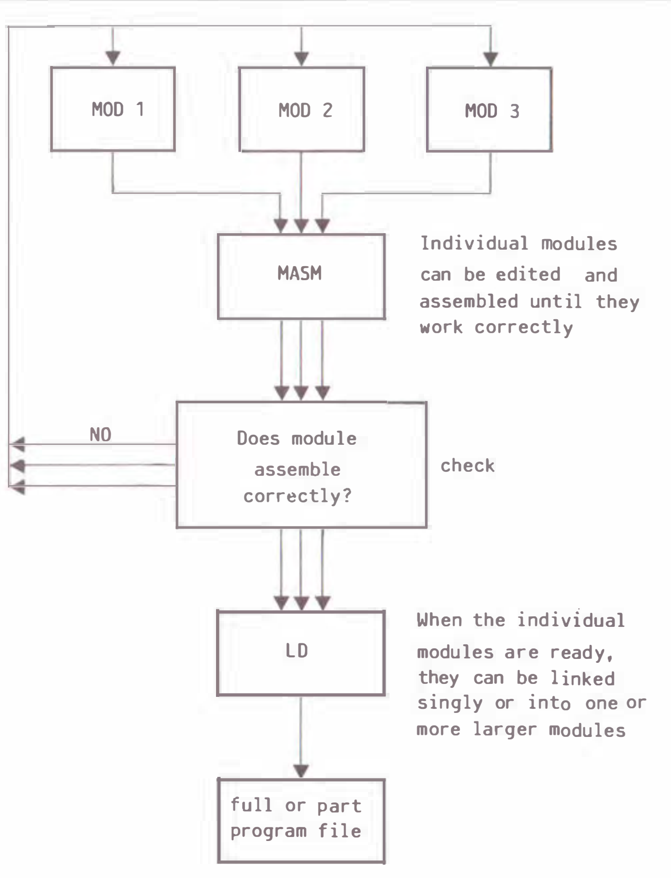


Fig. 1-3 The Assembly Process

MACRO CALLS

The Macro Assembler permits the writing of blocks of code for a set of frequently used instructions. The need for recoding these instructions each time they are required in the program is thus eliminated.

Such blocks of code are called macros. The instructions are the macro definition. Each time the set of instructions is needed, only a simple "call" to a macro is placed in the source file. The Macro Assembler expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are given only once; other occurrences are one-line calls.

Macros can be "nested," that is, a macro can be called from inside another macro block. Nesting of macros is limited only by memory.

See the figure entitled "Assembler Macros", which illustrates the above description.

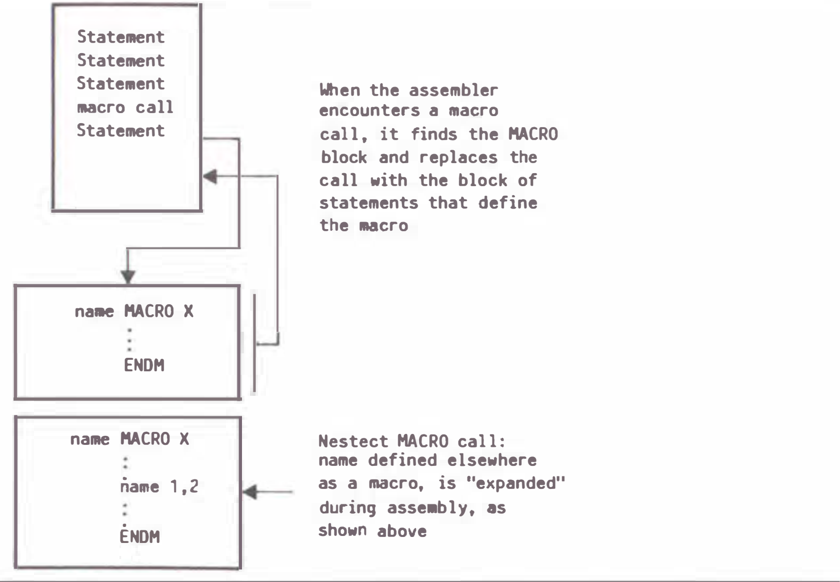


Fig. 1-4 Assembler Macros

The macro facility includes repeat, indefinite repeat, and indefinite repeat character directives for programming repeat block operations. The MACRO directive can also be used to alter the action of any instruction

or directive by using its name as the macro name. When any instruction or directive statement is placed in the program, MASM first checks the symbol table it created to see if the instruction or directive is a macro name. If it is, MASM "expands" the macro call statement by replacing it with the body of instructions in the macro's definition. If the name is not defined as a macro, MASM tries to match the name with an instruction or directive. The MACRO directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.

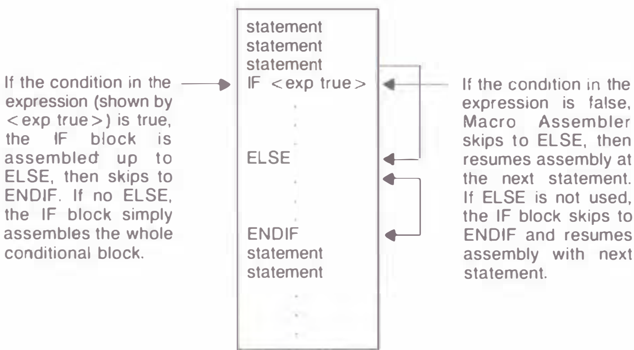
INSTRUCTIONS AND DIRECTIVES

Instructions and Directives are Macro Assembler statements. A statement is a specification to MASM as to what action to perform statements may be divided into three types:

- Macro calls: these are translated by MASM to become Instructions or Statements
- Instructions: these are translated by MASM to become machine instruction code, which the micro-processor obeys
- Directives: these define, give information and direct MASM. They are not translated into machine instruction code.

The Macro Assembler also supports an expanded set of conditional directives which are often used within macro definitions. Directives for evaluating a variety of assembly conditions can test assembly results and branch where required. Unneeded or unwanted portions of code will be left unassembled. The Macro Assembler can test for blank or nonblank arguments, for defined or undefined symbols, for equivalence, for first assembly pass or second, and can compare strings for identity or difference. Conditional error directives test for a specified condition and generate an error message, if the condition is true. Both kinds of conditional directives only test assembly time conditions.

The Macro Assembler's conditional assembly facility also supports the nesting of conditionals. Conditional assembly blocks can be nested up to 255 levels. See the figure entitled "Conditional Statements", which illustrates the implementation of conditional assembly blocks.



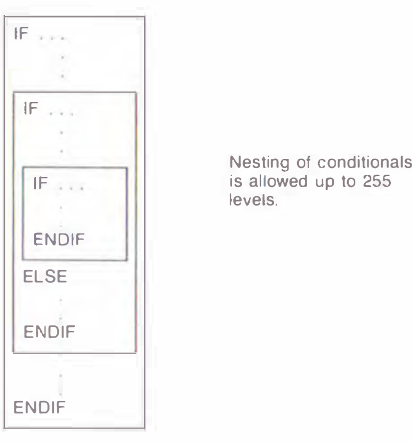
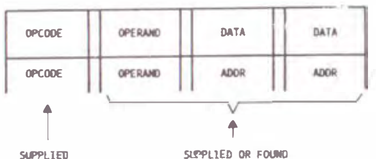


Fig. 1-5 Conditional Statements

INSTRUCTIONS

Instructions tell the command processor to perform some action. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = data and/or address inserted by the assembler from the information provided by expressions in instruction statements. (Opcode equates to the binary code for the action of an instruction)

Note that this manual does not contain detailed descriptions of instruction mnemonics and their characteristics. For this, you will need to consult other manuals, either the *8086 Assembler Reference Manual* or the *80286 Assembly Language Reference Manual.*

Note that the instruction set directives enable the instruction for the given microprocessor (see Chapter 4 for more details).

Appendix A contains an alphabetical listing of the instruction mnemonics.

The actual machine instruction code generated can depend on the type of its operands. Some instructions take only one type of operand. Other instructions can take several types of operands.

Single Type Operand Instructions

If you enter a typeless operand such as:

```
push [bx]
```

[bx] has no size, but PUSH only takes a word. The correct machine instruction code is generated, but a warning error message is given.

Multiple Type Operand Instructions

When the wrong type choice is given, the MASM displays an error message but generates the "correct" code. That is, it always outputs instructions, not just NOP instructions. For example, if you enter:

```
mov al,wordlbl
```

you may have meant one of the following three instructions:

1. mov ax,wordlbl

2. mov al,byte ptr wordlbl

3. mov al,<other>

MASM generates instruction (2) because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size". MASM accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some mundane debugging chores. An error message is still returned, however, because you may have mis-stated the operand MASM assumes is "correct".

DIRECTIVES

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions.

The directives have been divided into groups by the function they perform. Within each group, the directives are described alphabetically.

The groups are as shown below:

GROUP	MEANING
Instruction Set Directives	Directives in this group enable the instruction sets for the given microprocessors. See Chapter 4 for details.
Memory Directives	Directives in this group are used to define the organization that a program's code and data will have when loaded into memory. See Chapter 4 for details.
Conditional Directives	Directives in this group are used to test conditions of assembly before proceeding with assembly of a block of statements. This group contains all of the IF (and related) directives. See Chapter 5 for details.

Macro Directives	Directives in this group are used to create blocks of code called macros. This group also includes some special operators and directives that are used only inside macro blocks. The repeat directives also are considered macro directives for descriptive purposes. See Chapter 6 for details.
File Control and Listing Control Directives	Directives in this group are used to control the format and, to some extent, the content of source and object files, and listings that the assembler produces. See Chapter 7 for details.

Appendix B contains a summary of all the directives.

2. ELEMENTS OF THE ASSEMBLER

ABOUT THIS CHAPTER

The first part of this chapter describes the set of legal characters available for the writing of MASM source code. The syntax of constants and identifiers are described in the following part. The final section of the chapter describes the syntax of statements and comments. These are the elements of the assembler source code.

CONTENTS

INTRODUCTION	2-1
CHARACTER SET	2-1
CONSTANTS	2-1
INTEGERS	2-1
RANGE AND PRECISION OF INTEGERS	2-2
REAL NUMBERS	2-3
RANGE AND PRECISION OF REALS	2-4
IMPLEMENTATION OF FORMATS	2-5
ENCODED REAL NUMBERS	2-6
RULES FOR FORMATION	2-6
PACKED DECIMAL NUMBERS	2-7
CHARACTER AND STRING CONSTANTS	2-8
IDENTIFIERS	2-8
NAMES	2-9
RESERVED NAMES	2-9
STATEMENTS AND COMMENTS	2-10
STATEMENTS	2-10
COMMENTS	2-11
COMMENT DIRECTIVE	2-12

INTRODUCTION

All assembly language programs consist of one or more statements and comments. A statement or comment is a combination of characters, numbers, and names. Names and numbers are used to identify values in instruction statements. Characters are used to form the names or numbers, or to form character constants.

CHARACTER SET

MASM recognizes the following character set:

```
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
0 1 2 3 4 5 6 7 8 9
?  @  _  $ : . [ ] ( ) < > { }
+ - / * & % ! ' ~ | e = # ^ ; , ` "
```

CONSTANTS

A constant is an invariable value. The usage of constants as operands depends on the operator context. Check the particular instruction or directive in Appendix A, "Instruction Summary", or Appendix B, "Directive Summary".

INTEGERS

An integer is a whole number. The maximum size of an integer depends on its storage bit size.

Syntax

```
digits
digitsB
digitsQ
digitsO
digitsD
digitsH
```

An integer is a combination of binary, octal, decimal or hexadecimal digits and an optional radix. The digits are a combination of one or more digits of the specified radix, B, Q, O, D or H. If no radix is given, MASM uses the current default radix; see "Note", below. The following table defines which number base relates to each radix.

CONSTANT TYPE	RULES FOR FORMATION
Decimal (Base 10)	A sequence of digits 0 through 9, optionally followed by the letter D
Binary (Base 2)	A sequence of 0s and 1s followed by the letter B.
Octal (Base 8)	A sequence of digits 0 through 7 followed by either the letter O or the letter Q.
Hexadecimal (Base 16)	A sequence of digits 0 through 9 and/or letters A through F followed by the letter H. (Sequence must begin with 0 through 9 to distinguish hexadecimal numbers from symbols).

Note

The default radix can be changed by the RADIX directive. If the radix is changed Decimal digits must be followed by the letter D, while the new default radix digits do not need to be followed by a letter.

Examples

```
01011010B
132Q
5AH
90D
01111B
17O
0FH
15D
```

Range and Precision of Integers

The 8086/8088/80286 can operate on signed and unsigned integers in a byte (8-bits) and in a word (16-bits).

The 8087/80287 can operate on signed integers in a word (16-bits), in a double-word (32 bits) and in a quad-word (64 bits). The following table shows the range and precision of integers for different word lengths and formats:

FORMAT	WORD LENGTH	APPROXIMATE RANGE	SIGNIFICANT DIGITS (DECIMAL)
UNSIGNED	8	0 through 255	2 to 3
SIGNED	8	-128 through 127	2 to 3
UNSIGNED	16	0 through 65535	4 to 5
SIGNED (WORD INTEGER)	16	-32768 through 32767	4 to 5
UNSIGNED	32	0 through 4, 294, 967, 295	9 to 10
SIGNED (SHORT INTEGER)	32	-2 E9 through 2 E9	9
UNSIGNED	64	0 through 1.84E19 (approx.)	19 to 20
SIGNED	64	-9 E18 through 9 E18	18

Note

Unsigned 8 and 16 bit unsigned integers can be signed extended. Place the 8 bit unsigned integer in AL and the Sign in AH (FFH is equivalent to negative, and 00H is equivalent to positive). Or place the unsigned 16 bit integer in AX and the sign in DX (FFFFH is equivalent to negative and 0000H is equivalent to positive). You can then use the signed integer multiplication or division instruction IMUL or IDIV. See Appendix A "Instruction Summary" for operand details. This effectively doubles the range of these numbers.

REAL NUMBERS

Real numbers have non-ordinal values of a given range and precision.

A real number constant can be set using the following exponential notation.

Syntax

```
[+/-] digits. digits [E[+/-] digits]
```

Where

digits   Can be any combination of decimal digits

The digits before the decimal point are the integer part. The digits after the decimal point are the decimal fraction part. The digits after the E are the exponent. If the real number has an exponent, there is no need to have a decimal point and a decimal fraction.

Note

The constant will be stored in the old Microsoft format for reals, unless the code is assembled with the -R or -E options. When the -R or -E option has been used all real numbers arestored using IEEE formats. The range depends on the word length of the real.

Range and Precision of Reals

Real constants can be allocated by DD or DQ directives:

- DD allocates 32 bits (two words) for a single precision real
- DQ allocates 64 bits (four words) for a double precision real

The following table shows the range and precision of reals for different word lengths and formats:

FORMAT	WORD LENGTH	APPROXIMATE RANGE FOR POSITIVE NUMBERS	(DECIMAL DIGITS)
Microsoft Single Precision	32 bits	3.0E-39 to 1.7E38	6 to 7
Microsoft Double Precision	64 bits	3.0E-39 to 1.7E38	16 to 17
IEEE Singe Precision -short real	32 bits	1.175E-38 to 3.40E38	7 to 8
IEEE Double Precision -long real	64 bits	2.23E-308 to 1.80E308	15 to 16

Note

Negative numbers have the same range, but with the signs reversed. For example for the IEEE long real the negative numbers can range from approximately -1.80E308 to approximately -2.23E-308.

Examples

25.23
2.523E1
2523.0E-2

Implementation of Formats

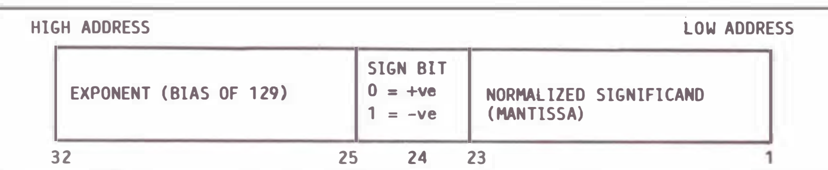
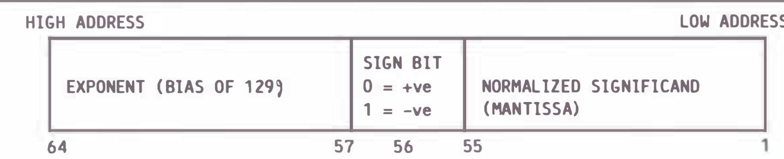


Fig. 2-1 Microsoft Single Precision Format



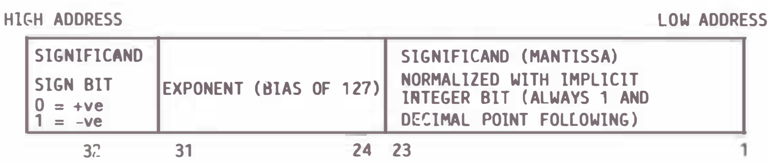


Fig. 2-2 Microsoft Double Precision Format

Fig. 2-3 IEEE Short Real (Single Precision) Format

Note that zero is represented by all the exponent and significant bits being set to zero.

HIGH ADDRESS		LOW ADDRESS
SIGNIFICAND SIGN BIT 0 = +ve 1 = -ve	EXPONENT (BIAS OF 1023)	SIGNIFICAND (MANTISSA) NORMALIZED WITH IMPLICIT INTEGER BIT (ALWAYS 1 AND DECIMAL POINT FOLLOWING)
64	63 53	52 1

Fig. 2-4 IEEE Long Real (Double Precision) Format

Note that zero is represented by all the exponent and significant bits being set to zero.

ENCODED REAL NUMBERS

An encoded real is a hexadecimal real number of 8, 16 or 20 digits, which represents a real number in encoded form.

Rules For Formation

A sequence of digits 0 through 9 and/or letters A through F followed by the letter R. The sequence must begin with 0-9. Total number of digits must be 8, 16 or 20 digits except if a leading zero is supplied then the total number of digits must be 9, 17 or 21 digits.

Note

An encoded real number has a sign, a biased exponent, and a mantissa. These values are encoded as bit fields within the number. The exact size and meaning of each bit field depends on the number of bits in the number.

Encoded real numbers can only be used with the following directives:

DIRECTIVE	HEXADECIMAL DIGITS
DD	8 (9, with leading 0)
DQ	16 (17, with leading 0)
DT	20 (21, with leading 0)

Examples

DIRECTIVE	ENCODED REAL NUMBERS	DECIMAL EQUIVALENT
DD	3FB00000	1.0
DQ	3FF0000000000000	1.0

PACKED DECIMAL NUMBERS

A packed decimal number to be stored using Binary Coded Decimal conventions in ten bytes.

Syntax

```
[+|-]digits
```

Where

digits Can be any combination of decimal digits. There is a maximum of 18 digits.

Note

The leading byte is the sign byte, the high order bit is 0 for positive values and 1 for negative values.

Packed decimals can only be used with the DT directive

Examples

DIRECTIVE	PACKED DECIMAL	BCD
DT	1234567890	00000000001234567890
DT	-1234567890	80000000001234567890

CHARACTER AND STRING CONSTANTS

A character constant consists of a single ASCII character.

A string constant consists of one or more ASCII characters:

Syntax

```
{'|"} character [character...] {'|"}
```

Where

character   This is any ASCII character

Note

Opening and closing quotes must match.

String constants are case sensitive

To use quotations marks literally escape them by using quotation marks twice.

Examples

```
'a'
'ab'
"a"
"This is a message"
"specified """value""" not found"
'Can''t find the file'
```

IDENTIFIERS

An identifier name is used to identify:

- a segment
- a group
- a variable
- a label
- a constant defined with an EQU directive or equal sign (=) operator.

NAMES

An identifier is a combination of letters, digits and special characters.

Syntax

```
.|letter|_|?|$|@[letter|digit|_|?|$|@]...
```

Note

Only 31 characters are significant, the maximum number of characters are 255. A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an "at" sign (@). All lowercase letters are converted to uppercase by MASM unless the -Ml option is used during assembly or unless the name is declared with a PUBLIC or EXTRN directive and the -Mx option is used during assembly.

Examples

```
subrout3
Array
_main
```

RESERVED NAMES

A reserved name is any name that has a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, operator names and special characters. These names can be used only as defined and must not be redefined.

All upper- and lowercase combinations of these names are treated as the same name. For example, the names Length and LENGTH are the same name for the LENGTH operator.

The following lists all reserved names except instruction mnemonics. For a complete list of instruction mnemonics, see Appendix A, "Instruction Summary".

Reserved Names

%OUT	DQ	IFIDN	QWORD	.186	DS	IFNB	.RADIX
.286c	DT	IFNDEF	RECORD	.286p	DW	INCLUDE	REPT
.287	DWORD	IRP	.SALL	.8086	DX	IRPC	SEG
.8087	ELSE	LABEL	SEGMENT	=	END	.LALL	.SFCOND
AH	ENDIF	LE	SHL	AL	ENDM	LENGTH	SHORT
AND	ENDP	.LFCOND	SHR	ASSUME	ENDS	.LIST	SI
AX	EQ	LOCAL	SIZE	BH	EQU	LOW	SP
BL	ES	LT	SS	BP	EVEN	MACRO	STRUC
BX	EXITM	MASK	SUBTTL	BYTE	EXTRN	MOD	TBYTE
CH	FAR	NAME	.TFCOND	CL	GE	NE	THIS
COMMENT	GROUP	NEAR	TITLE	.CREF	GT	NOT	TYPE
CS	HIGH	OFFSET	.TYPE	CX	IF	OR	WIDTH
DB	IF1	ORG	WORD	DD	IF2	PAGE	.XALL
DH	IFB	PROC	.XCREF	DI	IFDEF	PTR	.XLIST
DL	IFDIF	PUBLIC	XOR	IFE	PURGE		

STATEMENTS AND COMMENTS

The syntax for Statements and Comments are described in the following sections.

STATEMENTS

A statement represents an action to be taken by the assembler, such as generating a machine instruction or generating one or more bytes of data.

Syntax

```
[name] mnemonic [operand]...] [;comment]
```

Where

name — This is an identifier name

mnemonic — This is a directive or an instruction

operand — This is a variables or constants. The exact syntax depends on the directive or instruction.

comment — This is a combination of characters terminated by <CR> <LF>.

Note

Statements have the following formatting rules:

- A statement can begin in any column.
- A statement must not be more than 128 characters in length and must not contain an embedded newline character. This means continuing a statement on multiple lines is not allowed.
- A statement must be terminated by a <CR> <LF>. This includes the last statement in the source file.

Examples

```
count   db      0
        mov     ax, bx
        assume  cs:_text, ds:DGROUP
_main   proc    far
```

COMMENTS

Comments describe the action of a program at the given point, but are otherwise ignored by the assembler and have no effect on assembly.

Syntax

```
; text
```

Where

text — can be any combination of characters preceded by a semicolon (;) and terminated by an embedded carriage-return/line-feed combination.

Note

Comments can be placed anywhere in a program, even on the same line as a statement. However, if the comment shares the line with a statement, it must be to the right of all names, mnemonics, and operands. A comment following a semicolon must not continue past the end of the line on which it begins; that is, it must not contain any embedded <CR> <LF> combination characters. For very long comments, the COMMENT directive can be used.

Examples

```
; This comment is alone on a line
        mov     ax, bx   ; This comment follows a statement
; Comments can contain reserved words like PUBLIC
```

COMMENT DIRECTIVE

Causes MASM to treat all text between delimiter and delimiter as a comment. Usually used for multiple-line comments.

Syntax

COMMENT delimiter text delimiter

Where

delimiter	Any legal character
text	Any legal characters not containing delimiter, can be one or more lines.

Note

Text is treated as comment and are ignored by MASM

Examples

```
comment *
   This comment continues until the
   next asterisk.
*
```

The preceding and following examples illustrate how blocks of text can be designated as comments.

```
comment +
   The assembler ignores the
   following MOV statement
+ mov   ax, 1
```

3. OPERANDS, OPERATORS AND EXPRESSIONS

ABOUT THIS CHAPTER

The first part of this chapter describes the syntax and meaning of MASM operands. The next section is a systematic description of the eleven types of MASM operator. Finally expression evaluation and precedence, forward reference and strong typing are described.

CONTENTS

INTRODUCTION	3-1
OPERANDS	3-1
CONSTANT OPERANDS	3-1
DIRECT MEMORY OPERANDS	3-2
RELOCATABLE OPERANDS	3-3
LOCATION COUNTER OPERANDS	3-4
REGISTER OPERANDS	3-4
FLAG REGISTER	3-5
BASED OPERANDS	3-6
INDEXED OPERANDS	3-7
BASED-INDEXED OPERANDS	3-9
STRUCTURE OPERANDS	3-10
RECORD OPERANDS	3-11
RECORD-FIELD OPERANDS	3-12
OPERATORS AND EXPRESSIONS	3-13
ARITHMETIC OPERATORS	3-13
RELATIONAL OPERATORS	3-15
LOGICAL OPERATORS	3-16
EXPRESSION EVALUATION	3-17
PRECEDENCE OF OPERATORS	3-17
INDEXED MEMORY OPERANDS	3-18
ATTRIBUTE OPERATORS	3-19

OVERRIDE OPERATORS	3-19
: (COLON) (SEGMENT OVERRIDE)	3-19
PTR (POINTER)	3-21
SHORT	3-23
THIS	3-23
HIGH, LOW	3-24
VALUE RETURNING OPERATORS	3-25
SEG	3-15
OFFSET	3-26
TYPE	3-27
.TYPE	3-27
LENGTH	3-28
SIZE	3-29
RECORD SPECIFIC OPERATORS	3-30
SHIFT_COUNT	3-31
WIDTH	3-32
MASK	3-32
EXPRESSION EVALUATION AND PRECEDENCE	3-34
OPERATOR PRECEDENCE	3-34
FORWARD REFERENCES	3-35
STRONG TYPING FOR MEMORY OPERANDS	3-37

INTRODUCTION

This chapter describes the syntax and meaning of operands and expressions used in assembly language statements and directives. Operands represent values, registers, or memory locations to be acted on by instructions or directives. Expressions combine operands with arithmetic, logical, bitwise, and attribute operators to calculate a value or memory location that can be acted on by an instruction or directive. Operators indicate what operations will be performed on one or more values in an expression to calculate the value of the expression.

OPERANDS

An operand is a constant, label, variable, or other symbol that is used in an instruction or directive to represent a value, register, or memory location to be acted on.

These are the available operand types:

1. Constant
2. Direct Memory
3. Relocatable
4. Location Counter
5. Register
6. Based
7. Indexed
8. Based-Indexed
9. Structure
10. Record
11. Record-Field

CONSTANT OPERANDS

This operand is or evaluates to a fixed value.

Syntax

```
number|string|constant_identifier|expression
```

Where

number	An invariable value, as described in the section "Constants" in Chapter 2.
string	Consists of one or more ASCII characters enclosed in quotes, as described in the section "Character and String Constants" in Chapter 2.
constant identifier	A constant defined with an EQU directive or equal-sign (=) operator.
expression	A legal expression that evaluates to a number or string constant.

Remarks

Constant operands, unlike other operands, represent values to be acted on, rather than memory addresses.

Examples

```
mov ax, 9          ; 9 is an integer constant

mov al, 'C'        ; C is a character or string constant

mov bx, 65535/3    ; 65535/3 is an expression

mov cx, count      ; count is a constant-identifier, previously
                   ; defined as count = 10
```

DIRECT MEMORY OPERANDS

This operand represents the absolute memory address of one or more bytes of memory.

---

segment:offset

Where

segment	Can be a segment register (CS, DS, SS or ES), a segment name or a group name.
offset	Must be an integer, a constant-identifier, expression, or a symbol that is or evaluates to a value from 0 through

65535.

Examples

```
mov dx, ss:0031H     ; 0031H is an integer constant
mov bx, data:0       ; data is a segment name
mov cx, DGROUP:block ; DGROUP is a group name block is a constant-
                     ; identifier or a symbol.
```

RELOCATABLE OPERANDS

This operand is any symbol that represents the memory address (segment and offset) of an instruction or of data to be acted upon.

Syntax

---

symbol

Where

symbol   Can be a label, variable name, segment or group name.

Remarks

Relocatable operands, unlike direct-memory operands, are relative to the start of the segment or group in which the symbol is defined and have no explicit absolute address, until the program has been linked.

Examples

```
call main                    ; main is a relocatable operand
mov bx, local                ; value is a relocatable operand
mov bx, offset DGROUP:table  ; returns the offset value within DGROUP.
                             ; This is a relocatable operand
mov cx, count                ; count has been previously defined with
                             ; the DW directive; it is therefore a
                             ; relocatable operand
```

LOCATION COUNTER OPERANDS

This is a special operand that during assembly, represents the current location within the current segment.

Syntax

$

Remarks

The location counter has the same attributes as a near label. It represents an instruction address that is relative to the current segment. Its offset is equal to the number of bytes generated for that segment to that point. After each statement in the segment has been assembled the assembler increments the location counter by the number of bytes generated.

Example

```
target   equ $
         mov ax,1
         .
         .
         .
         jmp target
```

REGISTER OPERANDS

This is the reserved name of a CPU register.

Syntax

---

registername

Where

registername	TYPE
AX BX CX DX	16-bit general purpose register. They can be used for any data or numeric manipulation.
AH BH CH DH	8-bit high registers of the preceding general purpose registers.
AL BL CL DL	8-bit low registers of the preceding general purpose registers.
CS DS SS ES	16-bit segment registers. They contain the current segment address of the code data, stack and extra segment, respectively. All instruction and data addresses are relatable to the segment address in one of these registers.
BX BP SI DI	16-bit base and index registers. These are general-purpose registers typically used for programs to program data. By default in address expressions: BP is relative to SS; BX, SI and DI are relative to DS; except DI is relative to ES for string expressions
SP	16-bit stack pointer register, it contains the current top-of-stack address. This address is relative to SS and is automatically modified by instructions that modify the stack.

Flag Register

This register is unnamed. It is a 16-bit register containing nine 1-bit flags as defined in the following table:

FLAG BIT	MEANING
0	Carry flag
2	Parity flag
4	Auxiliary flag
5	Trap flag
6	Zero flag
7	Sign flag
9	Interrupt-enable flag
10	Direction flag
11	Overflow flag

Although this register has no name, the contents of the register can be accessed using the LAHF, SAHF, PUSHF and POPF instructions. See Appendix A for more details.

BASED OPERANDS

This operand represents a memory address relative to one of the base registers. All the following syntaxes are equivalent:

Syntax

```
[displacement] indirect-baseregister

or

opening-bracket baseregister + displacement closing bracket

or

indirect-baseregister. displacement

or

indirect-baseregister + displacement
```

Where

displacement	Any immediate or direct-memory operand. In Syntax 1, if no displacement is given, its value is assumed to be 0.
indirect-baseregister	opening-bracket baseregister closing-bracket

opening-bracket	The actual bracket '['
closing-bracket	The actual bracket ']'
baseregister	Can be:
	BP The operand's address is relative to the segment pointed to by the SS register.
	BX The operand's address is relative to the segment pointed to by the DS register.

Remarks

The operand evaluates to a memory address which is the sum of the displacement and the contents of the given baseregister.

Examples

```
mov ax, [bp]      ; The contents of the stack pointed to by SS:BP are
                  ; moved into ax

mov ax, [bx]      ; The contents of the address pointed to by DS:BX
                  ; are moved into ax

mov ax, 12[bx]    ; The contents of the address pointed to by
                  ; DS(BX+12) are moved into ax

mov ax, fred[bp]  ; The contents of the address pointed to by
                  ; SS:(BP+fred) are moved into ax. Fred must have
                  ; an integer value
```

INDEXED OPERANDS

This operand represents a memory address relative to one of the index registers.

Syntax

[displacement] indirect-indexregister

or

opening-bracket indexregister+displacement closing-bracket

or

```
indirect-indexregister.displacement

or

indirect-indexregister+displacement
```

Where

displacement	Any immediate or direct-memory operands. In Syntax 1, if no displacement is given, its value is assumed to be 0.
indirect-indexregister	opening-bracket indexregister closing-bracket
opening-bracket	The actual bracket '['
closing-bracket	The actual bracket ']'
index register	Can be: SI The operand's address is relative to the segment pointed to by the DS register. DI The operand's address is relative to the segment pointed to by the DS register, except where the operand is used in string manipulating expressions, string. When the operand is used in string the operand is used in string expressions the operand's address is relative to the segment pointed to by the ES register.

Remarks

The operand evaluates to a memory address which is the sum of the displacement and the contents of the given index register.

Examples

```
mov ax, [si]      ; The contents of the address pointed to by DS:SI
                  ; are moved into ax

mov ax, [di]      ; The contents of the address pointed to by DS:DI
                  ; are moved into ax

mov ax, 12[di]    ; The contents of the address pointed to by
                  ; DS:(DI+12) are moved into ax

mov ax, fred[si]  ; The contents of the address pointed to by
```

```
                        ; DI:(SI+fred) are moved into ax. Fred must have
                        ; an integer value
```

BASED-INDEXED OPERANDS

This operand represents a memory address relative to any combination of base and index registers.

Syntax

[displacement] indirect_basepointer indirect_indexregister

or

opening-bracket basepointer+indexregister+displacement closing-bracket

or

opening bracket basepointer+indexregister closing_bracket. displacement

or

indirect_basepointer+indirect_indexregister+displacement

Note

See the previous sections "Based Operands" and "Indexed Operands" for an explanation of the syntax.

Remarks

The effective address is the sum of the contents of the given registers and the displacement.

Warning

Either base register can be combined with either index register, but combining two base or two index registers is not allowed.

Examples

```
mov ax,[bp][si]       ; The contents of the stack pointed to by
                      ; SS:(BP+SI) are moved into ax

mov ax,[bx+di]        ; The contents of the address pointed to by
```

```
                     ; DS:(BX+DI) are moved into ax

mov ax, 12[bp+di]    ; The contents of the address pointed to by
                     ; DS:(BP+DI+12) are moved into ax

mov ax, fred[bx][si] ; The contents of the address pointed to by
                     ; DS:(BX+SI+fred) are moved into ax

mov ax, fred[bx][bp] ; Error - base registers combined

mov ax, fred[di][si] ; Error - index registers combined
```

STRUCTURE OPERANDS

This operand represents the memory address of one member of a structure.

Syntax

```
variable.field
```

Where

variable	The name of a structure or a memory operand, which resolves to the address of a structure.
field	The name of a field within that structure.
	This is the "Structure Field Operator". See the relevant section in "Operators and Expressions" for more details.

Remarks

The effective address is the sum of the offsets of variable and field. The address is relative to the segment or group in which the variable is defined.

See the STRUC directive in Chapter 4 for an example.

Examples

In this example, "current_date" is assumed to be the structure defined in the following:

```
date  struc
```

```
          month dw ?
          day   dw ?
          year  dw ?
    date  ends
    current_date date <'ja', '01', '85'>
```

This structure is used in the following example:

```
    mov ax, current_date.day
    mov current_date.year, '86'
```

The next example shows how structure operands can be used to access values on the stack:

```
    stframe struc                    ; stack frame
     retadr dw  ?                    ; from lowest ...
     dest   dw  ?
     source dw  ?
     nbytes dw  ?                    ; ... to highest address
    stframe ends

    copy    proc near                ; push nbytes, source, dest before calling
            mov  bx,sp               ; load stack into base register
            mov  ax,ds
            mov  es,ax               ; (es) = data segment
            mov  di,ss:[bx].dest     ; (di) = destination
            mov  si,ss:[bx].source   ; (si) = source
            mov  cx,ss:[bx].nbytes   ; (cx) = nbytes
            rep  movsb               ; move bytes from ds:si to es:di
            ret
    copy    endp
```

RECORD OPERANDS

This operand refers to the value of a record type. Operands can be in expressions.

Syntax

```
recordname <[value][,[value]]...>
```

Where

recordname — As previously defined in a RECORD directive

value — The value to be placed into a field of the record. If more than one value is given, they must be separated by commas. Note that the enclosing angle brackets (<>)

are required even if no value is given. If no value is given for a field, the default value for that field is used.

See the RECORD directive in Chapter 4 for more details.

Examples

```
mov  ax, encode <l, 3, 2>
mov  cx, key <l, 7>
```

RECORD-FIELD OPERANDS

This operand represents the location of a field in its corresponding record.

Syntax

```
record-fieldname
```

Where

record-fieldname   The name of a previously defined record field.

Remarks

The operand evaluates to the bit position of the low-order-bit in the field and can be used as a constant operand.

See the RECORD directive in Chapter 4 for an example of this operand.

Examples

These examples assume that the record "recl" has been defined as follows:

```
rtype record fieldl:3,field2:6,field3:7
recl rtype<>
```

The following example copies 13, the shift conunt for fieldl, to ax:

```
mov  ax,fieldl
```

The next example copies 7, the shift count for field2, to cl, then uses the address of "recl", copied to dx, in a shift operation. Ths operation

adjusts "recl" so that field2 is at the lowest bit:

```
mov  dx,recl
mov  cl,field2
shr  dx,cl
```

OPERATORS AND EXPRESSIONS

An expression is a combination of operands and operators that evaluates to a single value. Operands in expressions can include any of the operands described in the previous section. The result of an expression can be a value or a memory location, depending on the types of operands and operators used.

Operators manipulate and compare operands.

ARITHMETIC OPERATORS

Eight arithmetic operators provide the common mathematical functions (add, subtract, divide, multiply, modulo, negation), plus two shift operators. The arithmetic operators are used to combine operands to form an expression that results in a data item or an address.

Except for + and - (binary), operands must be constants.

For plus (+), one operand must be a constant.

For minus (-), the first (left) operand may be a nonconstant, or both operands may be nonconstants. The right must be a constant if the left is a constant.

OPERATOR	MEANING
*	Multiply
/	Integer Division
MOD	Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolute. Example: `mov ax,100 mod 17` The value moved into ax will be 0FH (decimal 15).

SHR	Shift Right. SHR is followed by an integer which specifies the number of bit positions the value is to be shifted right. Example: `mov ax,1100000B shr 5` The value moved into ax will be 11B (03).
SHL	Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be shifted left. Example: `mov ax,0110B shl 5` The value moved into ax will be 01100 0000B (0C01H) 0110 01100
-(Unary Minus)	Indicates that the following value is negative, as in a negative integer.
+	Add. One operand must be a constant; one may be a non-constant.
-	Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be nonconstants. But the right may be a nonconstant only if the left is also a nonconstant and in the same segment.

Examples

```
14 * 4        ; equals 56
14 / 4        ; equals 3
14 MOD 4      ; equals 2
14 + 4        ; equals 18
14 - 4        ; equals 10
14 - +4       ; equals 10
14 - -4       ; equals 18
alpha + 5     ; adds 5 to alpha's offset
alpha - 5     ; subtracts 5 from alpha's offset
alpha - beta  ; subtracts beta's offset from alpha's
```

RELATIONAL OPERATORS

Relational operators compare two constant operands.

If the relationship between the two operands matches the operator, FFFFH is returned.

If the relationship between the two operands does not match the operator, zero is returned.

Relational operators are most often used with conditional directives and conditional instructions to direct program control.

OPERATOR	MEANING
EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less than or Equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.

Examples

```
1 EQ 0 ; false
1 NE 0 ; true
1 LT 0 ; false
1 LE 0 ; false
1 GT 0 ; true
1 GE 0 ; true
```

LOGICAL OPERATORS

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate the logical relationship defined by the logical operator.

Logical operators can be used two ways:

1. To combine operands in a logical relationship. In this case, all bits in the operands will have the same value (either 0000 or FFFFH). In fact, it is best to use these values for true (FFFFH) and false (0000) for the symbols you wiil use as operands, because in conditionals anything nonzero is true.
2. In bitwise operations the operation is performed on each bit in an expression rather than on the expression as a whole. The expressions must resolve to absolute values. In this case the logical operators act the same as the instructions of the same name.

OPERATOR	MEANING
NOT	Logical NOT. This is a unary operator. Returns true if right operand is false. Returns false if right operand is true. Bitwise it inverts the bits, and can be used to form the 1's complement.
AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values. Bitwise it carries out a Boolean AND on the parallel bits.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values. Bitwise it carries out a Boolean inclusive OR on the parallel bits.
XOR	Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values. Bitwise it carries out a Boolean XOR on the parallel bits.

Examples

```
NOT 11110000B              ; equals 00001111B
01010101B AND 11110000B ; equals 01010000B
01010101B OR 11110000B  ; equals 11110101B
01010101B XOR 11110000B ; equals 10100101B
```

Expression Evaluation

Expressions are evaluated higher precedence operators first, then left to right for equal precedence operators.

Parentheses can be used to alter precedence.

For example:

```
mov ax,101B shl 2*2   =   mov ax,00101000B

mov ax,101B shl (2*2) = mov ax,01010000B
```

SHL and * are equal precedence. Therefore, their functions are performed in the order the operators are encountered (left to right).

Precedence of Operators

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

- LENGTH, SIZE, WIDTH, MASK
  Entries inside: parentheses ( )
  square brackets [ ]
  Structure variable operand: variable.field
- Segment override operator: colon (:)
- PTR, OFFSET, SEG, TYPE, THIS
- HIGH, LOW
- *,/, MOD, SHL, SHR
- +, - (both unary and binary)
- EQ, NE, LT, LE, GT, GE
- Logical NOT
- Logical AND
- Logical OR, XOR

. SHORT, .TYPE

INDEXED MEMORY OPERANDS

Indexed memory operands use base and index registers, constants, displacement values, and variables, often in combination. When you combine indexed operands, you create an address expression.

Indexed memory operands use square brackets to indicate indexing (by a register or by registers) or subscripting (for example, FOO[5]). The square brackets are treated like plus signs (+). Therefore:

```
arg[5] is equivalent to arg+5

5[arg] is equivalent to 5+arg
```

The only difference between square brackets and plus signs occurs when a register name appears inside the square brackets. Then, the operand is indexed.

The types of indexed memory operands are:

```
Base registers:   [BX]  [BP]

  (BP has SS as its default segment register;
  all others have DS as default.)

Index registers:  [DI]  [SI]

[constant]:       Immediate in square brackets [8], [ARG]

+/- Displacement: 8-bit or 16-bit value.

  (Used only with another indexed operand.)
```

These elements may be combined in any order. The only restriction is that two base registers and two indexed registers cannot be combined:

```
[BX+BP]  ;  illegal
[SI+DI]  ;  illegal
```

Some examples of indexed memory operand combinations:

```
[BP+8]
[SI+BX][4]
16[DI+BP+3]
8[ARG]-8
```

More examples of equivalent forms:

```
5[BX][SI]
[BX+5][SI]
[BX+SI+5]
[BX]5[SI]
```

ATTRIBUTE OPERATORS

Attribute operators used as operands perform one of three functions:

- Override an operand's attributes
- Return the values of operand attributes
- Isolate record fields (record specific operators)

The following list shows all the attribute operators by type:

- Override operators
  PTR
  colon (:) (segment override)
  SHORT
  THIS
  HIGH
  LOW

- Value returning operators
  SEG
  OFFSET
  TYPE
  .TYPE
  LENGTH
  SIZE

- Record specific operators
  Shift count (Field name)
  WIDTH
  MASK

OVERRIDE OPERATORS

These operators are used to override the segment, offset, type, or distance of variables and labels.

: (colon) (Segment Override)

The segment override operator (:) overrides the assumed segment of an address expression (which may be a label, a variable, or other memory operand).

Syntax

```
segment_register : address_expression
```

or

```
segment_name : address_expression
```

or

```
group_name : address_expression
```

Where

segment-register	is one of the four segment register names: CS, DS, SS, ES.
segment-name	is a name defined by the SEGMENT directive.
group-name	is a name defined by the GROUP directive.

Remarks

The colon operator helps with forward references by telling the assembler to what a reference is relative (segment, group, or segment register).

MASM assumes that labels are addressable through the current CS register. MASM also assumes that variables are addressable through the current DS register, or possibly the ES register, by default. If the operand is in another segment and you have not alerted MASM through the ASSUME directive, you will need to use a segment override operator. Also, if you want to use a nondefault relative base (that is, not the default segment register), you will need to use the segment override operator for forward references. Note that if MASM can reach an operand through a nondefault segment register, it will use it, but the reference cannot be forward in this case.

The following table gives the default and alternative segment registers, and offset, for memory reference:

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT REGISTER	ALTERNATE SEGMENT REGISTER	OFFSET
Instruction Fetch	CS	-	IP
Stack Operation	SS	-	SP
Variable (except following	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	-	DI
BP Used as Base Register	SS	CS,DS,ES	Effective Address

Warning

Trying to override the default segment where no alternate segment register is shown in the above table will result in an error message.

Examples

```
mov   ax,es:[bx][si]
mov   _TEXT:far_label,ax
mov   ax,DGROUP:variable
mov   al,cs:0001H
```

PTR (Pointer)

The PTR operator assigns or overrides the type of a memory-variable or the distance of label.

Syntax

```
type PTR expression
```

Where:

type must be one of the following names or values:

BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	FFFFH
FAR	FFFEH

expression can be any of the following:

memory-expression — A legal expression (which could be a simple variable) which evaluates to a memory address.

label-expression — A legal expression (which could be a simple label) which evaluates to a location which could be jumped to or called.

register-expression — A legal expression incorporating a register reference, such as [BX], which can only be resolved at runtime.

constant-expression — A legal expression which evaluates to an integer constant offset to a segment register.

Remarks

The most important and frequent use for PTR is to assure that Macro Assembler understands what attribute the expression is supposed to have. This is especially true for the type attribute. Whenever you place forward references in your program, PTR will make clear the distance or type of the expression. This way you can avoid phase errors.

The second use of PTR is to access data by type other than the type in the variable definition. Most often this occurs in structures. If the structure is defined as WORD but you want to access an item as a byte, PTR is the operator for this. However, a much easier method is to enter a second statement that defines the structure in bytes, too. This eliminates the need to use PTR for every reference to the structure. Refer to the LABEL directive in the section on "Memory Directives" in Chapter 4.

Examples

```
call  far ptr subrout3
mov   byte ptr [array],1
add   al,byte ptr [full_word]
```

SHORT

SHORT overrides NEAR distance attributes of labels used as targets for the JMP instruction. SHORT tells MASM that the distance between the JMP statement and the "label" specified as its operand is not more than 127 bytes in either direction.

Syntax

```
SHORT label
```

The major advantage of using the SHORT operator is to save a byte. Normally, the "label" carries a 2-byte pointer to its offset in its segment. Because a range of 256 bytes can be handled in a single byte, the SHORT operator eliminates the need for the extra byte (which would carry 00 or FF anyway). However, you must be sure that the target is within +/-127 bytes of the JMP instruction before using SHORT.

Example

```
    jmp   short do_again   ; jump less than 128 bytes
```

THIS

The THIS operator creates an operand of a particular distance or type.

Syntax

```
THIS distance

or

THIS type
```

The argument to THIS may be:

- A distance (NEAR or FAR)

- A type (BYTE, WORD, DWORD, QWORD, or TBYTE)

Where

THIS distance	creates an operand with the distance attribute you specify, an offset equal to the current location counter.
THIS type	creates an operand with the type attribute you specify, an offset equal to the current location counter.

Examples

```
    tag     equ THIS BYTE ; same as TAG LABEL BYTE
    check=  THIS NEAR     ; same as SPOT_CHECK LABEL NEAR
```

HIGH, LOW

HIGH and LOW are byte isolation operators.

Syntax

HIGH expression

or

LOW expression

Where

HIGH	isolates the high 8 bits of an absolute 16-bit value or address expression.
LOW	isolates the low 8 bits of an absolute 16-bit value or address expression.

Examples

```
mov  ah,high word_value    ; move high byte of word_value
mov  al,low OABCDH         ; move OCDH
```

VALUE RETURNING OPERATORS

These operators return the attribute values of the operands that follow them but do not override the attributes.

The value returning operators take labels and variables as their arguments.

Because variables in Macro Assembler have three attributes, you need to use value returning operators to isolate single attributes, as follows:

SEG	isolates the segment base address
OFFSET	isolates the offset value
TYPE	isolates either type or distance
LENGTH and SIZE	isolate the memory allocation

SEG

SEG returns the segment value (segment base address) of the segment enclosing the label or variable.

Syntax

SEG label

or

SEG variable

Example

```
mov  ax,seg variable_name
mov  seg label_name
```

OFFSET

OFFSET returns the offset value of the variable or label within its segment (the number of bytes between the segment base address and the address where the label or variable is defined).

Syntax

OFFSET expression

Where:

expression can be any label, variable, segment name or other symbol.

Remarks

OFFSET is chiefly used to tell the assembler that the operand is an immediate operand.

OFFSET does not make the value a constant. Only ld, the XENIX link editor can resolve the final value.

OFFSET is not required with uses of the DW or DD directives. The assembler applies an implicit OFFSET to variables in address expressions following DW and DD.

The segment override operator (:) can be used to force OFFSET to return the number of bytes between the item in the expression and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group. See the second example, below.

Examples

```
    mov  bx,offset subrout3
    mov  bx,offset DGROUP:array
```

TYPE

The TYPE Operator returns the number of bytes of the variable type, if the operand is a variable; if the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Syntax

```
TYPE label

or

TYPE variable
```

If the operand is a variable, the following values are returned:

```
BYTE  = 1
WORD  = 2
DWORD = 4
QWORD = 8
TBYTE = 10
STRUC = the number of bytes declared by STRUC
```

Examples

```
mov  ax,type array
jmp  (type get_loc) ptr destiny
```

.TYPE

The .TYPE operator returns a byte that describes two characteristics of the "variable:" the mode, and whether it is External or not. The argument to .TYPE may be any expression (string, numeric, logical). If the expression is invalid, .TYPE returns zero.

Syntax

```
.TYPE variable
```

The byte that is returned contains attributes in bits 1,2, 6 and 8.

. the lower two bits (1 and 2) are the mode. If the lower two bits have the value:

0	the mode is Absolute
1	the mode is Program Related
2	the mode is Data Related

. (bit 8) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is local or public scope.

. The Defined bit is bit 6. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external.

If bits 6 and 8 are zero the expression is invalid.

.TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow; for example, when conditional assembly is involved.

Example

```
x   db   12
z   equ .type x
```

This example sets z to 22H (00100010B). Bit 0 is not set in z because x is not program-related. Bit 6 is set because x is defined. Bit 8 is not set because x is local. The remaining bits are never set.

LENGTH

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for the variable that constitutes its argument.

Syntax

```
LENGTH variable
```

Remarks

If the "variable" is defined by a DUP expression, LENGTH returns the number of type units duplicated; that is, the number that precedes the first DUP in the expression.

If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples

These examples assume the following definitions:

```
array  dw  100 dup(1)
table  dw  100 dup(1, 10 dup(?))
```

In the first of the following examples, LENGTH returns 100:

```
mov  cx,length array
```

In the next example, LENGTH also returns 100, but the returned value does not depend on any nested DUP operators:

```
mov  cx,length table
```

SIZE

syntax

```
SIZE variable
```

Where

SIZE returns the total number of bytes allocated for a variable. SIZE is the product of the value of LENGTH times the value of TYPE.

Example

The following is assumed:

```
array  dw  100 dup(1)
```

In the following example, SIZE returns 200:

```
mov  bx, size array
```

RECORD SPECIFIC OPERATORS

Record specific operators are used to isolate fields in a record.

Records are defined by the RECORD directive (see Section "Memory Directives" in Chapter 4). A record may be up to 16 bits long. The record is defined by fields, which may be from one to 16 bits long. To isolate one of the three characteristics of a record field, you use one of the record specific operators, as follows:

OPERATOR	MEANING
Shift count	Number of bits from low end of record to low end of field (number of bits to right shift the record to lowest bits of record)
WIDTH	The number of bits wide the field or record is (number of bits the field or record contains)
MASK	Value of record if field contains its maximum value and all other fields are zero (all bits in field contain 1; all other bits contain 0)

In the following discussions of the record specific operators, these symbols are used:

SYMBOL	MEANING
ARG	a record defined by the RECORD directive ARG RECORD FIELD1:3,FIELD2:6,FIELD3:7
BAN	a variable used to allocate ARG BAN ARG.
FIELD1, FIELD2, FIELD3	are the fields of the record ARG.

SHIFT_COUNT

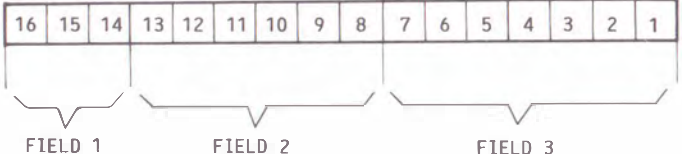
The shift count is derived from the record fieldname to be isolated.

Syntax

(record_fieldname)

The shift count is the number of bits the field must be shifted right to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (FOO) contains three fields (FIELD1, FIELD2, and FIELD3), the record can be diagrammed as follows:



In the above diagram,

FIELD1 has a shift count of 13.
FIELD2 has a shift count of 7.
FIELD3 has a shift count of 0.

When you want to isolate the value in one of these fields, you enter its name as an operand.

Example

```
    mov   dx,ban
    mov   cl,FIELD2
    shr   dx,cl
```

FIELD2 is now right-shifted, ready for access.

WIDTH

When a record_fieldname is given as the argument, WIDTH returns the width of a record field as the number of bits in the record field.

When a record is given as the argument, WIDTH returns the width of a record as the number of bits in the record.

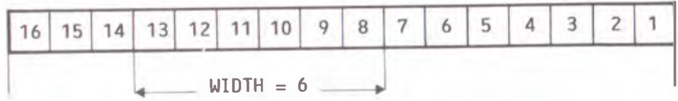
Syntax

WIDTH record_fieldname

or

WIDTH record

Using the diagram under shift count, WIDTH can be diagrammed as:



In the above diagram,

The WIDTH of FIELD1 equals 3.
The WIDTH of FIELD2 equals 6.
The WIDTH of FIELD3 equals 7.

Examples

The above record configuration is given by:

```
rtype  RECORD  field1:3,field2:6,field3:7
rec1   rtype <>
```

This configuration is assumed in the following example:

```
WIDTH field1   ; equals 3
WIDTH field2   ; equals 6
WIDTH field3   ; equals 7
WIDTH rtype    ; equals 16
```

In the next example, the number of bits in field2 is placed in the Count Register:

```
mov  cl, width field2
```

MASK

MASK accepts a field name as its only argument.

MASK returns a bit-mask defined by 1 for bit positions included by the field and 0 for bit positions not included. The value returned represents the maximum value for the record when the field is masked.

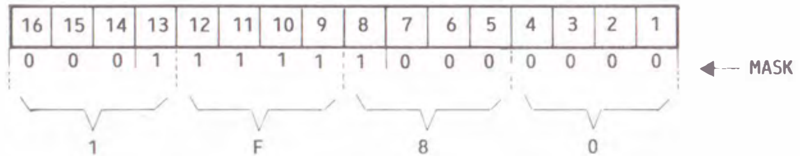
Syntax

```
MASK record_fieldname
```

Where

MASK accepts a field name as its only argument.

Using the diagram used for shift count, MASK can be diagrammed as:



The MASK of FIELD2 equals 1F80H.

Example

Note that the above configuration assumes the following definition:

```
rtype  RECORD  field1:3,field2:6,field3:7
rec1   rtype <>
```

This record definition gives the following values:

```
MASK field1  ; equals E000H
```

```
MASK field2  ; equals 1F80H
MASK field3  ; equals 003FH
MASK rtype   ; equals OFFFFH
```

EXPRESSION EVALUATION AND PRECEDENCE

Expressions are evaluated according to the rules of operator precedence and order. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order of evaluation can be overridden using enclosing parentheses. Operations in parentheses are always performed before any adjacent operations.

Operator Precedence

The following table lists the precedence of all operators. Operators on the same line have equal precedence.

PRECEDENCE	OPERATORS
Highest	
1	LENGTH, SIZE, WIDTH, MASK
2	( )
3	[ ]
4	:
5	PTR, OFFSET, SEG, TYPE, THIS
6	HIGH, LOW
7	*, /, MOD, SHL, SHR
8	+, - (binary)
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
Lowest	

Examples

```
8 / 4 * 2              ; equals 4
8 / (4 * 2)            ; equals 1
8 + 4 * 2              ; equals 16
(8 + 4) * 2            ; equals 24
8 EQ 4 AND 2 LT 3      ; equals 0000H (false)
8 EQ 4 OR 2 LT 3       ; equals OFFFFH (true)
```

FORWARD REFERENCES

Although MASM permits forward references to labels, variable names, segment names, and other symbols, such references can lead to assembly errors if not used properly. A forward reference is any use of a name before it has been formally declared. For example, in the JMP instruction below, the label "target" is a forward reference.

```
        jmp     target
        mov     ax, 0
    target:
```

Whenever MASM encounters an undefined name in pass 1, it assumes that the name is a forward reference. If only a name is given, MASM makes assumptions about that name's type and segment register, and uses these assumptions to generate code or data for the statement. For example, in the JMP instruction above, MASM assumes that "target" is an instruction label having NEAR type. It generates three bytes of instruction code for the instruction.

MASM bases its assumptions on the statement containing the forward reference. Errors can occur when these assumptions are incorrect. For example, if "target" were really a FAR label and not a NEAR label, the assumption made by MASM in pass 1 would cause a phase error. In other words, MASM would generate three bytes of instruction code for the JMP instruction in pass 1, if the distance of target only was resolved to FAR in pass 2, this would be after the incorrect number of bytes was generated, because jumps to FAR labels requires 5 bytes. To solve this problem the label should be declared: target PROC FAR.

To avoid errors with forward references, the segment override (:), PTR, and SHORT operators should be used to override the assumptions made by MASM whenever necessary. The following guidelines list when these operators should be used.

If a forward reference is a variable that is relative to the ES, SS, or CS register, then use the segment override operator (:) to specify the variable's segment register, segment, or group.

Examples

```
    mov  ax, ss:stacktop
    inc  data:time[1]
    add  ax, DGROUP:_I
```

If the segment override operator is not used, MASM assumes that the variable is DS relative.

If a forward reference is an instruction label in a JMP instruction, then use the SHORT operator if the instruction is less than 128 bytes from the point of reference.

Example

```
jmp  short target
```

If SHORT is not used, MASM assumes that the instruction is greater than 128 bytes away. This does not cause an error, but it does cause MASM to generate an extra NOP instruction that is not needed.

If a forward reference is an instruction label in a CALL or JMP instruction, then use the PTR operator to specify the label's type.

Examples

```
call  far ptr print
jmp   near ptr exit
```

MASM assumes that the label has NEAR type, so PTR need not be used for NEAR labels. If the label has FAR type, however, and PTR is not used, a phase error will result.

If the forward reference is a segment name with a segment override operator, use the GROUP statement to associate the segment name with a group name, then use the ASSUME statement to associate the group name with a segment register.

Example

```
DGROUP   segment stack
         assume  ss: DGROUP

code     segment
         .
         .
         .
         mov     ax, stack:stacktop
         .
         .
         .
```

If you do not associate a group with the segment name, MASM may ignore the segment override and use the default segment register for the variable. This usually results in a phase error in pass 2.

STRONG TYPING FOR MEMORY OPERANDS

MASM carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that any relocatable operand used in an instruction that operates on an implied data type must either have that type, or have an explicit type override (PTR operator).

For example, in the following program segment, the variable "string" is incorrectly used in a move instruction.

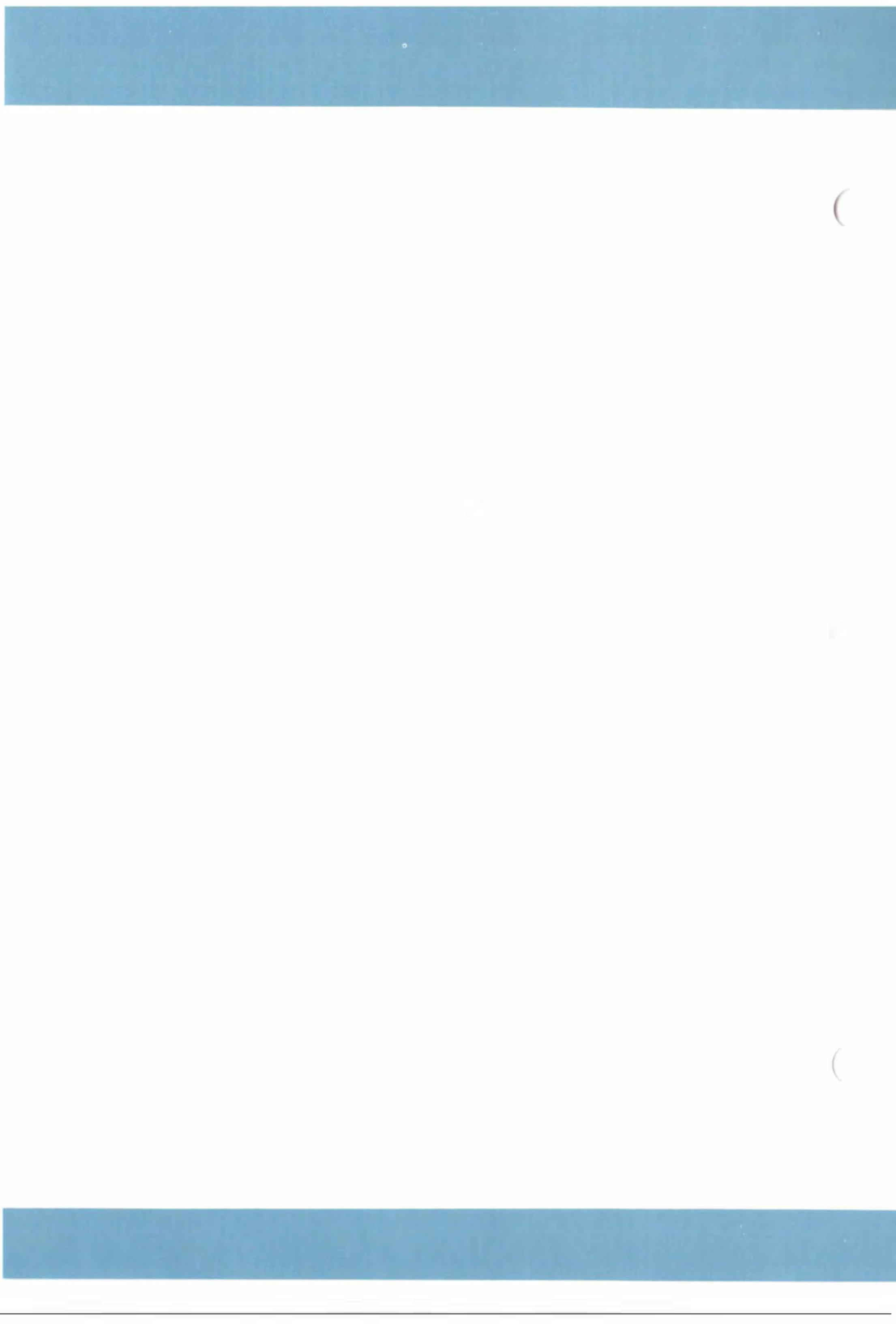
```
    string  db      "A message."
            mov     ax, string[1]
```

This statement will create an "Operand types must match" error since "string" has BYTE type and the instruction expects a variable having WORD type.

To avoid this error, the PTR operator must be used to override the variable's type. The statement

```
    mov     ax, word ptr string[1]
```

will assemble correctly and execute as expected.



4. INSTRUCTION SET AND MEMORY DIRECTIVES

ABOUT THIS CHAPTER

The first part of this chapter gives a fully annotated example of a Macro Assembler source file, exemplifying MASM's major features. The next section describes the Instruction Set Directives, which enable the instruction set for the given microprocessor. The major part of the chapter describes the Memory Directives, which define the organization oı a program's code and data.

CONTENTS

SOURCE FILES	4-1
INSTRUCTION SET DIRECTIVES	4-2
.8086	4-3
.186	4-3
.286	4-4
.287	4-4
MEMORY DIRECTIVES	4-5
ASSUME	4-5
DB, DW, DD, DQ, DT (DEFINE)	4-6
DUP	4-8
END	4-9
EQU	4-10
EQUALS SIGN	4-11
EVEN	4-12
EXTRN	4-12
GROUP	4-14
LABEL	4-15
ORG	4-17
PROC AND ENDP	4-17
PUBLIC	4-19
RECORD	4-20
SEGMENT AND ENDS	4-22
STRUC	4-26

SOURCE FILES

Every assembly language program consists of one or more "source" files: text files that contain statements that define the program's data and instructions. MASM reads source files and assembles the statements to create object modules. Ld, the XENIX V link editor, can be used to prepare these object modules for execution.

Source files must be in standard ASCII format: they must not contain control codes, and each line must be separated by a carriage-return/line-feed combination. Statements can be entered in upper- or lowercase. Sample code in this manual uses uppercase letters for MASM reserved words and for class types, but this is a convention, not a requirement.

All source files have the same form: zero or more program segments followed by an END directive (a source file containing only macros, structures, or records might have zero segments). The END directive, required in every source file, signals the end of the source file. The END directive also provides a way to define the program entry point or starting address (if any).

The following example illustrates the source file format. It is a complete assembly language program that uses XENIX V functions (or system calls) to print the message "Hello world" on the screen.

Example

```
_data   segment                     ; Program Data Segment
hello   db    "Hello.",10
tty     db    "/dev/tty",0
fd      dw    0
_data   ends

extrn   _open:near                  ; External entry points
extrn   _close:near
extrn   _write:near
extrn   _exit:near

_text   segment                     ; Program Code Segment
        assume cs:text, ds:DGROUP, ss:DGROUP, es:DGROUP

_main:                              ; Program Entry Point

        push  2                     ; fd = open("/dev/tty",2)
        push  offset DGROUP:tty
        call  _open
        add   sp,4
        mov   fd,ax

        push  7                     ; write(fd, &hello, 7)
        push  offset DGROUP:hello
        push  fd
```

```
        call   _write
        add    sp,6

        push   fd                     ; close(fd)
        call   _close
        add    sp,2

        push   0                      ; exit
        call   _exit
_text   ends

        end
```

The following main features of this source file should be noted:

1. The SEGMENT and ENDS statements, which define segments named "_data", and "_text".

2. The group statement defining a group "DGROUP" which contains the data segment "_data".

3. The variables "hello" and "tty" in the "_data" segment, defining the string to be displayed and the name of the file which is opened to do this.

4. The instruction label "_main" in the "_text" segment and its "public" declaration, which provides the necessary entry point for the runtime library to call.

5. The "assume" statements in the "_data" and "_text" segments, defining which segment registers will be associated with the labels, variables, and symbols defined within the segments.

INSTRUCTION SET DIRECTIVES

If the -r or -e options to MASM (see "XENIX V System and Application Software Development Tools Reference Manual", Section CP) are used, real numbers use 8087 or 80287 coprocessor format. If -e is used, the assembled object code must be linked with the appropriate maths library. See "MS-Macro Assembler Under XENIX V User Guide" for further details.

The instruction-set directives enable the instruction sets for the given microprocessors. When a directive is given, MASM will recognize and assemble any subsequent instructions belonging to that microprocessor.

The instruction-set directives, if used, must be placed at the beginning of the program source file to ensure that all instructions in the file are assembled using the same instruction set.

8086

Syntax

```
.8086
```

Remarks

The .8086 directive enables assembly of instructions for the 8086 and 8088 microprocessors. It also disables assembly of the instructions unique to the 80186 and 80286 processors. Similarly, the .8087 directive enables assembly of instructions for the 8087 floating-point coprocessor and disables assembly of instructions unique to the 80287 coprocessor.

Since MASM assembles 8086 instructions by default, the .8086 and .8087 directives are not required if the source files contain 8086 and 8087 instructions only. Using the default instruction sets ensures that your programs will be usable on all processors in the 8086/80186/80286 family. However, they will not take advantage of the more powerful instructions available on the 80186, 80286, and 80287 processors.

186

Syntax

```
.186
```

Remarks

The .186 directive enables assembly of the 8086 instructions plus the additional instructions for the 80186 microprocessor. This directive should be used for programs that will be executed by an 80186 microprocessor.

286

Syntax

```
.286{C|P}
```

Remarks

The .286C directive enables assembly of 8086 instructions and non-protected 80286 instructions (identical to the 80186 instructions). The .286P directive enables assembly of the protected instructions of the 80286 in addition to the 8086 and nonprotected 80286 instructions. The .286C directive should be used with programs that will be executed only by an 80286 microprocessor, but do not use the protected instructions of the 80286. The .286P directive can be used with programs that will be executed only by an 80286 processor using both protected and nonprotected instructions.

287

Syntax

```
.287
```

Remarks

The .287 directive enables assembly of instructions for the 80287 floating-point coprocessor. This directive should be used with programs that have floating-point instructions and are intended for execution only by an 80286 microprocessor.

Even though a source file may contain the .8087 or .287 directive, MASM also requires the -r or -e options in the MASM command line to define how to assemble floating point instructions. The -r option directs the assembler to generate the actual instruction code for the floating-point instruction. The -e option enables the assembler to generate code that can be used by a floating-point-emulator routine.

MEMORY DIRECTIVES

The following directives are used to structure and organize memory:

ASSUME

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register.

Syntax

```
ASSUME segment_register : segment_register[ , ...]

or

ASSUME NOTHING
```

Remarks

When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from 1 to 4 arguments to ASSUME.

The valid "segment_register" entries are:

CS, DS, ES, and SS.

The possible entries for "seg_name" are:

- The name of a segment declared with the SEGMENT directive
- The name of a group declared with the GROUP directive
- An expression: either "SEG variable_name" or "SEG label_name"
- The key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement

If ASSUME is not used or if NOTHING is typed for "seg_name", each reference to variables, symbols, labels, and so forth in a particular segment must be prefixed by a segment register. For example, type DS:FOO instead of simply FOO.

Examples

```
assume  cs:code
assume  cs:cgroup,ds:dgoup,ss:nothing,es:nothing
assume  nothing
```

DB,DW,DD,DQ,DT,(Define)

The DEFINE directives are used to define variables or to initialize portions of memory.

```
[name] DB initial_value,,,
[name] DW initial_value,,,
[name] DD initial_value,,,
[name] DQ initial_value,,,
[name] DT initial_value,,,
```

Remarks

If the optional "name" is entered, the DEFINE directives define the name as a variable. The argument "initial_value" can be any of the following:

Directive	initial_value
DB	integer; character string constant; DUP operator; constant expression; question mark (?).
DW	integer; string constant; DUP operator; constant expression; address expression; question mark (?).
DD	integer; real number; 1- or 2-character string constant; encoded real number; DUP operator; constant expression; address expression; question mark (?).

DQ	integer; real number; 1- or 2-character string constant; encoded real number; DUP operator; constant expression; question mark (?).
DT	integer expression; packed decimal, 1- or 2-character string constant; encoded real number; DUP operator; question mark (?).

Notes

A question mark (?) represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The DEFINE directives allocate memory in units specified by the second letter of the directive (each DEFINE directive may allocate one or more of its units at a time):

DB allocates one byte (8 bits)
DW allocates one word (2 bytes)
DD allocates two words (4 bytes)
DQ allocates four words (8 bytes)
DT allocates ten bytes

Examples

Define Byte (DB):

```
integer       db  16
string        db  'ab'
message       db  "Enter Your Name: "
constantexp   db  4*3
empty         db  ?
multiple      db  1, 2, 3, '$'
duplicate     db  10 dup(?)
high_byte     db  255
```

Define Word (DW):

```
integer       dw  16728
character     dw  'a'
string        dw  'bc'
constantexp   dw  4*3
addressexp    dw  string
empty         dw  ?
multiple      dw  1, 2, 3, '$'
duplicate     dw  10 dup(?)
high_word     dw  65535
arrayptr      dw  array
arrayptr2     dw  offset DGROUP:array
```

Define Doubleword (DD):

```
integer        dd  16728
character      dd  'a'
string         dd  'ab'
real           dd  1.5
encodedreal    dd  3f000000R
constantexp    dd  4*3
addsegexp      dd  real
empty          dd  ?
multiple       dd  1, 2, 3, '$'
duplicate      dd  10 dup(?)
high_double    dd  4294967295
```

Define Quadword (DQ):

```
integer        dq  16728
character      dq  'a'
string         dq  'ab'
real           dq  1.5
encodedreal    dq  3f00000000000000R
constantexp    dq  4*3
empty          dq  ?
multiple       dq  1, 2, 3, '$'
duplicate      dq  10 dup(?)
high_quad      dq  18446744073709551615
```

Define Tenbytes (DT):

```
packeddecimal dt  1234567890
integer       dt  16728D
character     dt  'a'
string        dt  'ab'
real          dt  1.5
encodedreal   dt  3f000000000000000000R
empty         dt  ?
multiple      dt  1, 2, 3, '$'
duplicate     dt  10 dup(?)
high_byte     dt  1208925819614629174706175D
```

DUP

The DUP statement specifies multiple occurences of one or more initial values.

Syntax

```
count DUP(initial_value,,,)
```

Remarks

The argument "count" specifies the number of times to repeat "initial_value". An initial value can be any exprssion that evaluates to an integer value, a character constant, or another DUP operator. If more than one initial value is given, they must be separated by commas (,). DUP operators may be nested to 17 levels.

Examples

The first example generates 00 bytes with value 1:

```
db  100 dup(1)
```

The second example generates 80 words of data. the first four words have the values 1, 2, 3, and 4 respectively. This pattern is repeated for the remaining words:

```
db  20 dup(1, 2, 3, 4)
```

The third example generates 125 bytes of data, each byte having the value 1:

```
db  5 dup(5 dup(5 dup(1)))
```

The last example generates 14 doublewords of uninitialised data:

```
dd  14 dup(?)
```

END

The END statement specifies the end of the program.

Syntax

```
END [exp]
```

Remarks

If "exp" is present, it is the start address of the program. If several modules are to be linked, only one module may specify the start of the program with the "END exp" statement.

If "exp" is not present, then no start address is passed to ld, the XENIX V link editor, for that program or module.

Statements occuring after an END statement are ignored.

Warning

If you fail to define an entry point for the main module, your program may not be able to initialize correctly. The program will assemble and link without error messages, but it may crash when you attempt to run it. Remember, one (and only one) module must define an entry point.

Example

```
    end
    end _start
```

EQU

EQU assigns the value of "exp" to "name".

Syntax

```
<name> EQU <exp>
```

Remarks

If "exp" is an external symbol, an error is generated. If "name" already has a value, an error is generated. If you want to be able to redefine a "name" in your program, use the equal sign (=) directive instead.

In many cases, EQU is used as a primitive text substitution, like a macro.

The argument "exp" may be any one of the following:

- A symbol. "Name" becomes an alias for the symbol in "exp". "Name" is shown as an Alias in the symbol table.

- An instruction name. Shown as an Opcode in the symbol table.

- A valid expression. Expressions that evaluate to a value in the range 0 to 65535 create values. (Shown as a Number or L (label) in the symbol table. Any number outside this range becomes text (shown as text in the symbol table).

- Any other entry, including text, index references, segment prefix and operands. Shown as Text in the symbol table.

Examples

```
integer      equ  16728
real         equ  3.14159
constantexp  equ  3*4
memoryop     equ  [bp]
mnemonic     equ  mov
addressexp   equ  real
string       equ  'Type Enter'
```

EQUALS SIGN

The equals sign (=) allows the user to set and to redefine symbols.

Syntax

```
name = exp
```

Where

exp must be a valid expression that evaluates to a value in the range 0 to 65535. It is shown as a Number or L (label) in the symbol table

The equal sign is like the EQU directive, except the user can redefine the symbol without generating an error. Redefinition may take place more than once, and redefinition may refer to a previous definition.

Examples

```
integer     =  16728
string      =  'ab'
constantexp =  3*4
addressexp  =  string
```

EVEN

The EVEN directive causes the program counter to go to an even boundary; that is, to an address that begins a word.

EVEN

Remarks

If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP instruction so that the counter will reach an even boundary.

An error results if EVEN is used with a byte-aligned segment.

Example

In the following example, EVEN increments the location counter and generates a NOP instruction (90H). This means that the offset of "test2" is 2, not 1.

```
        org   0
test1   db    1
        even
test2   dw    513
```

EXTRN

The EXTRN directive identifies a procedure or function that resides in another loaded module.

Syntax

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

Where

name is a symbol that is defined in another module. "Name" must have been declared PUBLIC in the module where "name" is defined.

type may be any one of the following, must be a valid type for "name:"

- BYTE, WORD, DWORD, QWORD, TBYTE,
- NEAR or FAR for labels or procedures (defined under a PROC directive)
- ABS for pure numbers (implicit size is WORD, but includes BYTE)

If the directive is given with a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments, then use either

ASSUME seg-reg:SEG name

or an explicit segment prefix.

Remarks

If a mistake is made and the symbol is not in the segment, ld will take the offset relative to the given segment, if possible. If the real segment is less than 64K bytes away from the reference, ld may find the definition. If the real segment is more than 64K bytes away, ld will fail to make the link between the reference and the definition and will return an error message.

Example

```
    extrn  tagn:near
    extrn  var1:word, var2:dword
```

GROUP

The GROUP directive collects the segments named after "GROUP (segment name)" under one name.

Syntax

```
name GROUP segment_name[,segment_name]...
```

Remarks

The GROUP directive collects the segments named after "GROUP (segment_names)" under one name. The GROUP is used by ld so that it knows which segments should be loaded together (the order the segments are named here does not influence the order in which the segments are loaded. The order in which the segments are loaded is determined by the CLASS designation of the SEGMENT directive, or by the order you name object modules in response to the ld Object Module : prompt).

All segments in a GROUP must fit into 64K bytes of memory. The assembler does not check this at all, but leaves the checking to ld.

The argument "segment_name" may be one of the following:

- A segment name, assigned by a SEGMENT directive. The name may be a forward reference.

- An expression: either "SEG var" or "SEG label". Both of these entries resolve themselves to a segment name (see SEG operator).

Once you have defined a group name, you can use the name:

- As an immediate value:

```
mov ax,DGROUP
mov ds,ax
```

DGROUP is the paragraph address of the base of DGROUP.

- In ASSUME statements:

```
assume ds:DGROUP
```

The DS register can now be used to reach any symbol in any segment of the group.

- As an operand prefix (for segment override):

```
mov    bx,offset DGROUP:foo
dw     DGROUP:foo
dd     DGROUP:foo
```

DGROUP: forces the offset to be relative to DGROUP, instead of to the segment in which FOO is defined.

Warning

A group name must not be used in more than one GROUP directive in any source file. If several segments within the source file belong to the same group all segment names must be given in the same GROUP directive.

Example

```
DGROUP  group _data, _BSS
        assume      ds:DGROUP

_data   segment word public 'data'
        .
        .
        .
_data   ends

_BSS    segment word public 'bss'
        .
        .
        .
_BSS    ends
        end
```

LABEL

By using LABEL to define a "name", you cause the assembler to associate the current segment offset with <name>.

Syntax

```
name LABEL type
```

Where

type varies depending on the use of "name"; "name" may be used for code or for data.

- For code (for example, as a JMP or CALL operand):

  "Type" may be either NEAR or FAR. "Name" cannot be used in data manipulation instructions without using a type override.

  If you wish, you can define a NEAR label using the "name:" form (the LABEL directive is not used in this case). If you are defining a BYTE or WORD NEAR label, you can place the "name:" in front of a Define directive.

  When using a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

- For data:

  "Type" may be BYTE, WORD, DWORD, QWORD, TWORD, "structure_name", or "record_name". When STRUC or RECORD name is used, "name" is assigned the size of the structure or record.

Example

For code:

```
subroutine  label far
subroutine: (first instruction)  ; colon = near label
```

Example

For Data:

```
barray  label byte
array   dw    100 dup(0)
        .
        .
        .
        add   al,barray[99]  ;  add 100th byte to al
        add   ax,array[98]   ;  add 50th word to ax
```

By defining the array two ways, you can access entries either by byte or by word. Also, you can use this method for STRUC. It allows you to place your data in memory as a table, and to access it without the offset of the STRUC.

Defining the array two ways also permits you to avoid using the PTR operator. The double defining method is especially effective if you access the data different ways. It is easier to give the array a second name than to remember to use PTR.

ORG

The location counter is set to the value of "exp", and the assembler assigns generated code starting with that value.

Syntax

```
ORG exp
```

Remarks

All names used in "exp" must be known on pass 1. The value of "exp" must either evaluate to an absolute or must be in the same segment as the location counter.

Example

```
    org  120H  ;  2-byte absolute value maximum=0FFFFH
    org  $+2   ;  skip 2 bytes
```

Example - ORG to a boundary (conditional):

```
    cseg    segment page
    begin   = $

    if ($-begin) mod 256       ;  if not already on a 256-byte boundary
            org($-begin) + 256 - (($-begin)mod 256)
    endif
```

See Chapter 5 "Conditional Directives," for an explanation of conditional assembly.

PROC and ENDP

The PROC and ENDP directives mark the beginning and end of a procedure.

Syntax

```
procname PROC {[NEAR]|[FAR]}
.
.
.
RET
name ENDP
```

Remarks

The default, if no operand is specified, is NEAR. Use FAR if:

- The procedure name is an operating system entry point
- The procedure will be called from code which has another ASSUME CS value

Each PROC block usually contains a RET statement.

The PROC directive, through the NEAR|FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL, and RETs to generate a NEAR or a FAR RET. PROC is used, therefore, for coding simplification so that the user does not have to worry about NEAR or FAR for CALLs and RETs.

A NEAR CALL or RETURN changes the IP but not the CS register. A FAR CALL or RETURN changes both the IP and the CS registers.

Procedures are executed either in line, from a JMP, or from a CALL.

PROCs may be nested, which means that they are put in line.

Combining the PUBLIC directive with a PROC statement (both NEAR and FAR), permits you to make external CALLs to the procedure or to make other external references to the procedure.

Example

```
_main   proc   near
        push   bp
        mov    bp,sp
        push   si
        push   di
        mov    ax,offset DGROUP:string
        push   ax
        call   _printf
        add    sp,2
        pop    di
        pop    si
```

```
        mov   sp,bp
        pop   bp
        ret
_main   endp
```

PUBLIC

Place a PUBLIC directive statement in any module that contains symbols you want to use in other modules without defining the symbol again. PUBLIC means the listed symbol(s), which are defined in the module where the PUBLIC statement appears, available for use by other modules to be linked with the module that defines the symbol(s). This information is passed to ld.

Syntax

```
PUBLIC symbol [,symbol]...
```

Where

symbol may be a number, a variable, a label (including PROC labels). It may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes.

Examples

```
      public   true, test, start
true  =        0FFFFH
test  db       1
start label    far
```

The following example is illegal:

```
            public   pie_bald, high_value
pie_bald    equ      3.1416
high_value  equ      999999999
```

RECORD

A record is a bit pattern you define to format bytes and words for bit-packing.

Syntax

```
recordname RECORD fieldname:width[ = exp][,fieldname:width[=exp]]...
```

Where

fieldname — is the name of the field. "Width" specifies the number of bits in the field defined by "fieldname". "Exp" contains the initial (or default) value for the field. Forward references are not allowed in a RECORD statement.

"Fieldname" becomes a value that can be used in expressions. When you use "fieldname" in an expression, its value is the shift count to move the field to the far right. Using the MASK operator with the "fieldname" returns a bit mask for that field.

width — is a constant in the range 1 to 16 that specifies the number of bits contained in the field defined by "fieldname". The WIDTH operator returns this value. If the total width of all declared fields is larger than 8 bits, then the assembler uses two bytes. Otherwise, only one byte is used.

exp — contains the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character as the "exp"

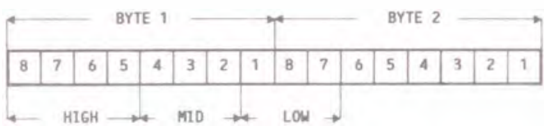
Remarks

The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

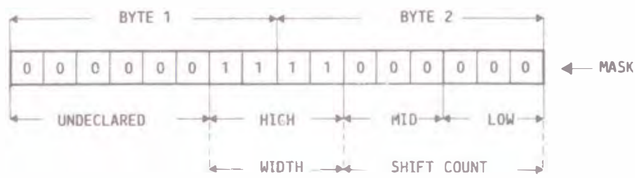
Example 1

```
foo  record high:4,mid:3,low:3
```

Initially, the bit map would be:



In this example the total bits are greater than 8, so the field requires two bytes; however, the total bits are less than 16, so the data is right shifted and all unused bits are at the left.



Example 2

```
item  record char:7='Q'
```

To initialize records, use the same method used for DB. The format is:

[<name>] <recordname> <[exp][,...]>

or

[<name>] <recordname> [<exp> DUP(<[exp][,...]>)

The name is optional. When given, name is a label for the first byte or word of the record storage area.

The recordname is the name used as a label for the RECORD directive.

The [exp] (both forms) contains the values you want placed into the fields of the record. In the latter case, the parentheses and angle brackets are required only around the second [exp] (following DUP). If [exp] is left blank, either the default value applies (the value given in the original record definition), or the value is indeterminate (when not initialized in the original record definition). For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields.

For example:

```
foo ,,7
```

From the previous example, the 7 would be placed into the LOW field of the record FOO. The fields HIGH and MID would be left as declared (in this case, uninitialized).

A record may be used in an expression (as an operand) in the form:

recordname<[value[,...]]>

The value entry is optional. The angle brackets must be coded as shown, even if the optional values are not given. A value entry is the value to be placed into a field of the record. For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields, as shown above.

Example 3

```
foo     record high:5,mid:3,low:3
        .
        .
        .
bax     foo                         ; leave undetermined here
jane    foo    10 dup(<16,8>)       ; high=16, mid=8, low=?
        .
        .
        .
        mov    dx,offset jane[2]    ; get beginning record address
        and    dx,mask mid
        mov    cl,mid
        shr    dx,cl
        mov    cl,width mid
```

SEGMENT AND ENDS

These directives mark the beginning and end of a program segment. At runtime, all instructions that generate code and data are in (separate) segments. Your program may be a segment, part of a segment, several segments, parts of several segments, or a combination of these. If a program has no SEGMENT statement, an ld error (invalid object) will result at link time.

A program segment is a collection of instructions and/or data whose addresses are all relative to the same segment register.

Syntax

```
segname SEGMENT [align] [combine] ['class']
.
.
.
segname ENDS
```

Where

segment name must be a unique, legal name. The segment name must not be a reserved word.

align may be PARA (paragraph - default), BYTE, WORD, or PAGE.

combine may be PUBLIC, COMMON, AT, exp, STACK, MEMORY, or no entry (which defaults to not combinable, called Private in the LINK section of the manual).

class name is used to group segments at link time.

All three operands are passed to ld.

The alignment type tells the Linker on what kind of boundary you want the segment to begin. The first address of the segment will be, for each alignment type:

PAGE - address is xxx00H (low byte is 0)

PARA - address is xxxx0H (low nibble is 0)

bit map - |x|x|x|x|0|0|0|0|

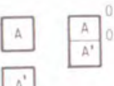
WORD - address is xxxxeH (e=even number;low bit is 0)

bit map - |x|x|x|x|x|x|x|0|

BYTE - address is xxxxxH (place anywhere)

The combine type tells ld how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

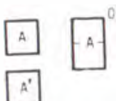
1. None (not combinable or Private)



Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base

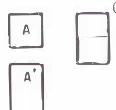
address.

2. PUBLIC AND STACK



Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. Combine type stack is treated the same as public. However the Stack Pointer is set to the first address of the first stack segment. Ld requires at least one stack segment. If you create a stack segment without using the STACK combine type, you must give instructions to load the segment address into SS.

3. COMMON



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

4. MEMORY

The memory combine type causes the segment(s) to be placed as the highest segments in memory. The first memory combinable segment encounter is placed as the highest segment in memory. Subsequent segments are treated the same as Common segments.

Note that this feature is not supported by ld. Ld treats Memory segments the same as Public segments.

5. AT exp

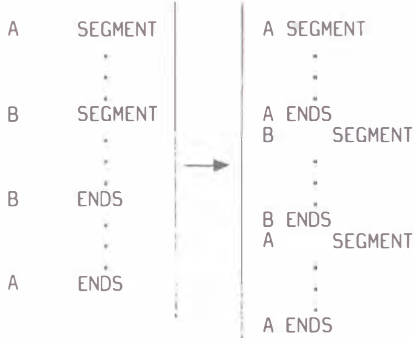
The segment is placed at the PARAGRAPH address specified in "exp". The expression may not be a forward reference. Also, the AT type may not be used to force loading at fixed addresses. Rather, the AT combine type permits labels and variables to be defined at fixed offsets within fixed areas of storage, such as ROM or the vector space in low memory.

Note that this restriction is imposed by MS-LINK and XENIX.

Class names must be enclosed in quotation marks. Class names may be any legal name.

Segment definitions may be nested. When segments are nested, the assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, the assembler takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. Overlapping segments are not permitted.

Example 1



The following arrangement is not allowed:

```
A       SEGMENT
          .
          .
          .
B       SEGMENT
          .
          .
          .
A       ENDS    ; this is illegal!
          .
          .
          .
B       ENDS
```

Example 2

In module A:

```
SEGA        SEGMENT PUBLIC 'CODE'
            ASSUME  CS:SEGA
              .
              .
              .
SEGA        ENDS
            END
```

In module B:

```
SEGA        SEGMENT PUBLIC 'CODE'
ASSUME      CS:SEGA
.                   ; Ld adds this segment to same
.                   ; named segment in module A (and others)
.                   ; if class name is the same.
SEGA        ENDS
END
```

Warning

Normally for a.out files you should provide at least one stack segment in a program. If no stack segment is declared, ld will display a warning message. However this message can be ignored if you are going to convert your a.out file to a .com file, or have another reason for not declaring a stack segment.

STRUC

The STRUC directive is very much like RECORD, except STRUC has a multiple byte capability.

```
structure_name STRUC
.
.
.
structure_name ENDS
```

Remarks

The allocation and initialization of a STRUC block are the same as for RECORDs.

Inside the STRUC/ENDS block, the Define directives (DB, DW, DD, DQ, DT) may be used to allocate space. The Define directives and Comments set off by semicolons (;) are the only statement entries allowed inside a STRUC block.

Any label on a Define directive inside a, STRUC/ENDS block becomes a "fieldname" of the structure. (This is how structure fieldnames are defined.) Initial values given to fieldnames in the STRUC/ENDS block are default values for the various fields. These field values are of two types: overridable or not overridable. A simple field, a field with only one entry (but not a DUP expression), is overridable. A multiple field, a field with more than one entry, is not overridable.

For example:

```
foo    db    1,2              ; is not overridable
baz    db    10 dup(?)        ; is not overridable
zoo    db    5                ; is overridable
```

If the "exp" following the Define directive contains a string, it may be overridden by another string. However, if the overriding string is shorter than the initial string, the assembler will pad with spaces. If the overriding string is longer, the assembler will truncate the extra characters.

Usually, structure fields are used as operands in some expression. The format for a reference to a structure field is:

---

variable.field

Where

variable represents an anonymous variable, usually set up when the structure is allocated. To allocate a structure, use the structure name as a directive with a label (the anonymous variable of a structure reference) and any override values in angle brackets:

```
foo    structure
       .
       .
       .
foo    ends
goo    foo ,7,,'joe'
```

.field represents a label given to a DEFINE directive inside a STRUC/ENDS block (the period must be coded as shown). The value of "field" will be the offset within the addressed structure.

Example

To define a structure:

```
s         struc
field1    db    1,2          ; not overridable
field2    db    10 dup(?)    ; not overridable
field3    db    5            ; overridable
field4    db    'dobosky'    ; overridable
s         ends
```

The Define directives in this example define the fields of the structure, and the order corresponds to the order values are given in the

initialization list when the structure is allocated. Every Define directive statement line inside a STRUC block defines a field, whether or not the field is named.

To allocate the structure:

```
dbarea s ,,7,'andy'      ;  overrides 3rd and 4th fields only
```

To refer to a structure:

```
mov   al,[bx].field3
mov   al,dbarea.field3
```

5. CONDITIONAL DIRECTVES

ABOUT THIS CHAPTER

This chapter begins by describeing the MASM Conditional Assembly Directives. The final part of the chapter describes the Conditional Error Directives, which help debug programs by checking for assembly-time errors.

CONTENTS

INTRODUCTION	5-1
IF	5-2
IFE	5-3
IF1	5-4
IF2	5-5
IFDEF	5-6
IFNDEF	5-7
IFB	5-8
IFNB	5-9
IFIDN	5-10
IFDIF	5-11
CONDITIONAL ERROR DIRECTIVES	5-12
.ERR, .ERR1 AND .ERR2	5-13
.ERRE AND .ERRNZ	5-14
.ERRDEF AND .ERRNDEF	5-14
.ERRB AND .ERRNB	5-15
.ERRIDN AND .ERRDIF	5-16

INTRODUCTION

The Conditional Assembly directives provide conditional assembly of blocks of statements within a source file. There are the following conditional directives:

DIRECTIVE	MEANING
IF	Test the value of an expression for truth
IFE	Test the value of an expression for false-hood
IF1	Tests for pass one of the current assembly
IF2	Tests for pass two of the current assembly
IFDEF	Test to see whether the given name has been defined
IFNDEF	Test to see whether the given name has not been defined
IFB	Grants assembly if its argument is blank
IFNB	Grants assembly if its argument exists
IFIDN	Compares the two arguments of the directive for identity
IFDIF	Compares the two arguments of the directive for non-identity
ELSE	An optional compliment to any of the IF directives
ENDIF	Must be used to mark the end of any conditional-assembly block

The IF directives and the ENDIF directive are used to enclose the statements to be considered for conditional assembly:

```
IF
statements
.
.
.
ELSE
statements
.
.
.
ENDIF
```

Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated Conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a Code 8, "Not in conditional block" error.

Each conditional block may include the optional ELSE directive, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a Code 7, "Already had ELSE clause" error.

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid Phase errors and incorrect evaluation. For IF and IFE the expression must involve values which were previously defined, and the expression must be absolute. If the name is

defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

The assembler evaluates the conditional statement to TRUE (which equals any nonzero value), or to FALSE (which equals 0000H). If the evaluation matches the condition defined in the conditional statement, the assembler either assembles the whole conditional block or, if the conditional block contains the optional ELSE directive, assembles from IF to ELSE; the ELSE to ENDIF portion of the block is ignored. If the evaluation does not match, the assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE directive, assembles only the ELSE to ENDIF portion; the IF to ELSE portion is ignored.

IF

This directive causes the statements in the conditional block to be assembled if the condition is true.

Syntax

```
IF expression
[tstatement]
.
.
.
[ELSE]
[tstatement]
.
.
.
ENDIF
```

Where

expression	A combination of operands and operators that evaluates to a single value, which is either zero (FALSE) or non-zero (TRUE).
tstatement(s)	When expression = TRUE, tstatement(s) are assembled.
fstatement(s)	When expression = FALSE, fstatement(s) are assembled.

Warning

The expression must resolve to an absolute value and must not contain forward references

Example

```
    if debug
        extrn dump:far
        extrn trace:far
        extrn breakpoint:far
    endif
```

IFE

This directive causes the statements in the conditional block to be assembled if the condition is false.

Syntax

```
IFE expression
[fstatement]
.
.
.
[ELSE]
[tstatement]
.
.
.
ENDIF
```

Where

expression	A combination of operands and operators that evaluates to a single value which is either zero (FALSE) or non-zero (TRUE).
fstatement(s)	When expression = FALSE, fstatement(s) are assembled.
tstatement(s)	When expression = TRUE, tstatement(s) are assembled.

Warning

The expression must resolve to an absolute value and must not contain forward references.

Example

```
    ife debug
         extrn dump:far
         extrn trace:far
         extrn breakpoint:far
    endif
```

IF1

This directive tests the current assembly pass, and grants assembly on pass 1 only. The directive takes no arguments.

Syntax

```
IF1
[p1statement]
.
.
.
[ELSE]
[p2statement]
.
.
.
ENDIF
```

Where

p1statement(s) On the first pass p1 statement(s) are assembled but not on the second assembly pass.

p2statement(s) On the second pass p2statement(s) are assembled but not on the first assembly pass.

Example

```
    ifl
        %out Pass l Starting
    endif
```

IF2

This directive tests the current assembly pass and grants assembly on pass 2. The directives take no arguments.

Syntax

```
IF2
[p2statement]
.
.
.
[ELSE]
[plstatement]
.
.
.
ENDIF
```

Where

p2statement(s) On the second pass p2statement(s) are assembled, but not on the first assembly pass.

plstatement(s) On the first pass plstatement(s) are assembled but not on the second assembly pass.

Example

```
    if2
        %out Pass 2 Starting
    endif
```

IFDEF

This directive tests whether or not the given "name" has been defined. It grants assembly if "name" is a label, variable, or symbol.

The "name" can be any valid name. Note that if "name" is a forward reference, it is considered undefined on pass 1, but defined on pass 2. This is a frequent cause of phase errors.

Syntax

```
IFDEF name
[dstatement]
.
.
.
[ELSE]
[ustatement]
.
.
.
ENDIF
```

Where

dstatement(s) When name has been defined as a label variable or symbol dstatement(s) are assembled.

ustatement(s) When name has not been defined ustatement(s) are assembled.

Example

```
    ifdef    buffer
    buffer   db   10 dup(?)
    endif
```

IFNDEF

This directive tests whether or not the given "name" has been defined. It grants assembly if "name" has not yet been defined.

The "name" can be any valid name. Note that if "name" is a forward reference, it is considered undefined on pass 1, but defined on pass 2. This is a frequent cause of phase errors.

Syntax

```
IFNDEF name
[ustatement]
.
.
.
[ELSE]
[dstatement]
.
.
.
ENDIF
```

Where

ustatement(s) When name has not been defined ustatement(s) are assembled.

dstatement(s) When name has been defined as a label, variable or symbol dstatement(s) are assembled.

Example

```
    ifndef   buffer
    buffer   db  10 dup(?)
    endif
```

IFB

This directive tests the given "argument". It grants assembly if "argument" is blank. Arguments can be any name, number, or expression. The angle brackets (< >) are required.

The IFB and IFNB directives are intended to be used in macro definitions. They can be used to control conditional assembly of statements in the macro based on the parameters passed in the macro call. In such cases, "arg" should be one of the dummy parameters listed by the MACRO directive.

Syntax

```
IFB <argument>
[bastatement]
.
.
.
[ELSE]
[astatement]
.
.
.
ENDIF
```

Where

<argument> This can be a name or a number or an expression.

bastatement(s) When argument is blank, bastatement(s) are assembled.

astatement(s) When argument is non-blank astatement(s) are assembled.

Example

```
ifb <x>
```

This example tests the argument "<x>". If this is in a macro definition, and no parameter was passed for x, the directive would grant assembly.

IFNB

This directive tests the given "argument". It grants assembly if "argument" is not blank. The "arg" can be any name, number, or expression. The angle brackets (< >) are required.

The IFB and IFNB directives are intended to be used in macro definitions. They can be used to control conditional assembly of statements in the macro based on the parameters passed in the macro call. In such cases, "arg" should be one of the dummy parameters listed by the MACRO directive.

Syntax

```
IFNB <argument>
[astatement]
.
.
.
[ELSE]
[bastatement]
.
.
.
ENDIF
```

Where

<argument> This can be a name or a number or an expression. The angle brackets (< >) are required.

astatement(s) When argument is non-blank, astatement(s) are assembled.

bastatement(s) When argument is blank, bastatement(s) are assembled.

Examples

```
    ifnb <&exit>
```

This example tests the argument "<&exit>". This is assumed to be in a macro definition. If no parameter is passed for exit, the directive does not grant assembly.

IFIDN

This directive tests to see if the two arguments are identical. It grants assembly if the arguments are identical.

Syntax

```
IFIDN <argument1>, <argument2>
[istatement]
.
.
.
[ELSE]
[dstatement]
.
.
.
ENDIF
```

Where

<argument1>
<argument2>
: These arguments can be any name, number or expression. The angle brackets (<>) are required.

istatement(s) When argument1 is identical with argument2 istatement(s) are assembled.

dstatement(s) When argument1 is not identical with argument2 dstatement(s) are assembled.

The arguments can be any names, numbers, or expressions. To be identical, each character in "arg1" must match the corresponding character in "arg2". The angle brackets (< >) are required.

The IFIDN and IFDIF directives are intended to be used in macro definitions. They can be used to control conditional assembly of statements in the macro based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the MACRO directive.

Example

```
    ifidn <x>, <y>
```

This example tests the arguments "<x>" and "<y>". If this is in a macro definition and the parameters passed for x and y are identical, the directive grants assembly.

IFDIF

This directive tests to see whether the two arguments are not identical. It grants assembly if the arguments are different.

Syntax

```
IFDIF <argument1>,<argument2>
[dstatement]
.
.
.
[ELSE]
[istatement]
.
.
.
ENDIF
```

Where

<argument1> <argument2>	These arguments can be any name, number or expression. The angle brackets (<>) are required.
dstatement(s)	When argument1's not identical with argument2 dstatement(s) are assembled.
istatement(s)	When argument1's identical with argument2 istatement(s) are assembled.

The arguments can be any names, numbers, or expressions. To be identical, each character in "arg1" must match the corresponding character in "arg2". The angle brackets (< >) are required.

The IFIDN and IFDIF directives are intended to be used in macro definitions. They can be used to control conditional assembly of

statements in the macro based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the MACRO directive.

Example

```
    ifdif <&exit> <case>
```

This example tests the arguments "<&exit>" and "<case>". This is assumed to be in a macro definition. If the parameters passed for "exit" and "case" are identical, the directive does not grant assembly.

CONDITIONAL ERROR DIRECTIVES

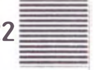
Conditional error directives can be used to debug programs and check for assembly-time errors. By inserting a conditional error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional error directives to test for boundary conditions in macros.

The conditional error directives, and the errors they produce, are listed in the following table:

DIRECTIVE	NUMBER	MESSAGE
.ERR1	87	Forced error - pass1
.ERR2	88	Forced error - pass2
.ERR	89	Forced error
.ERRE	90	Forced error - expression equals 0
.ERRNZ	91	Forced error - expression not equal 0
.ERRNDEF	92	Forced error - symbol not defined
.ERRDEF	93	Forced error - symbol defined
.ERRB	94	Forced error - string blank
.ERRNB	95	Forced error - string not blank
.ERRIDN	96	Forced error - strings identical
.ERRDIF	97	Forced error - strings different

Like other fatal assembler errors, those generated by conditional error directives cause the assembler to return exit code 7. If a fatal error is encountered during assembly, MASM will delete the object module. All conditional error directives except ERR1 generate fatal errors.

.ERR, .ERR1, and .ERR2



Syntax

```
.ERR
.ERR1
.ERR2
```

The .ERR, .ERR1, and .ERR2 directives force an error at the points at which they occur in the source file. The .ERR directive forces an error regardless of the pass, while the .ERR1 and .ERR2 directives force the error only on their respective passes. The .ERR1 directive only appears on the screen or in the listing file if you use the /D option to request a Pass 1 listing. Unlike other conditional error directives, it is not a fatal error.

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

Example

```
    ifdef    dos
             .
             .
             .
    else
             ifdef xenix
             .
             .
             .
             else
             .err
             endif
    endif
```

This example makes sure that either the symbol dos or the symbol xenix is defined. If neither is defined, the nested ELSE condition is assembled and an error message is generated. Since the .ERR directive is used, an error would be generated on each pass. You could use the .ERR2 directive if you wanted only a fatal error, or you could use the .ERR1 directive if you wanted only a warning error.

.ERRE and .ERRNZ

Syntax

```
.ERRE expression
.ERRNZ expression
```

The .ERRE and .ERRNZ directives test the value of an expression. The .ERRE directive generates an error if the expression is false (0). The .ERRNZ directive generates an error if the expression is true (nonzero). The expression must resolve to an absolute value and must not contain forward references.

Example

```
    buffer  macro   count,bname
            .erre   count le 128           ;  Allocate memory, but
            bname   db      count dup(0)   ;    no more than 128 bytes
            endm

    buffer  128,buf1                       ; Data allocated - no error
    buffer  129,buf2                       ; Error generated
```

In this example, the .ERRE directive is used to check the boundaries of a parameter passed to the macro buffer. If count is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If count is greater than 128, the expression will be false (0) and the error will be generated.

.ERRDEF and .ERRNDEF

Syntax

```
.ERRDEF name
.ERRNDEF name
```

The .ERRDEF and .ERRNDEF directives test whether or not name has been defined. The .ERRDEF directive produces an error if name is defined as a

label, variable, or symbol. The .ERRNDEF directive produces an error if name has not yet been defined. If name is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

Example

```
    .errdef  symbol
    ifdef    config1
             .
             .
             .
             symbol  EQU  0
             .
             .
             .
    endif
    ifdef    config2
             .
             .
             .
             symbol  EQU  1
             .
             .
             .
    endif
    .errndef symbol
```

In this example, the .ERRDEF directive at the beginning of the conditional blocks makes sure that symbol has not been defined before entering the blocks. The .ERRNDEF directive at the end ensures that symbol was defined somewhere within the blocks.

.ERRB and .ERRNB

Syntax

```
.ERRB <string>
.ERRNB <string>
```

The .ERRB and .ERRNB directives test the given string. The .ERRB directive generates an error if string is blank. The .ERRNB directive generates an error if string is not blank. The string can be any name, number, or expression. The angle brackets (<>) are required.

These conditional error directives can be used within macros to test for the existence of parameters.

Example

```
    work  macro  realarg,testarg
          .errb  <realarg>          ;  Error if no parameters
          .errnb <testarg>          ;  Error if more than one parameter
          .
          .
          .
          endm
```

In this example, error directives are used to make sure that one, and only one, argument is passed to the macro. The .ERRB directive generates an error if no argument is passed to the macro. The .ERRNB directive generates an error if more than one argument is passed to the macro.

.ERRIDN and .ERRDIF

Syntax

```
.ERRIDN <string1>,<string2>
.ERRDIF <string1>,<string2>
```

The .ERRIDN and .ERRDIF directives test whether two strings are identical. The .ERRIDN directive generates an error if the strings are identical. The .ERRDIF generates an error if the strings are different. The strings can be names, numbers, or expressions. To be identical, each character in string1 must match the corresponding character in string2. String checks are case-sensitive. The angle brackets (<>) are required.

Example

```
    addem   macro ad1,ad2,sum
            .erridn <ax>,<ad2>       ;  Error if ad2 is 'ax'
            .erridn <AX>,<ad2>       ;  Error if ad2 is 'AX'
            mov     ax,ad1           ;  Would overwrite if ad2 were 'ax'
            add     ax,ad2
            mov     sum,ax           ;  Sum must be register or memory
            endm
```

In this example, the .ERRIDN directive is used to protect against passing the AX register as the second parameter, because the macro won't work if the AX register is passed as the second parameter. Note that the directive is used twice to protect against the two most likely spellings.

6. MACRO DIRECTIVES

ABOUT THIS CHAPTER

The first part of this chapter describes how macros are defined. The second part goes on to describe how they are called. The directives that allow macro handling are described, followed by a description of the special control operation macros.

CONTENTS

INTRODUCTION	6-1
MACRO DEFINITION	6-1
CALLING A MACRO	6-3
ENDM (END MACRO)	6-4
EXITM (EXIT MACRO)	6-5
LOCAL	6-6
PURGE	6-7
REPEAT DIRECTIVES	6-8
REPEAT	6-8
IRP (INDEFINATE REPEAT)	6-9
IRPC (INDEFINATE REPEAT CHARACTER)	6-10
SPECIAL MACRO OPERATORS	6-11
&	6-11
<TEXT>	6-12
;;	6-13
!	6-13
%	6-14

MACRO DIRECTIVES

INTRODUCTION

The macro directives allow you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition directive or one of the repetition directives, and end with the ENDM directive. All of the macro directives may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro directives of the Macro Assembler include:

Macro Definition:

    MACRO

Termination:

    ENDM
    EXITM

Unique symbols within macro blocks:

    LOCAL

Undefine a macro:

    PURGE

Repetitions:

REPT	(repeat)
IRP	(indefinite repeat)
IRPC	(indefinite repeat character)

The macro directives also include some special macro operators:

&	(ampersand)
;;	(double semicolon)
!	(exclamation mark)
%	(percent sign)

MACRO DEFINITION

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

Syntax

```
name MACRO [dummy[,dummy]...]
.
.
.
ENDM
```

Where

name is like a label and conforms to the rules for forming symbols. After the macro has been defined, name is used to invoke the macro.

dummy is formed as any other name is formed. A "dummy" is a place holder that is replaced by a parameter in a one-for-one text substitution when the macro block is used. You should include all "dummys" used inside the macro block on this line. The number of "dummys" is limited only by the length of a line. If you specify more than one "dummy", they must be separated by commas. Macro Assembler interprets a series of "dummys" in the same way as any list of symbol names.

Remarks

A "dummy" is always recognized exclusively as a "dummy". Even if a register name (such as AX or BH) is used as a "dummy", it will be replaced by a parameter during expansion.

One alternative is to list no "dummys":

```
name MACRO
MACRO
```

This type of macro block allows you to call the block repeatedly, even if you do not want or need to pass parameters to the block. In this case, the block will not contain any "dummys".

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable, or symbol. When Macro Assembler evaluates a statement, it first looks at the macro table it builds during pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: Macro Assembler evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, Macro Assembler will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

Example

```
addup   macro  xx,yy,zz
        mov    ax,xx        ; First parameter in ax
        add    ax,yy        ; Add next two parameters and
        mov    ax,zz        ;   leave the result in ax
        endm
```

CALLING A MACRO

To use a macro, enter a macro call statement with the following format.

Syntax

```
name [<parameter, [,parameter]...>]
```

Where

name — is the name of the macro block.

parameter — replaces a dummy on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter.

For example:

```
allocblock 1,2,3,4,5
```

passes five parameters to the macro, but

```
allocblock <1,2,3,4,5>
```

passes only one. The number of parameters in the macro call statement need not be the same as the number of "dummys" in the MACRO definition. If there are more parameters than "dummys", the extras are ignored. If there are fewer, the extra "dummys" will be made null. The assembled code will include the macro block after each macro call statement.

Examples

```
    allocblock  1,2,3,4,5
```

This example passes five numeric parameters to the macro called "allocblock".

The second example passes a single parameter to "allocblock". The parameter is a list of five numbers:

```
    allocblock <1,2,3,4,5>
```

The third example passes three parameters to the macro called "addup".

```
    addup   bx,2,count
```

Assuming that "addup" is defined:

```
    addup   macro  ad1,ad2,ad3
            mov    ax,ad1          ;  First parameter in ax
            mov    ax,ad2          ;  Add next two parameters and
            mov    ax,ad3          ;    leave the result in ax
            endm
```

the assembler will expand the macro to give:

```
    mov    ax,bx
    mov    ax,2
    mov    ax,count
```

ENDM (End Macro)

ENDM tells the assembler that the MACRO or Repeat block is ended.

Syntax

```
ENDM
```

Remarks

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM directive. Otherwise, the "Unterminated REPT/IRP/IRPC/MACRO" message is generated at the end of each pass. An unmatched ENDM also causes an error.

If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Example

```
addup   macro   xx,yy,zz
        mov     ax,xx
        mov     ax,yy
        mov     ax,zz
        endm
```

EXITM (Exit Macro)

The EXITM directive is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional directive.

Syntax

```
EXITM
```

Remarks

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested witnin another block, the outer level continues to be expanded.

Example

```
alloc   macro   times
        x       =
        rept    times           ; Repeat upto 256 times
                ife x-0FFH      ; Does x = 256 yet ?
```

```
                exitm          ;  If so, quit
                else
                db  x          ;  Else allocate x
                endif
        x       =   x + 1      ;  Increment x
        endm
        endm
```

LOCAL

The LOCAL directive is allowed only inside a macro definition block. A LOCAL statement must precede all other types of statements in the macro definition.

Syntax

LOCAL dummy[,dummy]...

Remarks

When LOCAL is executed, the assembler creates a unique symbol for each "dummy" and substitutes that symbol for each occurrence of the "dummy" in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ??0000 to ??FFFF. Users should avoid the form ??nnnn for their own symbols.

Example

```
loop    macro   count,y
        local   a
        mov     ax,y
a:      mov     cx,count
        inc     ax
        jnz     a
        endm
```

PURGE

PURGE deletes the definition of the macro(s) listed after it.

Syntax

```
PURGE macro_name [,macro-name]...
```

Remarks

PURGE provides three benefits:

- It frees text space of the macro body.
- It returns any instruction mnemonics or directives that were redefined by macros to their original function.
- It allows you to "edit out" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method allows you to use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you will not use in this program.

  It is not necessary to PURGE a macro before redefining it. Simply place another MACRO statement in your program, reusing the macro name.

Examples

The first example deletes the macro called "add" from the assembler's memory:

```
purge add
```

The second example deletes three macros called "mac1", "mac2" and "mac9":

```
purge mac1,mac2,mac9
```

Repeat Directives

The directives in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the Repeat directives and MACRO directive are:

- MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

- MACRO allows parameters to be passed to the macro block when a MACRO is called; hence, parameters can be changed.

Repeat directive parameters must be assigned as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be performed for every program execution, then Repeat directives are convenient. With the MACRO directive, you must call in the MACRO each time it is needed.

Note that each Repeat directive must be matched with the ENDM directive to terminate the repeat block.

REPEAT

Repeats block of statements between REPT and ENDM <exp> times.

Syntax

```
REPT exp
.
.
.
ENDM
```

Where

exp — is evaluated as a 16-bit unsigned number. If "exp" contains an External symbol or undefined operands, an error is generated.

Example

```
x       =       0
        rept    10
x       =       x + 1
        db      x
        endm
```

This example repeats the equals sign (=) and DB directives ten times. The resulting statements create ten bytes of data whose values range from 1 to 10.

IRP (Indefinite Repeat)

Syntax

```
IRP dummy , <parameters>
.
.
.
ENDM
```

Remarks

Parameters must be enclosed in angle brackets. Parameters may be any legal symbol, string, numeric, or character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of "dummy" in the block. If a parameter is null (i.e., <>), the block is processed once with a null parameter.

Example

```
    irp     x,<0,1,2,3,4,5,6,7,8,9>
            db 10 dup(x)
    endm
```

This example generates the same bytes (DB 1 to DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code

as above, illustrates the removal of one level of brackets from the parameters:

```
    alloc    macro    x
             irp      y,<x>
             db       y
             endm
             endm
```

When the macro call statement

```
    alloc <0,1,2,3,4,5,6,7,8,9>
```

is assembled, the macro expansion becomes:

```
    irp     y,<0,1,2,3,4,5,6,7,8,9>
    db      y
    endm
```

The angle brackets around the parameters will be removed, and all items are passed as a single parameter.

IRPC (Indefinite Repeat Character)

Syntax

```
IRPC dummy , string
.
.
.
ENDM
```

Remarks

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of "dummy" in the block.

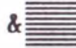
Example

```
    irpc    x,0123456789
            db    x + 1
    endm
```

This example generates the same effect as the previous two examples, repeating the db directive ten times, once for each character in the "0123456789" string. The resulting statements produce ten bytes of data, having the values 1 to 10.

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.



Syntax

&

The ampersand concatenates text or symbols. (The ampersand may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by an ampersand. To form a symbol from text and a dummy, put an ampersand between them.

For example:

```
    errgen     macro   y,x
    error&x    db      'Error &y - &x'
               endm
```

In this example, MASM replaces "&x" with the value of the actual parameter passed to errgen. If the macro is called with the statement

```
    errgen   1,wait
```

the macro is expanded to

```
    errorwait   db   'Error 1 - Wait'
```

In Macro Assembler, the ampersand will not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, extra ampersands may be needed. You need to supply as many ampersands as there are levels of nesting.

In the following macro definition the substitute operator is used twice with "z" to ensure that its replacement occurs while the IRP directive is being processed:

```
    alloc   macro          x
```

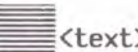
```
        irp     z,<1,2,3>
x&&z    db      z
endm
```

The dummy parameter "x" is here replaced immediately the macro is called. The dummy parameter "z", however, is not replaced until the IRP directive is processed. This means that the parameter is replaced once for each number in the IRP parameter list. If the macro is called with

```
alloc   var
```

the expanded macro will be

```
var1   db   1
var2   db   2
var3   db   3
```



<text>

Syntax

<text>

Angle brackets cause MASM to treat the text between the angle brackets as a single literal. Placing parameters to a macro call inside angle brackets; or placing the list of parameters following the IRP directive inside angle brackets causes two results:

- All text within the angle brackets is seen as a single parameter, even if commas are used.

- Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows.

One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets around parameters as there are levels of nesting.

;;

Syntax

```
;;text
```

In a macro or repeat block, a comment preceded by two semicolons is not saved as a part of the expansion.

The default listing condition for macros is .XALL (see Chapter 7, "File and Listing Control Directives"). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments inside macro and repeat blocks are saved and listed. This can be the cause of an "out of memory error." To avoid this error, place double semicolons before comments inside macro and repeat blocks, unless you specifically want a comment to be retained.

!

Syntax

```
!character
```

An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

%

Syntax

```
%text
```

The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference, with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must evaluate to an absolute (non-relocatable) constant.

Example

```
printe   macro   msg,n
         %out    *msg,n*
         endm

syml     equ     100
sym2     equ     200
         printe  <syml + sym2 =>,%(syml + sym2)
```

In this example, the macro call

```
printe <syml + sym2 =>,%(syml + sym2)
```

passes the text literal "syml + sym2" to the dummy parameter "msg". It passes the value 300 (the result of the expression "syml + sym2") to the dummy "n".

7. LISTING DIRECTIVES

ABOUT THIS CHAPTER

This chapter describes the File Control and Listing Control Directives, which are used to control the format and contents of source and object files, and listings that the assembler produces.

CONTENTS

INTRODUCTION	7-1
INCLUDE	7-1
.RADIX	7-2
%OUT	7-3
NAME	7-3
TITLE	7-4
SUBTITLE	7-5
PAGE	7-6
.LIST AND .XLIST	7-7
.LFCOND	7-8
.SFCOND	7-8
.TFCOND	7-9
.LALL	7-9
.SALL	7-10
.XALL	7-10
.CREF AND .XCREF	7-11

INTRODUCTION

These are the directives described in this chapter:

DIRECTIVE	MEANING
INCLUDE	Include a source file
.RADIX	Change default input radix
%OUT	Display message on console
NAME	Copy name to object file
TITLE	Set program-listing title
SUBTTL	Set program-listing subtitle
PAGE	Set program-listing page size and line width
.LIST	List statements in program listing
.XLIST	Suppress listing of statements
.LFCOND	List false conditional in program listing
.SFCOND	Suppress false-conditional listing
.TFCOND	Toggle false-conditional listing
.LALL	Include macro expansions in program listing
.SALL	Suppress listing of macro expansions
.XALL	Exclude comments from macro listing
.CREF	List symbols in cross-reference file
.XCREF	Suppress symbol listing

Directives in this group are used to control the format and, to some extent, the content of source and object files, and listings that the assembler produces. Format control directives: allow the use of a different default input radix for numbers; and allow the programmer to insert page breaks and direct page headings. Content control directives allow inclusion or suppression of various content types, such as macro expansions, from files and listings. Listing directives turn on and off the listing of all or part of the assembled file.

INCLUDE

The INCLUDE directive inserts source code from an alternate assembly language source file into the current source file during assembly.

Syntax

```
INCLUDE filename
```

Use of the INCLUDE directive eliminates the need to repeat an often-used sequence of statements in the current source file.

The "filename" is any valid file specification for the operating system. If the device designation is other than the default, the source filename specification must include it. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the INCLUDE directive statement. When end-of-file is reached, assembly resumes with the next statement following the INCLUDE directive.

Nested INCLUDES are allowed (the file inserted with an INCLUDE statement may contain an INCLUDE directive). However, this is not a recommended practice with small systems because of the amount of memory that may be required.

The file specified must exist. If the file is not found, an error is displayed, and the assembly aborts.

On a Macro Assembler listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See Section "Formats of Listings and Symbol Tables," for a description of listing file formats.

Example

```
include entry
include include/record
include /usr/include/as/stdio
```

.RADIX

The RADIX directive permits you to change the input radix to any base in the range 2 to 16.

Syntax

```
.RADIX exp
```

The default input base (or radix) for all constants is decimal.

"Exp" is always in decimal radix, regardless of the current input radix.

Examples

```
.radix  16

.radix  2
```

%OUT

The text is listed on the terminal during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

Syntax

```
%OUT text
```

Remarks

%OUT will output on both passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See Section "Conditional Directives," for descriptions of the IF1 and IF2 directives.

Example

```
if1
     %out first pass -- okay
endif
```

NAME

Declares the name of a module

Syntax

NAME module_name

Where

module-name must not be a reserved word. The module name may be any length, but Macro Assembler uses only the first six characters and truncates the rest.

Remarks

The module name is passed to ld, the XENIX link editor, but otherwise has no significance for the assembler. MASM does check to see if more than one module name has been declared.

Every module has a name. MASM derives the module name from:

- A valid NAME directive statement

- If the module does not contain a NAME statement, MASM uses the first six characters of a TITLE directive statement. The first six characters must be legal as a name. If no TITLE directive is found, the default name "A" is used.

Example

```
    name  main
```

TITLE

TITLE specifies a title to be listed on the first line of each page.

Syntax

TITLE text

Where

text may be up to 60 characters long. If more than one TITLE is given, an error results. The first six characters of the title, if legal, are used as the module name, unless a NAME directive is used.

Example

```
    title   progl -- 1st program
```

If the NAME directive is not used, the module name is now "progl -- 1st program". This title text will appear at the top of every page of the listing.

SUBTITLE

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title.

Syntax

```
SUBTTL text
```

text is truncated after 60 characters.

Remarks

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the "text" from the previous SUBTTL with the "text" from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for "text".

Example

```
    subttll   SPECIAL I/O ROUTINE
```

This example creates a subtitle "SPECIAL I/O ROUTINE". The next example creates a blank subtitle:

```
    subttl
```

PAGE

This directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break by putting a form-feed character in the listing file at the end of the page.

Syntax

```
PAGE [length][, width]

or

PAGE [+]
```

Remarks

The PAGE directive with either the length or width argument does not start a new listing page.

The value of "length", if included, becomes the new page length (measured in lines per page) and must be in the range 10 to 255. The default page length is 50 lines per page.

The value of "width", if included, becomes the new page width (measured in characters) and must be in the range 60 to 132. The default page width is 80 characters.

The plus sign (+) increments the major page number and resets the minor page number to one. Page numbers are in the form major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Examples

```
page
```

This example creates a page-break. The next example,

```
page  58,60
```

sets the maximum page length to 58 lines, and the maximum width to 60 characters. The third example:

```
page  ,132
```

sets the maximum width to 132 characters. The curent page length remains unaltered. The fourth example:

```
page +
```

increments the current section number and sets the page number to 1.

.LIST and .XLIST

The .LIST directive lists all lines with their code (the default condition). .XLIST suppresses all listing.

Syntax

.LIST

and

.XLIST

Remarks

If you specify a listing file following the Listing: prompt, a listing file with all the source statements included will be printed.

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

The .XLIST directive overrides all other listing directives. Nothing will be listed, even if another listing directive (other than .LIST) is encountered.

Example

```
.xlist    ;  listing suspended here
.
.
.
.list     ;  listing resumed here
```

.LFCOND

Syntax

```
.LFCOND
```

Remarks

The .LFCOND directive assures the listing of conditional expressions that evaluate false. This is the default condition.

.SFCOND

Syntax

```
.SFCOND
```

Remarks

The .SFCOND directive suppresses portions of the listing that contain conditional false expressions.

Example

```
.sfcond
if 0
.
.              ; This block will not be listed
.
endif
.lfcond
if 0
.
.              ; This block will be listed
.
endif
```

.TFCOND

Syntax

```
.TFCOND
```

Remarks

The .TFCOND directive toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the -X option when the assembler is running. When -X is used, .TFCOND will cause false conditionals to list. When -X is not used, .TFCOND will suppress false conditionals.

.LALL

Syntax

```
.LALL
```

Remarks

The .LALL directive lists the complete macro text for all expansions, including lines that do not generate code. Comments preceded by two semicolons ";;" will not be listed.

Examples

```
.sall  ;  No macros listed here

.lall  ;  Macros listed in full
```

.SALL

Syntax

```
.SALL
```

Remarks

The .SALL directive suppresses listing of all text and object code produced by macros.

.XALL

Syntax

```
.XALL
```

Remarks

The .XALL directive lists source code and object code produced by a macro, but source lines which do not generate code are not listed. .XALL is the default.

Example

```
    .xall    ;  Macros listed by generated code or data only
```

.CREF and .XCREF

Syntax

```
.CREF

and

.XCREF [variable list]
```

Characteristics

The .CREF directive is the default condition. .CREF remains in effect until Macro Assembler encounters .XCREF.

The .XCREF directive without arguments turns off the .CREF (default) directive. .XCREF remains in effect until Macro Assembler encounters .CREF. Use .XCREF to suppress the creation of cross-references in selected portions of the file. Use .CREF to restart the creation of a cross-reference file after using the .XCREF directive.

If you include one or more variables following .XCREF, these variables will not be placed in the listing or cross-reference file. All other cross-referencing, however, is not affected by an .XCREF directive with arguments. Separate the variables with commas.

Neither .CREF nor .XCREF without arguments takes effect unless you specify a cross-reference file when running the assembler. ".XCREF variable list" suppresses the variables from the symbol table listing regardless of the creation of a cross-reference file.

Example

```
.xcref one,two,three
```

A. INTRODUCTION

ABOUT THIS APPENDIX

This appendix is a list of the error messages returned by the MASM Macro Assembler and the ld link editor.

CONTENTS

MACRO ASSEMBLER MESSAGES A-1

LINKER MESSAGES A-7

INTRODUCTION

MASM is an assembler for the 8086/186/286 family of microprocessors, capable of assembling instructions for the 8086, 186, and 286 microprocessors and the 8087 and 287 floating point coprocessors. MASM will assemble any program written for an 8086, 186, or 286 microprocessor environment as long as the program uses the instruction syntax described in this chapter.

By default, MASM recognizes 8086 and 8087 instructions only. If a source program contains 186, 286, or 287 instructions, one or more Instruction Set directives must be used in the source file to enable assembly of the instructions. The following sections list the syntax of all instructions recognized by MASM and the Instruction Set directives.

Abbreviations used in the syntax descriptions are:

SYMBOL	MEANING
accum	accumulator: AX or AL
reg	byte or word register byte: AL, AH, BL, BH, CL, CH, DL, DH word: AX, BX, CX, DX, SI, DI, BP, SP
segreg	segment register: CS, DS, SS, ES
r/m	general operand: register, memory address, indexed operand, based operand, or based indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	memory operand: label, variable, or symbol
label	instruction label

8086 INSTRUCTIONS

The following is a complete list of the 8086 instructions. MASM assembles all 8086 instructions by default.

SYMBOL	ACTION
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC accum, immed	Add immediate with carry to accumulator
ADC r/m, immed	Add immediate with carry to operand
ADC r/m, reg	Add register with carry to operand
ADC reg, r/m	Add operand with carry to register
ADD accum, immed	Add immediate to accumulator
ADD r/m, immed	Add immediate to operand
ADD r/m, reg	Add register to operand
ADD reg, r/m	Add operand to register
AND accum, immed	Bitwise AND immediate with accumulator
AND r/m, immed	Bitwise AND immediate with operand
AND r/m, reg	Bitwise AND register with operand
AND reg, r/m	Bitwise AND operand with register
CALL label	Call instruction at label

CALL r/m	Call instruction indirect
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP accum, immed	Compare immediate with accumulator
CMP r/m, immed	Compare immediate with operand
CMP r/m, reg	Compare register with operand
CMP reg, r/m	Compare operand with register
CMPS src, dest	Compare strings
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC r/m	Decrement operand
DEC reg	Decrement 16-bit register
DIV r/m	Divide accumulator by operand
ESC immed, r/m	Escape with 6-bit immediate and operand
HLT	Halt
IDIV r/m	Integer divide accumulator by operand
IMUL r/m	Integer multiply accumulator by operand
IN accum, immed	Input from port (8-bit immediate)
IN accum, DX	Input from port given by DX
INC r/m	Increment operand
INC reg	Increment 16-bit register
INT 3	Software interrupt 3 (encoded as one byte)
INT immed	Software Interrupt 0 through 255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA label	Jump on above
JAE label	Jump on above or equal
JB label	Jump on below
JBE label	Jump on below or equal
JC label	Jump on carry
JCXZ label	Jump on CX zero
JE label	Jump on equal
JG label	Jump on greater
JGE label	Jump on greater or equal
JL label	Jump on less than
JLE label	Jump on less than or equal
JMP label	Jump to instruction at label
JMP r/m	Jump to instruction indirect
JNA label	Jump on not above
JNAE label	Jump on not above or equal
JNB label	Jump on not below
JNBE label	Jump on not below or equal
JNC label	Jump on no carry
JNE label	Jump on not equal
JNG label	Jump on not greater
JNGE label	Jump on not greater or equal
JNL label	Jump on not less than
JNLE label	Jump on not less than or equal
JNO label	Jump on not overflow

JNP label	Jump on not parity
JNS label	Jump on not sign
JNZ label	Jump on not zero
JO label	Jump on overflow
JP label	Jump on parity
JPE label	Jump on parity even
JPO label	Jump on parity odd
JS label	Jump on sign
JZ label	Jump on zero
LAHF	Load AH with flags
LDS r/m	Load operand into DS
LEA r/m	Load effective address of operand
LES r/m	Load operand into ES
LOCK	Lock bus
LODS src	Load string
LODSB	Load byte from string into AL
LODSW	Load word from string into AX
LOOP label	Loop
LOOPE label	Loop while equal
LOOPNE label	Loop while not equal
LOOPNZ label	Loop while not zero
LOOPZ label	Loop while zero
MOV accum, mem	Move memory to accumulator
MOV mem, accum	Move accumulator to memory
MOV r/m, immed	Move immediate to operand
MOV r/m, reg	Move register to operand
MOV r/m, segreg	Move segment register to operand
MOV reg, immed	Move immediate to register
MOV reg, r/m	Move operand to register
MOV segreg, r/m	Move operand to segment register
MOVS dest, src	Move string
MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL r/m	Multiply accumulator by operand
NEG r/m	Negate operand
NOP	No operation
NOT r/m	Invert operand bits
OR accum, immed	Bitwise OR immediate with accumulator
OR r/m, immed	Bitwise OR immediate with operand
OR r/m, reg	Bitwise OR register with operand
OR reg, r/m	Bitwise OR operand with register
OUT DX, accum	Output to port given by DX
OUT immed, accum	Output to port (8-bit immediate)
POP r/m	Pop 16-bit operand
POP reg	Pop 16-bit register from stack
POP segreg	Pop segment register
POPF	Pop flags
PUSH r/m	Push 16-bit operand
PUSH reg	Push 16-bit register onto stack
PUSH segreg	Push segment register
PUSHF	Push flags
RCL r/m, 1	Rotate left through carry by 1 bit
RCL r/m, CL	Rotate left through carry by CL
RCR r/m, 1	Rotate right through carry by 1 bit
RCR r/m, CL	Rotate right through carry by CL

REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET [immed]	Return after popping bytes from stack
ROL r/m, 1	Rotate left by 1 bit
ROL r/m, CL	Rotate left by CL
ROR r/m, 1	Rotate right by 1 bit
ROR r/m, CL	Rotate right by CL
SAHF	Store AH into flags
SAL r/m, 1	Shift arithmetic left by 1 bit
SAL r/m, CL	Shift arithmetic left by CL
SAR r/m, 1	Shift arithmetic right by 1 bit
SAR r/m, CL	Shift arithmetic right by CL
SBB accum, immed	Subtract immediate and carry flag
SBB r/m, immed	Subtract immediate and carry flag
SBB r/m, reg	Subtract register and carry flag
SBB reg, r/m	Subtract operand and carry flag
SCAS dest	Scan string
SCASB	Scan string for byte in AL
SCASW	Scan string for word in AX
SHL r/m, 1	Shift left by 1 bit
SHL r/m, CL	Shift left by CL
SHR r/m, 1	Shift right by 1 bit
SHR r/m, CL	Shift right by CL
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS dest	Store string
STOSB	Store byte in AL at string
STOSW	Store word in AX at string
SUB accum, immed	Subtract immediate from accumulator
SUB r/m, immed	Subtract immediate from operand
SUB r/m, reg	Subtract register from operand
SUB reg, r/m	Subtract operand from register
TEST accum, immed	Compare immediate bits with accumulator
TEST r/m, immed	Compare immediate bits with operand
TEST r/m, reg	Compare register bits with operand
TEST reg, r/m	Compare operand bits with register
WAIT	Wait
XCHG accum, reg	Exchange accumulator with register
XCHG r/m, reg	Exchange operand with register
XCHG reg, accum	Exchange register with accumulator
XCHG reg, r/m	Exchange register with operand
XLAT mem	Translate
XOR accum, immed	Bitwise XOR immediate with accumulator
XOR r/m, immed	Bitwise XOR immediate with operand
XOR r/m, reg	Bitwise XOR register with operand
XOR reg, r/m	Bitwise XOR operand with register

The String instructions (CMPS, LODS, MOVS, SCAS, and STOS) use the DS, SI, ES, and DI registers to compute operand locations. Source operands are assumed to be at DS:[SI]; destination operands at ES:[DI]. The operand type (BYTE or WORD) is defined by the instruction mnemonic. For example, CMPSB specifies BYTE operands and CMPSW specifies WORD operands.

For the CMPS, LODS, MOVS, SCAS, and STOS instructions, the "src" and "dest" operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The "src" operand can also be used to specify a segment override. The ES register for the destination operand cannot be overridden.

Examples

```
    cmps  word ptr string, word ptr es:0
    lods  byte ptr string
    mov   byte ptr es:0,  byte ptr string
```

The REP, REPE, REPNE, REPNZ, or REPZ instructions provide a way to repeatedly execute a String instruction for a given count or while a given condition is true. If a Repeat instruction immediately precedes a String instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false, or the CX register is equal to zero. The Repeat instruction decrements CX by one for each execution.

Example

```
    mov     cx, 10
    rep     scasb
```

In this example, SCASB is repeated ten times.

8087 INSTRUCTIONS

The following is a list of the 8087 instructions. MASM assembles all 8087 instructions by default.

SYMBOL	ACTION
F2XM1	Calculate 2uxd-1
FABS	Take absolute value of top of stack
FADD	Add real
FADD mem	Add real from memory
FADD ST, ST(i)	Add real from stack
FADD ST(i), ST	Add real to stack
FADDP ST(i), ST	Add real and pop stack
FBLD mem	Load 10-byte packed decimal on stack
FBSTP mem	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(i)	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop

FCOMP ST(i)	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIV mem	Divide real from memory
FDIV ST, ST(i)	Divide real from stack
FDIV ST(i), ST	Divide real in stack
FDIVP ST(i), ST	Divide real and pop stack
FDIVR	Reversed real divide
FDIVR mem	Reverse real divide from memory
FDIVR ST, ST(i)	Reverse real divide from stack
FDIVR ST(i), ST	Reverse real divide in stack
FDIVRP ST(i), ST	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FFREE ST	Free top of stack element
FFREE ST(i)	Free ith stack element
FIADD mem	Add 2 or 4-byte integer
FICOM mem	2 or 4-byte integer compare
FICOMP mem	2 or 4-byte integer compare and pop stack
FIDIV mem	2 or 4-byte integer divide
FIDIVR mem	Reversed 2 or 4-byte integer divide
FILD mem	Load 2, 4, or 8-byte integer on stack
FIMUL mem	2 or 4-byte integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST mem	Store 2 or 4-byte integer
FISTP mem	Store 2, 4, or 8-byte integer and pop stack
FISUB mem	2 or 4-byte integer subtract
FISUBR mem	Reversed 2 or 4-byte integer subtract
FLD mem	Load 4, 8, or 10-byte real on stack
FLD1	Load +1.0 onto top of stack
FLDCW mem	Load control word
FLDENV mem	Load 8087 environment (14-bytes)
FLDL2E	Load logd2ue onto top of stack
FLDL2T	Load logd2u10 onto top of stack
FLDLG2	Load logd10u2 onto top of stack
FLDLN2	Load logdeu2 onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
MUL mem	Multiply real from memory
FMUL ST, ST(i)	Multiply real from stack
FMUL ST(i), ST	Multiply real to stack
FMULP ST(i), ST	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor, with no WAIT
FNOP	No operation
FNSAVE mem	Save 8087 state (94 bytes) with no WAIT
FNSTCW mem	Store control word with no WAIT
FNSTENV mem	Store 8087 environment with no WAIT
FNSTSW mem	Store 8087 status word with no WAIT

FPATAN	Partial arctangent function
FPREM	Partial remainder
FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR mem	Restore 8087 state (94 bytes)
FSAVE mem	Save 8087 state (94 bytes) after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FST ST	Store real from top of stack
FST ST(i)	Store real from stack
FSTCW mem	Store control word with WAIT
FSTENV mem	Store 8087 environment after WAIT
FSTP mem	Store 4, 8, or 10-byte real and pop stack
FSTSW mem	Store 8087 status word after WAIT
FSUB	Subtract real
FSUB mem	Subtract real from memory
FSUB ST, ST(i)	Subtract real from stack
FSUB ST(i), ST	Subtract real to stack
FSUBP ST(i), ST	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBR mem	Reversed real subtract from memory
FSUBR ST, ST(i)	Reversed real subtract from stack
FSUBR ST(i), ST	Reversed real subtract in stack
FSUBRP ST(i), ST	Reversed real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack elements
FFREE ST	Exchange top of stack element
FFREE ST(i)	Exchange top of stack and *i*th element
FXTRACT	Extract exponent and significand
FYL2X	Calculate Y logd2ux
FYL2PI	Calculate Y logd2u(x+1)

186 INSTRUCTIONS

The 186 instruction set consists of all 8086 instructions plus the following instructions. The .186 directive can be used to enable these instructions for assembly.

SYMBOL	ACTION
BOUND reg, mem	Detect value out of range
ENTER immed16, immed8	Enter procedure
IMUL immed, reg	Integer multiply immediate byte into word register
IMUL r/m, immed	Integer multiply operand by immediate wird/byte
INS mem, DX	Input string from port DX
INSB mem, DX	Input byte string from port DX
INSW mem, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, mem	Output byte/word/string to port DX
OUTSB DX, mem	Output byte string to port DX
OUTSW DX, mem	Output word string to port DX

POPA	Pop all registers
PUSH immed	Push all immediate word/byte
PUSHA	Push all registers
RCL r/m, immed	Rotate left through carry immediate
RCR r/m, immed	Rotate right through carry immediate
ROL r/m, immed	Rotate left immediate
ROR r/m, immed	Rotate right immediate
SAL r/m, immed	Shift arithmetic left immediate
SAR r/m, immed	Shift arithmetic right immediate
SHL r/m, immed	Shift left immediate
SHR r/m, immed	Shift right immediate

286 NON-PROTECTED INSTRUCTIONS

The 286 non-protected instruction set consists of all 8086 instructions plus the following instructions. The .286c directive can be used to enable these instructions for assembly.

SYMBOL	ACTION
BOUND reg, mem	Detect value out of range
ENTER immed16, immed8	Enter procedure
IMUL immed, reg	Integer multiply immediate byte into word register
IMUL r/m, immed	Integer multiply operand by immediate word/byte
INS mem, DX	Input string from port DX
INSB mem, DX	Input byte string from port DX
INSW mem, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, mem	Output byte/word/string to port DX
OUTSB DX, mem	Output byte string to port DX
OUTSW DX, mem	Output word string to port DX
POPA	Pop all registers
PUSH immed	Push immediate word/byte
PUSHA	Push all registers
RCL r/m, immed	Rotate left through carry immediate
RCR r/m, immed	Rotate right through carry immediate
ROL r/m, immed	Rotate left immediate
ROR r/m, immed	Rotate right immediate
SAL r/m, immed	Shift arithmetic left immediate
SAR r/m, immed	Shift arithmetic right immediate
SHL r/m, immed	Shift left immediate
SHR r/m, immed	Shift right immediate

286 PROTECTED INSTRUCTIONS

The 286 protected instruction set consists of all 8086 and 286 non-protected instructions plus the following instructions. The .286p directive can be used to enable these instructions for assembly.

SYMBOL	ACTION
ARPL mem, reg	Adjust requested privilege level
CLTS	Clear task switched flag

LAR reg, mem	Load access rights
LGDT mem	Load global descriptor table (8 bytes)
LIDT mem	Load interrupt descriptor table (8 bytes)
LLDT mem	Load local descriptor table
LMSW mem	Load machine status word
LSL reg, mem	Load segment limit
LTR mem	Load task register
SGDT mem	Store global descriptor table (8 bytes)
SIDT mem	Store interrupt descriptor table (8 bytes)
SLDT mem	Store local descriptor table
SMSW mem	Store machine status word
STR mem	Store task register
VERR mem	Verify read access
VERW mem	Verify write access

287 INSTRUCTIONS

The 287 instruction set consists of all 8087 instructions plus the following additional instructions. The .287 directive can be used to enable these instructions for assembly.

SYMBOL	ACTION
FSETPM	Set Protected Mode
FSTSW AX	Store Status Word in AX (wait)
FNSTSW AX	Store Status Word in AX (no-wait)

B. DIRECTIVE AND OPERATOR SUMMARY

ABOUT THIS APPENDIX

This appendix lists all the MASM directives and all the MASM Operators.

CONTENTS

INTRODUCTION	B-1
DIRECTIVES	B-1
OPERATORS	B-4

DIRECTIVE AND OPERATOR SUMMARY

INTRODUCTION

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. There are the following directives:

.186	ELSE	IFDIF	PROC
.286c	END	IFE	PUBLIC
.286p	ENDIF	IFIDN	.RADIX
.287	ENDP	IFNB	RECORD
.8086	ENDS	IFNDEF	.SALL
.8087	EQU	INCLUDE	SEGMENT
=	EVEN	LABEL	.SFCOND
ASSUME	EXTRN	.LALL	STRUC
COMMENT	GROUP	.LFCOND	SUBTTL
.CREF	IF	.LIST	.TFCOND
DB	IF1	NAME	TITLE
DD	IF2	ORG	.XALL
DQ	IFB	%OUT	.XCREF
DT	IFDEF	PAGE	.XLIST
DW			

Any combination of upper and lowercase letters can be used when giving directive names in a source file.

DIRECTIVES

The following is a complete list of directive syntax and function:

.186 — Enables assembly of 186 instructions.

.286c — Enables assembly of 286 unprotected instructions.

.286p — Enables assembly of 286 protected instructions.

.287 — Enables assembly of 287 instructions.

.8086 — Enables assembly of 8086 instructions while disabling assembly of 186 and 286 instructions.

.8087 — Enables assembly of 8087 instructions while disabling assembly of 287 instructions.

name = expression — Assigns the numeric value of "expression" to "name".

ASSUME seg-reg : seg-name ,,, — Selects the given segment register "seg-reg" to be the default segment register for all symbols in the named segment or group. If "seg-name" is

	NOTHING, no register is selected.
COMMENT delim text delim	Treats all "text" between the given pair of delimiters, "delim", as a comment.
.CREF	Restores listing of symbols in the cross-reference listing file.
[name] DB initial-value ,,,	Allocates and initializes a byte (8 bits) of storage for each "initial-value".
[name] DW initial-value ,,,	Allocates and initializes a word (2 bytes) of storage for each given "initial-value".
[name] DD initial-value ,,,	Allocates and initializes a doubleword (4 bytes) of storage for each given "initial-value".
[name] DQ initial-value ,,,	Allocates and·initializes a quadword (8 bytes) of storage for each given "initial-value".
[name] DT initial-value ,,,	Allocates and initializes 10 bytes of storage for each given "initial-value".
ELSE	Marks the beginning of an alternate block within a conditional block.
END [expression]	Marks the end of the module and optionally sets the program entry point to "expression".
ENDIF	Terminates a conditional block.
name EQU expression	Assigns the "expression" to the given "name".
name ENDP	Marks the end of a procedure definition.
name ENDS	Marks the end of a segment or structure type definition.
EVEN	If necessary, increments the location counter to an even value and generates one NOP instruction (90h)
EXTRN name : type ,,,	Defines an external variable, label, or symbol named "name" and whose type is "type".
name GROUP seg-name ,,,	Associates a group name. "name", with one or more segments.

Directive	Description
IF expression	Grants assembly if the "expression" is non-zero (true).
IF1	Grants assembly on pass 1 only.
IF2	Grants assembly on pass 2 only.
IFB < arg >	Grants assembly if the "arg" is blank.
IFDEF name	Grants assembly if "name" is a previously defined label, variable, or symbol.
IFDIF < arg1 >, < arg2 >	Grants assembly if the arguments are different.
IFE expression	Grants assembly if the "expression" is 0 (false).
IFIDN < arg1 >, < arg2 >	Grants assembly if the arguments are identical.
IFNB < arg >	Grants assembly if the "arg" is not blank.
IFNDEF name	Grants assembly if "name" has not yet been defined.
INCLUDE filename	Inserts source code from the source file given by "filename" into the current source file during assembly.
name LABEL type	Creates a new variable or label by assigning the current location counter value and the given "type" to "name".
.LALL	Lists all statements in a macro.
.LFCOND	Restores the listing of conditional blocks.
.LIST	Restores listing of statements in the program listing.
NAME module-name	Sets the name of the current module to "module-name".
ORG expression	Sets the location counter to "expression".
%OUT text	Displays "text" at the user's terminal.
name PROC type	Marks the beginning of a procedure definition.

PUBLIC name ,,,	Makes the variable, label, or absolute symbol given by "name" available to all other modules in the program.
.RADIX expression	Sets the input radix for numbers in the source file to "expression".
recordname RECORD fieldname : width [= exp]	
	Defines an record type for a 8- or 16-bit record that contains one or more fields.
.SALL	Suppresses listing of all macro expansions.
name SEGMENT align combine	
	Marks the beginning of a program segment named "name" and having segment attributes "align", "combine", and "class".
.SFCOND	Suppresses listing of any subsequent conditional blocks whose IF condition is false.
name STRUC	Marks the beginning of a type definition for a structure.
PAGE length , width	Sets the line length and character width of the program listing.
PAGE +	Increments section page numbering.
PAGE	Generates a page break in the listing.
SUBTTL text	Defines the listing subtitle.
.TFCOND	Sets the default mode for listing of conditional blocks.
TITLE text	Defines the program listing title.
.XALL	Lists only those macro statements that generate code or data.
.XCREF name ,,,	Suppresses the listing of symbols in the cross-reference listing file.
.XLIST	Suppresses listing of subsequent source lines to the program listing.

OPERATORS

The operators recognized by MASM are listed by precedence in the following table. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order can be overridden using enclosing parentheses.

PRECEDENCE	OPERATORS
(Highest)	
1	LENGTH, SIZE, WIDTH, MASK
2	( )
3	[ ]
4	:
5	PTR, OFFSET, SET, TYPE, THIS
6	HIGH, LOW
7	*, /, MOD, SHL, SHR
8	+, - (binary)
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
(Lowest)	

The syntax of each operator is shown in the following list:

expression1 * expression 2	Multiply expression1 by expression2.
expression1/expression2	Divide expression1 by expression2.
expression1 + expression2	Add expression1 to expression2.
expression1 - expression2	Subtract expression2 from expression1.
+expression	Retain the current sign of expression.
-expression	Reverse the sign of expression.
segmentregister:expression	Override the default segment of expression with segmentregister.
segmentname:expression	Override the default segment of expression with segmentname.
groupname:expression	Override the default segment of expression with groupname.
variable.field	Add the offset to field to the offset of variable.
expression1[expression2]	Add the value of expression1 to the value of expression2.
&dummyparameter	Replace dummyparameter with its actual parameter value.

dummyparameter&	Replace dummyparameter with its actual parameter value.
<text>	Treat text as a single literal element.
!character	Treat character as a literal character rather than as an operator or symbol.
%text	Treat text as an expression and compute its value rather than treating it as a string.
;;test	Make text into a comment that will not be listed in expanded macros.
expression1 AND expression2	Do a bitwise Boolean AND on expression1 and expression2.
count DUP (initialvalue)	Specify count number of declarations of initialvalue.
expression1 EQ expression2	Return true (0FFFFh) if expression1 equals expression2, or return false (0) if it does not.
expression1 GE expression2	Return true (0FFFFh) if expression1 is greater than or equal to expression2, or return false (0) if it is not.
expression1 GT expression2	Return true (0FFFFh) if expression1 is greater than expression2, or return false (0) if it is not.
HIGH expression	Return the high byte of expression.
expression1 LE expression2	Return true (0FFFFh) if expression1 is less than or equal to expression2, or return false (0) if it is not.
LENGTH variable	Return the length of variable in the size in which the variable was declared.
LOW expression	Return the low byte of expression.
expression1 LT expression2	Return true (0FFFFh) if expression1 is less than expression2, or return false (0) if it is not.
MASK recordfieldname	Return a bit mask in which the bits for recordfieldname are set and all other bits are not set.
MASK record	Return a bit mask in which the bits used in record are set and all other bits are not set.

expression1 MOD expression2	Return the remainder of dividing expression1 by expression2.
expression1 NE expression2	Return true (OFFFFh) if expression1 does not equal expression2, or return false (0) if it does.
NOT expression	Reverse all bits of expression.
OFFSET expression	Return the offset of expression.
expression1 OR expression2	Do a bitwise Boolean OR on expression1 and expression2.
type PTR expression	Force the expression to be treated as havig the specified type.
SEG expression	Return the segment of expression.
expression SHL count	Shift the bits of expression left count number of bits.
SHORT label	Set type of label to short (having a distance less than 128 bytes from the current location-counter value).
expression SHR count	Shift the bits of expression right count number of bits.
SIZE variable	Return the total number of bytes allocated for variable.
THIS type	Create an operand of spelcified type whose offset and segment values are equal to the current location-counter value.
TYPE expression	Return the type of expression.
.TYPE expression	Return a byte defining the mode and scope of expression.
WIDTH recordfieldname	Return the width in bits of the current recordfieldname.
WIDTH record	Return the width in bits of the current record.
expression1 XOR expression2	Do a bitwise Boolean XOR on expression1 and expression2.

C. SEGMENT NAMES FOR HIGH-LEVEL LANGUAGES

ABOUT THIS CHAPTER

This appendix describes the naming conventions used to form assembly language source files that are compatible with object modules produced by the cc C language compiler and other high-level language compilers.

CONTENTS

INTRODUCTION	C-1
TEXT SEGMENTS	C-1
SMALL MODEL PROGRAMS	C-2
MIDDLE AND LARGE MODEL PROGRAMS	C-2
DATA SEGMENTS - NEAR	C-3
DATA SEGMENTS - FAR	C-4
BSS SEGMENTS	C-5
CONSTANT SEGMENTS	C-6

SEGMENT NAMES FOR HIGH-LEVEL LANGUAGES

INTRODUCTION

This appendix describes the naming conventions used to form assembly language source files that are compatible with object modules produced by the cc C language compiler and otherhigh-level language compilers.

High-level language modules have the following four predefined segment types:

TEXT	for program code
DATA	for program data
BSS	for uninitialized space
CONST	for constant data

Any assembly language source file that is to be assembled and linked to a high-level language module must use these segments as described in the following sections.

High-level language modules also have three different memory models:

Small	for single code and data segments
Middle	for multiple code segment but a single data segment
Large	for multiple code and data segments

Assembly language source files to be assembled for a given memory model must use the naming conventions given in the following sections.

TEXT SEGMENTS

Syntax

```
name_text  segment byte public 'CODE'
           statements
           .
           .
           .
name_text  ends
```

A text segment defines a module's program code. It contains "statements" that define instructions and data within the segment. A text segment must have the name "name_text" where "name" can be any valid name. For middle and large module programs, the module's own name is recommended. For small model programs, only TEXT is allowed.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the CS segment register. Therefore, the statement

```
    assume cs:  name_text
```

must appear at the beginning of the segment. This statement ensures that each label and variable declared in the segment will be associated with the CS segment register (see the section, ASSUME Directive in Chapter 4).

Text segments should have BYTE alignment and PUBLIC combination type, and must have the class name CODE. These define loading instructions that are passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see the section, "SEGMENT AND ENDS" in Chapter 4.

Small Model Programs

Only one text segment is allowed. The segment must not exceed 64 Kbytes.

Example

```
_text   segment byte public 'CODE'
        assume  cs:_text
_main   proc near
        .
        .
        .
_main   endp
_text   ends
```

Middle and Large Model Programs

Multiple text segments are allowed, however, no segment can be greater than 64 Kbytes. To distinguish one segment from another, each should have its own name. Since most modules contain only one text segment, the module's name is often used as part of the text segment's name. All procedure and statement labels should have the FAR type, unless they will only be accessed from within the same segment.

Example

```
sample_text     segment byte public 'CODE'
        assume  cs:sample_text
_main   proc    far
        .
        .
        .
_main   endp
sample_text     ends
```

DATA SEGMENTS - NEAR

Syntax

```
_DATA SEGMENT WORD PUBLIC 'DATA'
statements
.
.
.
_DATA ENDS
```

A near data segment defines initialized data that is in the segment pointed to by the DS segment register when the program starts execution. The segment is NEAR because all data in the segment is accessible without giving an explicit segment value. All programs have exactly one near data segment. Only large model programs can have additional data segments.

A near data segment's name must be "_DATA". The segment can contain any combination of data statements defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the predefined group DGROUP Therefore, the statements

```
DGROUP  group _data
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the data segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME" and "GROUP" in Chapter 4).

Near data segments must be WORD aligned, must have PUBLIC combination type, and must have the class name DATA. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section "SEGMENT and ENDS" in Chapter 4.

Example

```
DGROUP  group    _data
        assume   ds: DGROUP

_data   segment word public 'DATA'
count   dw       0
array   dw       10 dup(1)
string  db       "Type CANCEL then press RETURN", 0ah, 0
_data   ends
```

DATA SEGMENTS - FAR

Syntax

```
name_DATA  SEGMENT WORD PUBLIC 'FAR_DATA'
statements
.
.
.
name_DATA  ENDS
```

A far data segment defines data or data space that can be accessed only by specifying an explicit segment value. Only large model programs can have far data segments.

A far data segment's name must be "name_DATA" where "name" can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data statements defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the ES segment register. When accessing a variable in a far data segment, the ES register must be set to the appropriate segment value. Also, the segment override operator must be used with the variable's name.

Far data segments must be WORD aligned, must have PUBLIC combination type, and should have the class name FAR_DATA. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section "SEGMENT AND ENDS" in Chapter 4.

Example

```
array_data        segment word public 'FAR_DATA'
array    dw       0
         dw       1
         dw       2
         dw       4
table    dw       1600 dup(?)
array_data        ends
```

BSS SEGMENTS

Syntax

```
_BSS  SEGMENT WORD PUBLIC 'BSS'
statements
.
.
.
_BSS  ENDS
```

A BSS segment defines uninitialized data space. A BSS segment's name must be _BSS. The segment can contain any combination of data statements defining variables to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group DGROUP. Therefore, the statements

```
    DGROUP  group _BSS
            assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the bss segment will be associated with the DS segment register and DGROUP (see the sections, ASSUME Directive and GROUP Directive in Chapter 3).

The group name DGROUP must not be defined in more than one GROUP directive in a source file. If a source file contains both a DATA and BSS segment, the directive

```
    DGROUP  group  _data, _BSS
```

should be used.

A bss segment must be WORD aligned, must have PUBLIC combination type, and must have the class name BSS. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, SEGMENT and ENDS Directives, in Chapter 3.

Example

```
    DGROUP  group   _BSS
            assume ds: DGROUP

    _BSS    segment word public 'BSS'
    count   dw      ?
    array   dw      10 dup(?)
    string  db      30 dup(?)
    _BSS    ends
```

CONSTANT SEGMENTS

Syntax

```
CONST  SEGMENT WORD PUBLIC 'CONST'
statements
.
.
.
CONST  ENDS
```

A constant segment defines constant data that will not change during program execution. Constant segments are typically used in large model programs to hold the segment values of far data segments.

The constant segment's name must be CONST. The segment can contain any combination of data statements defining constants to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group DGROUP. Therefore, the statements

```
DGROUP  group  const
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the constant segment will be associated with the DS segment register and DGROUP (see the sections, ASSUME Directive and GROUP Directive in Chapter 3).

The group name DGROUP must not be defined in more than one GROUP directive in a source file. If a source file contains a DATA, BSS, and CONST segment, the directive

```
DGROUP  group  _data, _BSS, const
```

should be used.

A constant segment must be WORD aligned, must have PUBLIC combination type, and must have the class name CONST. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section "SEGMENT AND ENDS" in Chapter 4.

Example

```
DGROUP  group   const
        assume  ds: DGROUP
```

```
const   segment word public 'CONST'
seg1    dw      array_data
seg2    dw      message_data
const   ends
```

In this example, the constant segment receives the segment values of two far data segments: ARRAY_DATA and MESSAGE_DATA. These data segments must be defined elsewhere in the module.

NOTICE

Ing. C. Olivetti & C., S.p.A. reserves the right to make any changes in the product described in this manual at any time and without notice.

This manual is licensed to the Customer under the conditions contained in the User License enclosed with the Program to which the manual refers.

Code 4022950 Z (1)
Printed in Italy

olivetti