Microsoft®
Macro Assembler

Reference Manual

for the MS™-DOS Operating System

Microsoft Corporation

Document Number: 8451-300-00

Contents

Chapter 1
Introduction

1.1 Overview

This manual describes the usage and input syntax of the Microsoft® Macro Assembler, MASM. The assembler produces relocatable object modules from 8086, 186, or 286 assembly language source files. The relocatable object modules can be linked, using the Microsoft Linker, LINK, to create executable programs for the MS-DOS operating system.

MASM is an assembler for the 8086/186/286 family of microprocessors. It can assemble the instructions of the 8086, 186, and 286 microprocessors, and the 8087 and 287 floating point coprocessors. It has a powerful set of assembly language directives that give the programmer complete control of the segmented architecture of the 8086, 186, and 286 microprocessors. MASM instruction syntax allows a wide variety of operand data types, including integers, strings, packed decimals, floating point numbers, structures, and records.

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

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

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

This manual does not teach assembly language programming, nor does it give a detailed description of 8086, 186, and 286 instructions. For information on these topics, you will need other references.

1.2 What You Need

This manual is intended to be used with the *Microsoft Macro Assembler User's Guide*. The guide explains the steps required to create executable programs from source files.

You also need to know the function and operation of the instructions in the instruction sets of the 8086/186/286 family of microprocessors. For an explanation of these instruction sets, you will need to turn to one of the many books that define these instructions. For your convenience, a complete list of the instruction names and syntax for all processors is given in Appendix A, "Instruction Summary."

1.3 Notational Conventions

This manual uses the following notational conventions to define the assembly language syntax:

Convention	Meaning
Roman	Indicates command or parameter names that must be typed as shown. In most cases, upper and lowercase letters can be freely intermixed.
Italics	Indicates a placeholder, that is, a name that you must replace with the value or filename required by the program.
...	Ellipses. Indicates that you can repeat the preceding item any number of times.
,,,	Indicates that you can repeat the preceding item any number of times as long as you separate the items with a comma.
[]	Brackets. Indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action.
|	Vertical bar. Indicates that only one of the separated items can be used. You must make a choice between the items.

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.

The following sections describe what characters can be used in a program and 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

An integer represents an integer number. It is a combination of binary, octal, decimal, or hexadecimal *digits* and an optional radix. The *digits* are a combination of one or more digits of the specified radix: B, Q, O, D, or H. If no radix is given, MASM uses the current default radix (typically decimal). The following table lists the digits that can be used with each radix. (Radix can be either upper or lowercase.)

Radix	Type	Digits
B	Binary	0 1
Q 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

Hexadecimal numbers must always start with a decimal digit (0-9). The hexadecimal digits A through F can be given as either upper or lower case.

The maximum number of digits in an integer depends on the instruction or directive in which the integer is used.

You can set the default radix by using the **.RADIX** directive. See the section, ".RADIX Directive," in Chapter 9.

Examples

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

2.4 Real Numbers

Syntax

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

A real number represents a number having an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits before the decimal point (.) represent the integer part, and those after the point represent the fraction. The digits after the exponent mark (E) represent the exponent. The exponent is optional. If an exponent is given, the plus (+) and minus (–) signs can 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 depends on the directive.

Examples

```
25.23  2.523E1        2523.0E-2
```

2.5 Encoded Real Number

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 field, 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).

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

Example

```
3F800000              ; 1.0 for DD
3FF0000000000000      ; 1.0 for DQ
```

2.6 Packed Decimal Numbers

Syntax

[+ |-] *digits*

A packed decimal number represents a decimal integer that is 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

```
1234567890          ; encoded as 0000000001234567890
-1234567890         ; encoded as 8000000001234567890
```

2.7 Character and String Constants

Syntax

'characters'
"characters"

A character constant is a constant composed of a single ASCII character. A string constant is a constant composed of two or more ASCII characters. The constant must be enclosed in matching single quotation or double quotation marks.

Single quotation marks must be encoded twice when given in constants that are enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when given in constants that are enclosed by double quotation marks.

Examples

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

2.8 Names

Syntax

characters...

A name is a combination of letters, digits, and special characters that can be used in instruction statements to labels, variables, and symbols. Names have the following formatting rules:

1. A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).

2. A name can have any combination of upper and lowercase letters. All lowercase letters are converted to uppercase unless the **–ML** or **–MX** option is used.

3. 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 that has a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and predefined group and segment names. These names can be used only as defined and must not be redefined.

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

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

All upper and lowercase combinations of these names are considered to be the same name. For example, the names "Length" and "LENGTH" are the same name for the LENGTH operator.

2.10 Statements

Syntax

[*name*] *mnemonic* [*operands*]

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

Statements have the following formatting rules:

1. A statement can begin in any column.

2. A statement must not be more than 128 characters in length and must not contain an embedded newline character. This means continuing a statement on multiple lines is not allowed.

3. A statement must be terminated by a newline character. This includes the last statement in the source file.

Examples

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

2.11 Comments

Syntax

;*text*

A comment is any combination of characters preceded by a semicolon (;) and terminated by a newline character. Comments let a programmer describe the action of a program at the given point. Comments are otherwise ignored by the assembler and have no effect on assembly.

Comments can be placed anywhere in a program, including on the same line as a statement. The comment must be placed after all names, mnemonics, and operands have been given. A comment must not be longer than one line, that is, it must not contain any embedded newline 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 *delim text delim*

The COMMENT directive causes MASM to treat all *text* between the given pair of delimiters (*delim*) as a comment. The delimiter character must be the first non-blank 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.

The COMMENT directive is typically used for multiple line comments. Although text can appear on the same line as the last delimiter, any text after the delimiter is ignored.

Example

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

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

Chapter 3
Program Structure

3.1 Introduction

The Program Structure directives let a programmer define the organization that a program's code and data will have when loaded into memory.

There are the following Program Structure directives:

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

The following sections describe these directives in detail. They also describe the Instruction Set directives that define which instruction set is to be used during assembly.

3.2 Source Files

Every assembly language program consists of one or more source files. A source file is simply a text file that contains statements that define the program's data and instructions. MASM reads source files and assembles the statements to create "object modules" that can be prepared for execution by the system linker.

All source files have the same form — zero or more program "segments" followed by an END statement. The **END** statement, required in every source file, signals the end of the source file. It also provides a way to define the program entry point or starting address (if any). All other statements in a source file are optional.

The following example illustrates the source file format. It is a complete assembly language program that uses MS-DOS system calls to print the message "Hello." on the system display.

```
        .8086

DATA    segment                         ; Program Data Segment
STRING db       "Hello.", 13, 10, "$"
DATA    ends

CODE    segment                         ; Program Code Segment
        assume cs:CODE, ds:DATA
START:                                  ; Program Entry Point
        mov     ax, seg DATA
        mov     ds, ax
        mov     dx, offset STRING
        mov     ah, 9
        int     21h
        mov     ah, 4ch
        int     21h
CODE    ends

STACK   segment stack                   ; Program Stack Segment
        assume ss:STACK
        dw      64 dup(?)
STACK   ends

        end     START
```

The main features of this source file are:

1. The .8086 directive, enabling the 8086 instruction set for assembly

2. The SEGMENT and ENDS statements, defining segments named DATA, CODE, and STACK

3. The variable STRING in the DATA segment, defining the string to be displayed

4. The instruction label START in the CODE segment, marking the start of the program instructions

5. The DW statement in the STACK segment, defining the uninitialized data space to be used for the program stack

6. The ASSUME statements in the DATA, CODE, and STACK segments, defining which segment registers will be associated with the labels, variables, and symbols defined within the segments

7. The END statement, defining START as the program entry point

3.3 Instruction Set Directives

Syntax

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

The instruction set directives enable/disable 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, should be placed at the beginning of the program source file. This ensures that all instructions in the file are assembled using the same set.

The .8086 directive enables assembly of instructions for the 8086 microprocessor. It also disables assembly of 186 and 286 instructions. Similarly, the .8087 directive enables assembly of instructions for the 8087 floating point coprocessor and disables assembly of 287 instructions. 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.

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

The .286c directive enables assembly of non-protected instructions for the 286 microprocessor. (These are identical to the 186 instructions). The .286p directive enables assembly of the protected instructions of the 286. The .286c directive should be used with programs that will be executed by a 286 microprocessor but do not ac-

cess the 286's protected instructions. The .286p directive can be used with programs that will be executed by a 286.

The .287 directive enables assembly of instructions for the 287 floating point coprocessor. This directive should be used with programs that have floating point instructions and will be executed by a 286 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 directs it to generate a software interrupt code to a floating point emulator routine.

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* define program loading instructions that are to be used by the linker when forming the executable program. These options are described later.

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_TEXT   segment word public 'CODE'
_main  proc far
       .
       .
       .

CONST  segment word public 'CONST'   ; nested segment
seg1   dw      ARRAY_DATA
CONST  ends                          ; end nesting

       mov     es, seg1
       push    es
       mov     ax, es:pointer
       push    ax
       call    _printf
       add     sp, 4
       .
       .
       .
       ret
_main  endp
SAMPLE_TEXT   ends
```

This example contains two segments: "SAMPLE_TEXT" and "CONST". The "CONST" segment is nested within the "SAMPLE_TEXT" segment.

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.

Program Loading Options

The optional *align* 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. It can be any one of the following:

BYTE use any byte address
WORD use any word address (2 bytes/word)
PARA use paragraph addresses (16 bytes/paragraph)
PAGE use page addresses (1024 bytes/page)

If no *align* is given, PARA is used by default. The actual start address is computed when the program is loaded, and the linker guarantees that the address will be on the given boundary.

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

PUBLIC

Concatenates all segments having the same name and forms 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 and forms a single, contiguous segment. All addresses in the new segment are relative to the SS segment register. The Stack Pointer (SP) register is set to an address in the segment.

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.

MEMORY

Places all segments having the same name in the highest physical segment in memory. If more than one MEMORY segment is given, the segments are overlapped as with COMMON segments.

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. AT segments typically contain no code or initialized data. Instead, they represent address templates that can be placed over code or data already in memory, such as code and data found in ROM devices. The labels and variables in the AT segments can then be used to access the fixed instructions and data.

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

Note

The linker requires at least one stack segment in a program.

The optional *class* 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.

Example

```
        assume  cs:_TEXT
_TEXT   segment word public 'CODE'
        .
        .
        .
_TEXT    ends
```

This example illustrates the general form of a text segment for a small module program. The segment name is "_TEXT". The segment alignment and combine type are "word" and "public," respectively. The class is "CODE."

3.5 END Directive

Syntax

END *expression*

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

The optional *expression* defines the program entry point. The entry point defines 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.

Examples

```
end
end     _main
```

3.6 GROUP Directive

Syntax

name GROUP *seg-name,,,*

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 that are relative to the beginning of the group instead of to the beginning of the segments in which they are defined. The *seg-name* must be the name of a segment defined using the SEGMENT directive, or a SEG expression. 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 the object modules are given to the linker.

Segments in a group do not have to be contiguous. This means that 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 65,535. If the segments of a group are contiguous, the group can occupy up to 64 Kbytes of memory.

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

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  _DATA, _BSS
      assume ds:DGROUP

_DATA  segment word public 'DATA'
         .
         .
         .
_DATA  ends
_BSS   segment word public 'BSS'
         .
         .
         .
_BSS   ends
        end
```

3.7 ASSUME Directive

Syntax

ASSUME *seg-reg* : *seg-name* ,,,
ASSUME NOTHING

The ASSUME directive selects the given segment register *seg-reg* to be the default segment register for all labels and variables defined in the segment or group given by *seg-name*. 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 4 selections: one selection for each of the four segment registers. The *seg-reg* can be any one of the segment register names: CS, DS, ES, or SS. The *seg-name* must be one of the following:

— The name of a segment previously defined with the SEGMENT directive.

— The name of a group previously defined with the GROUP directive.

— The keyword NOTHING.

The keyword NOTHING cancels the current segment selection. The directive "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, i.e., 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
org    $+2
```

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 instruction (90h). If the location counter is already even, the directive is ignored.

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

Example

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

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

3.10 PROC and ENDP Directives

Syntax

name PROC *type*
statements
name ENDP

The PROC and ENDP directives mark the beginning and end of a procedure. A procedure is a block of instructions that form 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 *type* can be either NEAR or FAR. NEAR is assumed if no *type* 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** statement to return control to the point of call. Nested procedures are allowed.

Example

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

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 that contain multiple fields, such as structures and records.

4.2 Label Declarations

Label declarations create "labels." A label is simply a name that represents the address of a given instruction. Labels can be used in JMP, CALL, and other execution control instructions to direct program execution to the associated instruction.

4.2.1 Near Label Declarations

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 and not previously defined. Furthermore, the segment containing the declaration must be associated with the CS segment register (see 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 as long as it immediately precedes the instruction.

Examples

```
start:
loop:   inc   4[bp]
```

4.2.2 Procedure Labels

Syntax

name PROC [NEAR|FAR]

The PROC directive creates a label *name* and sets its type to NEAR or FAR. The label then represents the address of the following instruction and can be used in JMP, CALL, or LOOP instruction to direct execution control to the given instruction.

When the PROC label definition is encountered, the assembler sets the label's value to the value of the current 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 JMP, CALL, and LOOP instruction in the enclosing segment only. FAR labels can be used in any segment of the program.

4.3 Data Declarations

The Data Declaration directives let a programmer generate data for a program. The directives translate numbers, strings, and expressions into individual bytes, words, or other units of data. The encoded data is copied to the program object file.

There are the following data declaration directives:

DB	Data Byte
DW	Data Word
DD	Data Doubleword
DQ	Data Quadword
DT	Data Ten-byte Word

The following sections describe these directives in detail.

4.3.1 DB Directive

Syntax

[*name*] DB *initial-value*,,,

The DB directive allocates and initializes a byte (8 bits) of storage for each given *initial-value*. The *initial-value* can be an integer, a character string constant, a DUP operator, a constant expression, or 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 a *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 *initial-value*,,,

The DW directive allocates and initializes a word (2 bytes) of storage for each given *initial-value*. An *initial-value* can be an integer, a 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 a *name* is given, the directive creates a variable of type WORD whose offset value is the current location counter value.

String constants must not exceed two characters in length. The last (or only) character in the string is placed in the low-order byte, and either zero 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 *initial-value* ,,,

The DD directive allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value*. An *initial-value* can be an integer, a real number, a 1- or 2-character string constant, an encoded real number, a DUP operation, 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 a *name* is given, the directive creates a variable of type DWORD whose offset value is the current location counter value.

String constants must not exceed two characters in length. 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 *initial-value* ,,,

The DQ directive allocates and initializes a quadword (8 bytes) of storage for each given *initial-value*. An *initial-value* can be an integer, a real number, a 1- or 2-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 expressions are given, they must be separated by commas (,).

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

String constants must not exceed two characters in length. 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 *initial-value* ,,,

The DT directive allocates and initializes 10 bytes of storage for each given *initial-value*. An *initial-value* can be an integer expression, a packed decimal, a 1- or 2-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 expressions are given, they must be separated by commas (,).

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

String constants must not exceed two characters in length. 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.

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(*initial-value*,,,)

The DUP operator is a special operator that can be used with the Data Declaration and other directives to specify multiple occurrences of one or more initial values. The *count* defines the number of times to repeat the *initial-value*. An initial value can be any expression that evalutes 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.

Examples

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

This example generates 100 bytes with value 1.

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

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

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

This example generates 125 bytes of data, each byte having the value 1.

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

This example generates 14 doublewords of uninitialized data.

4.4 Symbol Declarations

The Symbol Declaration directives let a programmer create and use symbols. A symbol is a descriptive name that represents a number, text, an instruction, or an address. Symbols make programs easier to read and maintain by letting descriptive names represent values. A symbol can be used anywhere its corresponding value is allowed.

There are the following Symbol Declaration directives:

=	Assign Absolutes
EQU	Equate Absolutes, Aliases, or Text Symbols
LABEL	Instruction or Data Labels

The following sections describe the directives in detail.

4.4.1 = Directive

Syntax

name = *expression*

The = 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 the *name* with the value of the given *expression*.

The *expression* can be an integer, a 1- or 2-character string constant, a constant expression, or an address expression. Its value must not exceed 65,535. The *name* must be either a unique name, or a name that was previously defined using the = 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 the *expression* to the given *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, not 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 integer values in the range 0 to 65,535 create absolute symbols and cause the assembler 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 before attempting to assemble the statement containing the name.

Symbols defined using EQU directive cannot be redefined.

Examples

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

```
subroutine    label  far
barray        label  byte
```

4.5 Type Declarations

The Type Declaration directives let a programmer define data types that can be used to create program variables that consist 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.

There are the following Type Declaration directives

STRUC and ENDS	Structure Types
RECORD	Record Types

The following sections describe the directives in detail.

4.5.1 STRUC and ENDS Directives

Syntax

name STRUC
    field-definitions
name ENDS

The STRUC and ENDS directives mark the beginning and end of a type definition for a structure. Structure type definitions define the name of a structure type and the number, type, and default value of the fields contained in the type. Once defined, structure types may be used to declare structure variables.

The *name* defines the new name of the structure type. It must be unique. The *field-definitions* define the structure's fields. Any number of field definitions can be given. The definitions must have the form

[*name*] DB *default-value*,,,
[*name*] DW *default-value*,,,
[*name*] DD *default-value*,,,
[*name*] DQ *default-value*,,,
[*name*] DT *default-value*,,,

The optional *name* defines the field name, the DB, DW, DD, DQ, and DT directives define the size of each field, and *default-value* de-

fines 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 structure type definition can contain field definitions and comments only. It must not contain any other statements. This means structures cannot be nested.

Example

```
table  struc
       count  db      10
       value  dw      10 DUP(?)
       name   db      'font3'
table  ends
```

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

4.5.2 RECORD Directive

Syntax

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

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 *exp* is the initial (or default) value for the field. Any number of *field:width= exp* combinations can be given with the record as long as each is separated from the preceding with a comma (,). The sum of the widths for all fields must not exceed 16.

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 = *exp* 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 *exp*. The *exp* 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 in the high end of the record.

Examples

```
encode  RECORD      high:4, mid:3, low:3
```

This example creates a record type "encode" having three fields: "high," "mid," and "low." The record occupies 16 bits of memory. The "high" field is in bits 6 to 9, "mid" in bits 3 to 5, and "low" in bits 0 to 2. The remaining high-order bits are unused.

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

This example creates a record type "item" having two fields: "char" and "weight." These values are initialized to the letter Q and the number 2, respectively.

4.6 Structure and Record Declarations

Structure and record declarations let a programmer generate a block of data bytes that have many elements or fields. A structure or record declaration consists of a previously-defined structure or record type name and a set of initial values.

The following sections describe these declarations in detail.

4.6.1 Structure Declarations

Syntax

[*name*] *strucname* < [*initial-value*],,, >

A structure variable is a variable that has one or more fields of different sizes. The *name* is the name of the variable, *strucname* is the name of a structure type that has been created using the STRUC directive, and *initial-value* 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, MASM allocates space for the structure, but does not create a name that you can use to access the structure.

The *initial-value* 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 with commas. If the DUP operator 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 structure. If an initial value is left blank, MASM automatically uses the default initial value of the field. This is defined by the structure type. If there is no default value, the field is uninitialized.

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.

Examples

```
struct1         table     <>
```

This 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.

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

This example creates a structure variable named "struct2." Its type is also "table." The initial value for the first field is set to zero. The default values defined by the structure type are used for the remaining two fields.

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

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

4.6.2 Record Declarations

Syntax

[*name*] *recordname* < [*initial-value*],,, >

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 *initial-value* is one or more values defining the initial value of the record. One *initial-value* can be given for each field in the record.

The *name* is optional. If not given, MASM allocates space for the record, but does not create a variable that you can use to access the record.

The *initial-value* can be an integer, string constant, or any expression that evaluates to a value that is no larger than can be represented in the specified field width. Angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated with commas. If the DUP operator 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, MASM 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.

Examples

```
rec1   encode <>
```

This example creates a record 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.

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

This example creates a variable "table" containing 10 records of the record type "item." The fields in these records are all set to the initial values A and 2.

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

This 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.

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 are combinations of operands and arithmetic, logical, bitwise, and attribute operators. An expression evaluates to a value or memory location to be acted on by an instruction or directive.

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.

There are the following operand types:

Constant
Direct Memory
Relocatable
Location Counter
Register
Based
Indexed
Based Indexed
Structure
Record
Record Field

5.2.1 Constant Operands

Syntax

number|*string*|*expression*

An 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
```

5.2.2 Direct Memory Operands

Syntax

segment : *offset*

A direct memory operand is a pair of segment and offset values that represent the absolute memory address of one or more bytes of memory. The *segment* can be a segment register name (CS, DS, SS, or ES), a segment name, or a group name. The *offset* must be an integer, absolute symbol, or expression that evaluates to a value within the range 0 to 65,535.

Examples

```
mov     dx, ss:0031H
mov     bx, DATA:0
mov     cx, 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 data to be acted on. 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, local
mov    bx, offset DGROUP:table
```

5.2.4 Location Counter

$

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 that have been generated for that segment to that point. After each statement in the segment has been assembled, the assembler increments the offset by the number of bytes generated.

Example

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

5.2.5 Register Operands

Syntax

reg-name

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 *reg-name* can be any one of the following:

ax	ah	al	bx	bh	bl
cx	ch	cl	dx	dh	dl
cs	ds	ss	es	sp	bp
di	si				

Any combination of upper and lower case 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 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 address 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 that are typically used for pointers to program data.

The 16-bit flag register contains nine 1-bit flags whose positions and meaning are defined in the following table:

Flag Bit	Meaning
0	carry flag
2	parity flag
4	auxiliary flag
5	trap flag
6	zero flag
7	sign flag
9	interrupt-enable flag
10	direction flag
11	overflow flag

Although no name exists for the 16-bit flag register, the contents of the register can be accessed using the **LAHF**, **SAHF**, **PUSHF**, and **POPF** instructions.

5.2.6 Based Operands

Syntax

disp[bp]
disp[bx]

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

The effective address of a based operand is the sum of the *disp* 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. The following illustrate a few of these forms:

[*disp*][bp]
[bp + *disp*]
[bp].*disp*
[bp]+*disp*

In each case, the effective address is the sum of *disp* 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

disp[si]
disp[di]

An indexed operand represents a memory address that is relative to one of the index registers: **si** or **di**. The *disp* can be any immediate or direct memory operand. It must evaluate to an absolute number or memory address. If no *disp* is given, zero is assumed.

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

Indexed operands have a variety of alternate forms. The following illustrate a few of these forms:

[*disp*][di]
[di + *disp*]
[di].*disp*
[di]+*disp*

In each case, the effective address is the sum of *disp* 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

disp[bp][si]
disp[bp][di]
disp[bx][si]
disp[bx][di]

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

The effective address of a based indexed operand is the sum of the *disp* value and the contents of the given registers. If the **bp** 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. The following illustrate a few of these forms:

[*disp*][bp][di]
[bp+di+*disp*]
[bp+di].*disp*
[di]+*disp*+[bp]

In each case, the effective address is the sum of *disp* and the contents of the given registers.

Examples

```
mov    ax, [ bp ][ si ]
mov    ax, [ bx + di ]
mov    ax, 12[ bp + di ]
mov    ax, fred[ bx ][ si ]
```

5.2.9 Structure Operands

Syntax

variable.field

A structure operand represents the memory address of one member of a structure. The *variable* must be the name of a structure or must be a memory operand that resolves to the address of a structure, and *field* must be the name of a field within that structure.

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.

In the following examples, "current_date" is assumed to be the structure defined by the following:

```
date    struc
        month  dw ?
        day    dw ?
        year   dw ?
date    ends
current_date date <'ja','01','84'>
```

Example

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

Structure operands are often used to access values on the stack. One method is to copy the current stack address into the **bp** register, then use "[bp].*member*" to access elements on the stack. This method makes all values on the stack available in any desired format.

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. 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.

Examples

```
mov    ax, encode <1,3,2>
mov    cx, key <,7>
```

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 following examples, assume that the record "rec1" is defined as:

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

Example

```
mov    ax, field1
```

This example copies 13, the shift count for field1, to ax.

```
mov     dx,rec1
mov     cl,field2
shr     dx,cl
```

This example copies 7, the shift count for field2, to cl, then uses the address of "rec1," copied to dx, in a shift operation. This operation adjusts rec1 so that field2 is now at the lowest bit.

5.3 Expressions

An expression is a combination of operands and operators that evaluates to a single value. Operands in expressions can be 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.

MASM 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.

The following sections describe the operators in detail. The attribute operators are described individually.

5.3.1 Arithmetic Operators

Syntax

exp1 * *exp2*
exp1 / *exp2*
exp1 MOD *exp2*
exp1 + *exp2*
exp1 - *exp2*
+ *exp*
- *exp*

Arithmetic operators provide the common mathematical operations. The operators have the following meanings:

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

For all arithmetic operators except + and –, the expressions *exp1* and *exp2* 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.

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 the given *expression* right or left by *count* number of bits. Bits shifted off the end of the expression are lost. If *count* is greater than or equal to 16, the result is 0.

Examples

```
01110111B SHL 3          ; equals 10111000B
01110111B SHR 3          ; equals 00001110B
```

5.3.3 Relational Operators

Syntax

exp1 EQ *exp2*
exp1 NE *exp2*
exp1 LT *exp2*
exp1 LE *exp2*
exp1 GT *exp2*
exp1 GE *exp2*

The relational operators compare the expressions *exp1* and *exp2* and return true (0FFFFH) if the given condition is satisfied, or false (0000H) if it is not. The expressions must resolve to absolute values. The operators have the following meanings:

Operator	Condition is satisfied when:
EQ	Operands are equal.
NE	Operands are not equal.
LT	Left operand is less than right.
LE	Left operand is less than or equal to right.
GT	Left operand is greater than right.
GE	Left operand is greater than or equal to right.

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

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 *exp*
exp1 AND *exp2*
exp1 OR *exp2*
exp1 XOR *exp2*

The logical operators perform bitwise operations on the given 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.

The operators have the following meanings:

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

Examples

```
NOT   11110000B                    ; equals 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 as-

sumed. 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]
mov     ax, array[4]
mov     string[LAST], al
mov     cx, DGROUP:[1]
```

Note that the last example is identical to the statement "mov cx, DGROUP:1."

5.3.6 PTR Operator

Syntax

type PTR *expression*

The PTR operator forces the variable or label given by the *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:

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 not used, MASM assumes a default size or distance for the reference. The PTR operator is also used to give instructions access to variables in ways

that would otherwise generate errors, for example, accessing the high-order byte of a WORD size variable.

Examples

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

5.3.7 Segment Override Operator

Syntax

segment-register : *expression*
segment-name : *expression*
group-name : *expression*

The segment override operator (:) forces the address of a given variable or label to be computed using the beginning of the given *segment-register*, *segment-name*, or *group-name*. If a *segment-name* or *group-name* 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 *segment-register* must be one of 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 SHORT Operator

Syntax

SHORT *label*

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

Example

```
jmp    short repeat
```

5.3.9 THIS Operator

Syntax

THIS *type*

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

NEAR	FAR
BYTE	WORD
DWORD	QWORD
TBYTE	

The THIS operator is typically used with the EQU or = directive to create labels and variables. This is similar to using the LABEL directive to create labels and variables.

Examples

```
tag    equu    this byte
```

This example is equivalent to the statement "TAG LABEL BYTE".

```
check =      this near
```

This example is equivalent to the statement "CHECK LABEL NEAR".

5.3.10 HIGH and LOW Operators

Syntax

HIGH *expression*
LOW *expression*

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

Examples

```
mov    ah, high word_value
mov    al, low 0FFFFH
```

5.3.11 SEG Operator

Syntax

SEG *expression*

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

Example

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

5.3.12 OFFSET Operator

Syntax

OFFSET *expression*

The OFFSET operator returns the offset of the given *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 return 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 the *expression* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group. See the second example below.

Examples

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

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

5.3.13 TYPE Operator

Syntax

TYPE *expression*

The TYPE operator returns a number representing the type of the given *expression*. If the *expression* is a variable, the operator returns the size of the operand in bytes. If the *expression* is a label, the operator returns 0FFFFH if the label is NEAR, and 0FFFEH if the label is FAR. Note that the return value can be used to specify the type for a PTR operator. See the second example below.

Examples

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

5.3.14 .TYPE Operator

Syntax

.TYPE *expression*

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

The variable's attributes are returned in bits 0, 1, 5, and 7 as follows:

Bit Position	If Bit=0	If Bit=1
0	Absolute	Program related
1	Not Data related	Data related
5	Not defined	Defined
7	Local scope	External scope

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

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

Example

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

This example sets z to 34.

5.3.15 LENGTH Operator

Syntax

LENGTH *variable*

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

Only variables that have been defined using the DUP operator return values greater than one. The return value is always the number that precedes the first DUP operator.

In the following examples, assume the definitions:

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

Examples

```
mov    cx, length array
```

In this example, LENGTH returns 100.

```
mov    cx, length table
```

In this example, LENGTH returns 100. The return value does not depend on any nested DUP operators.

5.3.16 SIZE Operator

Syntax

SIZE *variable*

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

In the following example, assume the definition:

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

Example

```
mov    bx, size array
```

In this example, SIZE returns 200.

5.3.17 WIDTH Operator

Syntax

WIDTH *record-fieldname* | *record*

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

In the following examples, assume that the record "rec1" is defined as:

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

Example

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

5.3.18 MASK Operator

Syntax

MASK *record-fieldname* | *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 record bit. All other bits contain 0. The *record-fieldname* must be the name of a record field.

In the following examples, assume that the record "rec1" is defined as:

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

Example

```
MASK field1    ;equals E000H
MASK field2    ;equals 1F80H
MASK field3    ;equals 003FH
MASK rtype     ;equals 0FFFFH
```

5.3.19 Expression Evaluation and Precedence

Expressions are evaluated according to the rules of operator precedence and order. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order of evaluation can be overridden using enclosing parentheses. Operations in parentheses are always performed before any adjacent operations. The following table lists the precedence of all operators. Operators on the same line have equal precedence.

Precedence	Operators
Highest	
1	LENGTH, SIZE, WIDTH, MASK
2	( )
3	[ ]
4	:
5	PTR, OFFSET, SEG, TYPE, THIS
6	HIGH, LOW
7	*, /, MOD, SHL, SHR
8	+, -
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)
```

5.4 Forward References

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

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

Whenever MASM encounters an undefined name in pass 1, it assumes that the name is a forward reference. If only a name is given, MASM makes assumptions about that name's type and segment re-

gister, and uses these assumptions to generate code or data for the statement. For example, in the JMP instruction above, MASM assumes that "target" is an instruction label having NEAR type. It generates three bytes of instruction code for the instruction.

MASM bases its assumptions on the statement containing the forward reference. Errors can occur when these assumptions are incorrect. For example, if "target" were really a FAR label and not a NEAR label, the assumption made by MASM in pass 1 would cause a phase error. In other words, MASM would generate five bytes of instruction code for the JMP instruction in pass 2 but only three in pass 1.

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

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

Examples

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

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

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

Example

```
jmp     short target
```

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

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

Examples

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

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

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

Example

```
dgroup segment stack
       assume ss: dgroup

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

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

5.5 Strong Typing for Memory Operands

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

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

```
string db     "A message."

       mov    ax, string[1]
```

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

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

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

will assemble correctly and execute as expected.

Chapter 6
Global Declarations

6.1 Introduction

The Global Declaration directives let a programmer 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 can be used only in the source files in which they are defined) into "global" symbols that are available to all other modules.

There are the following Global Declaration directives:

PUBLIC
EXTRN

The PUBLIC directive is used in public declarations. A public declaration transforms a locally defined symbol into a global symbol, making it available to other modules. The EXTRN directive is used in external declarations. An external declaration makes a global symbol's name and type known in a source file, letting the global symbol 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. The following sections describe the Global Declaration directives in detail.

6.2 PUBLIC Directive

Syntax

PUBLIC *name,,,*

The PUBLIC directive makes the variable, label, or absolute symbol given 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.

MASM 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 MASM to preserve lowercase letters when copying to the object file. See Chapter 2 in the *Microsoft Macro Assembler User's Guide* for more information.

Example

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

6.3 EXTRN Directive

Syntax

EXTRN *name:type ,,,*

The EXTRN directive defines an external variable, label, or symbol named *name* and whose type is *type*. An external item is any variable, label, or symbol that has been publicly declared in another module of the program.

The *name* must be the name of a variable, label, or symbol defined in another module of the program and listed in a PUBLIC directive of the corresponding source file. 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 reserved for symbols that represent absolute numbers.

Although the actual address is not determined until link time, 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. In this case, 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 an external variable's or label's default segment.

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 "startmod" and "printmod." The "startmod" module is the program's main module. Execution starts at the instruction labeled "start" in "startmod," and passes to the instruction labeled "print" in "printmod" where a DOS system call is used to print the message "Hello" at the system console. Execution then returns to the instruction labeled "exit" in "startmod."

Startmod Module:

```
        NAME   startmod
        public start, exit
        extrn  main:near

stack   segment       word public '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, seg DATA
        mov    ds, ax
        jmp    print
exit:
        mov    ah, 4ch
        int    21h
code    ends
        end    start
```

Printmod Module:

```
        NAME   printmod
        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
        mov    ah, 9
        int    21h
        jmp    exit
code    ends

        end
```

In this example, "startmod" publicly declares two symbols, "start" and "exit," making the symbols available to the other source file in the program. Both of these symbols are locally defined as instruction labels later in the source file, and therefore can be used as instruction labels in the other source file. The "startmod" file also contains an external declaration of the symbol "print." This declaration defines "print" to be a near label and is assumed to have been publicly declared in the other source file. The label is used in a JMP instruction given later in the file.

The "printmod" file contains a public declaration of the symbol "print" and an external declaration of the symbol "exit." In this case, "print" is locally defined as a near label and matches the external declaration given to it in "startmod." The symbol "exit" is declared to be a near label, matching its definition in "startmod."

Before this program can be executed, these source files must be assembled individually, then linked together using the system linker.

Chapter 7
Conditional Assembly

7.1 Introduction

The Conditional Assembly directives provide conditional assembly of blocks of statements within a source file. There are the following conditional directives:

```
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 form:

```
IF
        statements
ELSE
        statements
ENDIF
```

where the *statements* can be any valid statements, including other conditional blocks. The ELSE directive is optional.

MASM assembles the statements in the conditional block only if the condition that satisfies the corresponding IF directive is met. If the conditional block contains an ELSE directive, however, MASM will assemble only the statements up to the ELSE directive. The statements following the ELSE directive are assembled only if the IF condition is not met. An ENDIF directive must mark the end of the conditional block. No more than one ELSE for each IF directive is allowed.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested ELSE directive always belongs to the nearest, preceding IF directive.

7.2 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 *expression* is non-zero (true). The IFE directive grants assembly if *expression* is 0 (false). 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
```

7.3 IF1 and IF2 Directives

Syntax

IF1
IF2

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

Example

```
ifi
        %out Pass 1 Starting
endif
```

7.4 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 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. This is a frequent cause of phase errors.

Example

```
ifndef  BUFFER
BUFFER db      10 dup(?)
endif
```

7.5 IFB and IFNB Directives

Syntax

IFB <*arg*>
IFNB <*arg*>

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

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

Examples

```
IFB <X>
```

This example tests the argument "<X>." If this is in a macro definition and no parameter was passed for X, the directive would grant assembly.

```
IFNB <&EXIT>
```

This example tests the argument "<&EXIT>." This is assumed to be in a macro definition. If no parameter is passed for EXIT, the directive does not grant assembly.

7.6 IFIDN and IFDIF Directives

Syntax

IFIDN <*arg1*>, <*arg2*>
IFDIF <*arg1*>, <*arg2*>

The IFIDN and IFDIF directives test *arg1* and *arg2*. 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 *arg1* must match the corresponding character in *arg2*. The angle brackets (< >) are required.

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

Examples

```
IFIDN <X>, <Y>
```

This example tests the arguments "<X>" and "<Y>." If this is in a macro definition and the parameters passed for X and Y are identical, the directive grants assembly.

```
IFDIF <&EXIT>, <CASE>
```

This example tests the arguments "<&EXIT>" and "<CASE>." This is assumed to be in a macro definition. If the parameters passed for EXIT and CASE are identical, the directive does not grant assembly.

Chapter 8
Macro Directives

8.1 Introduction

This chapter explains how to create and use macros in your source files. There are the following macro directives:

MACRO
LOCAL
PURGE
REPT
IRP
IRPC
EXITM
ENDM

The MACRO directive lets you write a named block of source statements, then use that name in your source file to represent the statements. MASM automatically replaces each occurrence of a macro name with the statements given in the macro definition. This means you can place a block of statements any where in your source file any number of times by simply defining it once, then giving the name where you need it. The LOCAL directive lets you define unique labels for a macro, and the PURGE directive lets you control the macros you define.

The REPT, IRP, and IRPC directives also let you create contiguous blocks of repeated statements. You control the number of times the statements are repeated. You can repeat them a given number of times, once for each parameter in a list, or once for each character in a string.

The macro directives use a special set of macro operators:

&	Ampersand
;;	Double semicolon
!	Exclamation mark
%	Percent sign

When used in a macro definition, these operators carry out special control operations, such as text substitution.

8.2 MACRO and ENDM Directives

Syntax

name MACRO [*dummy-parameter,,,*]
statements
ENDM

The MACRO and ENDM directives create a macro having the given *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 *dummy-parameter* is a name that acts as a placeholder for values to be passed to the macro when it is called. Any number of dummy parameters can be given, 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 parameter can be used any number of times in these statements.

A macro is "called" any time the macro's name appears in a source file (macros names in comments are ignored). MASM copies the statements in the macro definition to the point of call, replacing any dummy parameters in these statements with values passed in the call. Dummy parameters are replaced according to their order in the definition. Thus, the dummy parameter occupying the fourth parameter position in the macro definition is replaced by the fourth actual parameter in the macro 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 be 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. A macro definition cannot contain a call to itself.

Notes

Remember that 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, 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.

MASM assembles the statements in the macro only if the macro is called, and only at the point they are inserted into the source file. This means all addresses in the assembled code are 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 immediately after these directives causes MASM to begin to create macros named TITLE, SUBTTL, and NAME. To avoid this problem, you should alter the word MACRO in some way when using it in a title or name. For example, add a hyphen to the word, "- MACRO."

Examples

```
add     MACRO   xx,yy,zz
        mov     ax, xx
        add     ax, yy
        mov     zz, ax
        ENDM
```

This example defines a macro named "add", that contains three statements and uses three dummy parameters. The dummy parameters are "xx," "yy," and "zz." These parameters will be replaced with actual values when the macro is called.

8.3 Macro Calls

Syntax

name [*actual-parameter*,,,]

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 *actual-parameter* 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 with 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, any remaining dummy parameters are replaced with nothing. This means MASM removes the dummy parameter name for the macro statements, but does nothing else.

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
```

This example passes five numeric parameters to the macro ALLOCBLOCK.

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

This example passes one parameter to ALLOCBLOCK. This parameter is a list of five numbers.

```
add    1, inc, linecount
```

This example passes three parameters to the macro "add." The first parameter is a number, but the second and third are symbols. MASM replaces the corresponding dummy parameters with exactly what is typed here.

8.4 LOCAL Directive

Syntax

LOCAL *dummy-name,,,*

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

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

??xxxx

where *xxxx* is a hexadecimal number in the range 0000 to FFFF.

The LOCAL directive is typically used to create unique labels for macros. Normally, if a macro containing a label is used more than once, MASM will display a multiply-defined label error message since the same label will appear in both expansions. To avoid this problem, all labels can be local dummy names. MASM guarantees that these names will be replaced with unique names whenever the macro is expanded, no matter how often the macro is expanded.

Note

The directive can be used only in a macro definition, and it must precede all other statements in the definition.

Example

```
loop   MACRO   count, y
       LOCAL   A
       mov     ax, y
A:     mov     cx, count
       inc     ax
       jnz     A
       ENDM
```

In this example, the LOCAL directive defines a dummy name "A" that is replaced with a unique name each time the macro is expanded. "A" is used in two places in the macro: as a statement label, and as the target of a JNZ instruction.

8.5 PURGE Directive

Syntax

PURGE *macro-name* ,,,

The PURGE directive deletes the current definition of the macro *macro-name* from memory. Any subsequent call to that macro causes MASM to generate an error.

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

The PURGE directive is often used with a "macro library" to let you choose those macros from the library 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.

Examples

```
PURGE   add
```

This example deletes the macro named "add" from the assembler's memory.

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

This example deletes the macros named "mac1," "mac2," and "mac9."

8.6 REPT and ENDM Directives

Syntax

REPT *expression*
statements
ENDM

The REPT and ENDM directives define a block of *statements* that are 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 = and DB directives 10 times. The resulting statements create 10 bytes of data whose values range from 1 to 10.

8.7 IRP and ENDM Directives

Syntax

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

The IRP and ENDM directives define a block of *statements* that are to be repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummy* 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, you must separate them with commas. The *statements* can be any valid assembler statements. The dummy 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 replaces all occurrences of dummy in these statements with the current parameter. If a null parameter (<>) is found in the list, the dummy 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,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

This example repeats the DB directive 10 times, once for each number in the list. The resulting statements create 10 bytes of data having the values 1 through 10.

Notes

If an empty parameter is found in the list, MASM repeats the statements once, replacing the dummy with no value. If the entire list is empty, MASM skips the repeat block.

If 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, 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 <1,2,3,4,5,6,7,8,9,10>
```

the macro expansion becomes

```
IRP    Y,<1,2,3,4,5,6,7,8,9,10>
DB     Y
ENDM
```

That is, the macro removes the brackets from the actual parameter before replacing the dummy. This means you must provide the angle brackets for the parameter list yourself.

8.8 IRPC and ENDM Directives

Syntax

```
IRPC dummy,string
     statements
ENDM
```

The IRPC and ENDM directives define a block of *statements* that are repeated once for each character in the *string*. The *dummy* 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 dummy 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 replaces all occurrences of dummy in these statements with the current character.

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.

8.9 EXITM Directive

Syntax

EXITM

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

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

Example

```
alloc  MACRO  times
       x  =   0
              REPT   times
              IFE    x-0FFH
                     EXITM
              ELSE
                     DB     x
              ENDIF
              x  =   x+1
              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.

8.10 Substitute Operator

Syntax

&dummy-parameter
or
dummy-parameter&

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

Example

```
errgen      MACRO   Y, X
error&X             db      'Error &Y - &X'
            ENDM
```

In this example, MASM replaces &X with the value of the actual parameter passed to 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
```

8.11 Literal Text Operator

Syntax

<text>

The literal text operator directs MASM to treat *text* as a single literal. 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 ; or & 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.

8.12 Literal Character Operator

Syntax

!character

The literal character operator forces MASM to treat *character* as a literal. For example, you can force MASM to treat special characters such as ; or & literally. Therefore, !; is equivalent to <;>.

8.13 Expression Operator

Syntax

%text

The expression operator (%) causes MASM 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 assembler 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 the actual expression.

Example

```
printe MACRO   msg,n
       %OUT    * msg,n *
       ENDM
sym1   EQU     100
sym2   EQU     200
       printe  <sym1 + sym2 = >,%(sym1 + sym2)
```

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 "n."

8.14 Macro Comment

Syntax

;;*text*

The 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.

Regular comments, unlike macro comments, can be copied to each macro expansion by specifying the .XALL directive in the source file.

Chapter 9
File Control

9.1 Introduction

This chapter describes File Control directives. These directives provide control of the source, object, and listing files read and created by MASM during an assembly.

There are the following File Control directives:

INCLUDE	Include a Source File
.RADIX	Alter 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
.LIST	List Statements in Program Listing
.XLIST	Suppress Listing Statements
.LFCOND	List False Conditional in Program Listing
.SFCOND	Suppress False Conditional Listing
.TFCOND	Toggle False Conditional Listing
.LALL	List Macro Expansions in Program Listing
.SALL	Suppress Listing Macro Expansion
.XALL	Exclude Comments from Macro Listing
.CREF	List Symbols in Cross Reference File
.XCREF	Suppress Symbol Listing

The following sections describe the 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 pathname must be given if the file is not in the current working directory. If the named file is not found, MASM displays an error message and stops.

When MASM encounters an INCLUDE directive, it opens the named file and begins to assemble its source statements immediately. When all statements have been read, MASM resumes with the next statement following the directive.

Nested INCLUDE directives are allowed. This means a file named by an INCLUDE directive can contain its own INCLUDE directives.

When a program listing is created, MASM marks included statements with the letter C.

Examples

```
include entry
include include\record
include \usr\include\as\stdio
```

9.3 .RADIX Directive

Syntax

.RADIX *expression*

The .RADIX directive sets the default input radix for numbers in the source file. The *expression* defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base. It must be within the range 2 to 16. The following lists some common values:

2	- binary
8	- octal
10	- decimal
16	- hexadecimal

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

Examples

```
.radix  16
.radix  2
```

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 numeric suffix is appended to the value.

The .RADIX directive does not affect the optional radix specifiers, B and D, used with integers 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. This means that numbers such as "0abcd" and "234b" are illegal even when the input radix is set to 16. ("a," "b," and "c" are not legal digits for a decimal number; similarly, "2," "3," and "4" are not legal for a binary number.)

9.4 %OUT Directive

Syntax

%OUT *text*

The %OUT directive directs MASM to display the *text* at the user's terminal. The directive is useful for displaying messages during 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 -- Okay
ENDIF
```

9.5 NAME Directive

Syntax

NAME *module-name*

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

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

Example

```
name main
```

If the NAME directive is not used, MASM creates a default module name using the first six characters of a TITLE directive. If no TITLE directive is found, the default name "A" is used.

9.6 TITLE Directive

Syntax

TITLE *text*

The TITLE directive defines 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.

Example

```
title PROG1 -- 1st Program
```

Note that the first six non-blank characters of the title will be used as the module name if the module does not contain a NAME directive.

9.7 SUBTITLE Directive

Syntax

SUBTTL *text*

The SUBTTL directive defines the listing subtitle. It directs MASM to copy *text* to the line immediately after 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 characters are 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   SPECIAL I/O ROUTINE
```

This example creates the subtitle "SPECIAL I/O ROUTINE."

```
subttl
```

This example creates a blank subtitle.

9.8 PAGE Directive

Syntax

PAGE *length, width*
PAGE +
PAGE

The PAGE directive sets the line length and character width of the program listing, increments section page numbering, or generates a page break in the listing.

If a *length* and *width* are given, PAGE 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 is 50. The *width* must be in the range 60 to 132. The default is 80. A *width* can be given without a *length* as long as the comma (,) precedes the *width*.

If a plus sign (+) is given, PAGE increments the section number and resets the page number to 1. Program listing page numbers have the form:

section-minor

By default, page numbers start at 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
```

This example creates a page break.

```
PAGE 58,60
```

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

```
PAGE ,132
```

This example sets the maximum width to 132 characters. The current page length remains unchanged.

```
PAGE +
```

This example increments the current section number and sets the page number to 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 section of a given 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 or not 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. The directives can be used like .LIST and .XLIST to suppress listing of the 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.

Examples

```
.SFCOND
IF 0
        ;This block will not be listed.
ENDIF
.LFCOND
IF 0
        ;This block will be listed.
ENDIF
```

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. MASM always 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, so comments are ignored.

The .SALL directive suppresses listing of all macro expansions. That is, MASM copies the macro call to the source listing, but does not copy the source lines that the call generates.

.XALL is in effect when MASM first begins execution.

Example

```
.SALL
        ;No macros listed here.
.LALL
        ;Macros listed in full.
.XALL
        ;Macros listed by generated code or data only.
```

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 function restores this generation.

If a *name* is given 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 be given, they must be separated with commas.

Example

```
.XCREF one, two, three
```

Appendix A
Instruction Summary

A.1 Introduction

MASM is an assembler for the 8086/186/286 family of microprocessors, capable of assembling instructions for the 8086, 186, and 286 microprocessors and the 8087 and 287 floating point coprocessors. MASM will assemble any program written for an 8086, 186, or 286 microprocessor environment as long as the program uses the instruction syntax described in this chapter.

By default, MASM recognizes 8086 and 8087 instructions only. If a source program contains 186, 286, or 287 instructions, one or more Instruction Set directives must be used in the source file to enable assembly of the instructions. The following sections list the syntax of all instructions recognized by MASM and the Instruction Set directives.

Abbreviations used in the syntax descriptions are:

Symbol	Meaning
accum	accumulator: AX or AL
reg	byte or word register byte: AL, AH, BL, BH, CL, CH, DL, DH word: AX, BX, CX, DX, SI, DI, BP, SP
segreg	segment register: CS, DS, SS, ES
r/m	general operand: register, memory address, indexed operand, based operand, or based indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	memory operand: label, variable, or symbol
label	instruction label

A.2 8086 Instructions

The following is a complete list of the 8086 instructions. MASM assembles all 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 double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC *r/m*	Decrement operand
DEC *reg*	Decrement 16-bit register
DIV *r/m*	Divide accumulator by operand
ESC *immed, r/m*	Escape with 6-bit immediate and operand
HLT	Halt

IDIV *r/m*	Integer divide accumulator by operand
IMUL *r/m*	Integer multiply accumulator by operand
IN *accum, immed*	Input from port (8-bit immediate)
IN *accum*, **DX**	Input from port given by DX
INC *r/m*	Increment operand
INC *reg*	Increment 16-bit register
INT 3	Software interrupt 3 (encoded as one byte)
INT *immed*	Software Interrupt 0 through 255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA *label*	Jump on above
JAE *label*	Jump on above or equal
JB *label*	Jump on below
JBE *label*	Jump on below or equal
JC *label*	Jump on carry
JCXZ *label*	Jump on CX zero
JE *label*	Jump on equal
JG *label*	Jump on greater
JGE *label*	Jump on greater or equal
JL *label*	Jump on less than
JLE *label*	Jump on less than or equal
JMP *label*	Jump to instruction at label
JMP *r/m*	Jump to instruction indirect
JNA *label*	Jump on not above
JNAE *label*	Jump on not above or equal
JNB *label*	Jump on not below
JNBE *label*	Jump on not below or equal
JNC *label*	Jump on no carry
JNE *label*	Jump on not equal
JNG *label*	Jump on not greater
JNGE *label*	Jump on not greater or equal
JNL *label*	Jump on not less than
JNLE *label*	Jump on not less than or equal
JNO *label*	Jump on not overflow
JNP *label*	Jump on not parity
JNS *label*	Jump on not sign
JNZ *label*	Jump on not zero
JO *label*	Jump on overflow
JP *label*	Jump on parity
JPE *label*	Jump on parity even
JPO *label*	Jump on parity odd
JS *label*	Jump on sign
JZ *label*	Jump on zero
LAHF	Load AH with flags

LDS *r/m*	Load operand into DS
LEA *r/m*	Load effective address of operand
LES *r/m*	Load operand into ES
LOCK	Lock bus
LODS *src*	Load string
LODSB	Load byte from string into AL
LODSW	Load word from string into AX
LOOP *label*	Loop
LOOPE *label*	Loop while equal
LOOPNE *label*	Loop while not equal
LOOPNZ *label*	Loop while not zero
LOOPZ *label*	Loop while zero
MOV *accum, mem*	Move memory to accumulator
MOV *mem, accum*	Move accumulator to memory
MOV *r/m, immed*	Move immediate to operand
MOV *r/m, reg*	Move register to operand
MOV *r/m, segreg*	Move segment register to operand
MOV *reg, immed*	Move immediate to register
MOV *reg, r/m*	Move operand to register
MOV *segreg, r/m*	Move operand to segment register
MOVS *dest, src*	Move string
MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL *r/m*	Multiply accumulator by operand
NEG *r/m*	Negate operand
NOP	No operation
NOT *r/m*	Invert operand bits
OR *accum, immed*	Bitwise Or immediate with accumulator
OR *r/m, immed*	Bitwise Or immediate with operand
OR *r/m, reg*	Bitwise Or register with operand
OR *reg, r/m*	Bitwise Or operand with register
OUT DX, *accum*	Output to port given by DX
OUT *immed, accum*	Output to port (8-bit immediate)
POP *r/m*	Pop 16-bit operand
POP *reg*	Pop 16-bit register from stack
POP *segreg*	Pop segment register
POPF	Pop flags
PUSH *r/m*	Push 16-bit operand
PUSH *reg*	Push 16-bit register onto stack
PUSH *segreg*	Push segment register
PUSHF	Push flags
RCL *r/m*, **1**	Rotate left through carry by 1 bit
RCL *r/m*, **CL**	Rotate left through carry by CL
RCR *r/m*, **1**	Rotate right through carry by 1 bit

RCR *r/m*, **CL**	Rotate right through carry by CL
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET [*immed*]	Return after popping bytes from stack
ROL *r/m*, **1**	Rotate left by 1 bit
ROL *r/m*, **CL**	Rotate left by CL
ROR *r/m*, **1**	Rotate right by 1 bit
ROR *r/m*, **CL**	Rotate right by CL
SAHF	Store AH into flags
SAL *r/m*, **1**	Shift arithmetic left by 1 bit
SAL *r/m*, **CL**	Shift arithemetic 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 imeediate from accumulator
SUB *r/m, immed*	Subtract immediate from operand
SUB *r/m, reg*	Subtract register from operand
SUB *reg, r/m*	Subtract operand from register
TEST *accum, immed*	Compare immediate bits with accumulator
TEST *r/m, immed*	Compare immediate bits with operand
TEST *r/m, reg*	Compare register bits with operand
TEST *reg, r/m*	Compare operand bits with register
WAIT	Wait
XCHG *accum, reg*	Exchange accumulator with register
XCHG *r/m, reg*	Exchange operand with register
XCHG *reg, accum*	Exchange register with accumulator

XCHG *reg, r/m*	Exchange register with operand
XLAT *mem*	Translate
XOR *accum, immed*	Bitwise Xor immediate with accumulator
XOR *r/m, immed*	Bitwise Xor immediate with operand
XOR *r/m, reg*	Bitwise Xor register with operand
XOR *reg, r/m*	Bitwise Xor operand with register

The String instructions (CMPS, LODS, MOVS, SCAS, and STOS) use the DS, SI, ES, and DI registers to compute operand locations. Source operands are assumed to be at DS:[SI]; destination operands at ES:[DI]. The operand type (BYTE or WORD) is defined by the instruction mnemonic. For example, CMPSB specifies BYTE operands and CMPSW specifies WORD operands. For the CMPS, LODS, MOVS, SCAS, and STOS instructions, the *src* and *dest* operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The *src* operand can also be used to specify a segment override. The ES register for the destination operand cannot be overridden.

Examples

```
cmps  word ptr string, word ptr es:0
lods  byte ptr string
mov   byte ptr es:0,  byte ptr string
```

The REP, REPE, REPNE, REPNZ, or REPZ instructions provide a way to repeatedly execute a String instruction for a given count or while a given condition is true. If a Repeat instruction immediately precedes a String instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false, or the CX register is equal to zero. The Repeat instruction decrements CX by one for each execution.

Example

```
mov    cx, 10
rep    scasb
```

In this example, SCASB is repeated ten times.

A.3 8087 Instruction Mnemonics

The following is a list of the 8087 instructions. MASM assembles all 8087 instructions by default.

Syntax	Action
F2XM1	Calculate 2^X-1
FABS	Take absolute value of top of stack
FADD	Add real
FADD *mem*	Add real from memory
FADD ST, ST(*i***)**	Add real from stack
FADD ST(*i***), ST**	Add real to stack
FADDP ST(*i***), ST**	Add real and pop stack
FBLD *mem*	Load 10-byte packed decimal on stack
FBSTP *mem*	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(*i***)**	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop
FCOMP ST(*i***)**	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIV *mem*	Divide real from memory
FDIV ST, ST(*i***)**	Divide real from stack
FDIV ST(*i***), ST**	Divide real in stack
FDIVP ST(*i***), ST**	Divide real and pop stack
FDIVR	Reversed real divide
FDIVR *mem*	Reverse real divide from memory
FDIVR ST, ST(*i***)**	Reverse real divide from stack
FDIVR ST(*i***), ST**	Reverse real divide in stack
FDIVRP ST(*i***), ST**	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FFREE ST	Free top of stack element
FFREE ST(*i***)**	Free *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 elements
FFREE ST	Exchange top of stack element
FFREE ST(*i***)**	Exchange top of stack and *i*th element
FXTRACT	Extract exponent and significand
FYL2X	Calculate Y $\log_2 x$
FYL2PI	Calculate Y $\log_2(x+1)$

A.4 186 Instruction Mnemonics

The 186 instruction set consists of all 8086 instructions plus the following instructions. The .186 directive can be used to enable these instructions for assembly.

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
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
PUSHA	Push all registers
POPA	Pop all registers

A.5 286 Non-Protected Instruction Mnemonics

The 286 non-protected instruction set consists of all 8086 instructions plus the following instructions. The .286c directive can be used to enable these instructions for assembly.

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
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
PUSHA	Push all registers
POPA	Pop all registers

A.6 286 Protected Instruction Mnemonics

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

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
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.7 287 Instruction Mnemonics

The 287 instruction set consists of all 8087 instructions plus the following additional instructions. The .287 directive can be used to enable these instructions for assembly.

Syntax	Action
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. There are the following directives:

.186	ELSE	IFDIF	PROC
.286c	END	IFE	PUBLIC
.286p	ENDIF	IFIDN	.RADIX
.287	ENDP	IFNB	RECORD
.8086	ENDS	IFNDEF	.SALL
.8087	EQU	INCLUDE	SEGMENT
=	EVEN	LABEL	.SFCOND
ASSUME	EXTRN	.LALL	STRUC
COMMENT	GROUP	.LFCOND	SUBTTL
.CREF	IF	.LIST	.TFCOND
DB	IF1	NAME	TITLE
DD	IF2	ORG	.XALL
DQ	IFB	%OUT	.XCREF
DT	IFDEF	PAGE	.XLIST
DW			

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

The following is a complete list of directive syntax and function.

.186 Enables assembly of 186 instructions.

.286c Enables assembly of 286 unprotected instructions.

.286p Enables assembly of 286 protected instructions.

.287 Enables assembly of 287 instructions.

.8086 Enables assembly of 8086 instructions while disabling assembly of 186 and 286 instructions.

.8087 Enables assembly of 8087 instructions while disabling assembly of 287 instructions.

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

ASSUME *seg-reg*:*seg-name*,,,
Selects the given segment register *seg-reg* to be the default segment register for all symbols in the named segment or group. If *seg-name* is NOTHING, no register is selected.

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

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

[*name*] **DB** *initial-value*,,,
Allocates and initializes a byte (8 bits) of storage for each *initial-value*.

[*name*] **DW** *initial-value*,,,
Allocates and initializes a word (2 bytes) of storage for each given *initial-value*.

[*name*] **DD** *initial-value*,,,
Allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value*.

[*name*] **DQ** *initial-value*,,,
Allocates and initializes a quadword (8 bytes) of storage for each given *initial-value*.

[*name*] **DT** *initial-value*,,,
Allocates and initializes 10 bytes of storage for each given *initial-value*.

ELSE Marks the beginning of an alternate block within a conditional block.

END [*expression*]
Marks the end of the module and optionally sets the program entry point to *expression*.

ENDIF Terminates a conditional block.

name **EQU** *expression*
Assigns the *expression* to the given *name*.

name **ENDP** Marks the end of a procedure definition.

name **ENDS** Marks the end of a segment or structure type definition.

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

EXTRN *name*:*type*,,,
Defines an external variable, label, or symbol named *name* and whose type is *type*.

name **GROUP** *seg-name*,,,
Associates a group name *name* with one or more segments.

IF *expression* Grants assembly if the *expression* is non-zero (true).

IF1 Grants assembly on pass 1 only.

IF2 Grants assembly on pass 2 only.

IFB < *arg* > Grants assembly if the *arg* is blank.

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

IFDIF < *arg1* >, < *arg2* >
Grants assembly if the arguments are different.

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

IFIDN < *arg1* >, < *arg2* >
Grants assembly if the arguments are identical.

IFNB < *arg* > Grants assembly if the *arg* is not blank.

IFNDEF *name*
Grants assembly if *name* has not yet been defined.

INCLUDE *filename*
Inserts source code from the source file given by *filename* into the current source file during assembly.

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

.LALL Lists all statements in a macro.

.LFCOND Restores the listing of conditional blocks.

.LIST Restores listing of statements in the program listing.

NAME *module-name*
Sets the name of the current module to *module-name*.

ORG *expression*
Sets the location counter to *expression*.

%OUT *text* Displays *text* at the user's terminal.

name **PROC** *type*
Marks the beginning of a procedure definition.

PUBLIC *name*,,,
Makes the variable, label, or absolute symbol given by *name* available to all other modules in the program.

.RADIX *expression*
Sets the input radix for numbers in the source file to *expression.*

recordname **RECORD** *fieldname*: *width* [= *exp*] ,,,
Defines an record type for a 8- or 16-bit record that contains one or more fields.

.SALL Suppresses listing of all macro expansions.

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

.SFCOND Suppresses listing of any subsequent conditional blocks whose IF condition is false.

name **STRUC** Marks the beginning of a type definition for a structure.

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

PAGE + Increments section page numbering.

PAGE Generates a page break in the listing.

SUBTTL *text* Defines the listing subtitle.

.TFCOND Sets the default mode for listing of conditional blocks.

TITLE *text* Defines the program listing title.

.XALL Lists only those macro statements that generate code or data.

.XCREF *name*,,,
Suppresses the listing of symbols in the cross-reference listing file.

.XLIST Suppresses listing of subsequent source lines to the program listing.

Appendix C
Segment Names For High-Level Languages

C.1 Introduction

This appendix describes the naming conventions used to form assembly language source files that are compatible with object modules produced by the Microsoft C, Pascal, and Fortran language compilers (version 3.0 or later).

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

TEXT	for program code
DATA	for program data
BSS	for uninitialized space
CONST	for constant data

Any assembly language source file that is to be assembled and linked to a high-level language module must use these segments as described in the following sections.

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

Small	for single code and data segments
Middle	for multiple code segment but a single data segment
Large	for multiple code and data segments

Assembly language source files to be assembled for a given memory model must use the naming conventions given in the following sections.

C.2 Text Segments

Syntax

```
name_TEXT   SEGMENT BYTE PUBLIC 'CODE'
            statements
name_TEXT   ENDS
```

A text segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *name*_TEXT, where *name* can be any valid name. For middle and large module programs, the module's own name is recommended. For small model programs, only "_TEXT" is allowed.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the CS segment register. Therefore, the statement

```
assume cs: name_TEXT
```

must appear at the beginning of the segment. This statement ensures that each label and variable declared in the segment will be associated with the CS segment register (see the section, "ASSUME Directive" in Chapter 3).

Text segments should have "BYTE" alignment and "PUBLIC" combination type, and must have the class name "CODE." These define loading instructions that are passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," in Chapter 3.

Small Model Programs. Only one text segment is allowed. The segment must not exceed 64 Kbytes. If the segment's complete definition is distributed among several modules, the statement

```
IGROUP      group   _TEXT
```

should be used at the beginning of each module to ensure that the segment is placed in a single 64 Kbyte physical segment. All procedure and statement labels should have the NEAR type.

Example

```
IGROUP group  _TEXT
      assume cs:IGROUP

_TEXT  segment       byte public 'CODE'
_main  proc near
       .
       .
       .
_main  endp
_TEXT  ends
```

Middle and Large Model Programs. Multiple text segments are allowed, however, no segment can be greater than 64 Kbytes. To distinguish one segment from another, each should have its own name. Since most modules contain only one text segment, the module's name is often used as part of the text segment's name. All procedure and statement labels should have the FAR type, unless they will only be accessed from within the same segment.

Example

```
SAMPLE_TEXT   segment       byte public 'CODE'
      assume cs:SAMPLE_TEXT
_main  proc   far
       .
       .
       .
_main  endp
SAMPLE_TEXT   ends
```

C.3 Data Segments – Near

Syntax

```
_DATA SEGMENT WORD PUBLIC 'DATA'
        statements
_DATA ENDS
```

A near data segment defines initialized data that is in the segment pointed to by the DS segment register when the program starts execution. The segment is "near" because all data in the segment is accessible without giving an explicit segment value. All programs have exactly one near data segment. Only large model programs can have additional data segments.

A near data segment's name must be "_DATA." The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the predefined group "DGROUP". Therefore, the statements

```
DGROUP      group _DATA
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the data segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive" in Chapter 3).

Near data segments must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "DATA." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," in Chapter 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

*name*_DATA	SEGMENT WORD PUBLIC 'FAR_DATA'
	statements
*name*_DATA	ENDS

A far data segment defines data or data space that can be accessed only by specifying an explicit segment value. Only large model programs can have far data segments.

A far data segment's name must be *name*_DATA, where *name* can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the ES segment register. When accessing a variable in a far data segment, the ES register must be set to the appropriate segment value. Also, the segment override operator must be used with the variable's name (see the section, "Attribute Operators" in Chapter 5).

Far data segments must be "WORD" aligned, must have "PUBLIC" combination type, and should have the class name "FAR_DATA." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," in Chapter 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

```
_BSS    SEGMENT WORD PUBLIC 'BSS'
        statements
_BSS    ENDS
```

A bss segment defines uninitialized data space. A bss segment's name must be "_BSS." The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group "DGROUP". Therefore, the statements

```
DGROUP      group _BSS
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the bss segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive" in Chapter 3).

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 BSS segment, the directive

```
DGROUP group _DATA, _BSS
```

should be used.

A bss segment must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "BSS." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," in Chapter 3.

Example

```
DGROUP group  _BSS
      assume ds: DGROUP

_BSS   segment       word public 'BSS'
count  dw      ?
array  dw      10 dup(?)
string db      30 dup(?)
_BSS   ends
```

C.6 Constant Segments

Syntax

```
CONST  SEGMENT WORD PUBLIC 'CONST'
       statements
CONST  ENDS
```

A constant segment defines constant data that will not change during program execution. Constant segments are typically used in large model programs to hold the segment values of far data segments.

The constant segment's name must be "CONST." The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group "DGROUP". Therefore, the statements

```
DGROUP      group CONST
       assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the constant segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive" in Chapter 3).

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 be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "CONST." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," in Chapter 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