Microsoft®
Macro Assembler

for the MS-DOS® Operating System

Reference Manual

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410610002-400-R00-1285

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Overview

This reference manual describes the syntax and structure of assembly language for **MASM**, the Microsoft® Macro Assembler. **MASM** is an assembler for the Intel® 8086/80186/80286 family of microprocessors. It can assemble the instructions of 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 **LINK**, the Microsoft 8086 Object Linker, to create executable programs for the MS-DOS® operating system.

MASM is a macro assembler. It has a full set of macro directives that let you create and use macros in a source file. The directives instruct **MASM** to repeat common blocks of statements, or replace macro names with the blocks of statements they represent. **MASM** also has conditional directives that provide for selective exclusion of portions of a source file from assembly, or inclusion of 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 could lead to errors or unwanted results.

MASM produces object modules compatible with object modules created by many high-level-language compilers. Thus, programs can be constructed by combining **MASM** object modules with object modules created by C, Pascal, FORTRAN, or other language compilers.

1.2 About This Manual

This reference manual supplements the *Microsoft Macro Assembler User's Guide,* which explains program operation and the steps required to create executable programs from source files.

This reference manual does not teach assembly-language programming, nor does it give detailed descriptions of the 8086, 80186, and 80286 instruction sets. For further information on these topics, other references are available. Some of these are listed in the introduction to the *Microsoft Macro Assembler User's Guide.*

Chapter 1 concludes with an explanation of notational conventions used throughout the *Microsoft Macro Assembler Reference Manual.* Chapter 2 discusses the elements of the assembler, reserved words, characters that can be used in a program, and how to form numbers, names, statements and comments compatible with the assembler. Chapter 3 details the program-structure directives, which allow definition of code and data organization, and the instruction-set directives used for specifying which instruction set or sets will be used during assembly. Chapter 4 explains generating data for programs, declaration of labels, variables and other symbols, and type definition for data blocks. Chapter 5 deals with combining operators and operands into expressions for assembly-language statements and directives. Chapter 6 covers the global-declaration directives that allow transformation of local symbols into global symbols available to all program modules. Chapters 7 and 8 discuss the uses of, and relationship between, conditional-assembly directives and macro directives. Chapter 9 explains the file-control directives and how to use them to control source files and the files read and created by **MASM** during assembly.

Appendix A provides a list of the instruction names and syntax for the 8086/80186/80286 family of processors. For quick reference, the Microsoft Macro Assembler package also includes a copy of Intel Corporation's *8086/8088/8087/80186/80188 Programmer's Pocket Reference Guide.* Appendix B lists the directives you can use in **MASM** source files, while Appendix C gives some guidance on linking **MASM** object files to object files from high-level-language compilers.

1.3 Notational Conventions

This manual uses the following notational conventions in defining assembly-language syntax, and in presenting examples:

Convention	Meaning
Bold type	Bold type indicates commands, parameter names, or symbols that must be typed as shown. In most cases, upper- and lowercase letters can be freely intermixed. One exception is text within double

quotation marks ("*text*"). Text in quotation marks is usually case-sensitive.

Examples

[*displacement*] [**DI**]
[**DI**+*displacement*]
[**DI**].*displacement*
[**DI**]+*displacement*

Note that in the examples above, the brackets must be typed as shown. The register name **DI** must also be typed as shown, though you could use lowercase letters. The plus sign (+) in the second and fourth examples, and the period (.) in the third example must be typed as shown.

Italics

Italics indicate a placeholder: a name that you must replace with the value or file name required by the program.

Example

/I*path*

In the example above, the slash (/) and the letter **I** must be entered as shown (except that the **I** could be lowercase). However, *path* is a placeholder representing a path name supplied by the user. You could enter any path name such as `B:\` or `\MASM\PROJECT1`. When a placeholder is used in a syntax example at the start of a section, the text below usually describes the types of values that can replace the placeholder.

⟦ ⟧

Double brackets indicate that the enclosed item is optional. Don't confuse double brackets with single brackets ([]), which must be typed as shown.

Example

BP ⟦*number*⟧ *address* ⟦*passcount*⟧ ⟦"*commands*"⟧

In the example, above, you must enter **BP** as shown. You must also enter a value for the *address* placeholder. Values for the placeholders *number*, *passcount*, and *commands* can be entered if you wish, or they can be left blank. If you enter a value for *commands*, it must be enclosed in quotation marks ("").

,,,

A series of commas indicates that you can repeat the preceding item type if you separate each of the items with commas.

Example

[[*name*]] *recordname* <[[*initialvalue*,,,]]>

In the example above, you may provide a *name* and you must provide a *recordname*. You may provide more than one *initialvalue* as long as you separate the values with commas. Note that you must type the angle brackets even if you do not provide any *initialvalue*.

|

A vertical bar between items indicates that only one of the separated items can be used. You must make a choice between the items.

Example

D [[*address* | *range*]]

In the example above, you must enter the letter **D**. You may enter either an *address* or a *range* (but not both).

Special typeface for examples

Example text in this manual is shown in a special typeface so that it will look more like listings on the screen or listings produced with a printer.

Examples that represent source code follow these conventions:

- Lowercase for symbols, labels, instructions, and registers
- Uppercase for reserved words
- Uppercase for hexadecimal digits
- Lowercase for radix indicators
- Upper- and lowercase for comments

These are conventions, not requirements. Your source code can use any combination of upper- and lowercase letters, though your code will be clearer if you choose a convention and use it consistently.

Examples

```
count   DB      0
        mov     ax,bx
print   PROC    near
```

Chapter 2
Elements of the Assembler

2.1 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.

Section 2.2 lists the characters that can be used in a program and Sections 2.3–2.12 describe how to form numbers, names, statements, and comments.

2.2 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

? @ _ $: . [ ] ( ) < > { }

+ - / * & % ! ' ~ ¦ \ = # ^ ; , ‘ "

2.3 Integers

Syntax

digits
*digits*B
*digits*Q
*digits*O
*digits*D
*digits*H
*digits*R

An integer is an integer number: a combination of binary, octal, decimal, or hexadecimal *digits* plus an optional radix. The *digits* are combinations of

one or more digits of the specified radix: **B**, **Q**, **O**, **D**, or **H**. The real number designator **R** can also be used. If no radix is given, the assembler uses the current default radix (decimal, unless you have changed it with the **.RADIX** directive). The radix specifier can be either upper- or lowercase; sample code in this manual uses lowercase. Table 2.1 lists the digits that can be used with each radix.

Table 2.1

Digits Used with Each Radix

Radix	Type	Digits
B	Binary	0 1
Q or **O**	Octal	0 1 2 3 4 5 6 7
D	Decimal	0 1 2 3 4 5 6 7 8 9
H	Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F
R	Real Number	0 1 2 3 4 5 6 7 8 9 A B C D E F

Hexadecimal numbers must always start with a decimal digit (0 to 9). If necessary, put a leading 0 at the left of the number to distinguish between hexadecimal numbers that start with a letter, and symbols. For example, `0ABCh` is interpreted as a hexadecimal number, but `ABCh` is interpreted as a symbol. The hexadecimal digits A through F can be either upper- or lowercase. Sample code in this manual uses uppercase.

The real number designator (**R**) can only be used with hexadecimal numbers consisting of 8, 16, or 20 significant digits (a leading 0 can be added).

The maximum number of digits in an integer depends on the instruction or directive in which the integer is used. The default radix can be specified by using the **.RADIX** directive (see Section 9.3).

Examples

```
01011010b       132q        5Ah         90d
01111b          17o         0Fh         15d
```

2.4 Real Numbers

Syntax

[[+|−]] *digits.digits*[[**E**[[+|−]]*digits*]]

A real number is a number consisting of an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits before the decimal point (.) represent the integer. Those following the point represent the fraction. The digits after the exponent mark (**E**) represent the exponent, which is optional. If an exponent is given, a plus (+) or minus (−) sign may be used to indicate its sign.

Real numbers can be used only with the **DD**, **DQ**, and **DT** directives. The maximum number of digits in the number and the maximum range of exponent values depend on the directive. See Sections 4.3.3, 4.3.4, and 4.3.5 in this reference manual.

Examples

```
25.23
2.523E1
2523.OE-2
```

2.5 Encoded Real Numbers

Syntax

*digits***R**

An encoded real number is an 8-, 16-, or 20-digit hexadecimal number that represents a real number in encoded format. 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. The *digits* must be hexadecimal digits. The number must begin with a decimal digit (0-9) and must be followed by the real number designator (**R**).

Encoded real numbers can be used only with the **DD**, **DQ**, and **DT** directives. The number of digits for the encoded numbers used with **DD**, **DQ**, and **DT** must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See Sections 4.3.3, 4.3.4, and 4.3.5.

Examples

```
DD      3F800000r               ; 1.0 for DD
DQ      3FF0000000000000r       ; 1.0 for DQ
```

2.6 Packed Decimal Numbers

Syntax

[[+|–]]*digits*

A packed decimal number represents a decimal integer to be stored in packed decimal format. Packed decimal storage has a leading-sign byte and 9 value bytes. Each value byte contains two decimal digits. The high-order bit of the sign byte is 0 for positive values, and 1 for negative values.

Packed decimals have the same format as other decimal integers, except that they can take an optional plus (+) or minus (–) sign and can be defined only with the **DT** directive. A packed decimal must not have more than 18 digits.

Examples

```
DT      1234567890      ; Encoded as 0000000001234567890h
DT      -1234567890     ; Encoded as 8000000001234567890h
```

2.7 Character and String Constants

Syntax

'*characters*'
"*characters*"

A character constant consists of a single ASCII (American Standard Code for Information Interchange) character. A string constant consists of two or more ASCII characters. Constants must be enclosed in right single quotation marks or double quotation marks. String constants are case-sensitive.

Single quotation marks must be encoded twice when used literally within constants that are also enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when used in constants that are also enclosed within double quotation marks.

Examples

```
'a'
'ab'
"a"
"This is a message."
'Can''t find file.'          ; Can't find file.
"Can't find file."           ; Can't find file.
"This ""value"" not found."  ; This "value" not found.
'This "value" not found.'    ; This "value" not found.
```

2.8 Names

Syntax

characters

A name is a combination of letters, digits, and special characters used as a label, variable, or symbol in an assembly-language statement. Names have the following formatting rules:

- A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).
- A name can have any combination of upper- and lowercase letters. All lowercase letters are converted to uppercase by the assembler, 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.
- A name can have any number of characters, but only the first 31 characters are used. All other characters are ignored.

Examples

```
subrout3
Array
_main
```

2.9 Reserved Names

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. 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.

Table 2.2 lists all reserved names except instruction mnemonics. For a complete list of instruction mnemonics, see Appendix A.

Table 2.2

Reserved Names

.186	DI	.ERRNZ	LENGTH	.SALL
.286c	DL	ES	.LFCOND	SEG
.286p	DQ	EVEN	.LIST	SEGMENT
.287	DS	EXITM	LOCAL	.SFCOND
.8086	DT	EXTRN	LOW	SHL
.8087	DW	FAR	LT	SHORT
=	DWORD	GE	MACRO	SHR
AH	DX	GROUP	MASK	SI
AL	ELSE	GT	MOD	SIZE
AND	END	HIGH	NAME	SP
ASSUME	ENDIF	IF	NE	SS
AX	ENDM	IF1	NEAR	STRUC
BH	ENDP	IF2	NOT	SUBTTL
BL	ENDS	IFB	OFFSET	TBYTE
BP	EQ	IFDEF	OR	.TFCOND
BX	EQU	IFDIF	ORG	THIS
BYTE	.ERR	IFE	%OUT	TITLE
CH	.ERR1	IFIDN	PAGE	TYPE
CL	.ERR2	IFNB	PROC	.TYPE
COMMENT	.ERRB	IFNDEF	PTR	WIDTH
.CREF	.ERRDEF	INCLUDE	PUBLIC	WORD
CS	.ERRDIF	IRP	PURGE	.XALL
CX	.ERRE	IRPC	QWORD	.XCREF
DB	.ERRIDN	LABEL	.RADIX	.XLIST
DD	.ERRNB	.LALL	RECORD	XOR
DH	.ERRNDEF	LE	REPT	

2.10 Statements

Syntax

[*name*] *mnemonic* [*operands*] [;*comment*]

A statement is a combination of an optional *name*, a mandatory instruction or directive *mnemonic*, one or more optional *operands*, and an optional *comment*. A statement represents an action to be taken by the assembler, such as generating a machine instruction or generating 1 or more bytes of data.

Statements are formed according to the following rules:

- A statement can begin in any column.
- A statement must not have more than 128 characters and must not contain an embedded carriage-return/line-feed combination. In other words, continuing a statement on multiple lines is not allowed.
- All statements except the last one in the file must be terminated by a carriage-return/line-feed combination.

Examples

```
count   DB      0
        mov     ax,bx
        ASSUME  cs:_text,ds:DGROUP
print   PROC    near
```

2.11 Comments

Syntax

; *text*

A comment is any combination of characters preceded by a semicolon (;) and terminated by an embedded carriage-return/line-feed combination. Comments describe the action of a program at the given point, but are otherwise ignored by the assembler and have no effect on assembly.

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 carriage-return/line-feed 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.
```

2.12 COMMENT Directive

Syntax

COMMENT *delimiter*
text
delimiter [*text*]

The **COMMENT** directive causes the assembler to treat all *text* between *delimiter* and *delimiter* as a comment. The *delimiter* character must be the first nonblank character after the **COMMENT** keyword. The text is all remaining characters up to the next occurrence of the delimiter. The text must not contain the delimiter character.

The **COMMENT** directive is typically used for multiple-line comments. Although text can appear anywhere on the same line as the last *delimiter*, all text on the same line as the last *delimiter* is ignored by the assembler.

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 statement
following the last delimiter
+ mov   ax, 1
```

Chapter 3
Program Structure

3.1 Introduction

The program-structure directives let you define the organization that a program's code and data will have when loaded into memory. The program-structure directives include the following:

Directive	Meaning
SEGMENT	Segment definition
ENDS	Segment end
END	Source-file end
GROUP	Segment groups
ASSUME	Segment registers
ORG	Segment origin
EVEN	Segment alignment
PROC	Procedure definition
ENDP	Procedure end

Section 3.2 and Sections 3.4–3.10 describe these directives in detail. Section 3.3 describes the instruction-set directives, which let you specify the instruction set or sets to be used during assembly.

3.2 Source Files

Every assembly-language program is created from one or more "source" files: text files that contain statements defining the program's data and instructions. **MASM** reads source files and assembles the statements to create object modules. **LINK**, the Microsoft 8086 Object Linker, can then 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 MS-DOS functions (or system calls) to print the message `Hello world` on the screen.

Example

```
data    SEGMENT                 ; Program Data Segment
string  DB      "Hello world",13,10,"$"
data    ENDS

code    SEGMENT                 ; Program Code Segment
        ASSUME cs:code,ds:data
start:                          ; Program Entry Point
        mov     ax,data         ; Load data segment location
        mov     ds,ax           ;   into DS register
        mov     dx,OFFSET string ; Load string location
        mov     ah,09h          ; Call string display
        int     21h
        mov     ah,4Ch          ; Call terminate function
        int     21h
code    ENDS

stack   SEGMENT stack           ; Program Stack Segment
        DW      64 DUP(?)       ; Define stack space
stack   ENDS

        END     start           ; Mark end and define start
```

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

1. The **SEGMENT** and **ENDS** statements, which define segments named `data`, `code`, and `stack`.
2. The variable `string` in the `data` segment, which defines the string to be displayed. The variable data are defined in the `data` segment. They include the quoted dollar sign (`"$"`) required by the MS-DOS display-string function, as well as the ASCII codes for a carriage-return/line-feed combination.

3. The instruction label `start` in the `code` segment, which marks the start of the program instructions.
4. The **DW** statement in the `stack` segment, which defines the uninitialized data space to be used for the program stack.
5. The **ASSUME** statement for the `data` and `code` segments, which specifies which segment registers will be associated with the labels, variables, and symbols defined within the segments. An assume statement is not needed for the `stack` segment since the combine type `stack` tells **MASM** that the segment is associated with the **SS** register. See Section 3.4.2 for more information on combine types.
6. The first two code instructions, which load the address of the data segment into the **DS** register. These instructions are not necessary for the code and stack segments because the code-segment address is always loaded into the **CS** register and the stack-segment address is automatically loaded into the **SS** register when you use the **stack** combine type.
7. The last two instructions in the `code` segment, which use MS-DOS function `4Ch` to return to DOS. While there are other techniques for returning to DOS, this is the one recommended for most assembly-language programs.
8. The **END** directive, which indicates the end of the source file, and specifies `start` as the program entry point.

3.3 Instruction-Set Directives

Syntax

.8086
.8087
.186
.286c
.286p
.287

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 all instructions in the file are assembled using the same instruction set.

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 and 8087 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.

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 only by an 80186 microprocessor.

The **.286c** directive enables assembly of 8086 instructions and nonprotected 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.

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** option 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. See Sections 2.3.12 and 2.3.13 of the *Microsoft Macro Assembler User's Guide.*

3.4 SEGMENT and ENDS Directives

Syntax

name **SEGMENT** [[*align*]] [[*combine*]] ['*class*']
name **ENDS**

The **SEGMENT** and **ENDS** directives mark the beginning and end of a program segment. A program segment is a collection of instructions and/or data whose addresses are all relative to the same segment register.

The *name* defines the name of the segment. This name can be unique or be the same name given to other segments in the program. Segments with identical names are treated as the same segment.

The optional *align*, *combine*, and *class* types give the linker instructions on how to set up segments. They should be specified in order, but it is not necessary to enter all types, or any type, for a given segment.

Note

Don't confuse the **byte** and **word** align types with the **BYTE** and **WORD** reserved words used to specify data type with operators such as **THIS** and **PTR**. Also, the **page** align type and the **public** combine type should not be confused with the **PAGE** and **PUBLIC** directives. The distinction should be clear from context since the align and combine types are only used on the same line as the **SEGMENT** directive. To make the difference even clearer, align and combine types are shown with lowercase letters in this manual, although you can actually enter them in either case.

Sections 3.4.1–3.4.4 describe the three program-loading options and give an example program. Segment nesting is also explained in Section 3.4.5. Some of the information in this section is also discussed in Section 3.4 of the *Microsoft Macro Assembler User's Guide.*

3.4.1 Align Type

The optional *align* type defines the alignment of the given segment. The alignment defines the range of memory addresses from which a starting address for the segment can be selected. The align type can be any one of the following:

Align Type	Meaning
byte	Use any byte address
word	Use any word address (2 bytes/word)
para	Use paragraph addresses (16 bytes/paragraph)
page	Use page addresses (256 bytes/page)

If no *align* type is given, **para** is used by default. The actual start address is not computed until the program is loaded. The linker ensures that the address will be on the given boundary.

3.4.2 Combine Type

The optional *combine* type defines how to combine segments having the same name. The combine type can be any one of the following:

Combine Type	Meaning
public	Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the new segment.
stack	Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the **public** combine type, except that all addresses in the new segment are relative to the **SS** segment register. The stack pointer (**SP**) register is initialized to the ending address of the segment. Stack segments should normally use the **stack** type, since this automatically initializes the **SS** register. If you create a stack segment and do not use the **stack** type, you must give instructions to load the segment address into the **SS** register.

common	Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address. If data are declared in more than one segment having the same name and **common** type, the most recently declared data replace any previously declared data.
memory	Is treated by the Microsoft 8086 Object Linker (**LINK**) exactly like a **public** segment. **MASM** allows you to define segments with **memory** type even though **LINK** does not support a separate **memory** type. This feature is provided for compatibility with other linkers that may support a combine type conforming to the Intel definition of **memory** type.
at *address*	Causes all label and variable addresses defined in the segment to be relative to the given *address*. The *address* can be any valid expression, but must not contain a forward reference, that is, a reference to a symbol defined later in the source file. An **at** segment typically contains no code or initialized data. Instead, it represents an address template that can be placed over code or data already in memory, such as the screen buffer. The labels and variables in the **at** segments can then be used to access the fixed instructions and data.

If no *combine* type is given, the segment is not combined. Instead, it receives its own physical segment when loaded into memory.

Note

Normally you should provide at least one stack segment in a program. If no stack segment is declared, **LINK** will display a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment.

3.4.3 Class Type

The optional *class* type defines which segments are to be loaded in contiguous memory. Segments having the same class name are loaded into memory one after another. All segments of a given class are loaded before segments of any other class. The *class* name must be enclosed in single quotation marks ('). Class names are not case-sensitive unless the **/ML** or **/MX** option is used during assembly, or the **/NOIGNORECASE** option is used when linking.

Note

The names assigned for class types of segments should not be used for other symbol definitions in the source file. For example, if you give a segment the class name `'CONSTANT'`, you should not give the name `constant` to any variable or labels in the source file. If you do, the error `Symbol already different kind` will be generated.

If class types are not specified, **LINK** copies segments to the executable file in the same order they are encountered in the object files. This order is maintained throughout the program unless **LINK** encounters two or more segments having the same class name. Segments having identical class names belong to the same class, and are copied as contiguous blocks to the executable file.

Example

```
DATAX   segment 'DATA'
DATAX   ends

TEXT    segment 'CODE'
TEXT    ends

DATAZ   segment 'DATA'
DATAZ   ends
```

In the preceeding example-program fragment, the segments `DATAX` and `DATAZ` both have class type `'DATA'`. As a result, both segments are copied to the executable file before the `TEXT` segment.

All segments belong to a class. Segments for which no class name is explicitly stated have the null-class name, and will be loaded as contiguous blocks with other segments having the null-class name. **LINK** imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64K.

Since **LINK** processes modules in the order in which it receives them on the command line, you may not always be able to easily specify the order in which you want segments to be loaded. For example, assume your program has four segments that you want loaded in the following order: CODE, DATA, CONST, STACK. The CODE, CONST, and STACK segments are defined in the first module of your program, but the DATA segment is defined in the second module. **LINK** will not put the segments in the proper order because it will first load the segments encountered in the first module.

You can avoid this problem by creating and assembling a dummy program file containing empty segment definitions in the order in which you wish to load your real segments. Once this file is assembled, you can give it as the first object file in any invocation of **LINK**. The linker will automatically load the segments in the order given.

For example, the following dummy program file defines the loading order of segments in a program having segments named CODE, DATA, CONST, and STACK.

```
CODE    segment para public 'CODE'
CODE    ends
DATA    segment para public 'DATA'
DATA    ends
CONST   segment para public 'CONST'
CONST   ends
STACK   segment para stack 'STACK'
STACK   ends
```

The dummy program file must contain definitions for all classes to be used in your program. If it does not, **LINK** will choose a default loading order which may or may not correspond to the order you desire. When linking your program, the dummy program must be the first object file specified in the **LINK** command line.

Do not use a dummy program file with Microsoft C, Pascal, FORTRAN, or compiled BASIC. These languages follow the MS-DOS segment-ordering convention described in Section 3.3.15 of the *Microsoft Macro Assembler User's Guide*. This loading order must not be modified.

Another way to control segment order is with the **MASM /A** option. This option directs **MASM** to write segments to the object file in alphabetical order. You can give segments names with alphabetical order that matches the order in which you want them loaded and then use the **/A** option. To make this strategy work with multiple-module programs, you should define all segments in the first module specified in the **LINK** command line. Some of the definitions may be dummy segments. See Section 2.3.1 of the *Microsoft Macro Assembler User's Guide* for more information on the **/A** option.

Note

Some previous versions of the assembler ordered segments alphabetically by default. If you have trouble assembling and linking source-code listings from books or magazines, try using the **/A** option. Listings written for the old version assemblers may not work without this option.

3.4.4 Program Example

The following source code illustrates one way in which the *align* and *combine* types can be used. Figure 3.1 (following the example below) shows the way **LINK** would load the given program into memory. The **memory** combine type is not shown since it is the same as **public**. The *class* types are not used in the sample program, but they are illustrated in Section 3.4.3 and in the example in Section 3.6.

Note

Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict.

Example

```
                NAME module_1

seg_a           SEGMENT word public
start:          .
                .
                .
seg_a           ENDS

seg_b           SEGMENT page stack
                .
                .
                .
seg_b           ENDS

seg_c           SEGMENT para common
                .
                .
                .
seg_c           ENDS

seg_d           SEGMENT at 0B800h
                .
                .
                .
seg_d           ENDS
                END start

                NAME module_2

seg_a           SEGMENT word public
                .
                .
                .
seg_a           ENDS

seg_b           SEGMENT page stack
                .
                .
                .
seg_b           ENDS

seg_c           SEGMENT para common
                .
                .
                .
seg_c           ENDS
                END
```

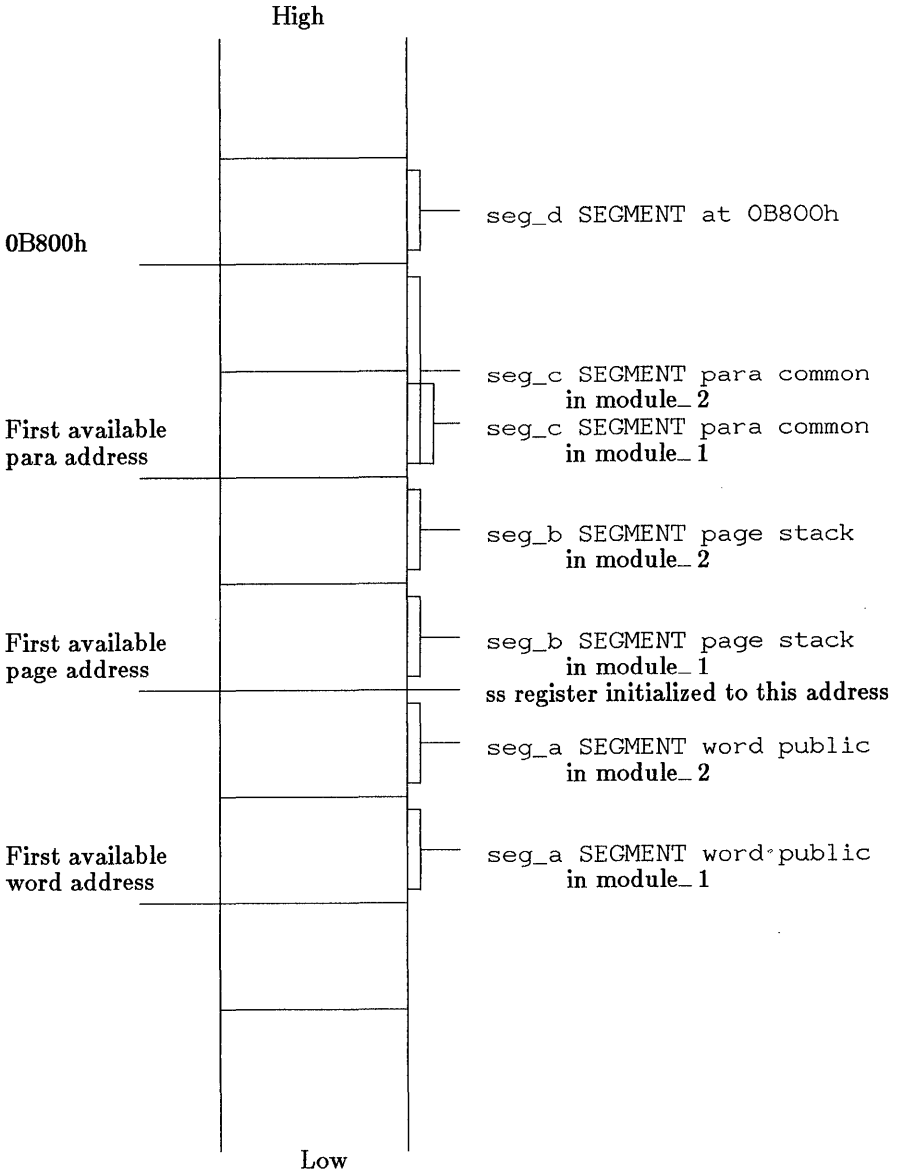


Figure 3.1 LINK Program Loading Order

3.4.5 Segment Nesting

Segments can be nested. When **MASM** encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, **MASM** continues assembly of the enclosing segment. Overlapping segments are not permitted.

Example

```
sample  SEGMENT word public 'CODE'      ; outside segment
main    PROC far
        .
        .
        .
const   SEGMENT word public 'CONST'     ; nested segment
array   DW      array_data
const   ENDS                            ; end nesting
        .
        .
        .
        RET
main    ENDP
sample  ENDS
```

This example-code fragment contains two segments: a code segment called `sample` and a data segment called `const`. The `const` segment is nested within the `sample` segment.

3.5 END Directive

Syntax

END [*expression*]

The **END** directive marks the end of a module. The assembler ignores any statements following this directive.

The optional *expression* defines the program entry point, the address at which program execution is to start. If the program has more than one module, only one of these modules can define an entry point. The module with the entry point is called the "main module". If no entry point is given, none is assumed.

Note

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.

Examples

```
        end
        end     start
```

3.6 GROUP Directive

Syntax

name **GROUP** *segmentname,,,*

The **GROUP** directive associates a group *name* with one or more segments, and causes all labels and variables defined in the given segments to have addresses relative to the beginning of the group rather than to the beginning of the segments in which they are defined. The *segmentname* must be the name of a segment defined using the **SEGMENT** directive, or a **SEG** expression (see Sections 3.4 and 5.3.12). The *name* must be unique.

The **GROUP** directive does not affect the order in which segments of a group are loaded. Loading order depends on each segment's class, or on the order in which object modules are given to the linker. Section 3.4.5 of the *Microsoft Macro Assembler User's Guide* also discusses groups and how they are handled by the linker.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65535. Therefore, if the segments of a group are contiguous, the group can occupy up to 64K of memory.

Group names can be used with the **ASSUME** directive (Section 3.7) and as an operand prefix with the segment override operator (:) (Section 5.3.7).

Note

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   aseg,bseg
        ASSUME  ds:dgroup

aseg    SEGMENT byte public 'DATA1'
          .
sym_a:    .
          .
aseg    ENDS

bseg    SEGMENT byte public 'DATA2'
          .
sym_b:    .
          .
bseg    ENDS

cseg    SEGMENT byte public 'DATA1'
          .
sym_c:    .
          .
cseg    ENDS
        END
```

The order in which **LINK** will load these segments is shown in Figure 3.2. **LINK** loads `aseg` first because it occurs first in the source file. Next, **LINK** loads `cseg` because it has the same class type as `aseg`. **LINK** loads `bseg` last. However, `aseg` and `bseg` are declared part of the same group, despite their separation in memory. This means that the symbols `sym_a` and `sym_b` have offsets from the beginning of the group, which is also the beginning of `aseg`. The offset of `sym_c` is from the beginning of `cseg`. This sample is intended to illustrate the way **LINK** organizes segments in a group, rather than to show a typical use of a group.

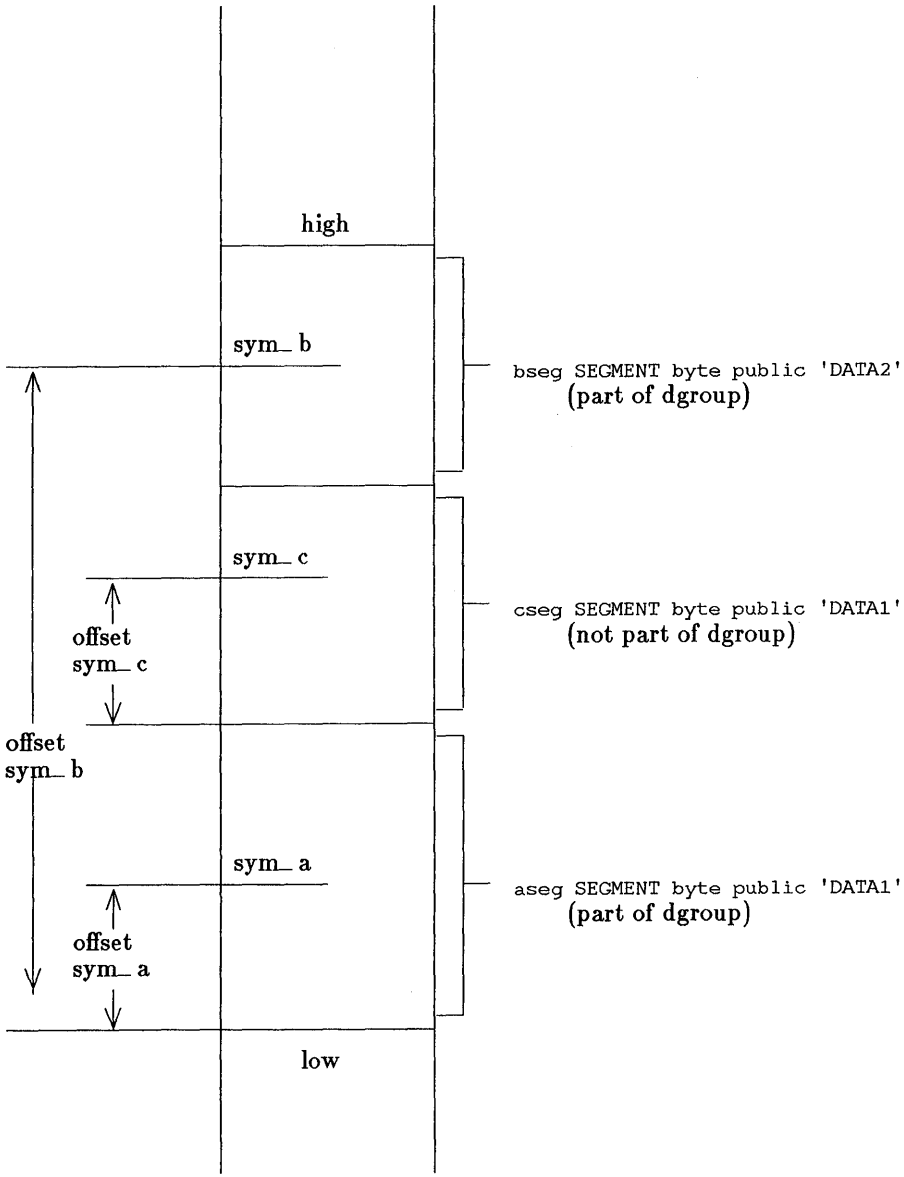


Figure 3.2 LINK Segment Loading Order

3.7 ASSUME Directive

Syntax

ASSUME *segmentregister*:*segmentname*,,,
ASSUME NOTHING

The **ASSUME** directive specifies *segmentregister* as the default segment register for all labels and variables defined in the segment or group given by *segmentname*. Subsequent references to the label or variable will automatically assume the selected register when the effective address is computed.

The **ASSUME** directive can define up to four selections: one for each of the four segment registers. The *segmentregister* can be any one of the segment register names: **CS**, **DS**, **ES**, or **SS**. The *segmentname* must be one of the following:

- The name of a segment that was previously defined with the **SEGMENT** directive
- The name of a group that was previously defined with the **GROUP** directive
- The keyword **NOTHING**

The keyword **NOTHING** cancels the current segment selection. The statement `ASSUME NOTHING` cancels all register selections made by a previous **ASSUME** statement.

Note

The segment-override operator (:) can be used to override the current segment register selected by the **ASSUME** directive.

Examples

```
ASSUME cs:CODE
ASSUME cs:cgroup,ds:dgroup,ss:nothing,es:nothing
ASSUME NOTHING
```

3.8 ORG Directive

Syntax

ORG *expression*

The **ORG** directive sets the location counter to *expression.* Subsequent instruction and data addresses begin at the new value.

The *expression* must resolve to an absolute number. In other words, all symbols used in the expression must be known on the first pass of the assembler. The location-counter symbol ($) can also be used.

Examples

```
        ORG     120h
        mov     ax,dx
```

In the first example, the statement `mov ax,dx` begins at byte 120h in the current segment.

```
        ORG     $+2
array   DW      100 dup (0)
```

In the second example, the variable `array` is declared to start at the address 2 bytes beyond the current address. See Section 5.2.4 for more information on the location-counter symbol ($).

3.9 EVEN Directive

Syntax

EVEN

The **EVEN** directive aligns the next data or instruction byte on a word boundary. If the current value of the location counter is odd, the directive increments the location counter to an even value and generates one **NOP** (no operation) instruction. If the location counter is already even, the directive does nothing.

Note

The **EVEN** directive must not be used in **byte**-aligned segments.

Example

```
        ORG     0
test1   DB      1
        EVEN
test2   DW      513
```

In this example, the **EVEN** directive tells **MASM** to increment the location counter, and generates a single **NOP** instruction (90h). This means the offset of `test2` is 2, not 1, as it would be without the **EVEN** directive.

3.10 PROC and ENDP Directives

Syntax

name **PROC** [*distance*]
    statements
name **ENDP**

The **PROC** and **ENDP** directives mark the beginning and end of a procedure. A procedure is a block of instructions that forms a program subroutine. Every procedure has a *name* with which it can be called.

The *name* must be a unique name, not previously defined in the program. The optional *distance* can be either **NEAR** or **FAR**. **NEAR** is assumed if no *distance* is given. The *name* has the same attributes as a label, and can be used as an operand in a jump, call, or loop instruction.

Any number of *statements* can appear between the **PROC** and **ENDP** statements. The procedure should contain at least one **RET** directive to return control to the point of call. Nested procedures are allowed.

Example

```
        push    ax              ; Push third parameter
        push    bx              ; Push second parameter
        push    cx              ; Push first parameter
        call    addup           ; Call the procedure
        add     sp,6            ; Destroy the pushed parameters
        .
        .
        .
addup   PROC    near            ; Return address for near call
                                ;   takes two bytes
        push    bp              ; Save base pointer - takes two more
                                ;   so parameters start at 4th byte
        mov     bp,sp           ; Load stack into base pointer
        mov     ax,[bp+4]       ; Get first parameter
                                ;   4th byte above pointer
        add     ax,[bp+6]       ; Get second parameter
                                ;   6th byte above pointer
        add     ax,[bp+8]       ; Get third paramter
                                ;   8th byte above pointer
        pop     bp              ; Restore base
        RET                     ; Return
addup   ENDP
```

In this example, three numbers are passed as parameters for the procedure addup. Parameters are often passed to procedures by pushing them before the call so that the procedure can read them off the stack.

Note

The parameter-passing method in this example conforms to the standard used in Microsoft high-level languages. As a result, this procedure could be traced using the Stack Trace command (**K**) of the Microsoft Symbolic Debug Utility (**SYMDEB**), described in Section 4.6.28 of the *Microsoft Macro Assembler User's Guide.*

Chapter 4
Types and Declarations

4.1 Introduction

This chapter explains how to generate data for a program; how to declare labels, variables, and other symbols that refer to instruction and data locations; and how to define types that can be used to generate data blocks containing multiple fields, such as structures and records.

4.2 Label Declarations

Label declarations create "labels." A label is a name that represents the address of an instruction. Labels can be used in jump, call, and loop instructions to direct program execution to the instruction at the address of the label.

4.2.1 Near-Label Declarations

Syntax

name:

A near-label declaration creates an instruction label that has **NEAR** type. The label can be used in subsequent instructions in the same segment to pass execution control to the corresponding instruction.

The *name* must be unique, not previously defined, and it must be followed by a colon (:). Furthermore, the segment containing the declaration must be associated with the **CS** segment register (see Section 3.7 for information on the **ASSUME** directive). The assembler sets the name to the current value of the location counter.

A near-label declaration can appear on a line by itself or on a line with an instruction. Labels must be declared with the **PUBLIC** or **EXTRN** directive if they are located in one module but called from another module (see Chapter 6).

Examples

```
start:
cycle:  inc     si
```

4.2.2 Procedure Labels

Syntax

name **PROC** [*distance*]

The **PROC** directive creates a label *name* and optionally assigns it a *distance*. The distance can be **NEAR** or **FAR**. The label then represents the address of the first instruction of a procedure. The label can be used in a **CALL** instruction (or in a jump or loop instruction) to direct execution control to the first instruction of the procedure. If you do not specify the type for a procedure, the assembler assumes **NEAR** as the default.

When the **PROC** label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to **NEAR** or **FAR**. If the label has **FAR** type, the assembler also sets its segment value to that of the enclosing segment.

NEAR labels can be used with jump, call, or loop instructions to transfer program control to any address in the current segment. **FAR** labels can be used to transfer program control to an address in any segment outside the current segment.

Labels must be declared with the **PUBLIC** and **EXTRN** directive if they are located in one module but called from another module (see Chapter 6).

4.3 Data Declarations

The data-declaration directives let you generate data for a program. The directives translate numbers, strings, and expressions into individual bytes, words, or other units of data. The encoded data are copied to the object file.

The data-declaration directives are listed below:

Directive	Meaning
DB	Define byte
DW	Define word
DD	Define doubleword
DQ	Define quadword
DT	Define ten bytes

Sections 4.3.1–4.3.5 describe these directives in detail.

4.3.1 DB Directive

Syntax

[*name*] **DB** *initialvalue*,,,

The **DB** directive allocates and initializes a byte (8 bits) of storage for each *initialvalue*. The *initialvalue* can be an integer, a character string constant, a **DUP** operator, a constant expression, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **BYTE** whose offset value is the current location-counter value.

A string constant can have any number of characters, as long as it fits on a single line. When the string is encoded, the characters are stored in the order given, with the first character in the constant at the lowest address and the last at the highest.

Examples

```
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
```

4.3.2 DW Directive

Syntax

[*name*] **DW** *initialvalue*,,,

The **DW** directive allocates and initializes a word (2 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a one- or two-character string constant, a **DUP** operator, a constant expression, an address expression, or a question mark (?). The question mark represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **WORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte. Either 0 or the first character is placed in the high-order byte.

Examples

```
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
```

4.3.3 DD Directive

Syntax

[*name*] **DD** *initialvalue*,,,

The **DD** directive allocates and initializes a doubleword (4 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a real number, a one- or two-character string constant, an encoded real number, a **DUP** operator, a constant expression, an address expression, or a question mark

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

The *name* is optional. If *name* is given, the directive creates a variable of type **DWORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Examples

```
integer         DD      16728
character       DD      'a'
string          DD      'bc'
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
```

4.3.4 DQ Directive

Syntax

[*name*] **DQ** *initialvalue*,,,

The **DQ** directive allocates and initializes a quadword (8 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a real number, a one- or two-character string constant, an encoded real number, a **DUP** operator, a constant expression, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **QWORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Examples

```
integer         DQ      16728
character       DQ      'a'
string          DQ      'bc'
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
```

4.3.5 DT Directive

Syntax

[*name*] **DT** *initialvalue*,,,

The **DT** directive allocates and initializes 10 bytes of storage for each *initialvalue*. The *initialvalue* can be an integer expression, a packed decimal, a one- or two-character string constant, an encoded real number, a **DUP** operator, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **TBYTE** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Note

The **DT** directive assumes that constants with decimal digits are packed decimals, not integers. If you want to specify a 10-byte integer, you must follow the number with the letter that specifies the number system you are using (for example, "D" or "d" for decimal or "H" or "h" for hexadecimal).

Examples

```
packeddecimal   DT      1234567890
integer         DT      16728d
character       DT      'a'
string          DT      'bc'
real            DT      1.5
encodedreal     DT      3F000000000000000000r
empty           DT      ?
multiple        DT      1,2,3,'$'
duplicate       DT      10 dup(?)
high_tbyte      DT      120892581961462917470617 5d
```

4.3.6 DUP Operator

Syntax

count **DUP**(*initialvalue*,,,)

The **DUP** operator is a special operator that can be used with the data-declaration directives and other directives to specify multiple occurrences of one or more initial values. The *count* sets the number of times to define *initialvalue*. The initial value can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. If more than one initial value is given, the values must be separated by commas (,). **DUP** operators can be nested up to 17 levels. The initial value (or values) must always be placed within parentheses.

Examples

```
DB      100     DUP(1)
```

The first example generates 100 bytes with initial value 1.

```
DW      20      DUP( 1,2,3,4 )
```

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

```
DB      5     DUP( 5 DUP( 5 DUP (1)))
```

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

```
        DD      14      DUP (?)
```

The final example generates 14 doublewords of uninitialized data.

4.4 Symbol Declarations

The symbol-declaration directives let you create and use symbols. A symbol is a descriptive name representing a number, text, an instruction, or an address. Symbols make programs easier to read and maintain by using descriptive names to represent values. A symbol can be used anywhere its corresponding value is allowed.

The symbol declaration directives are listed below:

Directive	Meaning
=	Assign absolutes
EQU	Equate absolutes, aliases, or text symbols
LABEL	Create instruction or data labels

Sections 4.4.1–4.4.3 describe the directives in detail.

4.4.1 Equal-Sign (=) Directive

Syntax

name=expression

The equal-sign (=) directive creates an absolute symbol by assigning the numeric value of *expression* to *name*. An absolute symbol is simply a name that represents a 16-bit value. No storage is allocated for the number. Instead, the assembler replaces each subsequent occurrence of *name* with the value of *expression*. The value is variable during assembly, but is a constant at run time.

The expression can be an integer, a one- or two-character string constant, a constant expression, or an address expression. Its value must not exceed 65535. The name must be either a unique name, or a name previously defined using the equal-sign (=) directive.

Absolute symbols can be redefined at any time.

Examples

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

4.4.2 EQU Directive

Syntax

name **EQU** *expression*

The **EQU** directive creates absolute symbols, aliases, or text symbols by assigning *expression* to *name*. An absolute symbol is a name that represents a 16-bit value; an alias is a name that represents another symbol; and a text symbol is a name that represents a character string or other combination of characters. The assembler replaces each subsequent occurrence of the name with either the text or the value of the expression, depending on the type of expression given.

The name must be a unique name, one which has not been previously defined. The expression can be an integer, a string constant, a real number, an encoded real number, an instruction mnemonic, a constant expression, or an address expression. Expressions that evaluate to values in the range 0 to 65535 create absolute symbols and cause **MASM** to replace the name with a value. All other expressions cause the assembler to replace the name with text.

The **EQU** directive is sometimes used to create simple macros. Note that the assembler replaces a name with text or a value before attempting to assemble the statement containing the name.

Symbols defined using the **EQU** directive cannot be redefined.

Examples

```
k           EQU  1024           ; Replaced with value
pi          EQU  3.14159        ; Replaced with text
matrix      EQU  20 * 30        ; Replaced with value
staptr      EQU  [bp]           ; Replaced with text
clearax     EQU  xor ax,ax      ; Replaced with text
prompt      EQU  'Type Enter'   ; Replaced with text
bpt         EQU  BYTE PTR       ; Replaced with text
```

4.4.3 LABEL Directive

Syntax

name **LABEL** *type*

The **LABEL** directive creates a new variable or label by assigning the current location-counter value and the given *type* to *name.*

The name must be unique and not previously defined. The type can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR

The type can also be the name of a valid structure type.

Examples

```
barray          LABEL   BYTE
warray          DW      100 DUP(?)
```

In this example, `barray` and `warray` refer to the same data. The data can be accessed by byte with `barray` or by word with `warray`.

4.5 Type Declarations

The type-declaration directives let you define data types that can be used to create program variables consisting of multiple elements or fields. The directives associate one or more named fields with a given type name. The type name can then be used in a data declaration to create a variable of the given type.

The type-declaration directives are listed below:

Directive	Declaration
STRUC and **ENDS**	Structure types
RECORD	Record types

Sections 4.5.1 and 4.5.2 describe these directives in detail.

4.5.1 STRUC and ENDS Directives

Syntax

name **STRUC**
fielddefinitions
name **ENDS**

The **STRUC** and **ENDS** directives mark the beginning and end of a type definition for a structure. A type definition for a structure defines the name of a structure type and the number, type, and default values of the fields contained in the structure.

A structure definition creates a template for data. Though this template is used by **MASM** during assembly, it does not in itself create any data. Data can only be created when you declare a structure, as described in Section 4.6.1.

The *name* defines the new name of the structure type. It must be unique. The *fielddefinitions* define the structure's fields. Any number of field definitions can be given. The definitions must have one of the following forms:

[*name*] **DB** *defaultvalue*,,,
[*name*] **DW** *defaultvalue*,,,
[*name*] **DD** *defaultvalue*,,,
[*name*] **DQ** *defaultvalue*,,,
[*name*] **DT** *defaultvalue*,,,

The optional *name* specifies the field name; the **DB**, **DW**, **DD**, **DQ**, and **DT** directives define the size of each field; and *defaultvalue* defines the value to be given to the field if no initial value is given when the structure variable is declared. The name must be unique, and, once defined, represents the offset from the beginning of the structure to the corresponding field.

The default value can define a number, character or string constant, or symbol. It may also contain the **DUP** operator to define multiple values for the field. If the default value is a string constant, the field has the same number of bytes as characters in the string. If multiple default values are given, they must be separated by commas (,).

A definition of a structure type can contain field definitions and comments only. It must not contain any other statements. Therefore, structures cannot be nested.

Example

```
table   STRUC
        count   DB      10
        value   DW      10 DUP(?)
        tname   DB      'font3'
table   ENDS
```

In this example, the fields are `count`, `value`, and `tname`. The `count` field is a single-byte value initialized to 10; `value` is an array of 10 uninitialized word values; and `tname` is a character array of 5 bytes initialized to `'font3'`. The field names `count`, `value`, and `tname` have the offset values 0, 1, and 21, respectively.

4.5.2 RECORD Directive

Syntax

recordname **RECORD** *fieldname:width*[=*expression*],,,

The **RECORD** directive defines a record type for an 8- or 16-bit record that contains one or more fields. The *recordname* is the name of the record type to be used when creating the record; *fieldname* is the name of a field in the record, *width* is the number of bits in the field; and *expression* is the initial (or default) value for the field.

Any number of *fieldname:width=expression* combinations can be given for a record, as long as each is separated from its predecessor by a comma (,). The sum of the widths for all fields must not exceed 16 bits.

The width must be a constant in the range 1 to 16. If the total width of all declared fields is larger than 8 bits, then the assembler uses 2 bytes. Otherwise, only 1 byte is used.

If =*expression* is given, it defines the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character for *expression*. The expression must not contain a forward reference to any symbol.

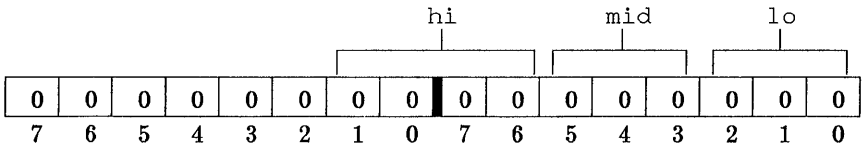
In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be initialized to 0 in the high end of the record.

The **RECORD** directive creates a template for data. This template is used by the assembler during assembly, but it does not in itself create any data. Data can only be created when you declare a record, as described in Section 4.6.2.

Examples

```
encode  RECORD  hi:4, mid:3, lo:3
```

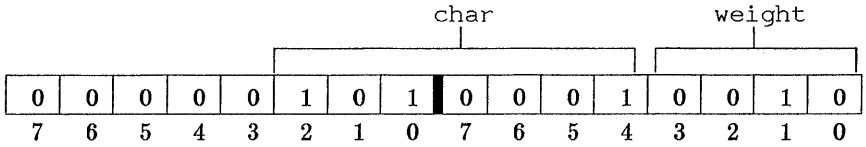
The example above creates a record type `encode` having three fields: `hi`, `mid`, and `lo`. Each record declared using this type will occupy 16 bits of memory. The `hi` field will be in bits 6 to 9 (bit 9 is bit 1 in the high byte); the `mid` field will be in bits 3 to 5; and the `lo` field will be in bits 0 to 2. The remaining high-order bits will be unused. The bit diagram below shows what the record type will look like:



Since no initial values are given, the record type has all bits set to 0. Note that this is only a template maintained by the assembler. No data are created.

```
item    RECORD  char:7='Q', weight:4=2
```

The example above creates a record type `item` having two fields: `char` and `weight`. These values are initialized to the letter `Q` and the number `2`, respectively. Unused bits are set to 0, as shown in the bit diagram below.



4.6 Structure and Record Declarations

Structure and record declarations allow you to generate blocks of data bytes with many elements or fields. A structure or record declaration consists of the name of a previously defined structure or record, and a set of initial values.

Sections 4.6.1–4.6.2 describe these declarations in detail.

4.6.1 Structure Declarations

Syntax

[[*name*]] *structurename* <[[*initialvalue*,,,]]>

A structure variable is a variable with one or more fields of different sizes. The *name* is the name of the variable; *structurename* is the name of a structure type created using the **STRUC** directive; and *initialvalue* is one or more values defining the initial value of the structure. One initial value can be given for each field in the structure.

The *name* is optional. If not given, the assembler allocates space for the structure, but does not create a name you can use to access the structure.

The *initialvalue* can be an integer, string constant, or expression that evaluates to a value having the same type as the corresponding field. The angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated by commas (,). If the **DUP** operator (see Section 4.3.6) is used, only the values within the parentheses need to be enclosed in angle brackets.

You need not initialize all fields in a structure. If an initial value is left blank, the assembler automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is uninitialized. Section 5.2.9 illustrates several ways to use structure data after they have been declared.

Note

You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was defined. For example, assume the following structure definition:

```
strings     STRUC
            buffer  DB 100 DUP (?)   ; Can't override
            crlf    DB 13,10         ; Can't override
            query   DB 'Filename: '  ; String <= can override
            endmark DB 36            ;
strings     ENDS
```

The `buffer` and `crlf` variables cannot be overridden because they have multiple values. The `query` variable can be overridden as long as the overriding data are no longer than `query` (10 bytes). Similarly, the `endmark` field can be overridden by any byte value.

Examples

```
struct1 table     <>
```

The preceding example creates a structure variable named `struct1` whose type is given by the structure type `table`. The initial values of the fields in the structure are set to the default values for the structure type, if any. For example, if `table` were defined with the structure definition in the example in Section 4.5.1, the first byte of `struct1` would be 10; 10 uninitialized words would follow; and finally would come the byte string `font3`.

```
struct2 table    <0,,>
```

The second example creates a structure variable named `struct2`. Its type is also `table`. The initial value for the first field is set to `0`. The default values defined by the structure type are used for the remaining two fields. If `table` were defined with the structure definition in the example in Section 4.5.1, the initial value of `0`, set with the structure declaration above, would override the initial value of `10`, set with the original structure definition.

```
struct3 table    10 DUP(<0,,>)
```

This final example creates a variable, `struct3`, containing 10 structures of the type `table`. The first field in each structure is set to the initial value of `0`. All remaining fields receive the default values.

4.6.2 Record Declarations

Syntax

[*name*] *recordname* <[*initialvalue*,,,]>

A record variable is an 8- or 16-bit value whose bits are divided into one or more fields. The *name* is the name of the variable; *recordname* is the name of a record type that has been created using the **RECORD** directive; and *initialvalue* is one or more values defining the initial value of the record. One *initialvalue* can be given for each field in the record.

The name is optional. If no *name* is given, **MASM** allocates space for the record, but does not create a variable that you can use to access the record.

The optional *initialvalue* can be an integer, string constant, or any expression that resolves to a value no larger than can be represented in the field width specified when the record was defined. Angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated by commas (,). If the **DUP** operator (see Section 4.3.6) is used, only the values within the parentheses need to be enclosed in angle brackets. You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is defined by the record type. If there is no default value, the field is uninitialized.

Sections 5.2.10 and 5.2.11 illustrate ways to use record data after it has been defined.

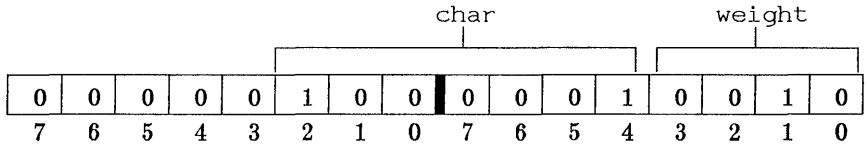
Examples

```
rec1    encode  <>
```

The first example creates a variable named `rec1` whose type is given by the record type `encode`. The initial values of the fields in the record are set to the default values for the record type, if any. For example, if `encode` were defined with the definition in the example in Section 4.5.2, `rec1` would be 0, since the fields were not initialized in the definition.

```
table   item    10 DUP(<'A',2>)
```

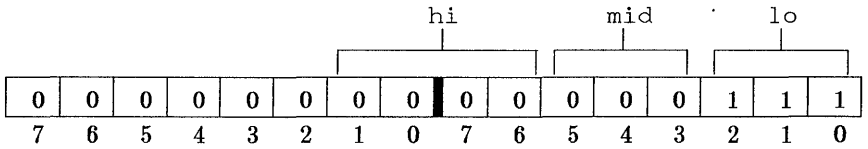
This second example creates a variable named `table` containing `10` records of the record type `item`. The fields in these records are all set to the initial values `A` and `2`. If the `item` definition from the example in Section 4.5.2 were used, the `A` would override the initial value of `Q` in the record definition.



The bit diagram above shows the value of the 10 bytes created by the record declaration.

```
passkey encode  <,,7>
```

The final example creates a record variable named `passkey`. Its type is `encode`. The initial values for the first two fields are the default values defined by the record type. The initial value for the third field is `7`. If the record definition from Section 4.5.2 were used, the first two fields would remain 0, since they were not initialized. The bit diagram below shows what the record looks like.



Chapter 5
Operands and Expressions

5.1 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.

5.2 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.

The operand types are listed below:

Constant

Direct-memory

Relocatable

Location-counter

Register

Based

Indexed

Based-indexed

Structure

Record

Record-field

5.2.1 Constant Operands

Syntax

number|string|expression

A constant operand is a number, string constant, symbol, or expression that evaluates to a fixed value. Constant operands, unlike other operands, represent values to be acted on, rather than memory addresses.

Examples

```
mov     ax,9
mov     al,'c'
mov     bx,65535/3
mov     cx,count
```

Note that `count` in the last example is a constant only if it was defined with the **EQU** or equal-sign (=) operator. If `count` is a symbol representing a relocatable value or address, it is not a constant.

5.2.2 Direct-Memory Operands

Syntax

segment:offset

A direct-memory operand is a pair of segment and offset values that represents the absolute memory address of 1 or more bytes of memory. The *segment* can be a segment register (**CS**, **DS**, **SS**, or **ES**), a segment name, or a group name. The *offset* must be an integer, absolute symbol, or expression that resolves to a value within the range 0 to 65535.

Examples

```
mov     dx,ss:0031h
mov     bx,data:0
mov     ax,DGROUP:block
```

5.2.3 Relocatable Operands

Syntax

symbol

A relocatable operand is any symbol that represents the memory address (segment and offset) of an instruction or of data to be acted upon. 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 value until the program has been linked.

Examples

```
call    main
mov     bx,value
mov     bx,OFFSET dgroup:table
mov     cx,count
```

Note that `count` in the last example is a relocatable operand if it was defined with the **DW** directive. If `count` was defined with the **EQU** or equal-sign (=) operator, it is a constant.

5.2.4 Location-Counter Operand

Syntax

$

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

```
help     DB       'Program options:',13,10
F1       DB       '  F1       This help screen',13,10
F2       DB       '  F2       Save file',13,10
         .
         .
         .
F10      DB       '  F10      Exit program',13,10,'$'
DISTANCE =       $-help
```

In this example, the location counter forces the assembler to count the total length of a group of declared strings, saving the programmer the trouble of counting each byte.

5.2.5 Register Operands

Syntax

registername

A register operand is the name of a CPU register. Register operands direct instructions to carry out actions on the contents of the given registers. The *registername* can be any of the register names in Table 5.1.

Table 5.1

Register Operands

Register Operand Type	Register Name			
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
8-bit high registers	**AH**	**BH**	**CH**	**DH**
8-bit low registers	**AL**	**BL**	**CL**	**DL**
16-bit segment	**CS**	**DS**	**SS**	**ES**
16-bit pointer and index	**SP**	**BP**	**SI**	**DI**

Any combination of upper- and lowercase letters is allowed.

The **AX**, **BX**, **CX**, and **DX** registers are 16-bit, general-purpose registers. They can be used for any data or numeric manipulation. The **AH**, **BH**,

CH, **DH** registers represent the high-order 8 bits of the corresponding general-purpose registers. Similarly, **AL**, **BL**, **CL**, and **DL** represent the low-order 8 bits of the general-purpose registers.

The **CS**, **DS**, **SS**, and **ES** registers are the segment registers. They contain the current segment addresses of the code, data, stack, and extra segments, respectively. All instruction and data addresses are relative to the segment address in one of these registers.

The **SP** register is the 16-bit stack-pointer register. The stack pointer contains the current top-of-stack address. This address is relative to the segment address in the **SS** register and is automatically modified by instructions that access the stack.

The **BX**, **BP**, **DI**, and **SI** registers are 16-bit, base and index registers. These are general-purpose registers typically used for pointers to program data. Address expressions using the **BP** register have offsets in the **SS** segment by default. Expressions using **BX**, **SI**, or **DI** have offsets in the **DS** segment by default. The **DI** register always has an offset in the **ES** segment when used with string instructions.

The unnamed, 16-bit flag register contains nine 1-bit flags whose positions and meanings are defined in Table 5.2.

Table 5.2

Flag Positions

Flag Bit	Meaning
0	Carry flag
2	Parity flag
4	Auxiliary flag
6	Zero flag
7	Sign flag
8	Trap flag
9	Interrupt-enable flag
10	Direction flag
11	Overflow flag

Although the 16-bit flag register has no name, the contents of the register can be accessed using the **LAHF**, **SAHF**, **PUSHF**, and **POPF** instructions. See Appendix A.2, 8086 Instructions.

5.2.6 Based Operands

Syntax

displacement[**BP**]
displacement[**BX**]

A based operand represents a memory address relative to one of the base registers: **BP** or **BX**. The *displacement* can be any immediate or direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of a based operand is the sum of the displacement value and the contents of the given register. If **BP** is used, the operand's address is relative to the segment pointed to by the **SS** register. If **BX** is used, the address is relative to the segment pointed to by the **DS** register.

Based operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**BP**]
[**BP**+*displacement*]
[**BP**].*displacement*
[**BP**]+*displacement*

In each case, the effective address is the sum of the displacement and the contents of the given register.

Examples

```
mov     ax,[bp]
mov     ax,[bx]
mov     ax,12[bx]
mov     ax,fred[bp]
```

5.2.7 Indexed Operands

Syntax

displacement[**SI**]
displacement[**DI**]

An indexed operand represents a memory address relative to one of the index registers: **SI** or **DI**. The *displacement* can be any immediate or

direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of an indexed operand is the sum of the displacement value and the contents of the given register. The address is relative to the segment pointed to by the **DS** register.

Indexed operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**DI**]
[**DI**+*displacement*]
[**DI**].*displacement*
[**DI**]+*displacement*

In each case, the effective address is the sum of the displacement and the contents of the given register.

Examples

```
mov     ax,[si]
mov     ax,[di]
mov     ax,12[di]
mov     ax,fred[si]
```

5.2.8 Based-Indexed Operands

Syntax

displacement[**BP**][**SI**]
displacement[**BP**][**DI**]
displacement[**BX**][**SI**]
displacement[**BX**][**DI**]

A based-indexed operand represents a memory address relative to a combination of base and index registers. The *displacement* can be any immediate or direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of a based-indexed operand is the sum of the displacement value and the contents of the given registers. If the **BD** register is used, the address is relative to the segment pointed to by the **SS** register. Otherwise, the address is relative to the segment pointed to by the **DS** register.

Based-indexed operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**BP**][**DI**]
[**BP**+**DI**+*displacement*]
[**BP**+**DI**].*displacement*
[**DI**]+*displacement*+[**BP**]

In each case, the effective address is the sum of the displacement and the contents of the given registers. 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]
mov     ax,[bx+di]
mov     ax,12[bp+di]
mov     ax,fred[bx][si]
mov     ax,fred[bx][bp]    ; Error - base registers combined
mov     ax,fred[di][si]    ; Error - index registers combined
```

5.2.9 Structure Operands

Syntax

variable.field

A structure operand represents the memory address of one member of a structure. The *variable* must either be the name of a structure or it must be a memory operand that resolves to the address of a structure. The *field* must be the name of a field within that structure. The *variable* is separated from *field* by the structure field-name operator (.), which is described in Section 5.3.8.

The effective address of a structure operand 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.

Examples

```
date    STRUC
        month   DW  ?
        day     DW  ?
        year    DW  ?
date    ENDS

current_date  date <'ja','01','84'>

        mov     ax,current_date.day
        mov     current_date.year,'85'
```

In the example above, the structure is first defined and declared. The first **MOV** instruction puts `'01'` (the value of `current_date.day`) in the **AX** register. The next instruction puts the value `'85'` in the variable `current_date.year`.

```
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
```

In this example, structure operands are used to access values on the stack.

Note

The procedure in the example above does not conform to the method of passing parameters used in Microsoft high-level languages. As a result, you could not use the **SYMDEB** Stack Trace command (**K**) in this case procedure. See Section 4.6.27 in the *Microsoft Macro Assembler User's Guide.*

5.2.10 Record Operands

Syntax

recordname <[[*value*]],,,>

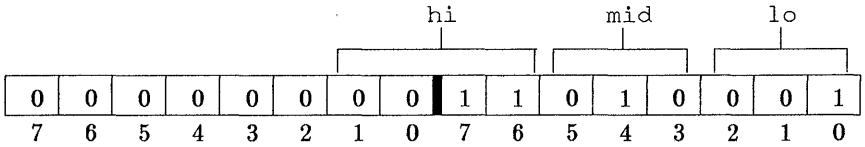
A record operand refers to the value of a record type. The operands can be in expressions. The *recordname* must be the name of a record type defined in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, the values must be separated by commas (,). Values include expressions or symbols that evaluate to constants. The enclosing angle brackets (< >) are required, even if no value is given. If no value for a field is given, the default value for that field is used. In the next example, assume the following record definition:

```
encode    RECORD hi:4, mid:3, lo:3
```

Example

```
rec1    encode <3,2,1>
        mov     ax,rec1
```

In this example, a constant with the value 209 (0D1h) is moved into the **AX** register. The following bit diagram illustrates how the value is obtained:



Using record operands is similar to declaring a record and then using the declared data except that, in using record operands, you are using constant data. See Section 4.6.2 for information on declaring record data.

5.2.11 Record-Field Operands

Syntax

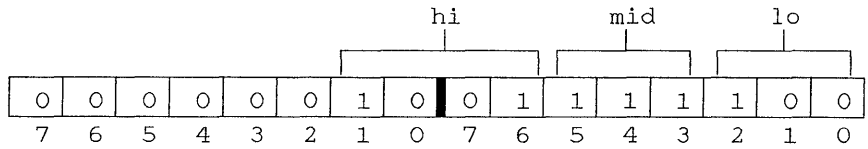
record-fieldname

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

The *record-fieldname* must be the name of a previously defined record field. In the next example, assume the following record definition and declaration:

```
encode  RECORD  hi:4, mid:3, lo:3
rec1    encode  <9,7,4>
```

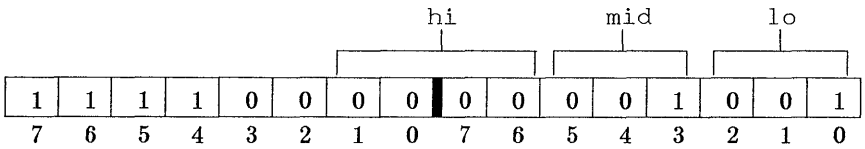
At this point `rec1` has a value of 636 (27Ch), shown in this bit diagram:



Example

```
mov     cl,hi
mov     dx,rec1
ror     dx,cl
mov     rec1,dx
```

This example copies 6, the shift count for `hi`, to register **CL**. The contents of `rec1` are copied to **DX**. The shift count of field three (`hi`) is then used to rotate the value of `rec1` so that the value of `hi` is now at the lowest bit. The new value is then put back into `rec1`. At this point `rec1` has a value of 61449 (0F009h), as shown in the bit diagram below.



5.3 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 this chapter. The result of an expression can be a value or a memory location, depending on the types of operands and operators used.

The assembler provides a variety of operators. Arithmetic, shift, relational, and bitwise operators manipulate and compare the values of operands. Attribute operators manipulate the attributes of operands, such as their type, address, and size.

Sections 5.3.1–5.3.4 describe the arithmetic, relational, and logical operators in detail. Attribute operators are described in Sections 5.3.5–5.3.19. In addition to the operators described here, you can use the **DUP** operator (Section 4.3.6) and the special macro operators (Section 8.3).

5.3.1 Arithmetic Operators

Syntax

*expression1***expression2*
expression1/*expression2*
*expression1***MOD***expression2*
expression1+*expression2*
expression1–*expression2*
+*expression*
–*expression*

Arithmetic operators provide the common mathematical operations. Table 5.3 lists the operators and their meanings.

Table 5.3
Arithmetic Operators

Operator	Meaning
+	Positive (unary)
–	Negative (unary)
*	Multiplication
/	Integer division
MOD	Remainder after division (modulus)
+	Addition
–	Subtraction

For all arithmetic operators except + and –, *expression1* and *expression2* must be integer numbers. The + operator can be used to add an integer number to a relocatable memory operand. The – operator can be used to subtract an integer number from a relocatable memory operand. The – operator can also be used to subtract one relocatable operand from another, but only if the operands refer to locations within the same segment. The result is an absolute value.

Note

The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to designate addition or subtraction). The unary plus and minus have a higher level of precedence, as shown in Table 5.7 in Section 5.4.

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             ; Add 5 to alpha's offset
```

```
alpha - 5                ; Subtract 5 from alpha's offset
alpha - beta             ; Subtract beta's offset from alpha's
```

5.3.2 SHR and SHL Operators

Syntax

expression **SHR** *count*
expression **SHL** *count*

The **SHR** and **SHL** operators shift *expression* right or left by *count* number of bits. Bits shifted off the end of the expression are lost. If the count is greater than or equal to 16, the result is 0. The bits will be shifted by 8 or 16 bits, depending on whether the value being shifted is a word or a byte.

Note

Do not confuse the assembler's **SHR** and **SHL** operators with the processor instructions having the same names.

Examples

```
mov ax,01110111b SHL 3          ; Move 0000000111011100b
mov ah,01110111b SHR 3          ; Move 00001110b
```

Notice that 16 bits are shifted into a word register (ax) in the first example. In the second example, only 8 bits are shifted because the register (ah) holds only 1 byte.

5.3.3 Relational Operators

Syntax

expression1 **EQ** *expression2*
expression1 **NE** *expression2*
expression1 **LT** *expression2*
expression1 **LE** *expression2*
expression1 **GT** *expression2*
expression1 **GE** *expression2*

The relational operators compare *expression1* and *expression2* and return true (0FFFFh) if the condition specified by the operator is satisfied, or false (0000h) if it is not. The expressions must resolve to absolute values. Table 5.4 lists the operators and the values they return if the specified condition is satisfied.

Table 5.4

Relational Operators

Operator	Returned Value
EQ	True (0FFFFh) if expressions are equal.
NE	True (0FFFFh) if expressions are not equal.
LT	True (0FFFFh) if left expression is less than right.
LE	True (0FFFFh) if left expression is less than or equal to right.
GT	True (0FFFFh) if left expression is greater than right.
GE	True (0FFFFh) if left expression is greater than or equal to right.

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

Note

The **EQ** and **NE** operators treat their arguments as 16-bit numbers. Numbers specified with the 16th bit on are considered negative (0FFFFh is -1). Therefore, the expression `-1 EQ 0FFFFh` is true, while the expression `-1 NE 0FFFFh` is false.

The **LT**, **LE**, **GT**, and **GE** operators treat their arguments as 17-bit numbers, where the 17th bit specifies the sign. Therefore, 0FFFFh is the largest positive unsigned number (65535); it is not –1. The expression `1 GT -1` is true (0FFFFh), while the expression `1 GT 0FFFFh` is false (0).

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
```

5.3.4 Bitwise Operators

Syntax

NOT *expression*
expression1 **AND** *expression2*
expression1 **OR** *expression2*
expression1 **XOR** *expression2*

The logical operators perform bitwise operations on expressions. In a bitwise operation, 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.

Table 5.5 lists the logical operators and their meanings:

Table 5.5

Logical Operators

Operator	Meaning
NOT	Inverse
AND	Boolean AND
OR	Boolean OR
XOR	Boolean exclusive OR

Examples

```
NOT   11110000b       ; Equals 1111111100001111b or 00001111b
01010101b   AND   11110000b     ; Equals 01010000b
01010101b   OR    11110000b     ; Equals 11110101b
01010101b   XOR   11110000b     ; Equals 10100101b
```

5.3.5 Index Operator

Syntax

[*expression1*][*expression2*]

The index operator, [ ], adds the value of *expression1* to *expression2*. This operator is identical to the + operator, except that *expression1* is optional.

If *expression1* is given, the expression must appear to the left of the operator. It can be any integer value, absolute symbol, or relocatable operand. If no *expression1* is given, the integer value 0 is assumed. If *expression1* is a relocatable operand, *expression2* must be an integer value or absolute symbol. Otherwise, *expression2* can be any integer value, absolute symbol, or relocatable operand.

The index operator is typically used to index elements of an array, such as individual characters in a character string.

Examples

```
mov   al,string[3]      ; Move 4th element of string
mov   ax,array[4]       ; Move 5th element of array
mov   string[last],al   ; Move into LAST element of string
mov   cx,DGROUP:[1]     ; Move 2nd byte of DGROUP
```

Note that the last example is identical to the following statement:

```
mov   cx, dgroup:1.
```

5.3.6 PTR Operator

Syntax

type **PTR** *expression*

The **PTR** operator forces the variable or label given by *expression* to be treated as a variable or label having the type given by *type*. The type must be one of the following names or values:

Type	Value
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

The expression can be any operand. The **BYTE**, **WORD**, and **DWORD** types can be used with memory operands only. The **NEAR** and **FAR** types can be used with labels only.

The **PTR** operator is typically used with forward references to explicitly define what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. The **PTR** operator is also used to enable instructions to access variables in ways that would otherwise generate errors. For example, you could use the **PTR** operator to access the high-order byte of a **WORD** size variable.

Section 5.6 discusses how the **PTR** operator can be used to avoid errors associated with strong type checking. These errors include `Illegal size for item` and `Operand types must match`.

Examples

```
call    FAR PTR subrout3
mov     BYTE PTR [array],1
add     al,BYTE PTR [full_word]
```

In these examples the **PTR** operator overrides a previous data declaration. The procedure `subrout3` might have been declared **NEAR**, while `array` and `full_word` could have been declared with the **DW** directive.

5.3.7 Segment-Override Operator

Syntax

segmentregister:*expression*
segmentname:*expression*
groupname:*expression*

The segment-override operator (:) forces the address of a given variable or label to be computed using the beginning of the given *segmentregister*, *segmentname*, or *groupname*. If either *segmentname* or *groupname* is given, the name must have been assigned to a segment register with a previous **ASSUME** directive and defined using a **SEGMENT** or **GROUP** directive. The *expression* can be an absolute symbol or relocatable operand. The *segmentregister* must be **CS**, **DS**, **SS**, or **ES**.

By default, the effective address of a memory operand is computed relative to the **DS**, **SS**, or **ES** register, depending on the instruction and operand type. Similarly, all labels are assumed to be **NEAR**. These default types can be overridden using the segment-override operator.

Examples

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

5.3.8 Structure Field-Name Operator

Syntax

variable.field

The structure field-name operator (.) is used to designate a field within a structure. The *variable* is an operand (often a previously declared structure variable) and *field* is the name of a field within a structure. This operator is equivalent to the addition operator (+) in based or indexed operands.

Example

```
inc     month.day
mov     time.min,0
mov     [bx].dest
```

5.3.9 SHORT Operator

Syntax

SHORT *label*

The **SHORT** operator sets the type of the given *label* to **SHORT.** Short labels can be used in **JMP** instructions whenever the distance from the label to the instruction is not more than 127 bytes. Instructions using short labels are 1 byte smaller than identical instructions using near labels.

Example

```
jmp     SHORT do_again  ; Jump less than 128 bytes
```

5.3.10 THIS Operator

Syntax

THIS *type*

The **THIS** operator creates an operand whose offset and segment values are equal to the current location-counter value and whose type is given by *type.* The *type* can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR

The **THIS** operator is typically used with the **EQU** or equal-sign (=) directive to create labels and variables. This is similar to using the **LABEL** directive to create labels and variables.

Examples

```
tag   EQU     THIS BYTE
```

The preceding example is equivalent to the statement `tag LABEL BYTE`.

```
check =       THIS NEAR
```

The final example is equivalent to the statement `check LABEL NEAR`.

5.3.11 HIGH and LOW Operators

Syntax

HIGH *expression*
LOW *expression*

The **HIGH** and **LOW** operators return the high and low 8 bits, respectively, of *expression*. The **HIGH** operator returns the high-order 8 bits of *expression*; the **LOW** operator returns the low-order 8 bits. The expression can be any value.

Examples

```
mov     ah,HIGH word_value      ; Move high byte of word_value
mov     al,LOW OABCDh           ; Move OCDh
```

5.3.12 SEG Operator

Syntax

SEG *expression*

The **SEG** operator returns the segment value of *expression*. The expression can be any label, variable, segment name, group name, or other symbol.

Examples

```
mov     ax,SEG variable_name
mov     ax,SEG label_name
```

5.3.13 OFFSET Operator

Syntax

OFFSET *expression*

The **OFFSET** operator returns the offset of *expression*. The expression can be any label, variable, segment name, or other symbol. The returned value is the number of bytes between the item and the beginning of the segment in which it is defined. For a segment name, the returned value is the offset from the start of the segment to the most recent byte generated for that segment.

The segment-override operator (:) can be used to force **OFFSET** to return the number of bytes between the item in *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
```

The returned value is always a relative value that is subject to change by the linker when the program is actually linked.

5.3.14 TYPE Operator

Syntax

TYPE *expression*

The **TYPE** operator returns a number representing the type of *expression*. If *expression* is a variable, the operator returns the size of the operand in bytes. If *expression* is a label, the operator returns 0FFFFh if the label is **NEAR**, and 0FFFEh if the label is **FAR**. Note that the returned value can be used to specify the type for a **PTR** operator, as in the second of the following two examples.

Examples

```
mov     ax,TYPE array
jmp     (TYPE get_loc) PTR destiny
```

5.3.15 .TYPE Operator

Syntax

.TYPE *expression*

The **.TYPE** operator returns a byte that defines the mode and scope of *expression*. If *expression* is not valid, **.TYPE** returns a 0.

Table 5.6 lists the variable's attributes as returned in bits 0, 1, 5, and 7.

Table 5.6

.TYPE Operator and Variable Attributes

Bit Position	If Bit = 0	If Bit = 1
0	Not program-related	Program-related
1	Not data-related	Data-related
5	Not defined	Defined
7	Local or public scope	External scope

If both the scope bit and defined bit are zero, *expression* is not valid.

The **.TYPE** operator is typically used with conditional directives, where an argument may need to be tested in order to make a decision regarding program flow.

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 1 is set because x is data-related. Bit 5 is set

because x is defined. Bit 7 is not set because x is local. The remaining bits are never set.

5.3.16 LENGTH Operator

Syntax

LENGTH *variable*

The **LENGTH** operator returns the number of **BYTE**, **WORD**, **DWORD**, **QWORD**, or **TBYTE** elements in *variable*. The size of each element depends on the variable's defined type.

Only variables defined using the **DUP** operator return values that are greater than 1. The returned value is always the number preceding the first **DUP** operator.

In the next two examples, assume the following definitions:

```
array  DW    100   DUP(1)
table  DW    100   DUP(1,10 DUP(?))
```

Examples

```
        mov     cx,LENGTH array
```

In the preceding example, LENGTH returns 100.

```
        mov     cx,LENGTH table
```

In the final example, **LENGTH** returns 100. The returned value does not depend on any nested **DUP** operators.

5.3.17 SIZE Operator

Syntax

SIZE *variable*

The **SIZE** operator returns the total number of bytes allocated for *variable*. The returned value is equal to the value of **LENGTH** times the value of **TYPE**.

In the next example, assume the following definition:

```
array DW    100    DUP(1)
```

Example

```
      mov   bx,SIZE array
```

In this example, **SIZE** returns 200.

5.3.18 WIDTH Operator

Syntax

WIDTH *recordfieldname|record*

The **WIDTH** operator returns the width (in bits) of the given record field or record. The *recordfieldname* must be the name of a field defined in a record. The *record* must be the name of a record.

In the next examples, assume the following record definition and record declaration:

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

Examples

```
wid1     =  WIDTH field1      ; Equals 3
wid2     =  WIDTH field2      ; Equals 6
wid3     =  WIDTH field3      ; Equals 7
widrec   =  WIDTH rtype       ; Equals 16
```

Remember, the field name represents the bit count. For example, `field1` equals 13 (the width of `field2` plus the width of `field3`) while `WIDTH field1` equals 3.

5.3.19 MASK Operator

Syntax

MASK *recordfieldname|record*

The **MASK** operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a field bit. All other bits contain 0.

The *recordfieldname* must be the name of a field defined in a record.

In the next example, assume the following record definition and record declaration:

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

Example

```
m1   =  MASK field1  ; Equals E000h  (1110000000000000b)
m2   =  MASK field2  ; Equals 1F80h  (1111110000000b)
m3   =  MASK field3  ; Equals 007Fh  (1111111b)
mrec =  MASK rtype   ; Equals 0FFFFh (1111111111111111b)
```

5.4 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 by using enclosing parentheses. Operations in parentheses are always performed before any adjacent operations. Table 5.7 lists the precedence of all operators. Operators on the same line have equal precedence.

Table 5.7

Operator Precedence

Precedence	Operators
(Highest)	
1	**LENGTH, SIZE, WIDTH, MASK, (), [], <>**
2	. (structure field-name operator)
3	:
4	**PTR, OFFSET, SEG, TYPE, THIS**
5	**HIGH, LOW**
6	+,– (unary)
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 0FFFFh (true)
```

5.5 Forward References

Although the assembler 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 declared. For example, in the **JMP** instruction below, the label `target` is a forward reference.

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

Whenever the assembler encounters an undefined name in Pass 1, it assumes that the name is a forward reference. If only a name is given, the assembler 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 3 bytes of instruction code for the instruction.

The assembler 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 the assembler in Pass 1 would cause a phase error. In other words, the assembler would generate 5 bytes of instruction code for the **JMP** instruction in Pass 2 but only 3 in Pass 1.

To avoid errors with forward references, the segment override (:), **PTR**, and **SHORT** operators should be used whenever necessary to override the assumptions made by the assembler. The following guidelines list situations in which 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, the assembler assumes that the variable is relative to the **DS** register.

- 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, the assembler assumes that the instruction is greater than 128 bytes away. This does not cause an error, but it does cause the assembler to generate an extra, and unnecessary, **NOP** instruction.

- 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     FAR PTR exit
```

The assembler assumes that the label has **NEAR** type, so **PTR** need not be used for **NEAR** labels. If the label has **FAR** type, however, and **FAR 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  GROUP   stack
        ASSUME  ss:dgroup

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

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

5.6 Strong Typing for Memory Operands

The assembler 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 result in an `Operand types must match` error message 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 following statement will assemble correctly and execute as expected:

```
mov   ax,WORD PTR string[1]
```

Note

Many assembly-language program listings in books and magazines are written for assemblers with weak typing for operands. These programs may produce error messages such as `Illegal size for item` or `Operand types must match` when assembled as listed using the Microsoft Macro Assembler. You can correct lines that produce errors by using the **PTR** operator to assign the correct size to variables.

Chapter 6
Global Declarations

6.1 Introduction

The global-declaration directives allow you to define labels, variables, and absolute symbols that can be accessed globally, that is, from all modules in a program. Global declarations transform "local" symbols (labels, variables, and other symbols that are specific to the source files in which they are defined) into "global" symbols that are available to all other modules of the program.

The two global-declaration directives are **PUBLIC** and **EXTRN**. The **PUBLIC** directive is used in public declarations, which transform locally defined symbols into global symbols, making them available to other modules. The **EXTRN** directive is used in external declarations, making a global symbol's name and type known in a source file so that the global symbol may be used in that file. Every global symbol must have a public declaration in exactly one source file of the program. A global symbol can have external declarations in any number of other source files. Sections 6.2–6.4 describe and demonstrate the global-declaration directives in detail.

6.2 PUBLIC Directive

Syntax

PUBLIC *name,,,*

The **PUBLIC** directive makes the variable, label, or absolute symbol specified by *name* available to all other modules in the program. The name must be the name of a variable, label, or absolute symbol defined within the current source file. Absolute symbols, if given, can only represent 1- or 2-byte integer or string values.

The assembler converts all lowercase letters in *name* to uppercase before copying the name to the object file. The **/ML** and **/MX** options can be used in the **MASM** command line to direct the assembler to preserve lowercase letters when copying public and external symbols to the object file. Sections 2.3.7 and 2.3.8 of the *Microsoft Macro Assembler User's Guide* describe the **/ML** and **/MX** options.

Symbols must be declared public before they can be used for symbolic debugging. See Section 4.2 of the *Microsoft Macro Assembler User's Guide* for details on how to prepare and use symbol files with **SYMDEB**.

Example

```
        PUBLIC   true,status,start,clear
true    =        OFFFFH
status  DB       1
start   LABEL    FAR
clear   PROC     NEAR
```

The values declared public in this example include an absolute symbol, a variable, a label, and a procedure.

6.3 EXTRN Directive

Syntax

EXTRN *name:type,,,*

The **EXTRN** directive defines an external variable, label, or symbol of the specified *name* and *type*. An external item is any variable, label, or symbol that has been declared with a **PUBLIC** directive in another module of the program. The *type* must match the type given to the item in its actual definition. It can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR
ABS

The **ABS** type is for symbols that represent absolute numbers.

Although the actual address is not determined until the object files are linked, the assembler may assume a default segment for the external item, based on where the **EXTRN** directive is placed in the module. If the directive is placed inside a segment, the external item is assumed to be relative to that segment, and the item's public declaration (in some other module)

must be in a segment having the same name and attributes. If the directive is outside all segments, no assumption is made about what segment the item is relative to, and the item's public declaration can be in any segment in any module. In either case, the segment-override operator (:) can be used to override the default segment of an external variable or label.

Example

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

6.4 Program Example

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules, named `main` and `task`. The `main` module is the program's initializing module. Execution starts at the instruction labeled `start` in `main`, and passes to the instruction labeled `print` in `task`. An MS-DOS system call in the `task` module is used to print `Hello` on the screen. Execution then returns to the instruction labeled `exit` in the `main` module.

Main Module

```
        NAME    main
        PUBLIC  exit
        EXTRN   print:near

stack   SEGMENT word stack 'STACK'
        DW      64 DUP(?)
stack   ENDS

data    SEGMENT word public 'DATA'
data    ENDS

code    SEGMENT byte public 'CODE'
        ASSUME  cs:code,ds:data
start:
        mov     ax,data             ; Load segment location
        mov     ds,ax               ;   into DS register
        jmp     print               ; Go to PRINT in other module
```

```
exit:
        mov     ah, 4Ch          ; Call terminate function
        int     21h
code    ENDS
        END     start
```

Task Module

```
        NAME    task
        PUBLIC  print
        EXTRN   exit:near

data    SEGMENT word public 'DATA'
string  DB      "Hello",13,10,"$"
data    ENDS

code    SEGMENT byte public 'CODE'
        ASSUME  cs:code, ds:data
print:
        mov     dx,OFFSET string ; Load string location
        mov     ah,09h           ; Call string display function
        int     21h
        jmp     exit             ; Go back to other module
code    ENDS
        END
```

In this example, the symbol `exit` is declared public in the `main` module so that it can be accessed from another source module (`task` in the example). The `main` module also contains an external declaration of the symbol `print`. This declaration defines `print` to be a near label so that it can be accessed from the `main` module, even though it is assumed to be located and declared public in another source module. A **JMP** instruction later in the module has this label as its destination.

The symbol `print` is declared public in the `task` module so that it can be accessed from another module (`main` in the example). The symbol `exit` is defined as a near label so that it can be accessed from this module, even though it is assumed to be located and declared public in the other module.

Before this program can be executed, these source files must be assembled individually, then linked together using **LINK**.

Chapter 7
Conditional Directives

7.1 Introduction

The Microsoft Macro Assembler provides two types of conditional directives. Conditional-assembly directives test for a specified condition and assemble a block of statements if the condition is true. Conditional error directives test for a specified condition and generate an error if the condition is true.

Both kinds of conditional directives only test assembly-time conditions. They cannot test run-time conditions since these are not known until an executable program is run. Only expressions that evaluate to constants during assembly can be compared or tested.

Since macros and conditional-assembly directives are often used together, you may need to refer to Chapter 8 to understand some of the examples in this chapter. In particular, conditional directives are frequently used with the special macro operators described in Section 8.3.

7.2 Conditional-Assembly Directives

The conditional-assembly directives include the following:

IF
IFE
IF1
IF2
IFDEF
IFNDEF
IFB
IFNB
IFIDN
IFDIF
ELSE
ENDIF

The **IF** directives and the **ENDIF** and **ELSE** directives can be used to

enclose the statements to be considered for conditional assembly. The conditional block takes the following form:

IF
statements
[**ELSE**
statements]
ENDIF

The *statements* following **IF** can be any valid statements, including other conditional blocks. The **ELSE** directive and its *statements* are optional. **ENDIF** ends the block.

The statements in the conditional block are assembled only if the condition specified by the corresponding **IF** directive is satisfied. If the conditional block contains an **ELSE** directive, only the statements up to the **ELSE** directive will be assembled. The statements following the **ELSE** directive are assembled only if the **IF** condition is not met. An **ENDIF** directive must mark the end of any conditional-assembly block. No more than one **ELSE** directive is allowed for each **IF** directive.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested **ELSE** directive always belongs to the nearest preceding **IF** directive that does not have its own **ELSE**.

7.2.1 IF and IFE Directives

Syntax

IF *expression*
IFE *expression*

The **IF** and **IFE** directives test the value of an *expression*. The **IF** directive grants assembly if the value of *expression* is true (nonzero). The **IFE** directive grants assembly if the value of *expression* is false (0). 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
```

In this example, the variables within the block will only be declared external if the symbol debug evaluates to true (nonzero).

7.2.2 IF1 and IF2 Directives

Syntax

IF1
IF2

The **IF1** and **IF2** directives test the current assembly pass. The **IF1** directive grants assembly only on Pass 1. **IF2** grants assembly only on Pass 2. The directives take no arguments.

Example

```
IF1
      %OUT Beginning Pass 1
ELSE
      %OUT Beginning Pass 2
ENDIF
```

7.2.3 IFDEF and IFNDEF Directives

Syntax

IFDEF *name*
IFNDEF *name*

The **IFDEF** and **IFNDEF** directives test whether or not the given *name* has been defined. The **IFDEF** directive grants assembly only if *name* is a label, variable, or symbol. The **IFNDEF** directive 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.

Example

```
IFDEF   buffer
        buf1   DB 10 DUP(?)
ENDIF
```

In this example, `buf1` is allocated only if `buffer` has been previously defined. One way to use this conditional block would be to leave `buffer` undefined in the source file and define it if you needed it by using the **/D***symbol* option when you start **MASM**. For example, if the conditional block is in `test.asm`, you could start the assembler with the command line:

```
MASM test /Dbuffer;
```

The symbol `buffer` would be defined, and as a result the conditional-assembly block would allocate `buf1`. However, if you didn't need `buf1`, you could use the command line:

```
MASM test;
```

7.2.4 IFB and IFNB Directives

Syntax

IFB <*argument*>
IFNB <*argument*>

The **IFB** and **IFNB** directives test *argument*. The **IFB** directive grants assembly if *argument* is blank. The **IFNB** directive grants assembly if *argument* is not blank. The arguments can be any name, number, or expression. The angle brackets (< >) are required.

The **IFB** and **IFNB** directives are intended for use in macro definitions. They can control conditional-assembly of statements in the macro, based on the parameters passed in the macro call. In such cases, *argument* should be one of the dummy parameters listed by the **MACRO** directive.

Example

```
pushall     MACRO   reg1,reg2,reg3,reg4,reg5,reg6
            IFNB    <reg1>          ;; If parameter not blank
                    push    reg1    ;;   push one register and repeat
                    pushall reg2,reg3,reg4,reg5,reg6
            ENDIF
            ENDM

pushall     ax,bx,si,ds
pushall     cs,es
```

In this example, `pushall` is a recursive macro that continues to call itself until it encounters a blank argument. Any register or list of registers (consisting of up to six registers) can be passed to the macro for pushing.

7.2.5 IFIDN and IFDIF Directives

Syntax

IFIDN <*argument1*>,<*argument2*>
IFDIF <*argument1*>,<*argument2*>

The **IFIDN** and **IFDIF** directives compare *argument1* and *argument2*. The **IFIDN** directive grants assembly if the arguments are identical. The **IFDIF** directive grants assembly if the arguments are different. The arguments can be any names, numbers, or expressions. To be identical, each character in *argument1* must match the corresponding character in *argument2*. Case is significant. The angle brackets (< >) are required. The arguments must be separated by a comma (,).

The **IFIDN** and **IFDIF** directives are intended for use in macro definitions. They can control conditional assembly of macro statements, based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the **MACRO** directive.

Example

```
divide    MACRO   numerator, denominator
          IFDIF   <denominator>,<0>   ;; If not dividing by zero
          mov     ax, numerator       ;;   divide AX by BX
          mov     bx, denominator
          div     bx                  ;; Result in accumulator
          ENDIF
          ENDM

divide    6,%test
```

In this example, a macro uses the **IFDIF** directive to check against dividing by a constant that evaluates to 0. The macro is then called, using a percent sign (%) on the second parameter so that the value of the parameter, rather than its name, will be evaluated. See Section 8.3.4 for a discussion of the expression (%) operator.

If the parameter `test` was previously defined with the statement

```
test        EQU     0
```

then the condition fails and the code in the block will not be assembled. However, if the parameter `test` was defined with the statement

```
test        DW      0
```

error 42, `Constant was expected`, will be generated. This is because the assembler has no way of knowing the run-time value of `test`. Remember, conditional directives can only evaluate constants that are known at assembly time.

7.3 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 Table 7.1.

Table 7.1

Conditional Error Directives

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.

7.3.1 .ERR, .ERR1, and .ERR2 Directives

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.

7.3.2 .ERRE and .ERRNZ Directives

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.

7.3.3 .ERRDEF and .ERRNDEF Directives

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.

7.3.4 .ERRB and .ERRNB Directives

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.

7.3.5 .ERRIDN and .ERRDIF Directives

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.

Chapter 8
Macro Directives

8.1 Introduction

This chapter explains how to create and use macros in your source files. It discusses the macro directives and the special macro operators. Since macros are closely related to conditional directives, you may need to review Chapter 7 to follow some of the examples in this chapter.

Macro directives enable you to write a named block of source statements, then use that name in your source file to represent the statements. During assembly, **MASM** automatically replaces each occurrence of the macro name with the statements in the macro definition. You can place a block of statements anywhere in your source file any number of times by simply defining a macro block once, then inserting the macro name at each location where you want the macro block to be assembled. You can also pass parameters to macros.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls that macro. Macros can be kept in a separate file and made available to the program through an **INCLUDE** directive (see Section 9.2).

Often a task can be done by either a macro or procedure. For example, the `Addup` procedure shown in Section 3.10 does the same thing as the `Addup` macro in Section 8.2.1. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower.

8.2 Macro Directives

The macro directives are listed below:

MACRO
ENDM
LOCAL
PURGE
REPT

IRP

IRPC

EXITM

The **MACRO** and **ENDM** directives designate the beginning and end of a macro block. The **LOCAL** directive lets you define labels used only within a macro, and the **PURGE** directive lets you delete previously defined macros. The **EXITM** directive allows you to exit from a macro before all the statements in the block are expanded.

The **REPT**, **IRP**, and **IRPC** directives let you create contiguous blocks of repeated statements. These repeat blocks are frequently placed within macros, but they can also be used independently. You can control the number of repetitions by specifying a number; or by allowing the block to be repeated once for each parameter in a list; or by having the block repeated once for each character in a string.

8.2.1 MACRO and ENDM Directives

Syntax

name **MACRO** [[*dummyparameter*,,,]]
statements
ENDM

The **MACRO** and **ENDM** directives create a macro having *name* and containing the given *statements.*

The name must be a valid name and must be unique. It is used in the source file to invoke the macro. The *dummyparameter* is a name that acts as a placeholder for values to be passed to the macro when it is called. Any number of *dummyparameters* can be specified, but they must all fit on one line. If you give more than one, you must separate them with commas (,). The statements are any valid **MASM** statements, including other macro directives. Any number of statements can be used. The dummy parameters can be used any number of times in these statements.

A macro is "called" any time its name appears in a source file (macro names in comments are ignored). **MASM** copies the statements in the macro definition to the point of the call, replacing any dummy parameters in these statements with actual parameters passed in the call.

Macro definitions can be nested. This means a macro can be defined within another macro. **MASM** does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Macro definitions can contain calls to other macros. These nested macro calls are expanded like any other macro call, but only when the outer macro is called. Macro definitions can also be recursive: they can call themselves, as illustrated in the example in Section 7.2.4.

Example

```
addup   MACRO   ad1,ad2,ad3
        mov     ax, ad1         ;; First parameter in AX
        add     ax, ad2         ;; Add next two parameters
        add     ax, ad3         ;;   and leave sum in AX
        ENDM
```

The preceding example defines a macro named addup, which uses three dummy parameters to add three values and leave their sum in the **AX** register. The three dummy parameters will be replaced with actual values when the macro is called.

MASM assembles the statements in the macro only if the macro is called, and only at the point in the source file from which it is called. Thus, all addresses in the assembled code will be relative to the macro call, not the macro definition. The macro definition itself is never assembled.

You must be careful when using the word **MACRO** after the **TITLE**, **SUBTTL**, and **NAME** directives. Since the **MACRO** directive overrides these directives, placing the word macro immediately after these directives would cause the assembler to begin to create macros named **TITLE**, **SUBTTL**, and **NAME**. For example, the line:

```
TITLE Macro File
```

may be intended to give an include file the title "Macro File", but its effect will be to create a macro called TITLE that accepts the dummy parameter File. Since there will be no corresponding **ENDM** directive, an error will usually result.

To avoid this problem, you should alter the word macro in some way when using it in a title or name. For example, change the spelling or add an underline character (MAKRO or _MACRO).

Note

MASM replaces all occurrences of a dummy parameter's name, even if you do not intend it to. For example, if you use a register name such as **AX** or **BH** for a dummy parameter, **MASM** replaces all occurrences of that register name when it expands the macro. If the macro definition contains statements that use the register, not the dummy, the macro will be incorrectly expanded.

Note

Macros can be redefined. You need not purge the first macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no lines between the **ENDM** directive of the nested redefinition and the **ENDM** directive of the original macro. The following example may produce incorrect code:

```
dostuff     MACRO
            .
            .
            .
            dostuff     MACRO
                        .
                        .
                        .
                        ENDM
            ;; Comments or statements not allowed
            ENDM
```

To correct the error, remove the line between the **ENDM** directives.

8.2.2 Macro Calls

Syntax

name [[*actualparameter*,,,]]

A macro call directs **MASM** to copy the statements of the macro *name* to the point of call and to replace any dummy parameters in these statements with the corresponding actual parameters. The *name* must be the name of a macro defined earlier in the source file. The *actualparameter* can be any name, number, or other value. Any number of actual parameters can be given, but they must all fit on one line. Multiple parameters must be separated by commas, spaces, or tabs.

MASM replaces the first dummy parameter with the first actual parameter, the second with the second, and so on. If a macro call has more actual parameters than dummy parameters, the extra actual parameters are ignored. If a call has fewer actual parameters than dummy parameters, any remaining dummy parameters are replaced with a null (blank) string. You can use the **IFB**, **IFNB**, **.ERRB**, and **.ERRNB** directives to have your macros check for null strings and take appropriate action. See Sections 7.2.4 and 7.3.4.

If you wish to pass a list of values as a single actual parameter, you must place angle brackets (< >) around the list. The items in the list must be separated by commas (,).

Examples

```
allocblock 1,2,3,4,5
```

The first example passes five numeric parameters to the macro called `allocblock`.

```
allocblock <1,2,3,4,5>
```

The second example passes one parameter to `allocblock`. The parameter is a list of five numbers.

```
addup       bx, 2, count
```

The final example passes three parameters to the macro `addup`. **MASM** replaces the corresponding dummy parameters with exactly what is typed in the macro call parameters. Assuming that `addup` is the same macro defined at the end of Section 8.2.1, the assembler would expand the macro to the following code:

```
mov     ax, bx
add     ax, 2
add     ax, count
```

See Section 2.4 of the *Microsoft Macro Assembler User's Guide* for an example of how macros are shown in listing files.

8.2.3 LOCAL Directive

Syntax

LOCAL *dummyname,,,*

The **LOCAL** directive creates unique symbol names for use in macros. The *dummyname* is a name for a placeholder that is to be replaced by a unique name when the macro is expanded. At least one *dummyname* is required. If you give more than one, you must separate the names with commas (,). A *dummyname* can be used in any statement within the macro.

MASM creates a new actual name for the dummy name each time the macro is expanded. The actual name has the following form:

??number

The *number* is a hexadecimal number in the range 0000 to FFFF. Do not give other symbols names in this format, since doing so will produce a label or symbol with multiple definitions. In listings, the dummy name is shown in the macro definition, but the actual names are shown for each expansion of the macro.

The **LOCAL** directive is typically used to create a unique label that will only be used in a macro. Normally, if a macro containing a label is used more than once, **MASM** will display an error message indicating the file contains a label or symbol with multiple definitions, since the same label will appear in both expansions. To avoid this problem, all labels in macros should be dummy names declared with the **LOCAL** directive.

Note

The **LOCAL** directive can be used only in a macro definition, and it must precede all other statements in the definition. If you try to put a comment line or an instruction before the **LOCAL** directive, a warning error will result.

Example

```
power     MACRO   factor,exponent
          LOCAL   again,gotzero   ;; Declare symbols for macro
          mov     cx,exponent     ;; Exponent is count for loop
          mov     ax,1            ;; Multiply by 1 first time
          jcxz    gotzero         ;; Get out if exponent is zero
          mov     bx,factor
again:    mul     bx              ;; Multiply until done
          loop    again
gotzero:
          ENDM
```

In this example, the **LOCAL** directive defines the dummy names `again` and `gotzero`. These names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, `again` will be assigned the name `??0000` and `gotzero` will be assigned `??0001`. The second time through `again` will be assigned `??0002` and `gotzero` will be assigned `??0003`, and so on.

8.2.4 PURGE Directive

Syntax

PURGE *macroname,,,*

The **PURGE** directive deletes the current definition of the macro called *macroname*. Any subsequent call to that macro causes the assembler to generate an error.

The **PURGE** directive is intended to clear memory space no longer needed by a macro. If *macroname* is an instruction or directive mnemonic, the directive name is restored to its previous meaning.

The **PURGE** directive is often used with a "macro library" to let you choose those macros from the library that you really need in your source file. A macro library is simply a file containing macro definitions. You add this library to your source file using the **INCLUDE** directive, then remove unwanted definitions using the **PURGE** directive.

It is not necessary to **PURGE** a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, any macro can purge itself as long as the **PURGE** directive is on the last line of the macro.

Examples

```
PURGE    addup
```

The first example deletes the macro named `addup`.

```
PURGE    mac1, mac2, mac9
```

The second example deletes the macros named `mac1` , `mac2` , and `mac9`.

8.2.5 REPT and ENDM Directives

Syntax

REPT *expression*
statements
ENDM

The **REPT** and **ENDM** directives enclose a block of *statements* to be repeated *expression* number of times. The expression must evaluate to a 16-bit unsigned number. It must not contain external or undefined symbols. The statements can be any valid statements.

Example

```
x       =       0
        REPT    10
x       =       x + 1
        DB      x
        ENDM
```

This example repeats the equal-sign (=) and **DB** directives 10 times. The resulting statements create 10 bytes of data whose values range from 1 to 10.

8.2.6 IRP and ENDM Directives

Syntax

IRP *dummyname*,<*parameter*,,,>
statements
ENDM

The **IRP** and **ENDM** directives designate a block of *statements* to be repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummyname* is a name for a placeholder to be replaced by the current *parameter*. The parameter can be any legal symbol, string, numeric, or character constant. Any number of parameters can be given. If you give more than one parameter, you must separate them with commas (,). The angle brackets (< >) around the parameter list are required. The *statements* can be any valid assembler statements. The *dummyname* can be used any number of times in these statements.

When **MASM** encounters an **IRP** directive, it makes one copy of the statements for each parameter in the enclosed list. While copying the statements, it substitutes the current parameter for all occurrences of *dummyname* in these statements. If a null parameter (< >) is found in the list, the dummy name is replaced with a null value. If the parameter list is empty, the **IRP** directive is ignored and no statements are copied.

Example

```
IRP     x,<0,1,2,3,4,5,6,7,8,9>
        DB    10 DUP(x)
ENDM
```

This example repeats the **DB** directive 10 times, duplicating the numbers in the list once for each repetition. The resulting statements create 100 bytes of data with the values 0 through 9 duplicated 10 times.

Notes

Assume an **IRP** directive is used inside a macro definition and the parameter list of the **IRP** directive is also a dummy parameter of the macro. In this case, you must enclose that dummy parameter within angle brackets. For example, in the following macro definition, the dummy parameter x is used as the parameter list for the **IRP** directive:

```
alloc       MACRO   x
            IRP     y,<x>
            DB      y
            ENDM
            ENDM
```

If this macro is called with

```
alloc <0,1,2,3,4,5,6,7,8,9>
```

the macro expansion becomes

```
IRP         y,<0,1,2,3,4,5,6,7,8,9>
DB          y
ENDM
```

The macro removes the brackets from the actual parameter before replacing the dummy parameter. You must provide the angle brackets for the parameter list yourself.

8.2.7 IRPC and ENDM Directives

Syntax

IRPC *dummyname,string*
statements
ENDM

The **IRPC** and **ENDM** directives enclose a block of *statements* that is repeated once for each character in *string*. The *dummyname* is a name for a placeholder to be replaced by the current character in the string. The string can be any combination of letters, digits, and other characters. The string should be enclosed with angle brackets (< >) if it contains spaces,

commas, or other separating characters. The statements can be any valid assembler statements. The dummyname can be used any number of times in these statements.

When **MASM** encounters an **IRPC** directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of *dummyname* in these statements.

Example

```
IRPC    x,0123456789
        DB      x + 1
ENDM
```

This example repeats the **DB** directive 10 times, once for each character in the string `0123456789`. The resulting statements create 10 bytes of data having the values 1 through 10.

8.2.8 EXITM Directive

Syntax

EXITM

The **EXITM** directive tells the assembler to terminate macro or repeat-block expansion and continue assembly with the next statement after the macro call or repeat block. The **EXITM** directive is typically used with **IF** directives to allow conditional expansion of the last statements in a macro or repeat block.

When **EXITM** is encountered, the assembler exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If **EXITM** is encountered in a macro or repeat block nested in another macro or repeat block, **MASM** returns to expanding the outer level block.

Example

```
alloc MACRO  times
      x      =     0
      REPT   times          ;; Repeat up to 256 times
             IFE   x - 0FFh ;; Does x = 255 yet?
             EXITM          ;; If so, quit
             ELSE
             DB    x        ;; Else allocate x
             ENDIF
      x      =     x + 1    ;; Increment x
      ENDM
      ENDM
```

This example defines a macro that creates no more than 255 bytes of data. The macro contains an **IFE** directive that checks the expression `x-0FFh`. When this expression is 0 (x equal to 255), the **EXITM** directive is processed and expansion of the macro stops.

8.3 Macro Operators

The macro and conditional directives use the following special set of macro operators:

Operator	Definition
&	Substitute operator
< >	Literal-text operator
!	Literal-character operator
%	Expression operator
;;	Macro comment

When used in a macro definition or a conditional-assembly directive, these operators carry out special control operations, such as text substitution. They are described in Sections 8.3.1–8.3.5.

8.3.1 Substitute Operator

Syntax

&dummyparameter

or

dummyparameter&

The substitute operator (&) forces **MASM** to replace *dummyparameter* with its corresponding actual parameter value. The operator is used anywhere a dummy parameter immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

Example

```
errgen     MACRO    y,x
error&x    DB       'Error &y - &x'
           ENDM
```

In the example above, **MASM** replaces `&x` with the value of the actual parameter passed to the macro `errgen`. If the macro is called with the statement

```
           errgen  1,wait
```

the macro is expanded to

```
errorwait DB      'Error 1 - wait'
```

Note

For complex, nested macros, you can use extra ampersands (&) to delay the actual replacement of a dummy parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with z to make sure its replacement occurs while the **IRP** directive is being processed:

```
alloc   MACRO     x
        IRP       z,<1,2,3>
        x&&z      DB    z
        ENDM
        ENDM
```

In this example, the dummy parameter x is replaced immediately when the macro is called. The dummy parameter z, however, is not replaced until the **IRP** directive is processed. This means 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
```

8.3.2 Literal-Text Operator

Syntax

<text>

The literal-text operator directs **MASM** to treat *text* as a single literal element regardless of whether it contains commas, spaces, or other separators. The operator is most often used with macro calls and the **IRP** directive to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force **MASM** to treat special characters such as the semicolon (;) or the ampersand (&) literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

8.3.3 Literal-Character Operator

Syntax

!*character*

The literal-character operator forces the assembler to treat *character* as a literal character. For example, you can use it to force **MASM** to treat special characters such as the semicolon (;) or the ampersand (&) literally. Therefore, !; is equivalent to <;>.

8.3.4 Expression Operator

Syntax

%*text*

The expression operator (%) causes the assembler to treat *text* as an expression. **MASM** computes the expression's value, using numbers of the current radix, and replaces *text* with this new value. The *text* must represent a valid expression.

The expression operator is typically used in macro calls where the programmer needs to pass the result of an expression to the macro instead of to the actual expression.

Example

```
printe  MACRO   msg,num
        IF2                     ;; On pass 2 only
        %OUT    * &msg&num *    ;; Display message and number
        ENDIF                   ;;   to screen
        ENDM

sym1    EQU     100
sym2    EQU     200

        printe  <sym1 + sym2 = >,%(sym1 + sym2) ; Macro call
```

In this example, the macro call

```
        printe  <sym1 + sym2 = >,%(sym1 + sym2)
```

passes the text literal `sym1 + sym2 =` to the dummy parameter `msg`. It passes the value 300 (the result of the expression `sym1 + sym2`) to the dummy parameter `num`. The result is that **MASM** displays the message `sym1+sym2=300` when it reaches the macro call during the assembly. The **%OUT** directive, which sends a message to the screen, is described in Section 9.4 and the **IF2** directive is described in Section 7.2.2.

8.3.5 Macro Comment

Syntax

;;text

A macro comment is any text in a macro definition that does not need to be copied in the macro expansion. All *text* following the double semicolon (;;) is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the **.LALL**, **.XALL**, and **.SALL** directives described in Section 9.11.

Chapter 9
File Control Directives

9.1 Introduction

This chapter describes the MASM file-control directives, which provide control of the source, object, and listing files read and created by the assembler.

The file-control directives include the following:

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

Sections 9.2–9.12 describe these directives in detail.

9.2 INCLUDE Directive

Syntax

INCLUDE *filename*

The **INCLUDE** directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must name an existing file. A full or partial path name may be given if the file is not in the current working directory. **MASM** first looks for the "include" file (the source file specified by *filename)* in any paths specified with the **MASM /I** option, then it checks the current directory. If the named file is not found, the assembler displays an error message and stops.

When the assembler encounters an **INCLUDE** directive, it opens the specified source file and immediately begins assembling its statements. When all statements have been read, **MASM** continues assembly with the statement immediately following the directive.

Nested **INCLUDE** directives are allowed. A file named by an **INCLUDE** directive can contain **INCLUDE** directives. **MASM** marks included statements with the letter C in listings.

Directories can be specified in **INCLUDE** path names with either the backslash (\) or the forward slash (/). This is for XENIX® compatibility.

You should specify a file name, but no path name with the **INCLUDE** directive if you plan to set a search path with the **MASM /I** option. The **/I** option is discussed in Section 2.3.6 of the *Microsoft Macro Assembler User's Guide.*

Examples

```
INCLUDE entry                  ; File name
INCLUDE b:\include\record      ; Path name
INCLUDE /include/as/stdio      ; Path name
INCLUDE localinc\define.inc    ; Partial path name
```

9.3 .RADIX Directive

Syntax

.RADIX *expression*

The **.RADIX** directive sets the input radix for numbers in the source file. The *expression* is a number in the range 2 to 16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base. The most common bases are listed below:

Base	Number type
2	binary
8	octal
10	decimal
16	hexadecimal

The *expression* is always considered a decimal number, regardless of the current input radix. The default input radix is decimal.

Notes

The **.RADIX** directive does not affect the **DD**, **DQ**, or **DT** directives. Numbers entered in the expression of these directives are always evaluated as decimal unless a radix specifier is appended to the value.

The **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integer numbers. When **B** or **D** appears at the end of any integer, it is always considered to be a radix specifier even if the current input radix is 16.

For example, if the input radix is 16, the number `0ABCD` will be interpreted as 0ABC decimal, an illegal number, instead of as 0ABCD hexadecimal, as intended. Type `0ABCDh` to specify 0ABCD in hexadecimal. Similarly, the number `11B` will be treated as 11 binary, a legal number, but not 11B hexadecimal, as intended. Type `11Bh` to specify 11B in hexadecimal.

Examples

```
.RADIX  16
.RADIX  2
```

The first example sets the input radix to hexadecimal, while the second sets the input radix to binary.

9.4 %OUT Directive

Syntax

%OUT *text*

The **%OUT** directive instructs the assembler to display the *text* on the screen when it reaches the line containing the specified *text* during assembly. The directive is useful for displaying messages at specific points of a long assembly.

The **%OUT** directive generates output for both assembly passes. The **IF1** and **IF2** directives can be used to control when the directive is processed.

Example

```
IF1
        %OUT First Pass - OK
ENDIF
```

This sample block could be placed at the end of a source file so that the message `First Pass - OK` would be displayed at the end of the first pass, but ignored on the second pass.

9.5 NAME Directive

Syntax

NAME *modulename*

The **NAME** directive sets the name of the current module to *modulename*. A module name is used by the linker when displaying error messages.

The *modulename* can be any combination of letters and digits. Although the module name can be any length, only the first six characters are used. The name must be unique and not a reserved word.

If the **NAME** directive is not used, the assembler creates a default module name using the first six characters of the text specified in the **TITLE** directive. If no **TITLE** directive is found, the default name A is used.

Example

```
NAME Grafix
```

This example sets the module name to `Grafix`.

9.6 TITLE Directive

Syntax

TITLE *text*

The **TITLE** directive specifies the program-listing title. It directs **MASM** to copy *text* to the first line of each new page in the program listing. The text can be any combination of characters up to 60 characters in length.

No more than one **TITLE** directive per module is allowed. The first 6 nonblank characters of the title are used as the module name if the module does not contain a **NAME** directive.

Example

```
TITLE Graphics - First program
```

This example sets the title to `Graphics - First program`. If the module does not contain a **NAME** directive, the module name will be set to `Graphi` (the first six characters of `Graphics`.)

9.7 SUBTTL Directive

Syntax

SUBTTL *text*

The **SUBTTL** directive specifies the listing subtitle. It directs the assembler to copy *text* to the line immediately following the title on each new page in the program listing. The *text* can be any combination of characters. Only the first 60 characters are used. If no *text* is given, the subtitle line is left blank.

Any number of **SUBTTL** directives can be given in a program. Each new directive replaces the current subtitle with the new *text.*

Examples

```
SUBTTL Point Plotting Routines
```

The example above creates the subtitle `Point Plotting Routines`.

```
SUBTTL
```

The example above creates a blank subtitle.

9.8 PAGE Directive

Syntax

PAGE *length,width*
PAGE +
PAGE

The **PAGE** 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 in the listing.

If *length* and *width* are specified, the **PAGE** directive sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default page length is 50. The *width* must be in the range 60 to 132. The default page width is 80. If *width* is specified, but *length* is not, a comma (,) must precede *width*.

If a plus sign (+) follows **PAGE**, the section number is incremented and the page number is reset to 1. Program listing page numbers have the form

section-page

where *section* is the section number within the module, and *page* is the page number within the section. By default, section and page numbers begin with 1-1.

If no argument is given, **PAGE** starts a new output page in the program listing. It copies a form-feed character to the file and generates a title and subtitle line.

Examples

```
PAGE
```

The first example creates a page break.

```
PAGE 58,60
```

The second example sets the maximum page length to 58 lines, and the maximum width to 60 characters.

```
PAGE ,132
```

The third example sets the maximum width to 132 characters. The current page length (either the default of 50 or a previously set value) remains unchanged.

```
PAGE +
```

The final example increments the current section number and sets the page number to 1. For example, if the preceding page was 3-6, the new page would be 4-1.

9.9 .LIST and .XLIST Directives

Syntax

.LIST
.XLIST

The **.LIST** and **.XLIST** directives control which source-program lines are copied to the program listing. The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. The directives are typically used in pairs, to prevent a particular section of a source file from being copied to the program listing.

The **.XLIST** directive overrides all other listing directives.

Example

```
.XLIST          ; Listing suspended here
        .
        .
        .
.LIST           ; Listing resumes here
        .
        .
        .
```

9.10 .SFCOND, .LFCOND, and .TFCOND Directives

Syntax

.SFCOND
.LFCOND
.TFCOND

The **.SFCOND** and **.LFCOND** directives determine whether false-conditional blocks should be listed.

The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose **IF** condition is false. The **.LFCOND** directive restores the listing of these blocks. Like **.LIST** and **.XLIST**, false-conditional listing directives can be used to suppress listing of conditional blocks in sections of a program.

The **.TFCOND** directive sets the default mode for listing of conditional blocks. This directive works in conjunction with the **/X** option of the assembler. If **/X** is not given in the **MASM** command line, **.TFCOND** causes false-conditional blocks to be listed by default. If **/X** is given, **.TFCOND** causes false-conditional blocks to be suppressed. Every time a new **.TFCOND** is inserted in the source code, listing of false-conditionals is turned off if it was on, or on if it was off.

The **/X** option is discussed in Section 2.3.15 of the *Microsoft Macro Assembler User's Guide.*

Example

```
test1   DB      0       ; Symbol defined so all conditionals false

                        ; /X not used       /X used
.SFCOND
IFNDEF  test1           ; Not listing       Not listed
test2   DB      128
ENDIF
.LFCOND
IFNDEF  test1           ; Listed            Listed
test2   DB      128
ENDIF
.TFCOND
IFNDEF  test1           ; Listed            Not listed
test2   DB      128
ENDIF
.TFCOND
IFNDEF  test1           ; Not listed        Listed
test2   DB      128
ENDIF
```

In the example above, the listing for the last two conditionals would be reversed if the **/X** option were used. The first block with **.TFCOND** would not be listed and the second block would be listed.

9.11 .LALL, .XALL, and .SALL Directives

Syntax

.LALL
.XALL
.SALL

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the statements in macros that have been expanded in the source file. The assembler lists the full macro definition, but lists macro expansions only if the appropriate directive is set.

The **.LALL** directive causes **MASM** to list all the source statements in a macro, including comments preceded by a single semicolon (;), but not those preceded by a double semicolon (;;). The **.XALL** directive lists only those source statements that generate code or data. Comments are ignored.

The **.SALL** directive suppresses listing of all macro expansions. That is, the assembler copies the macro call to the source listing, but does not copy the source lines generated by the call.

The **.XALL** directive is in effect when **MASM** first begins execution.

For the sample listing below, assume that the following macro has been defined at the beginning of the source file:

```
tryout  MACRO
;;Macro comment line
; Normal comment line
        IF2                     ; No code or data
        ASSUME cs:code          ; No code or data
        DW      20 DUP(?)       ; Generates data
        mov     ax,bx           ; Generates code
        ENDIF                   ; No code or data
        ENDM
```

Assume also that the macro has been called once in the source file with each of the following macro listing directives:

```
.LALL
        tryout                  ; Call with .LALL
.XALL
        tryout                  ; Call with .XALL
.SALL
        tryout                  ; Call with .SALL
```

Example

```
                        .LALL
                                tryout
                    1           ; Normal comment line
                    1           IF2                 ; No code or data
                    1           ASSUME cs:code      ; No code or data
0005  0014[         1           DW      20 DUP(?)   ; Generates data
            ????    1
                    1
002D  8B C3         1           mov     ax,bx       ; Generates code
                    1           ENDIF               ; No code or data

                        .XALL
                                tryout
002F  0014[         1           DW      20 DUP(?)   ; Generates data
0057  8B C3         1           mov     ax,bx       ; Generates code

                        .SALL
                                tryout
```

Notice that the macro comment line is never listed in macro expansions. The normal comment line is listed only with the **.LALL** directive.

9.12 .CREF and .XCREF Directives

Syntax

.CREF
.XCREF [*name*,,,]

The **.CREF** and **.XCREF** directives control the generation of cross-references for the macro assembler's cross-reference file. The **.XCREF** directive suppresses the generation of label, variable, and symbol cross-references. The **.CREF** directive restores this generation.

If *name* is specified with **.XCREF**, only that label, variable, or symbol will be suppressed. All other names will be cross-referenced. The named label, variable, or symbol will also be omitted from the symbol table of the program listing. If two or more names are to be given, they must be separated by commas (,).

Example

```
.XCREF                  ; Suppress cross-referencing
      .                 ;   of symbols in this block
      .
      .
.CREF                   ; Restore cross-referencing
      .                 ;   of symbols in this block
      .
      .
.XCREF test1,test2      ; Don't cross-reference test1 or test2
      .                 ;   in this block
      .
      .
```

Appendixes

Appendix A
Instruction Summary

A.1 Introduction

The Microsoft Macro Assembler (**MASM**) is an assembler for the Intel 8086/80186/80286 family of microprocessors. It is capable of assembling instructions for the 8086, 8088, 80186, and 80286 microprocessors and the 8087 and 80287 floating-point coprocessors. Programs must use the instruction syntax described in this chapter.

By default, **MASM** recognizes the 8086 and 8087 instruction sets only (the 8088 set is identical to the 8086 set). If a source program contains 80186, 80286, or 80287 instructions, one or more instruction-set directives must be used in the source file to enable assembly of the additional instructions available in those instruction sets. Sections A.2–A.7 provide lists of the syntax of all instructions recognized by **MASM** with the various instruction-set directives.

Table A.1 explains the abbreviations used in the syntax descriptions.

Table A.1

Syntax Abbreviations

Abbreviation	Meaning
accum	One of the accumulators: **AX** or **AL**
reg	One of the byte or word registers Byte: **AL, AH, BL, BH, CL, CH, DL, DH** Word: **AX, BX, CX, DX, SI, DI, BP, SP**
segreg	One of the segment registers: **CS, DS, SS, ES**
r/m	One of the general operands: register, memory address, indexed operand, based operand, based-indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	One of the memory operands: label, variable, symbol
label	Instruction label
src	Source in string operations
dest	Destination in string operations

A.2 8086 Instructions

The 8086 instructions are listed below. (The 8088 instructions are identical to 8086 instructions.) **MASM** assembles 8086 instructions by default.

Syntax	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 doubleword
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 interrupts 0–255
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 (2's complement)
NOP	No operation

NOT *r/m*	Invert operand bits (1's complement)
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**
REP	Repeat
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**) may be 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**, and **REPZ** instructions provide ways 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 10 times.

A.3 8087 Instructions

The 8087 instructions are listed below. **MASM** assembles 8087 instructions by default.

Syntax	Action
F2XM 1	Calculate 2^X-
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*	Reversed real divide from memory
FDIVR ST,ST(*i***)**	Reversed real divide from stack
FDIVR ST(*i***),ST**	Reversed 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 *i*th 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 $\log_2 e$ onto top of stack
FLDL2T	Load $\log_2 10$ onto top of stack
FLDLG2	Load $\log_{10} 2$ onto top of stack
FLDLN2	Load $\log_e 2$ 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 element
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 $\log_2 x$
FYL2PI	Calculate Y $\log_2(x+1)$

A.4 80186 Instruction Mnemonics

The 80186 instruction set consists of all 8086 instructions plus the following instructions. The **.186** directive must be used to enable these instructions.

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16,immed8*	Enter procedure
IMUL *reg,immed*	Integer multiply register by immediate
IMUL *reg,r/m,immed*	Integer multiply general operand by immediate and store result in register
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 data onto stack
PUSHA	Push all registers
RCL *r/m,immed*	Rotate left through carry by immediate
RCR *r/m,immed*	Rotate right through carry by immediate
ROL *r/m,immed*	Rotate left by immediate

ROR *r/m,immed*	Rotate right by immediate
SAL *r/m,immed*	Shift arithmetic left by immediate
SAR *r/m,immed*	Shift arithmetic right by immediate
SHL *r/m,immed*	Shift left by immediate
SHR *r/m,immed*	Shift right by immediate

A.5 80286 Nonprotected Instructions

The 80286 nonprotected instruction set consists of all 8086 instructions plus the following instructions. The **.286c** directive must be used to enable these instructions.

Syntax	Action
BOUND *reg,mem*	Detect value out of range
ENTER *immed16,immed8*	Enter procedure
IMUL *reg,immed*	Integer multiply register by immediate
IMUL *reg,r/m,immed*	Integer multiply general operand by immediate and store result in register
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 data onto stack
PUSHA	Push all registers
RCL *r/m,immed*	Rotate left through carry by immediate
RCR *r/m,immed*	Rotate right through carry by immediate

ROL *r/m,immed*	Rotate left by immediate
ROR *r/m,immed*	Rotate right by immediate
SAL *r/m,immed*	Shift arithmetic left by immediate
SAR *r/m,immed*	Shift arithmetic right by immediate
SHL *r/m,immed*	Shift left by immediate
SHR *r/m,immed*	Shift right by immediate

A.6 80286 Protected Instruction Mnemonics

The 80286 protected instruction set consists of all 8086 and 80286 non-protected instructions plus the following instructions. The **.286p** directive must be used to enable these instructions.

Syntax	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

A.7 80287 Instruction Mnemonics

The 80287 instruction set consists of all 8087 instructions plus the following additional instructions. The **.287** directive must be used to enable these instructions.

FSETPM	Set protected mode
FSTSW AX	Store status word in **AX** (wait)
FNSTSW AX	Store status word in **AX** (no-wait)

Appendix B
Directive Summary

B.1 Introduction

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. Table B.1 lists all directives.

Table B.1

Directives

.186	ENDP	IF1	ORG
.286c	ENDS	IF2	%OUT
.286p	EQU	IFB	PAGE
.287	.ERR	IFDEF	PROC
.8086	.ERR1	IFDIF	PUBLIC
.8087	.ERR2	IFE	PURGE
=	.ERRB	IFIDN	.RADIX
ASSUME	.ERRDEF	IFNB	RECORD
COMMENT	.ERRDIF	IFNDEF	REPT
.CREF	.ERRE	INCLUDE	.SALL
DB	.ERRIDN	IRP	SEGMENT
DD	.ERRNB	IRPC	.SFCOND
DQ	.ERRNDEF	LABEL	STRUC
DT	.ERRNZ	.LALL	SUBTTL
DW	EVEN	.LFCOND	.TFCOND
ELSE	EXITM	.LIST	TITLE
END	EXTRN	LOCAL	.XALL
ENDIF	GROUP	MACRO	.XCREF
ENDM	IF	NAME	.XLIST

Any combination of upper- and lowercase letters can be used when giving directive names in a source file.

B.2 MASM Directives

The directives you can use in MASM source code are listed below with the syntax and function of each. This list is for reference only. See the appropriate chapters in this manual for details.

.186

Enables assembly of 80186 and 8086 instructions.

.286c

Enables assembly of 80286 nonprotected instructions and 8086 instructions.

.286p

Enables assembly of 80286 protected instructions and 8086 instructions.

.287

Enables assembly of 80287 and 8087 instructions.

.8086

Enables assembly of 8086 instructions (and the identical 8088 instructions) while disabling assembly of instructions available only with 80186 and 80286. This is the default mode.

.8087

Enables assembly of 8087 instructions while disabling assembly of instructions available only with 80287. This is the default mode.

name = *expression*

Assigns the numeric value of *expression* to *name.*

ASSUME *segmentregister*:*segmentname*,,,

Selects *segmentregister* to be the default segment register for all symbols in the named segment or group. If *segmentname* is **NOTHING,** no register is selected.

COMMENT *delimiter text delimiter*

Treats as a comment all *text* between the given pair of delimiters *delimiter.*

.CREF

Restores listing of symbols in the cross-reference listing file.

[*name*] **DB** *initialvalue,,,*

Allocates and initializes a byte (8 bits) of storage for each *initialvalue.*

[*name*] **DW** *initialvalue,,,*

Allocates and initializes a word (2 bytes) of storage for each *initialvalue.*

[*name*] **DD** *initialvalue,,,*

Allocates and initializes a doubleword (4 bytes) of storage for each *initialvalue.*

[*name*] **DQ** *initialvalue,,,*

Allocates and initializes a quadword (8 bytes) of storage for each *initialvalue.*

[*name*] **DT** *initialvalue,,,*

Allocates and initializes 10 bytes of storage for each given *initialvalue.*

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.

ENDM

Terminates a macro or repeat block.

name **ENDP**

Marks the end of a procedure definition.

name **ENDS**

Marks the end of a segment or of a structure-type definition.

name **EQU** *expression*

Assigns *expression* to *name.*

.ERR

Generates error.

.ERR1

Generates error on Pass 1 only.

.ERR2

Generates error on Pass 2 only.

.ERRB <*argument*>

Generates error if the *argument* is blank.

.ERRDEF *name*

Generates error if *name* is a previously defined label, variable, or symbol.

.ERRDIF <*string1*>,<*string2*>

Generates error if the strings are different.

.ERRE *expression*

Generates error if the *expression* is false (0).

.ERRIDN <*string1*>,<*string2*>

Generates error if the strings are identical.

.ERRNB <*argument*>

Generates error if the *argument* is not blank.

.ERRNDEF *name*

Generates error if *name* has not yet been defined.

.ERRNZ *expression*

Generates error if *expression* is true (nonzero).

EVEN

If necessary, increments the location counter to an even value and generates one **NOP** instruction (90h).

EXITM

Terminates expansion of the current repeat or macro block and begins assembly of next statement outside the block.

EXTRN *name:type,,,*

Defines an external variable, label, or symbol called *name* whose type is *type*.

name **GROUP** *segmentname,,,*

Associates a group name *name* with one or more segments.

IF *expression*

Grants assembly if *expression* is true (nonzero).

IF1

Grants assembly on Pass 1 only.

IF2

Grants assembly on Pass 2 only.

IFB *<argument>*

Grants assembly if *argument* is blank.

IFDEF *name*

Grants assembly if *name* is a previously defined label, variable, or symbol.

IFDIF *<argument1>,<argument2>*

Grants assembly if the arguments are different.

IFE *expression*

Grants assembly if *expression* is false (0).

IFIDN *<argument1>,<argument2>*

Grants assembly if the arguments are identical.

IFNB *<argument>*

Grants assembly if *argument* 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.

IRP *dummyname, <parameter,,,>*

Marks start of a block that will be repeated for as many parameters as are given, with the current *parameter* replacing the placeholder *dummyname* on each repetition.

IRPC *dummyname,<string>*

Marks start of a block that will be repeated for as many characters as there are in *string*, with the current character replacing the placeholder *dummyname* on each repetition.

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.

LOCAL *dummyname,,,*

Declares *dummyname* within a macro as a placeholder for an actual name to be created when the macro is expanded.

name **MACRO** *dummyparameter,,,*

Marks the beginning of macro *name* and establishes each item called *dummyparameter* as a placeholder for the expressions passed when the macro is called.

NAME *modulename*

Sets the name of the current module to *modulename.*

PURGE *macroname,,,*

Deletes the named macros.

ORG *expression*

Sets the location counter to *expression.*

%OUT *text*

Displays *text* at the user's terminal.

name **PROC** *type*

Marks the beginning of procedure *name,* of specified *type.*

PAGE *length,width*

Sets line *length* and character *width* of the program listing.

PAGE +

Increments section-page numbering.

PAGE

Generates a page break in the listing.

PUBLIC *name,,,*

Makes each variable, label, or absolute symbol specified as *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*[=*expression*]*,,,*

Defines a record type for an 8- or 16-bit record that contains one or more fields.

REPT *expression*

Marks the start of a block that is to be repeated *expression* number of times.

.SALL

Suppresses listing of all macro expansions.

name **SEGMENT** [*align*] [*combine*] ['*class*']

Marks the beginning of a program segment called *name* and having segment attributes *align*, *combine*, and *class*.

.SFCOND

Suppresses listing of any subsequent conditional blocks whose **IF** condition evaluates to false (0).

name **STRUC**

Marks the beginning of a type definition for a structure.

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.

B.3 MASM Operators

The operators recognized by MASM are listed by precedence in Table B.2. 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.

Table B.2

Operator Precedence

Precedence	Operators
(Highest)	
1	**LENGTH, SIZE, WIDTH, MASK,** (), [], <>
2	. (structure field name operator)
3	:
4	**PTR, OFFSET, SEG, TYPE, THIS**
5	**HIGH, LOW**
6	+,– (unary)
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 * *expression2*

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 of *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.

;;text

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 (0FFFFh) 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 having 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 specified *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.*

Appendix C
Segment Names for High-Level Languages

C.1 Introduction

This appendix describes the naming conventions used to form assembly-language source files compatible with object modules produced by recent Microsoft language compilers. Compilers that use these conventions include the following:

Microsoft C Version 3.0 or later

Microsoft Pascal Version 3.3 or later

Microsoft FORTRAN Version 3.3 or later

High-level-language modules have the following four predefined segment types:

Type	Use
TEXT	For program code
DATA	For program data
BSS	For uninitialized space
CONST	For constant data

Any assembly-language source file to be assembled and linked to a high-level-language module must use these segments, as described in Sections C.2–C.6.

High-level-language modules also have three different memory models:

Model	Use
Small	For single code and data segments
Middle	For multiple code segments, but a single data segment
Large	For multiple code and multiple data segments

Assembly-language source files to be assembled for a given memory model must use the naming conventions detailed in Sections C.2–C.6.

C.2 Text Segments

Syntax

[*prefix*]_ **TEXT SEGMENT byte public 'CODE'**
    ASSUME cs:[*prefix*]_ **TEXT**
statements
[*prefix*]_ **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 *prefix*_ **TEXT**, where *prefix* can be any valid string. For middle- and large-model programs, the module's own name is recommended. For small-model programs, *prefix* is omitted; the segment must be called _ **TEXT**.

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 **ASSUME** statement 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 Section 3.7).

Text segments should have **byte** align type and **public** combine type, and must have the class name **'CODE'**. These define loading instructions to be passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see Sections 3.4.1, 3.4.2, and 3.4.3.

The following formats are used for each of the different memory models:

Model	Requirements
Small model	Only one text segment is allowed. The segment must not exceed 64K. All procedure and statement labels should have the **NEAR** type.

Example

```
_TEXT          SEGMENT byte public 'CODE'
               ASSUME cs:_TEXT
_main          PROC near
                .
                .
                .
_main          ENDP
_TEXT          ENDS
```

Middle or large model

Multiple text segments are allowed. However, no segment can exceed 64K. 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
```

C.3 Data Segments – Near

Syntax

DGROUP GROUP _DATA
ASSUME ds:DGROUP
_DATA SEGMENT word public 'DATA'
statements
_DATA ENDS

A near data segment defines initialized data 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 are 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 64K of data. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements 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 Sections 3.6 and 3.7).

Near data segments must have **word** align type, **public** combine 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 Sections 3.4.1–3.4.3.

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
```

C.4 Data Segments – Far

Syntax

*prefix*_**DATA SEGMENT word public 'FAR_DATA'**
statements
*prefix*_**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 *prefix*_**DATA**, where *prefix* can be any valid string. 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 64K 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 (see Section 5.3.7).

Far data segments must have **word** align type, **public** combine 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 Sections 3.4.1–3.4.3.

Example

```
ARRAY_DATA      SEGMENT word public 'FAR_DATA'
array   DW      0
        DW      1
        DW      2
        DW      4
table   DW      1600 DUP(?)
ARRAY_DATA      ENDS
```

C.5 BSS Segments

Syntax

DGROUP GROUP _BSS
ASSUME ds:DGROUP
_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 64K. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements 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 Sections 3.6 and 3.7).

Note

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 a BSS segment, the directive

```
DGROUP  GROUP _DATA,_BSS
```

should be used.

A **BSS** segment must have **word** align type, **public** combine 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 Sections 3.4.1–3.4.3.

Example

```
DGROUP  GROUP   _BSS
        ASSUME ds:DGROUP

_BSS    SEGMENT word public 'BSS'
```

```
count   DW      ?
array   DW      10 DUP(?)
string  DB      30 DUP(?)
_BSS    ENDS
```

C.6 Constant Segments

Syntax

DGROUP GROUP CONST
ASSUME ds:DGROUP
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 64K. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements 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 Sections 3.6 and 3.7).

Note

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 have **word** align type, **public** combine 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 Sections 3.4.1–3.4.3.

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.

Index (Reference Manual)

MICROSOFT®
16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken: