2.0

TURBO ASSEMBLER®

REFERENCE GUIDE

void run() /* The main program lo
displaycell(curcol, currow, HIGH
curcell = cell[curcol][currow];
showcelltype();
input = getkey();
DisplayChar:
cmp [byte
je Exit
inc
jmp
CharacterOutput

BORLAND

Turbo Assembler®

Version 2.0

Reference Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001

PRINTED IN THE USA.
R1
10 9 8 7

C O N T E N T S

T A B L E S

F I G U R E S

I N T R O D U C T I O N

This book is the second of the two books accompanying the Turbo Assembler package. Now that you've probably thoroughly perused the first manual (*User's Guide*) in this set, you'll want to look at this one for all the nitty-gritty information.

The *Reference Guide* is just that—a straight and to-the-point reference about Turbo Assembler. If you find you still need to know more about the basics of assembly language, go back to the *User's Guide* for some in-depth discussions.

Hardware and software requirements

Turbo Assembler runs on the IBM PC family of computers, including the XT, AT, and PS/2, along with all true compatibles. Turbo Assembler requires MS-DOS 2.0 or later and at least 256K of memory.

Turbo Assembler generates instructions for the 8086, 80186, 80286, 80386, and 80486 processors. It also generates floating-point instructions for the 8087, 80287, and 80387 numeric coprocessors.

What's in this manual

Here's what we discuss in this manual:

Chapter 1: Predefined symbols tells you about Turbo Assembler's equates.

Chapter 2: Operators describes the various operators Turbo Assembler provides.

Chapter 3: Directives provides, in alphabetical order, detailed information about all the Turbo Assembler directives.

Appendix A: Turbo Assembler syntax illustrates Turbo Assembler expressions (both MASM and Ideal modes) in modified Backus-Naur form (BNF).

Appendix B: Compatibility issues covers the differences between MASM and Turbo Assembler MASM mode.

Appendix C: Turbo Assembler highlights details Turbo Assembler's enhancements that add to those of MASM.

Appendix D: Turbo Assembler utilities describes three of the utilities that come with this package: MAKE, TLINK, and TLIB; information about GREP, TCREF, and OBJXREF are in files on your distribution disks.

Appendix E: Turbo Assembler error messages describes all the error messages that can be generated when using Turbo Assembler: information messages, fatal error messages, warning messages, and error messages.

Notational conventions

When we talk about IBM PCs or compatibles, we're referring to any computer that uses the 8088, 8086, 80186, 80286, and 80386 chips (all of these chips are commonly referred to as 80x86). When discussing PC-DOS, DOS, or MS-DOS, we're referring to version 2.0 or greater of the operating system.

All typefaces were produced by Borland's Sprint: The Professional Word Processor, output on a PostScript printer. The different typefaces displayed are used for the following purposes:

Italics	In text, italics represent labels, placeholders, variables, and arrays. In syntax expressions, placeholders are set in italics to indicate that they are user-defined.
Boldface	Boldface is used in text for symbols and command-line options.
CAPITALS	In text, this typeface represents instructions, directives, registers, and operators.
`Monospace`	Monospace type is used to display any sample code, text or code that appears on your screen, and any text that you must

	actually type to assemble, link, and run a program.
Keycaps	In text, keycaps are used to indicate a key on your keyboard. It is often used when describing a key you must press to perform a particular function; for example, "Press *Enter* after typing your program name at the prompt."

How to contact Borland

If, after reading this manual and using Turbo Assembler, you would like to contact Borland with comments, questions, or suggestions, we suggest the following procedures:

- The best way is to log on to Borland's forum on CompuServe: Type `GO BPROGB` at the main CompuServe menu and follow the menus to Turbo Assembler. Leave your questions or comments here for the support staff to process.
- If you prefer, write a letter detailing your problem and send it to

  Technical Support Department
  Borland International
  P.O. Box 660001
  1700 Green Hills Drive
  Scotts Valley, CA 95066 U.S.

- You can also telephone our Technical Support department at (408) 438-5300. To help us handle your problem as quickly as possible, have these items handy before you call:
  - product name and version number
  - product serial number
  - computer make and model number
  - operating system and version number

If you're not familiar with Borland's No-Nonsense License statement, now's the time to read the agreement at the front of this manual and mail in your completed product registration card.

C H A P T E R

1

Predefined symbols

Turbo Assembler provides a number of predefined symbols that you can use in your programs. These symbols can have different values at different places in your source file. They are similar to equated symbols that you define using the **EQU** directive. When Turbo Assembler encounters one of these symbols in your source file, it replaces it with the current value of that predefined symbol.

Some of these symbols are text (string) equates, some are numeric equates, and others are aliases. The string values can be used anywhere that you would use a character string, for example to initialize a series of data bytes using the **DB** directive:

```
NOW   DB   ??time
```

Numeric predefined values can be used anywhere that you would use a number:

```
IF ??version GT 100h
```

Alias values make the predefined symbol into a synonym for the value it represents, allowing you to use the predefined symbol name anywhere you would use an ordinary symbol name:

```
ASSUME  cs:@code
```

All the predefined symbols can be used in both MASM and Ideal mode.

If you use the **/ml** command-line option when assembling, you must use the predefined symbol names exactly as they are described on the following pages.

The following rule applies to predefined symbols starting with an at-sign (@): *The first letter of each word that makes up part of the symbol name is an uppercase letter (except for segment names); the rest of the word is lowercase*. As an example,

```
@CodeSize
@FileName
@WordSize
```

The exception is redefined symbols, which refer to segments. Segment names begin with an at-sign (@) and are all lowercase. For example,

```
@curseg
@fardata
```

For symbols that start with two question marks (??), the letters are all lowercase. For example,

```
??data
??version
```

$

Function Location counter operand

Remarks This special symbol represents the current location counter. The location counter is the current offset within the current segment during assembly. (Note: This operand has the same attribute as a near label.)

The location counter is an address that is incremented to reflect the current address as each statement in the source file is assembled.

Example

```
helpMessage   DB    'This is help for the program.'
helplength    =  $ - helpMessage
```

After these two lines are assembled, the symbol *helpLength* will equal the length of the help message.

@code

Function Alias equate for **.CODE** segment name

Remarks When you use the simplified segmentation directives (**.MODEL**, and so on), this equate lets you use the name of the code segment in expressions, such as **ASSUME**s and segment overrides.

Example

```
.CODE
mov ax,@code
mov dx,ax
ASSUME ds:@code
```

@CodeSize

Function Numeric equate that indicates code memory model

Remarks **@CodeSize** is set to 0 for the small and compact memory models that use near code pointers, and is set to 1 for all other models that use far code pointers.

You can use this symbol to control how pointers to functions are assembled, based on the memory model.

Example

```
     IF  @CodeSize EQ 0
procptr  DW   PROC1      ;pointer to near procedure
     ELSE
procptr  DD   PROC1      ;pointer to far procedure
     ENDIF
```

@Cpu

Function Numeric equate that returns information about current processor

Remarks The value returned by **@Cpu** encodes the processor type in a number of single-bit fields:

Bit	Description
0	8086 instructions enabled
1	80186 instructions enabled
2	80286 instructions enabled
3	80386 instructions enabled
7	Privileged instructions enabled (80286 and 80386)
8	8087 numeric processor instructions
10	80287 numeric processor instructions
11	80387 numeric processor instructions

The bits not defined here are reserved for future use. Mask them off when using **@Cpu** so that your programs will remain compatible with future versions of Turbo Assembler.

Since the 8086 processor family is upward compatible, when you enable a processor type with a directive like **.286**, the lower processor types (8086, 80186) are automatically enabled as well.

Note: This equate *only* provides information about the processor you've selected at assembly-time via the **.286** and related directives. The processor type your program is executing on at run time is not indicated.

Example

```
IPUSH = @Cpu AND 2  ;allow immediate push on 186 and above
IF IPUSH
PUSH 1234
ELSE
   mov  ax,1234
   push ax
ENDIF
```

@curseg

Function Alias equate for current segment

Remarks **@curseg** changes throughout assembly to reflect the current segment name. You usually use this after you have used the simplified segmentation directives (**.MODEL**, and so on).

Use **@curseg** to generate **ASSUME** statements, segment overrides, or any other statement that needs to use the current segment name.

Example

```
.CODE
ASSUME cs:@curseg
```

@data

Function Alias equate for near data group name

Remarks When you use the simplified segmentation directives (**.MODEL**, and so on), this equate lets you use the group name shared by all the near data segments (**.DATA**, **.CONST**, **.STACK**) in expressions such as **ASSUME**s and segment overrides.

Example

```
.CODE
mov ax,@data
mov ds,ax
ASSUME ds:@data
```

@DataSize

Function Numeric equate that indicates the data memory model

Remarks **@DataSize** is set to 0 for the tiny, small, and medium memory models that use near data pointers, set to 1 for the compact and large models that use far data pointers, and set to 2 for the huge memory model.

You can use this symbol to control how pointers to data are assembled, based on the memory model.

Example

```
IF  @DataSize EQ 1
    lea  si,VarPtr
    mov  al,[BYTE PTR si]
ELSE
    les  si,VarPtr
    mov  al,[BYTE PTR es:si]
ENDIF
```

??date

Function String equate for today's date

Remarks **??date** defines a text equate that represents today's date. The exact format of the date string is determined by the DOS country code.

See also **??time**

Example

```
asmdate DB  ??date   ;8-byte string
```

@fardata

Function Alias equate for initialized far data segment name

Remarks When you use the simplified segmentation directives (**.MODEL**, and so on), this equate lets you use the name of the initialized far data segment (**.FARDATA**) in expressions such as **ASSUME**s and segment overrides.

Example

```
mov ax,@fardata
mov ds,ax
ASSUME ds:@fardata
```

@fardata?

Function Alias equate for uninitialized far data segment name

Remarks When you use the simplified segmentation directives (**.MODEL**, and so on), this equate lets you use the name of the uninitialized far data segment (**.FARDATA?**) in expressions such as **ASSUME**s and segment overrides.

Example

```
mov ax,@fardata?
mov ds,ax
ASSUME ds:@fardata?
```

@FileName

Function Alias equate for current assembly file

See also **??filename**

??filename

Function String equate for current assembly file

Remarks **??filename** defines an eight-character string that represents the file name being assembled. If the file name is less than eight characters, it is padded with spaces.

Example

```
SrcName DB   ??filename   ;8-bytes always
```

@Model

Function Numeric equate that indicates the model currently in effect

Remarks **@Model** returns an integer indicating the current model:

- 0 = TINY
- 1 = SMALL
- 2 = COMPACT
- 3 = MEDIUM
- 4 = LARGE
- 5 = HUGE

See also **.MODEL, MODEL**

@Startup

Function Label that marks the beginning of startup code

Remarks **@Startup** is a near label, defined by the **.STARTUP** or **STARTUPCODE** directive, which marks the beginning of the startup code generated by that directive.

See also **.STARTUP, STARTUPCODE**

??time

Function String equate for the current time

Remarks **??time** defines a text equate that represents the current time. The exact format of the time string is determined by the DOS country code.

See also **??date**

Example

```
asmtime DB  ??time   ;8-byte string
```

??version

Function Numeric equate for this Turbo Assembler version

Remarks The high byte is the major version number, and the low byte is the minor version number. For example, V2.1 would be represented as 201h.

??version lets you write source files that can take advantage of features in particular versions of Turbo Assembler.

This equate also lets your source files know whether they are being assembled by MASM or Turbo Assembler, since **??version** is not defined by MASM.

Example

```
IFDEF ??version
;Turbo Assembler stuff
ENDIF
```

@WordSize

Function Numeric equate that indicates 16- or 32-bit segments

Remarks **@WordSize** returns 2 if the current segment is a 16-bit segment, or 4 if the segment is a 32-bit segment.

Example

```
IF @WordSize EQ 4
   mov esp,0100h
ELSE
   mov sp,0100h
ENDIF
```

C H A P T E R

2

Operators

Operators let you form complex expressions that can be used as an operand to an instruction or a directive. Operators act upon operands, such as program symbol names and constant values. Turbo Assembler evaluates expressions when it assembles your source file and uses the calculated result in place of the expression. One way you can use expressions is to calculate a value that depends on other values that may change as you modify your source file.

This chapter details the operators Turbo Assembler provides.

Arithmetic precision

Turbo Assembler uses 16- or 32-bit arithmetic, depending on whether you have enabled the 80386 processor with the **.386** or **.386P** directive. When you are assembling code for the 80386 processor or in Ideal mode, some expressions will yield different results than when they are evaluted in 16-bit mode; for example,

```
DW   (1000h * 1000h) / 1000h
```

generates a word of 1000h in 32-bit mode and a word of 0 in 16-bit mode. In 16-bit mode, multiplication results in an overflow condition, saving only the bottom 16 bits of the result.

Operator precedence

Turbo Assembler evaluates expressions using the following rules:

- Operators with higher precedence are performed before ones with lower precedence.
- Operators with the same precedence are performed starting with the leftmost one in the expression.
- If an expression contains a subexpression within parentheses, that subexpression is evaluated first because expressions within parentheses have the highest priority.

Ideal mode and MASM mode have a different precedence for some operators. The following two tables show the precedence of the operators in both modes. The first line in each table shows the operators with the highest priority, and the last line those operators with the lowest priority. Within a line, all the operators have the same priority.

Table 2.1 MASM mode operator precedence

<>, (), [], LENGTH, MASK, SIZE, WIDTH
. (structure member selector)
HIGH, LOW
+, – (unary)
: (segment override)
OFFSET, PTR, SEG, THIS, TYPE
***, /, MOD, SHL, SHR**
+, – (binary)
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
LARGE, SHORT, SMALL, .TYPE

Table 2.2
Ideal mode operator precedence

(), [], LENGTH, MASK, OFFSET, SEG, SIZE, WIDTH
HIGH, LOW
+, – (unary)
***, /, MOD, SHL, SHR**
+, – (binary)
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
: (segment override)
. (structure member selector)
HIGH (before pointer), **LARGE, LOW** (before pointer), **PTR, SHORT, SMALL, SYMTYPE**

The operators allowed in expressions follow in alphabetical order.

( ) Ideal, MASM

Function Marks an expression for early evaluation

Syntax `(expression)`

Remarks Use parentheses to alter the normal priority of operator evaluation. Any expression enclosed within parentheses will be evaluated before the operator(s) that comes before or after the parentheses.

See also **+, –, *, /, MOD, SHL, SHR**

Example

```
(3 + 4) * 5   ;evaluates to 35
3 + 4 * 5     ;evaluates to 23
```

* Ideal, MASM

Function Multiplies two integer expressions

Syntax *expression1 * expression2*

Remarks *expression1* and *expression2* must both evaluate to integer constants. The * operator can also be used between a register and a constant to support 386 addressing modes.

See also **+, –, /, MOD, SHL, SHR**

Example
```
SCREENSIZE = 25 * 80   ;# chars onscreen
```

The * operator can also be used between a register and a constant to support 386 addressing modes.

+ (Binary) Ideal, MASM

Function Adds two expressions

Syntax *expression1 + expression2*

Remarks At least one of *expression1* or *expression2* must evaluate to a constant. One expression can evaluate to an address.

See also **–, *, /, MOD, SHL, SHR**

Example
```
X    DW  4 DUP (?)
XPTR DW  X + 4      ;third word in buffer
```

+ (Unary) Ideal, MASM

Function Indicates a positive number

Syntax *+ expression*

Remarks This operator has no effect. It is available merely to make explicit positive constants.

See also **–, *, /, MOD, SHL, SHR**

Example
```
Foo  DB  +4   ;redundant +
```

– (Binary) Ideal, MASM

Function Subtracts two expressions

Syntax *expression1 - expression2*

Remarks There are three combinations of operands you can use with subtraction:

- *expression1* and *expression2* can both evaluate to integer constants.
- *expression1* and *expression2* can both be addresses as long as both addresses are within the same segment. When you subtract two addresses, the result is a constant. Ideal mode checks *expression1* and *expression2* for consistency much more stringently than MASM mode. In particular, Ideal mode correctly handles subtracting a segment from a far pointer, leaving just an offset fixup in cases where this is a valid operation.
- *expression1* can be an address and *expression2* can be a constant, but not vice versa. The result is another address.

See also **+, *, /, MOD, SHL, SHR**

Example

```
DATA  SEGMENT
         DW   ?
XYZ      EQU  10
VAL1     DW   XYZ  - 1       ;constant 9
VAL2     DW   ?
V1SIZE   DW   VAL2 - VAL1    ;constant 2
V1BEFORE DW   VAL1 - 2       ;points to DW before VAL1E
DATA     ENDS
```

– (Unary) Ideal, MASM

Function Changes the sign of an expression

Syntax *- expression*

Remarks *expression* must evaluate to a constant. If *expression* is positive, the result will be a negative number of the same magnitude. If *expression* is negative, the result will be a positive number.

See also **+, *, /, MOD, SHL, SHR**

Example

```
LOWTEMP  DB -10   ;pretty chilly
```

. Ideal, MASM

Function Selects a structure member

Syntax *memptr.fieldname*

Remarks In MASM mode, *memptr* can be any operand that refers to a memory location, and *fieldname* can be the name of any member in any structure or even a constant expression. If *memptr* is the name of a structure (like XINST), there's no requirement that *fieldname* be a member in that structure. This operator acts much like the + operator: It adds the offset within the structure of *fieldname* to the memory address of *memptr*, but it also gives it the size of field name.

In Ideal mode, its operation is much stricter. *memptr* must evaluate to a pointer to a structure, and *fieldname* must be a member of that structure. This lets you have different structures with the same field names, but a different offset and size. If you want to use a base and/or index register for *memptr*, you must precede it with a typecast for the name of the structure you want to access (take a look at the example that follows).

See also **STRUC**

Example

```
X  STRUC
MEMBER1 DB  ?
MEMBER2 DW  ?
X  ENDS
XINST   X   <>

;an instance of STRUC X
;MASM mode
   mov  [bx].Member2,1
;Ideal mode
   mov [(X PTR bx).Member2],1   ;notice typecast
```

/ Ideal, MASM

Function Divides two integer expressions

Syntax *expression1 / expression2*

Remarks *expression1* and *expression2* must both evaluate to integer constants. The result is *expression1* is divided by *expression2*; any remainder is discarded. You can get the remainder by using the **MOD** operator with the same operands you supplied to the / operator.

See also **+, –, *, MOD, SHL, SHR**

Example

```
X = 55 / 10   ;= 5 (integer divide)
```

: Ideal, MASM

Function Generates segment or group override

Syntax *segorgroup : expression*

Remarks The colon (:) forces the address of *expression* to be generated relative to the specified segment or group. You use this to force the assembler to use something other than its default method for accessing *expression*.

You can specify *segorgroup* in several ways:

- as a segment register: CS, DS, ES, or SS (or FS or GS if you have enabled the 80386 processor with the **P386** or **P386N** directive)
- as a segment name defined with the **SEGMENT** directive
- as a group name defined with the **GROUP** directive
- as an expression starting with the **SEG** operator

expression can be a constant- or a memory-referencing expression.

Example

```
        mov cl,es:[si+4]
VarPtr     DD   DGROUP:MEMVAR
```

? Ideal, MASM

Function Initializes with indeterminate data

Syntax `Dx ?`

Remarks *Dx* refers to one of the data allocation directives (**DB**, **DD**, and so on). Use **?** when you want to allocate data but don't want to explicitly assign a value to it.

You should use **?** when the data area is initialized by your program before being used. Using **?** rather than 0 makes the initialization method explicit and visible in your source code.

When you use **?** as the only value in or outside a **DUP** expression, no object code is generated. If you use **?** inside a **DUP** expression that also contains initialized values, it will be treated as a 0.

Using **?** outside of a **DUP** causes the program counter to be advanced but no data to be emitted. You can also use **?** for the same purpose in, for example, a structure:

```
x struc      ;declare structure
a db 1
x ends

xinst ?      ;undefined structure instance
```

See also **DUP**

Example

```
MyBuf DB 20 DUP (?)   ;allocate undefined area
```

[ ] operator — Ideal, MASM

Function Specifies addition or indexed memory operand

Syntax `[`*expression1*`] [`*expression2*`]`

Remarks This operator behaves very differently in MASM mode and in Ideal mode.

In MASM mode, it can act as an addition operator, simply adding *expression1* to *expression2*. The same limitations on operand combinations apply; for example, *expression1* and *expression2* can't both be addresses. **[ ]** can also indicate register indirect memory operands, using the BX, BP, SI, and DI registers. The indirect register(s) must be enclosed within the **[ ]**. An indirect displacement may appear either inside or outside the brackets.

In Ideal mode, **[ ]** means "memory reference." Any operand that addresses memory must be enclosed in brackets. This provides a clear, predictable, unambiguous way of controlling whether an operand is immediate or memory-referencing.

See also **+**

Example

```
; MASM mode
   mov al,BYTE PTR es:[bx]
   mov al,cs:10h
; Ideal mode
   mov al,[BYTE es:bx]
   mov al,[cs:10h]
```

AND — Ideal, MASM

Function Bitwise logical **AND**

Syntax `expression1 AND expression2`

Remarks Performs a bit-by-bit logical **AND** of each bit in *expression1* and *expression2*. The result has a 1 in any bit position that had a 1 in both expressions and a 0 in all other bit positions.

See also **NOT, OR, XOR**

Example

```
mov  al,11110000b AND 10100000B   ;loads 10100000B
```

BYTE — Ideal

Function Forces expression to be byte size

Syntax `BYTE expression`

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **BYTE** size, regardless of the original size of expression.

You usually use this operator to define the size of a forward-referenced expression, or to explicitly state the size of a register indirect expression from which the size cannot be determined.

In MASM mode, you must use the **PTR** directive preceded with the **BYTE** type to perform this function.

See also **PTR**

Example

```
mov [BYTE bx],1     ;byte immediate move
mov [BYTE X],1      ;forward reference
X   DB  0
```

CODEPTR — Ideal

Function Returns the default procedure address size

Mode Ideal

Syntax `CODEPTR expression`

Remarks **CODEPTR** returns the default procedure address size depending on the current model (**WORD** for models with **NEAR** code; **DWORD** for models with **FAR** code). **CODEPTR** can be used wherever **DATAPTR** is used.

See also **DATAPTR**

DATAPTR — Ideal

Function Forces expression to model-dependent size

Syntax `DATAPTR` *`expression`*

Remarks Declares expression to be a near or far pointer, depending on selected memory model.

See also **CODEPTR, PTR, UNKNOWN**

Example

```
mov  [DATAPTR bx],1
```

DUP — Ideal, MASM

Function Repeats a data allocation

Syntax *`count`* `DUP (`*`expression [,expression]...`*`)`

Remarks *count* defines the number of times that the data defined by the *expression*(s) will be repeated. The **DUP** operator appears after one of the data allocation directives (**DB**, **DW**, and so on).

Each expression is an initializing value that is valid for the particular data allocation type that **DUP** follows.

You can use the **DUP** operator again within an expression, nested up to 17 levels.

You must always surround the expression values with parentheses, ().

Example

```
WRDBUF  DW  40 DUP (1)          ;40 words initialized to 1
SQUARE  DB   4 DUP (4 DUP (0))  ;4x4 array of 0
```

DWORD — Ideal

Function Forces expression to be doubleword size

Syntax `DWORD` *`expression`*

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **DWORD** size, regardless of the original expression size.

You usually use this operator to define the size of a forward-referenced expression.

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **DWORD** type.

See also **PTR**

Example

```
call DWORD FPTR
```

EQ — Ideal, MASM

Function Returns true if expressions are equal

Syntax *`expression1`* `EQ` *`expression2`*

Remarks *expression1* and *expression2* must both evaluate to constants. **EQ** returns true (-1) if both expressions are equal and returns false (0) if they have different values.

EQ considers *expression1* and *expression2* to be signed 32-bit numbers, with the top bit being the sign bit. This means that –1 **EQ** 0FFFFFFFFh evaluates to true.

See also **GE, GT, LE, LT, NE**

Example

```
ALIE   =  4 EQ 3   ;= 0 (false)
ATRUTH =  6 EQ 6   ;= 1 (true)
```

FAR Ideal

Function Forces an expression to be a far code pointer

Syntax `FAR expression`

Remarks *expression* must be an address. The result is an expression that points to the same memory address but is a far pointer with both a segment and an offset, regardless of the original expression type.

You usually use this operator to call or jump to a forward-referenced label that is declared as **FAR** later in the source file.

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **FAR** type.

See also **NEAR**

Example

```
     call FAR ABC    ;forward reference
ABC  PROC FAR
```

FWORD Ideal

Function Forces expression to be 32-bit far pointer size

Syntax `FWORD expression`

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **FWORD** size, regardless of the original expression size.

You usually use this operator to define the size of a forward-referenced expression or to explicitly state the size of a register indirect expression from which the size cannot be determined.

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **FWORD** type.

See also **PTR, PWORD**

Example

```
.386
call FWORD [bx]    ;far indirect 48-bit call
jmp  FWORD funcp   ;forward reference
funcp DF  myproc   ;indirect pointer to PROC
```

GE — Ideal, MASM

Function Returns true if one expression is greater than another

Syntax *expression1* GE *expression2*

Remarks *expression1* and *expression2* must both evaluate to constants. **GE** returns true (-1) if *expression1* is greater than or equal to *expression2* and returns false (0) if it is less.

GE considers *expression1* and *expression2* to be signed 33-bit numbers, with the top bit being the sign bit. This means that 1 **GE** –1 evaluates to true, but 1 **GE** 0FFFFFFFFh evaluates to false.

See also **EQ, GT, LE, LT, NE**

Example

```
TROOTH =  5 GE 5
AFIB   =  5 GE 6
```

GT — Ideal, MASM

Function Returns true if one expression is greater than another

Syntax *expression1* GT *expression2*

Remarks *expression1* and *expression2* must both evaluate to constants. **GT** returns true (-1) if expression1 is greater than *expression2*, and returns false (0) if it is less than or equal.

GT considers *expression1* and *expression2* to be signed 33-bit numbers, with the top bit being the sign bit. This means that 1 **GT** –1 evaluates to true, but 1 **GT** 0FFFFFFFFh evaluates to false.

See also **EQ, GE, LE, LT, NE**

Example

```
AFACT =  10 GT 9
NOTSO =  10 GT 11
```

HIGH — Ideal, MASM

Function Returns the high part of an expression

Syntax `HIGH` *expression*

Ideal mode only:
`type HIGH` *expression*

Remarks **HIGH** returns the top 8 bits of *expression*, which must evaluate to a constant.

In Ideal mode, **HIGH** in conjunction with **LOW** becomes a powerful mechanism for extracting arbitrary fields from data items. *type* specifies the size of the field to extract from *expression* and can be any of the usual size specifiers (**BYTE, WORD, DWORD**, and so on). You can apply more than one **HIGH** or **LOW** operator to an expression; for example, the following is a byte address pointing to the third byte of the doubleword **DBLVAL:**

```
BYTE LOW WORD HIGH DBLVAL
```

See also **LOW**

Example

```
;MASM and Ideal modes
magic EQU   1234h
mov  cl,HIGH magic
Ideal
;Ideal mode  only
big  DD  12345678h
mov  ax,[WORD HIGH big]    ;loads 1234h into AX
```

LARGE — Ideal (386 modes only), MASM

Function Sets an expression's offset size to 32 bits

Syntax `LARGE` *expression*

Remarks *expression* is any expression or operand, which **LARGE** converts into a 32-bit offset. You usually use this to remove ambiguity about the size of an operation. For example, if you have enabled the 80386 processor with the **P386** directive, this code can be interpreted as either a far call with a segment and 16-bit offset or a near call using a 32-bit offset:

```
jmp [DWORD PTR ABC]
```

You can remove the ambiguity by using the **LARGE** directive:

```
jmp LARGE [DWORD PTR ABC]    ;32-bit offset near call
```

In this example, **LARGE** appears outside the brackets, thereby affecting the interpretation of the **DWORD** read from memory. If **LARGE** appears inside the brackets, it determines the size of the address from which to read the operand, not the size of the operand once it is read from memory. For example, this code means *XYZ* is a 4-byte pointer:

```
jmp LARGE [LARGE DWORD PTR XYZ]
```

Treat it as a 32-bit offset, and **JMP** indirect through that address, reading a **JMP** target address that is also a 32-bit offset.

By combining the **LARGE** and **SMALL** operators, both inside and outside brackets, you can effect any combination of an indirect **JMP** or **CALL** from a 16- or 32-bit segment to a 16- or 32-bit segment.

You can also use **LARGE** to avoid erroneous assumptions when accessing forward-referenced variables:

```
mov ax,[LARGE FOOBAR]    ;FOOBAR is in a USE32 segment
```

LARGE and **SMALL** can be used with other ambiguous instructions, such as **LIDT** and **LGDT**.

See also **SMALL**

Example

```
;MASM and Ideal modes
magic   EQU 1234h
mov bl, HIGH magic
Ideal

;Ideal mode only
big DD 12345678h
mov ax,[word HIGH big]    ;leads 1234h into AX
```

LE — Ideal, MASM

Function Returns true if one expression is less than or equal to another

Syntax `expression1 LE expression2`

Remarks *expression1* and *expression2* must both evaluate to constants. **LE** returns true (-1) if *expression1* is less than or equal to *expression2* and returns false (0) if it is greater.

LE considers *expression1* and *expression2* to be signed 33-bit numbers, with the top bit being the sign bit. This means that 1 **LE** –1 evaluates to false, but 1 **LE** 0FFFFFFFFh evaluates to true.

See also **EQ, GE, GT, LT, NE**

Example
```
YUP = 5 LT 6   ;true = -1
```

LENGTH — Ideal, MASM

Function Returns number of allocated data elements

Syntax `LENGTH` *`name`*

Remarks *name* is a symbol that refers to a data item allocated with one of the data allocation directives (**DB**, **DD**, and so on). **LENGTH** returns the number of repeated elements in *name*. If *name* was not declared using the **DUP** operator, it always returns 1.

LENGTH returns 1 even when *name* refers to a data item that you allocated with multiple items (by separating them with commas).

See also **SIZE, TYPE**

Example
```
MSG     DB   "Hello"
array   DW   10 DUP(0)
numbrs  DD   1,2,3,4
var     DQ   ?
lmsg = LENGTH MSG         ;= 1, no DUP
larray = LENGTH ARRAY     ;=10, DUP repeat count
lnumbrs = LENGTH NUMBRS   ;= 1, no DUP
lvar = LENGTH VAR         ;= 1, no DUP
```

LOW — Ideal, MASM

Function Returns the low part of an expression

Syntax `LOW` *`expression`*

Ideal mode only:
`type LOW` *`expression`*

Remarks **LOW** returns the bottom 8 bits of expression, which must evaluate to a constant.

In Ideal mode, **LOW** in conjunction with **HIGH** becomes a powerful mechanism for extracting arbitrary fields from data items. *type* specifies the size of the field to extract from *expression* and can be any of the usual size specifiers (**BYTE**, **WORD**, **DWORD**, and so on). You can apply more than one **LOW** or **HIGH** operator to an expression; for example,

```
  BYTE LOW WORD HIGH DBLVAL
```

is a byte address pointing to the third byte of the doubleword **DBLVAL**.

See also **HIGH**

Example

```
;MASM and Ideal modes
magic EQU   1234h
mov  bl,LOW magic
ideal
;Ideal mode only
big  DD  12345678h
mov  ax,[WORD LOW big]   ;loads 5678h into AX
```

LT — Ideal, MASM

Function Returns true if one expression is less than another

Syntax `expression1 LT expression2`

Remarks *expression1* and *expression2* must both evaluate to constants. **LT** returns true (-1) if *expression1* is less than *expression2* and returns false (0) if it is greater than or equal.

LT considers *expression1* and *expression2* to be signed 33-bit numbers, with the top bit being the sign bit. This means that 1 **LT** –1 evaluates to false, but 1 **LT** 0FFFFFFFFH evaluates to true.

See also **EQ, GE, GT, LE, NE**

Example

```
JA =  3 LT 4   ;true = -1
```

MASK — Ideal, MASM

Function Returns a bit mask for a record field

Syntax

```
MASK recordfieldname
MASK record
```

Remarks *recordfieldname* is the name of any field name in a previously defined record. **MASK** returns a value with bits turned on to correspond to the position in the record that *recordfieldname* occupies.

record is the name of a previously defined record. **MASK** returns a value with bits turned on for all the fields in the record.

You can use **MASK** to isolate an individual field in a record by **AND**ing the mask value with the entire record.

See also **WIDTH**

Example

```
STAT    RECORD A:3,b:4,c:5
NEWSTAT STAT <0,2,1>
mov  al,NEWSTAT          ;get record
and  al,MASK B           ;isolate B
mov  al,MASK STAT        ;get mask for entire record
```

MOD — Ideal, MASM

Function Returns remainder (modulus) from dividing two expressions

Syntax `expressions1 MOD expression2`

Remarks *expression1* and *expression2* must both evaluate to integer constants. The result is the remainder of *expression1* divided by *expression2*.

See also **+, –, *, /, SHL, SHR**

Example

```
REMAINS = 17 / 5   ;= 2
```

NE — Ideal, MASM

Function Returns true if expressions are not equal

Syntax `expression1 NE expression2`

Remarks *expression1* and *expression2* must both evaluate to constants. **NE** returns true (-1) if both expressions are not equal and returns false (0) if they are equal.

NE considers *expression1* and *expression2* to be signed 32-bit numbers, with the top bit being the sign bit. This means that –1 **NE** 0FFFFFFFFh evaluates to false.

See also **EQ, GE, GT, LE, LT**

Example

```
aint = 10 NE 10   ;false = 0
```

NEAR — Ideal

Function Forces an expression to be a near code pointer

Syntax `NEAR` *`expression`*

Remarks *expression* must be an address. The result is an expression that points to the same memory address but is a **NEAR** pointer with only an offset and no segment, regardless of the original expression type.

You usually use this operator to call or jump to a far label or procedure with a near jump or call instruction. See the example section for a typical scenario.

To perform this function in MASM mode, you must use the **PTR** directive preceded with the **NEAR** type.

See also **FAR**

Example

```
Ideal
PROC farp FAR
   ;body of procedure
ENDP farp
;still in same segment
push cs
call NEAR PTR farp   ;faster/smaller than far call
```

NOT — Ideal, MASM

Function Bitwise complement

Syntax `NOT` *`expression`*

Remarks **NOT** inverts all the bits in *expression*, turning 0 bits into 1 and 1 bits into 0.

See also **AND, OR, XOR**

Example

```
mov al,NOT 11110011b   ;loads 00001100b
```

OFFSET — Ideal, MASM

Function Returns an offset within a segment

Syntax `OFFSET` *`expression`*

Remarks *expression* can be any expression or operand that references a memory location. **OFFSET** returns a constant that represents the number of bytes between the start of the segment and the referenced memory location.

If you are using the simplified segmentation directives (**MODEL**, and so on) or Ideal mode, **OFFSET** automatically returns offsets from the start of the group that a segment belongs to. If you are using the normal segmentation directives, and you want an offset from the start of a group rather than a segment, you must explicitly state the group as part of *expression*. For example,

```
mov si,OFFSET BUFFER
```

is not the same as

```
mov si,OFFSET DGROUP:BUFFER
```

unless the segment that contains **BUFFER** happens to be the first segment in **DGROUP**.

See also **SEG**

Example

```
.DATA
msg  DB "Starting analysis"
.CODE
mov  si,OFFSET msg    ;address of MSG
```

OR — Ideal, MASM

Function Bitwise logical **OR**

Syntax *`expression1`* `OR` *`expression2`*

Remarks **OR** performs a bit-by-bit logical **OR** of each bit in *expression1* and *expression2*. The result has a 1 in any bit position that had a 1 in either or both expressions, and a 0 in all other bit positions.

See also **AND, NOT, XOR**

Example

```
mov  al,11110000b OR 10101010b ;loads 11111010b
```

PROC — Ideal

Function Forces an expression to be a near or far code pointer

Syntax `PROC` *`expression`*

Remarks *expression* must be an address. The result is an expression that points to the same memory address but is a near or far pointer, regardless of the original expression type. If you specified the **TINY, SMALL,** or **COMPACT** memory model with the **.MODEL** directive, the pointer will be near. Otherwise, it will be a far pointer.

You usually use **PROC** to call or jump to a forward-referenced function when you are using the simplified segmentation directives. The example section shows a typical scenario.

To perform this function in MASM mode, you must use the **PTR** directive preceded with the **PROC** type.

See also **FAR, NEAR**

Example

```
.MODEL large
.CODE
Ideal
call PROC Test1
     PROC Test1   ;actually far due to large model
```

PTR — Ideal, MASM

Function Forces expression to have a particular size

Syntax *`type`* `PTR` *`expression`*

Remarks *expression* must be an address. The result of this operation is a reference to the same address, but with a different size, as determined by *type.*

Typically, this operator is used to explicitly state the size of an expression whose size is undetermined, but required. This can occur if an expression is forward referenced, for example.

type must be one of the following in Ideal mode:

- **UNKNOWN, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPTR, CODEPTR,** or the name of a structure, for data
- **SHORT, NEAR, FAR, PROC** for code

In Ideal mode, you don't need to use the **PTR** operator. You can simply follow the *type* directly with *expression*.

In MASM mode, *type* can be any of the following numbers.

- For data:

0 = **UNKNOWN**	6 = **PWORD**
1 = **BYTE**	8 = **QWORD**
2 = **WORD**	10 = **TBYTE**
4 = **DWORD**	

- For code:

0FFFFh = **NEAR**	0FFFEh = **FAR**

Correspondingly, in MASM mode the following keywords have these values.

- For data:

UNKNOWN = 0	**FWORD** = 6
BYTE = 1	**QWORD** = 8
WORD = 2	**TBYTE** = 10
DWORD = 4	**DATAPTR** = 2 or 4 (depending on **MODEL** in use)
PWORD = 6	**CODEPTR** = 2 or 4 (depending on **MODEL** in use)
FWORD = 6	

- For code:

NEAR = 0FFFFh	**PROC** = 0FFFFh or 0FFFEh (depending on **MODEL** in use)
FAR = 0FFFEh	

See also **BYTE, CODEPTR, DATAPTR, DWORD, FAR, FWORD, NEAR, PROC, PWORD, QWORD, TBYTE, WORD**

Example

```
mov  BYTE PTR[SI],10   ;byte immediate mode
fld  QWORD PTR val     ;load quadword float
val  DQ  1234.5678
```

PWORD — Ideal, MASM

Function Forces expression to be 32-bit, far pointer size

See also **FWORD**

QWORD — Ideal

Function Forces expression to be quadword size

Syntax `QWORD` *`expression`*

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **QWORD** size, regardless of the original size of *expression.*

You usually use **QWORD** to define the size of a forward-referenced expression, or to explicitly state the size of a register indirect expression from which the size cannot be determined.

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **QWORD** type.

See also **PTR**

Example

```
fadd  [QWORD BX]   ;sizeless indirect
fsubp [QWORD X]    ;forward reference
.DATA
X  DQ   1.234
```

SEG — Ideal, MASM

Function Returns the segment address of an expression

Syntax `SEG` *`expression`*

Remarks *expression* can be any expression or operand that references a memory location. **SEG** returns a constant that represents the segment portion of the address of the referenced memory location.

See also **OFFSET**

Example

```
.DATA
temp   DW   0
.CODE
mov  ax,SEG temp
mov  ds,ax          ;set up segment register
ASSUME ds:SEG temp  ;tell assembler about it
```

SHL Ideal, MASM

Function Shifts the value of an expression to the left

Syntax `expression SHL count`

Remarks *expression* and *count* must evaluate to constants. **SHL** performs a logical shift to the left of the bits in *expression*. Bits shifted in from the right contain 0, and the bits shifted off the left are lost.

A negative count causes the data to be shifted the opposite way.

See also **SHR**

Example

```
mov al,00000011b SHL 3   ;loads 00011000B
```

SHORT Ideal, MASM

Function Forces an expression to be a short code pointer.

Syntax `SHORT expression`

Remarks *expression* references a location in your current code segment. **SHORT** informs the assembler that *expression* is within –128 to +127 bytes from the current code location, which lets the assembler generate a shorter **JMP** instruction.

You only need to use **SHORT** on forward-referenced **JMP** instructions, since Turbo Assembler automatically generates the short jumps if it already knows how far away *expression* is.

See also **FAR, NEAR**

Example

```
     jmp SHORT done  ;generate small jump instruction
                     ;less than 128 bytes of code here
Done:
```

SHR — Ideal, MASM

Function Shifts the value of an expression to the right

Syntax `expression SHR count`

Remarks *expression* and *count* must evaluate to constants. **SHR** performs a logical shift to the right of the bits in *expression*. Bits shifted in from the left contain 0, and the bits shifted off the right are lost.

A negative count causes the data to be shifted the opposite way.

See also **SHL**

Example

```
mov  al,80h SHR 2   ;loads 20h
```

SIZE — Ideal, MASM

Function Returns size of allocated data item

Syntax `SIZE name`

Remarks *name* is a symbol that refers to a data item allocated with one of the data allocation directives (**DB**, **DD**, and so on). In MASM mode, **SIZE** returns the value of **LENGTH** name multiplied by **TYPE** name. Therefore, it does not take into account multiple data items, nor does it account for nested **DUP** operators.

In Ideal mode, **SIZE** returns the byte count within a **DUP**. To get the byte count of **DUP**, use **LENGTH**.

See also **LENGTH, TYPE**

Example

```
msg     DB   "Hello"
array   DW   10 DUP(4 DUP (1), 0)
numbrs  DD   1,2,3,4
var     DQ   ?
;MASM mode
smsg = SIZE msg         ;1, string has length 1
sarray = SIZE array     ;= 20, 10 DUPS of DW
snumbrs = SIZE numbrs   ;4, length = 1, DD = 4 bytes
svar = SIZE var         ;= 8, 1 element, DQ = 8 bytes
;Ideal mode
smsg = SIZE msg         ;1, string has length 1
sarray = SIZE array     ;= 20, 10 DUPS of DW
snumbrs = SIZE numbrs   ;4, length = 1, DD = 4 bytes
svar = SIZE var         ;=8, 1 element, DQ = 8 bytes
```

SMALL Ideal (386 code generation only), MASM

Function Sets an expression's offset size to 16 bits

Syntax `small expression`

Remarks *expression* is any expression or operand. **SMALL** converts it into a 16-bit offset. You usually use this to remove ambiguity about the size of an operation. For example, if you have enabled the 80386 processor with the **P386** directive,

```
jmp [DWORD PTR ABC]
```

can be interpreted as either a far call with a segment and 16-bit offset or a near call using a 32-bit offset. You can remove the ambiguity by using the **SMALL** directive:

```
jmp small [DWORD PTR ABC]   ;16-bit offset far call
```

In this example, **SMALL** appears outside the brackets, thereby affecting the interpretation of the **DWORD** read from memory. If **SMALL** appears inside the brackets, it determines the size of the address from which to read the operand, not the size of the operand once it is read from memory. For example,

```
CODE  SEGMENT USE32
jmp   small [small DWORD PTR XYZ]
```

means *XYZ* is a 4-byte pointer that's treated as a 16-bit offset and segment, and **JMP** indirect through that address, reading a near **JMP** target address that is also a 16-bit offset.

By combining the **LARGE** and **SMALL** operators, both inside and outside brackets, you can effect any combination of an indirect **JMP** or **CALL** from a 16- or 32-bit segment to a 16- or 32-bit segment. **LARGE** and **SMALL** can also be used with other ambiguous instructions, such as **LIDT** and **LGDT**.

See also **LARGE**

SYMTYPE — Ideal

Function Returns a byte describing a symbol

Syntax `SYMTYPE <expression>`

Remarks **SYMTYPE** functions very similarly to **.TYPE**, with one minor difference: If *expression* contains an undefined symbol, **SYMTYPE** returns an error, unlike **.TYPE**.

See also **.TYPE**

TBYTE — Ideal

Function Forces expression to be 10-byte size

Syntax `TBYTE` *`expression`*

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **TBYTE** size, regardless of the original size of *expression*.

You usually use **TBYTE** to define the size of a forward-referenced expression, or to explicitly state the size of a register indirect expression from which the size cannot be determined.

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **TBYTE** type.

See also **PTR**

Example

```
fld [TBYTE bx]   ;sizeless indirect
fst [TBYTE X]    ;forward reference
X   DT    0
```

THIS — Ideal, MASM

Function Creates an operand whose address is the current segment and location counter

Syntax `THIS` *`type`*

Remarks *type* describes the size of the operand and whether it refers to code or data. It can be one of the following:

- **NEAR, FAR,** or **PROC** (**PROC** is the same as either **NEAR** or **FAR,** depending on the memory set using the **MODEL** directive)
- **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name

You usually use this operator to build **EQU** and = statements.

Example

```
ptr1  EQU THIS WORD   ;same as following statement
ptr2  LABEL WORD
```

.TYPE MASM

Function Returns a byte describing a symbol

Syntax `.TYPE name`

Remarks *name* is a symbol that may or may not be defined in the source file. **.TYPE** returns a byte that describes the symbol with the following fields:

Bit	Description
0	Program relative symbol
1	Data relative symbol
2	Constant
3	Direct addressing mode
4	Is a register
5	Symbol is defined
7	Symbol is external

If bits 2 and 3 are both zero, the expression uses register indirection (like [BX], and so on).

If .TYPE returns zero, the expression contained an undefined symbol.

.TYPE is usually used in macros to determine how to process different kinds of arguments.

See also **SYMTYPE**

Example

```
  IF (.TYPE ABC) AND 3     ;is it segment-relative?
     ASSUME  ds:SEG abc
     mov  ax,SEG abc
     mov  ds,ax
ENDIF
```

TYPE Ideal, MASM

Function Returns a number indicating the size or type of symbol

Syntax `TYPE` *`expression`*

Remarks **TYPE** returns one of the following values, based on the type of expression:

```
BYTE  1
WORD  2
DWORD 4
FWORD 6
PWORD 6
QWORD 8
TBYTE 10
NEAR  0FFFFHh
FAR   0FFFEh
constant 0
structure # of bytes in structure
```

See also **LENGTH, SIZE**

Example

```
bvar    DB  1
darray  DD  10 DUP (1)
X  STRUC
   DW   ?
   DT   ?
X  ENDS
fp EQU THIS FAR
tbvar = TYPE bvar      ;= 1
tdarray = TYPE darray  ;= 4
tx = TYPE x            ;=12
tfp = TYPE fp          ;0FFFEh
```

UNKNOWN Ideal

Function Removes type information from an expression

Syntax `UNKNOWN` *`expression`*

Remarks *expression* is an address. The result is the same expression, but with its type (**BYTE, WORD,** and so on) removed.

Use **UNKNOWN** to force yourself to explicitly mention a size whenever you want to reference a location. This is useful if you want to treat the location as a type of union, allowing the storage of many different data types. Incorrectly then, if you define another name without an explicit size to reference the location, the assembler can't use the original data allocation size.

You can also use an address with **UNKNOWN** size much like you would use register indirect memory-referencing for one operand, and pin down the size of the operation by using a register for the other operand. By defining a name as **UNKNOWN**, you can use it exactly as you would an anonymous register expression such as [BX].

To perform this function in MASM mode, you must use the **PTR** directive preceded by the **BYTE** type.

See also **PTR**

Example

```
   .DATA
workbuf DT  0               ;can hold up to a DT
workptr EQU UNKNOWN WORKBUF ;anonymous pointer
   .CODE
;EXAMPLE 1
   mov  [BYTE PTR WORKPTR],1 ;store a byte
   fstp [QWORD PTR WORKPTR]  ;store a qword
   mov  [WORKPTR],1          ;error--no type
;EXAMPLE 2
   mov al,[WORKPTR]          ;no complaint
   mov ax,[WORKPTR]          ;no complaint either!
```

WIDTH — Ideal, MASM

Function Returns the width in bits of a field in a record

Syntax

```
WIDTH recordfieldname
WIDTH record
```

Remarks *recordfieldname* is the name of any field name in a previously defined record. **WIDTH** returns a value of the number of bits in the record that *recordfieldname* occupies.

record is the name of a previously defined record. **WIDTH** returns a value of the total number of bits for all the fields in the record.

See also **MASK**

Example

```
;Macro determines maximum value for a field
maxval  MACRO FIELDNAME
value=2
   REPT WIDTH FIELDNAME - 1
value = value * 2;
   ENDM
```

```
value = value - 1
   ENDM
```

WORD — Ideal

Function Forces expression to be word size

Syntax `WORD expression`

Remarks *expression* must be an address. The result is an expression that points to the same memory address but always has **WORD** size, regardless of the original size of *expression.*

You usually use **WORD** to define the size of a forward-referenced expression, or to explicitly state the size of a register indirect expression from which the size cannot be determined.

To perform this function in MASM mode, you must use the **PTR** directive preceded with the **WORD** type.

See also **PTR**

Example

```
mov [WORD bx],1   ;word immediate move
mov [WORD X],1    ;forward reference
X   DW   0
```

XOR — Ideal, MASM

Function Bitwise logical exclusive **OR**

Syntax `expression1 XOR expression2`

Remarks **XOR** performs a bit-by-bit logical exclusive **OR** of each bit in *expression1* and *expression2*. The result has a 1 in any bit position that had a 1 in one expression but not in the other, and a 0 in all other bit positions.

See also **AND, NOT, OR**

Example

```
mov  al,11110000b XOR 11000011b    ;AL = 00110011b
```

The special macro operators

You use the special macro operators when calling macros and within macro and repeat-block definitions. You can also use them with the arguments to conditional assembly directives.

Here's a list of the special macro operators:

&	Substitute operator
<>	Literal text string operator
!	Quoted character operator
%	Expression evaluate operator
;;	Suppressed comment

The operators let you modify symbol names and individual characters so that you can either remove special meaning from a character or determine when an argument gets evaluated.

& Ideal, MASM

Function Substitute operator

Syntax `&name`

Remarks *name* is the value of the actual parameter in the macro invocation or repeat block. In many situations, parameter substitution is automatic, and you don't have to use this operator. You must use this operator when you wish substitution to take place inside a quoted character string, or when you want to "paste" together a symbol from one or more parameters and some fixed characters. In this case, the **&** prevents the characters from being interpreted as a single name.

Example

```
MAKEMSG  MACRO StrDef,NUM
MSG & NUM DB   '&StrDef'
ENDM
```

If you call this macro with

```
   MAKEMSG 9,<Enter a value: >
```

it expands to

```
   MSG9  DB 'Enter a value: '
```

<> Ideal, MASM

Function Literal text string operator

Syntax *<text>*

Remarks *text* is treated as a single macro or repeat parameter, even if it contains commas, spaces, or tabs that usually separate each parameter. Use this operator when you want to pass an argument that contains any of these separator characters.

You can also use this operator to force Turbo Assembler to treat a character literally, without giving it any special meaning. For example, it you wanted to pass a semicolon (;) as a parameter to a macro invocation, you would have to enclose it in angle brackets (<;>) to prevent it from being treated as the start of a comment.

Turbo Assembler removes the outside set of angle brackets each time a parameter is passed during the invocation of a macro. To pass a parameter down through several levels of macro expansion, you must supply one set of angle brackets for each level.

Example

```
MANYDB MACRO VALS
IRP    X,<VALS>
   ENDM
   ENDM
```

When calling this macro, you must enclose multiple values in angle brackets so they get treated as the single parameter *VALS*:

```
   MANYDB  <4,6,0,8>
```

The **IRP** repeat directive still has angle brackets around the parameter name because the set of brackets around the parameter are stripped when the macro is called.

! Ideal, MASM

Function Quoted character operator

Syntax !*character*

Remarks The ! operator lets you call macros with arguments that contain special macro operator characters. This is somewhat equivalent to enclosing the character in angle brackets. For example, **!&** is the same as **<&>**.

Example

```
MAKEMSG  MACRO StrDef,NUM
```

```
MSG & NUM DB   '&StrDef'
ENDM

MAKEMSG  <Can't enter !> 99>
```

In this example, the argument would have been prematurely terminated if the ! operator had not been used.

% Ideal, MASM

Function Expression evaluate operator

Syntax *%expression*

Remarks *expression* can be either a numeric expression using any of the operands and operators described in this chapter or it can be a text equate. If it is a numeric expression, the string that is passed as a parameter to the macro invocation is the result of evaluating the expression. If *expression* is a text equate, the string passed is the text of the text equate. The evaluated expression will be represented as a numerical string in the current **RADIX**.

Use this operator when you want to pass the string representing a calculated result, rather than the expression itself, to a macro. Also, a text macro name can be specified after the **%**, causing a full substitution of the text macro body for the macro argument.

Example

```
DEFSYM  MACRO NUM
   ???&NUM:
   ENDM

DEFSYM  %5+4
```

results in the following code label definition:

```
   ????9:
```

;; Ideal, MASM

Function Suppressed comment

Syntax *;;text*

Remarks Turbo Assembler ignores all *text* following the double semicolon (;;). Normal comments are stored as part of the macro definition and appear in the listing any time the macro is expanded. Comments that start with a double semicolon (;;) are not stored as part of the macro definition. This saves memory, particularly if you have a lot of macros that contain a lot of comments.

Example

```
SETBYTES   MACRO VarName,val
   VarName DB 10 DUP (val)   ;;this comment doesn't get saved
   ENDM
```

C H A P T E R

3

Directives

A source statement can either be an *instruction* or a *directive.* An *instruction source line* generates object code for the processor operation specified by the instruction mnemonic and its operands. A *directive source line* tells the assembler to do something unrelated to instruction generation, including defining and allocating data and data structures, defining macros, specifying the format of the listing file, controlling conditional assembly, and selecting the processor type and instruction set.

Some directives define a symbol whose name you supply as part of the source line. These include, for example, **SEGMENT**, **LABEL**, and **GROUP**. Others change the behavior of the assembler but do not result in a symbol being defined, for example, **ORG**, **IF**, **%LIST**.

The directives presented here appear in alphabetical order (excluding punctuation); for example, **.CODE** appears just before **CODESEG**.

The reserved keywords **%TOC** and **%NOTOC** do not perform any operation in the current version of Turbo Assembler. Future versions, however, will use these keywords, so you should avoid using them as symbols in your programs.

The directives fall into three categories:

1. ***The MASM-style directives:*** Turbo Assembler supports all MASM-style directives. When you use Turbo Assembler in Ideal mode, the syntax of some of these directives changes. For

these directives, the description notes the syntax for both modes.

2. ***The new Turbo Assembler directives:*** These directives provide added functionality beyond that provided by MASM.
3. ***Turbo Assembler directives that are synonyms for existing MASM directives:*** These synonyms provide a more organized alternative to some existing MASM directives. For example, rather than **.LIST** and **.XLIST**, you use **%LIST** and **%NOLIST**. As a rule, all paired directives that enable and disable an option have the form *xxxx* and *NOxxxx*. The synonyms also avoid using a period (.) as the first character of the directive name. The MASM directives that start with a period are not available in Turbo Assembler's Ideal mode, so you must use the new synonyms instead.

All Turbo Assembler directives that control the listing file start with the percent (%) character.

In the syntax section of each entry, the following typographical conventions are used:

- Brackets ([ ]) indicate optional arguments (you do not need to type in the brackets).
- Ellipses (...) indicate that the previous item may be repeated as many times as desired.
- Items in *italics* are placeholders that you replace with actual symbols and expressions in your program.

Sample Directive — Mode directive operates in

Function	Brief description of what the directive does.
Syntax	How the directive is used; italicized items are user-defined
Remarks	General information about the directive.
See also	Other related directives.
Example	Sample code using the directive.

.186 MASM

Function Enables assembly of 80186 instructions

Syntax

```
.186
```

Remarks **.186** enables assembly of the additional instructions supported by the 80186 processor. (Same as **P186.**)

See also **.8086, .286, .286C, .286P, .386, .386C, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

Example

```
.186
push 1    ;valid instruction on 186*
```

.286 MASM

Function Enables assembly of non-privileged 80286 instructions

Syntax

```
.286
```

Remarks **.286** enables assembly of the additional instructions supported by the 80286 processor in non-privileged mode. It also enables the 80287 numeric processor instructions exactly as if the **.287** or **P287** directive had been issued. (Same as **P286N** and **.286C.**)

See also **.8086, .186, .286C, .286P, .386, .386C, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

Example

```
.286
fstsw ax    ;only allowed with 80287
```

.286C

Function Enables assembly of non-privileged 80286 instructions

See also **.8086, .186, .286, .286P, .386, .386C, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

.286P MASM

Function Enables assembly of all 80286 instructions

Syntax
```
.286P
```

Remarks **.286P** enables assembly of all the additional instructions supported by the 80286 processor, including the privileged mode instructions. It also enables the 80287 numeric processor instructions exactly as if the **.287** or **P287** directive had been issued. (Same as **P286P.**)

See also **.8086, .186, .286, .286C, .386, .386C, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

.287 MASM

Function Enables assembly of 80287 coprocessor instructions

Syntax
```
.287
```

Remarks **.287** enables assembly of all the 80287 numeric coprocessor instructions. Use this directive if you know you'll never run programs using an 8087 coprocessor. This directive causes floating-point instructions to be optimized in a manner incompatible with the 8087, so *don't* use it if you want your programs to run using an 8087. (Same as **P287.**)

See also **.8087, .387, P8087, PNO87, P287, P387**

Example
```
.287
fsetpm     ;only on 287
```

.386 MASM

Function Enables assembly of non-privileged 80386 instructions

Syntax
```
.386
```

Remarks **.386** enables assembly of the additional instructions supported by the 80386 processor in non-privileged mode. It also enables the 80387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued. (Same as **P386N** and **.386C.**)

See also **.8086, .186, .286C, .286, .286P, .386C, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

Example
```
.386
```

```
stosd    ;only valid as 386 instruction
```

.386C

Function Enables assembly of 80386 instructions

See also **.8086, .186, .286C, .286, .286P, .386, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

.386P MASM

Function Enables assembly of all 80386 instructions

Syntax `.386P`

Remarks **.386P** enables assembly of all the additional instructions supported by the 80386 processor, including the privileged mode instructions. It also enables the 80387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued. (Same as **P386P.**)

See also **.8086, .186, .286C, .286, .286N, .286P, .386, .386C, P8086, P286, P286N, P286P, P386, P386N, P386P**

.387 MASM

Function Enables assembly of 80387 coprocessor instructions

Syntax `.387`

Remarks **.387** enables assembly of all the 80387 numeric coprocessor instructions. Use this directive if you know you'll never run programs using an 8087 coprocessor. This directive causes floating-point instructions to be optimized in a manner incompatible with the 8087, so *don't* use it if you want your programs to run using an 8087. (Same as **P387.**)

See also **.8087, .287, 8087, PNO87, P287, P387**

Example

```
.387
fsin    ;SIN() only available on 387
```

.8086 MASM

Function Enables assembly of 8086 instructions only

Syntax `.8086`

Remarks **.8086** enables assembly of the 8086 instructions and disables all instructions available only on the 80186, 80286, and 80386 processors. It also enables the 8087 coprocessor instructions exactly as if the **.8087** or **8087** had been issued.

This is the default instruction set mode used by Turbo Assembler when it starts assembling a source file. Programs assembled using this mode will run on all members of the 80x86 processor family. If you know that your program will only be run on one of the more advanced processors, you can take advantage of the more sophisticated instructions of that processor by using the directive that enables that processor's instructions. (Same as **P8086.**)

See also **.186, .286C, .286, .286P, .386C, .386, .386P, P8086, P286, P286N, P286P, P386, P386N, P386P**

.8087 MASM

Function Enables assembly of 8087 coprocessor instructions

Syntax `.8087`

Remarks **.8087** enables all the 8087 coprocessor instructions, and disables all those coprocessor instructions available only on the 80287 and 80387.

This is the default coprocessor instructions set used by Turbo Assembler. Programs assembled using this mode will run on all members of the 80x87 coprocessor family. If you know that your program will only be run on one of the more advanced coprocessors, you can take advantage of the more sophisticated instructions of that processor by using the particular directive that enables that processor's instructions. (Same as **P8087.**)

See also **.287, .387, 8087, PNO87, P287, P387**

Example

```
.8087
fstsw  MEMLOC     ;no FSTSW AX on 8087
```

: Ideal, MASM

Function Defines a near code label

Syntax `name:`

Remarks *name* is a symbol that you have not previously defined in the source file. You can place a near code label on a line by itself or at the start of a line before an instruction. You usually use a near code label as the destination of a **JMP** or **CALL** instruction from within the same segment.

The code label will only be accessible from within the current source file unless you use the **PUBLIC** directive to make it accessible from other source files.

This directive is the same as using the **LABEL** directive to define a **NEAR** label; for example *A:* is the same as *A* **LABEL NEAR**.

See also **LABEL**

Example

```
   . . .
   jne A       ;skip following instruction
   inc si
A:             ;JNE goes here
```

= Ideal, MASM

Function Defines a numeric equate

Syntax `name = expression`

Remarks *name* is assigned the result of evaluating *expression,* which must evaluate to either a constant or an address within a segment. *name* can either be a new symbol name, or a symbol that was previously defined using the = directive.

You can redefine a symbol that was defined using the = directive, allowing you to use the symbols as counters. (See the example that follows.)

You can't use = to assign strings or to redefine keywords or instruction mnemonics; you must use **EQU** to do these things.

The = directive has far more predictable behavior than the **EQU** directive in MASM mode, so you should use = instead of **EQU** wherever you can.

See also **EQU**

Example

```
BitMask = 1           ;initialize bit mask
BittBl  LABEL BYTE
  REPT 8
  DB   BitMask
BitMask = BitMask * 2  ;shift the bit to left
  ENDM
```

ALIGN — Ideal, MASM

Function Rounds up the location counter to a power-of-two address

Syntax `ALIGN boundary`

Remarks *boundary* must be a power of 2 (for example, 2, 4, 8, and so on).

If the location counter is not already at an offset that is a multiple of *boundary,* single byte **NOP** instructions are inserted into the segment to bring the location counter up to the desired address. If the location counter is already at a multiple of *boundary,* this directive has no effect.

You can't reliably align to a boundary that is more strict than the segment alignment in which the **ALIGN** directive appears. The segment's alignment is specified when the segment is first started with the **SEGMENT** directive.

For example, if you have defined a segment with

```
CODE SEGMENT PARA PUBLIC
```

you can say **ALIGN 16** (same as **PARA**) but you can't say **ALIGN 32,** since that is more strict than the alignment indicated by the **PARA** keyword in the **SEGMENT** directive. **ALIGN** generates a warning if the segment alignment is not strict enough.

See also **EVEN, EVENDATA**

Example

```
ALIGN  4              ;align to DWORD boundary for 386
BigNum  DD  12345678
```

.ALPHA — MASM

Function Sets alphanumeric segment-ordering

Syntax `.ALPHA`

Remarks You usually use **.ALPHA** to ensure compatibility with very old versions of MASM and the IBM assembler. The default behavior of these old assemblers is to emit segments in alphabetical order, unlike the newer versions. Use this option when you assemble source files written for old assembler versions.

If you don't use this directive, the segments are ordered in the same order that they were encountered in the source file. The **DOSSEG** directive can also affect the ordering of segments.

.ALPHA does the same thing as the **/A** command-line option. If you used the **/S** command-line option to force sequential segment-ordering, **.ALPHA** will override it.

See also **DOSSEG, .SEQ,**

Example

```
     .ALPHA
XYZ  SEGMENT
XYZ  ENDS
ABC  SEGMENT    ;this segment will be first
ABC  ENDS
```

ARG — Ideal, MASM

Function Sets up arguments on the stack for procedures

Syntax `arg` *`argument`* `[,`*`argument`*`] ... [=`*`symbol`*`] [RETURNS` *`argument`* `[,`*`argument`*`]]`

Remarks **ARG** usually appears within a **PROC/ENDP** pair, allowing you to access arguments pushed on the stack by the caller of the procedure. Each *argument* is assigned a positive offset from the BP register, presuming that both the return address of the function call and the caller's BP have been pushed onto the stack already.

argument describes an argument the procedure is called with. The language specified with the **.MODEL** directive determines whether the arguments are in reverse order on the stack. You must always list the arguments in the same order they appear in the high-level language function that calls the procedure. Turbo Assembler reads them in reverse order if necessary. Each *argument* has the following syntax (boldface items are literal):

```
argname[ [ count1 ] ] [:[distance] PTR] type] [:count2]]
```

argname is the name you'll use to refer to this argument throughout the procedure. *distance* is optional and can be either **NEAR** or **FAR** to indicate that the argument is a pointer of the indicated size. *type* is the data type of the argument and can be **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. *count1* and *count2* are the number of elements of the specified type. The total count is calculated as *count1* * *count2*.

If you don't specify *type,* **WORD** is assumed.

If you add **PTR** to indicate that the argument is in fact a pointer to a data item, Turbo Assembler emits this debug information for Turbo Debugger. Using **PTR** only affects the generation of additional debug information, not the code Turbo Assembler generates. You must still write the code to access the actual data using the pointer argument.

If you use **PTR** alone, without specifying **NEAR** or **FAR** before it, Turbo Assembler sets the pointer size based on the current memory model and, for the 386 processor, the current segment address size (16 or 32 bit). The size is set to **WORD** in the tiny, small, and medium memory models and to **DWORD** for all other memory models using 16-bit segments. If you're using the 386 and are in a 32-bit segment, the size is set to **DWORD** for tiny, small, and medium models and to **FWORD** for compact, large, and huge models.

The *argument* name variables remain defined within the procedure as BP-relative memory operands. For example,

```
Func1 PROC NEAR
      ARG A:WORD,B:DWORD:4,C:BYTE = D
```

defines *A* as *[BP+4]*, *B* as *[BP+6]*, *C* as *[BP+14]*, and *D* as 20.

Argument names that begin with the local symbol prefix when local symbols are enabled are limited in scope to the current procedure.

If you end the argument list with an equal sign (=) and a *symbol,* that *symbol* will be equated to the total size of the argument block in bytes. You can then use this value at the end of the procedure as an argument to the **RET** instruction, which effects a stack cleanup of any pushed arguments before returning (this is the Pascal calling convention).

Since it is not possible to push a byte-sized argument on the 8086 processor family, any arguments declared of type **BYTE** are considered to take 2 bytes of stack space. This agrees with the way high-level languages treat character variables passed as parameters. If you really want to specify an argument as a single byte on the stack, you must explicitly supply a count field, as in

```
ARG REALBYTE:BYTE:1
```

If you don't supply a count for **BYTE** arguments, a count of 2 is presumed.

The optional **RETURNS** keyword introduces one or more arguments that won't be popped from the stack when the procedure returns to its caller. Normally, if you specify the language as **PASCAL** or **TPASCAL** when using the **.MODEL** directive, all arguments are popped when the procedure returns. If you place arguments after the **RETURNS** keyword, they will be left on the stack for the caller to make use of, and then pop. In particular, you must define a Pascal string return value by placing it after the **RETURNS** keyword.

See also **LOCAL, PROC, USES**

Example A sample Pascal procedure:

```
fp  PROC FAR
    ARG  SRC:WORD,DEST:WORD = ARGLEN
    push bp
    mov  bp,sp
    mov  di,DEST
    mov  si,SRC
    ;<Procedure body>
    pop  bp
    ret  ARGLEN
fp  ENDP
```

ASSUME — Ideal, MASM

Function Associates segment register with segment or group name

Syntax

```
ASSUME segmentreg:name [,segmentreg:name]...
ASSUME segmentreg:NOTHING
ASSUME NOTHING
```

Remarks *segmentreg* is one of CS, DS, ES, or SS registers and, if you have enabled the 80386 processor with the **P386** or **P386N** directives, the FS and GS registers.

name can be one of the following:

- the name of a group as defined using the **GROUP** directive

- the name of a segment as defined using the **SEGMENT** directive or one of the simplified segmentation directives
- an expression starting with the **SEG** operator
- the keyword **NOTHING**

The **NOTHING** keyword cancels the association between the designated segment register and segment or group name. The **ASSUME NOTHING** statement removes all associations between segment registers and segment or group names.

You can set multiple registers in a single **ASSUME** statement, and you can also place multiple **ASSUME** statements throughout your source file.

See "The ASSUME Directive" in Chapter 9 of the *User's Guide* for a complete discussion of how to use it.

See also **GROUP, SEGMENT**

Example

```
DATA   SEGMENT
mov  ax,DATA
mov  ds,ax
ASSUME ds:DATA
```

%BIN — Ideal, MASM

Function Sets the width of the object code field in the listing file

Syntax `%BIN` *`size`*

Remarks *size* is a constant. If you don't use this directive, the instruction opcode field takes up 20 columns in the listing file.

Example

```
%BIN 12    ;set listing width to 12 columns
```

CATSTR — Ideal, MASM51

Function Concatenates several strings to form a single string

Syntax *`name`* `CATSTR` *`string`*`[,`*`string`*`]...`

Remarks *name* is given a value consisting of all the characters from each string combined into a single string.

Each string can be one of the following:

- a string argument enclosed in angle brackets, like <*abc*>

- a previously defined text macro
- a numeric string substitution starting with percent (%)

See also **INSTR, SIZESTR, SUBSTR**

Example
```
LETTERS  CATSTR  <abc>,<def>   ;LETTERS = "abcdef"
```

.CODE   MASM

Function Defines the start of a code segment

Syntax `.CODE [name]`

Remarks The **.CODE** directive indicates the start of the executable code in your module. You must first have used the **.MODEL** directive to specify a memory model. If you specify the medium, large, or huge memory model, you can follow the **.CODE** directive with an optional *name* that indicates the name of the segment; otherwise *name* is ignored. This way you can put multiple code segments in one file by giving them each a different name.

You can place as many **.CODE** directives as you want in a source file. All the different pieces with the same name will be combined to produce one code segment exactly as if you had entered all the code at once after a single **.CODE** directive.

Using the **.CODE** directive allows the CS register to access the current code segment. This behavior is exactly as if you had put this directive after each **.CODE** directive in your source file:

```
   ASSUME  cs:@code
```

See also **CODESEG, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

Example
```
.CODE            ;here comes the code
mov al,X
.DATA            ;switch to data segment
X DB ?
```

CODESEG — Ideal, MASM

Function Defines the start of the code segment

Remarks **CODESEG** is the same as **.CODE**.

See also **CODE, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

COMM — Ideal, MASM

Function Defines a communal variable

Syntax
```
COMM definition [,definition]...
```

Remarks Each *definition* describes a symbol and has the following format (boldface items are literal):

```
[distance] [language] symbolname[ [ count1 ] ]:type [:count2]
```

distance is optional and can be either **NEAR** or **FAR**. It specifies whether the communal variable is part of the near data space (**DGROUP**) or whether it occupies its own far segment. If you do not specify a *distance*, it will default to the size of the default data memory model. If you are not using the simplified segmentation directives (**.MODEL**, and so on), the default size is **NEAR**. With the tiny, small, and medium models, the default size is also **NEAR**; all other models are **FAR**.

language is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or **NOLANGUAGE** and defines any language-specific conventions to be applied to the symbol name. Using a language in the **COMM** directive temporarily overrides the current language setting (default or one established with the **.MODEL** directive). Note that you don't need to have a **.MODEL** directive in effect to use this feature.

symbolname is the symbol that is to be communal and have storage allocated at link time. If *distance* is FAR, then *symbolname* can also specify an array element size multiplier to be included in the total space computation:

```
name[multiplier]
```

type can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name.

The optional *count* specifies how many items this communal symbol defines. If you do not specify a *count*, one is assumed. The total space allocated for the communal variable is the count times the length specified by the *type* field, times the array element size multiplier (if it is present).

You can define more than one communal symbol by separating each definition with a comma (,).

Communal variables are allocated by the linker.

In MASM mode, communal symbols declared outside of any segment are presumed to be reachable via the DS register, which may not always be a valid assumption. Make sure that you either place the correct segment value in DS or use an explicit segment override when referring to these variables. In Ideal mode, Turbo Assembler correctly checks for whether the communal variable is addressable, using any of the current segment registers as described with the **ASSUME** directive.

Communal variables can't be initialized. Use the **GLOBAL** directive if you wish to initialize data items that are to be shared between modules. The linker also doesn't guarantee the allocation of communal variables in any particular order, so you can't make assumptions about data items allocated by **COMM** directives on sequential source lines.

See also **EXTRN, GLOBAL, PUBLIC, PUBLICDLL**

Example

```
COMM buffer:BYTE:512        ; 512 bytes allocated at link time

COMM FAR abc[41]:BYTE:10    ; 410 bytes (10 elements of 41
                            ; bytes) allocated at link time
```

COMMENT — MASM

Function Starts a comment block

Syntax

```
COMMENT delimiter [text]
   [text]
delimiter
```

Remarks **COMMENT** ignores all text between the first delimiter character and the line containing the next occurrence of the delimiter. *delimiter* is the first nonblank character after the **COMMENT** directive.

Example

```
COMMENT *
   Any old stuff
*
```

%CONDS — Ideal, MASM

Function Shows all statements in conditional blocks in the listing

Syntax
```
%CONDS
```

Remarks **%CONDS** is the default conditional listing mode if you don't use any listing control directives. (Same as **.LFCOND.**)

When **%CONDS** is in effect, the listing will show all statements within conditional blocks, even those blocks that evaluate as false and don't result in the evaluation of enclosed statements.

See also **.LFCOND, %NOCONDS, .SFCOND, .TFCOND**

Example
```
%CONDS
IF 0
  mov ax,1    ;in listing, despite "IF 0" above
ENDIF
```

.CONST — MASM

Function Defines constant data segment

Syntax
```
.CONST
```

Remarks The **.CONST** directive indicates the start of the segment in your program containing constant data. This is data your program requires, but it will not be changed when the program executes. You can put things such as prompt and message strings in this segment.

You don't have to use this directive when writing an assembler-only program. It exists so that you can write routines that interface to high-level languages and then use this for initializing constant data.

See also **.CODE, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL**

Example
```
.CONST
MSG DB  "Execution terminated"
```

CONST — Ideal, MASM

Function Defines constant data segment

See also **.CODE, .CONST, .DATA, DATA?, .FARDATA, .FARDATA?**

.CREF — MASM

Function Enables cross-reference listing (CREF)

Syntax `.CREF`

Remarks **.CREF** allows cross-reference information to be accumulated for all symbols encountered from this point forward in the source file. This directive reverses the effect of any **%XCREF** or **.XCREF** directives that inhibited the collection of cross-reference information.

Turbo Assembler includes cross-reference information in the listing file, as well as in a separate .XRF file.

See also **%CREF**

%CREF — Ideal, MASM

Function Enables cross-reference listing (CREF)

Syntax `%CREF`

See also **.CREF, %CREFALL, %CREFREF, %CREFUREF, %NOCREF, .XCREF**

Example

```
        %CREF
WVAL DW 0       ;CREF shows WVAL defined here
```

%CREFALL — Ideal, MASM

Function Lists all symbols in cross-reference

Syntax `%CREFALL`

Remarks **%CREFALL** reverses the effect of any previous **%CREFREF** or **%CREFUREF** directives that disabled the listing of unreferenced or referenced symbols. After issuing **%CREFALL**, all subsequent symbols in the source file will appear in the cross-reference listing.

By default, Turbo Assembler uses this mode when assembling your source file.

See also **%CREFREF, %CREFUREF**

Example

```
%CREFREF
ARG1 EQU [bp+4]    ;not referenced, won't be in listing
%CREFALL
ARG2 EQU [bp+6]    ;not referenced, appears anyway
ARG3 EQU [bp+8]    ;referenced, appears in listing
mov  ax,ARG3
END
```

%CREFREF — Ideal, MASM

Function Disables listing of unreferenced symbols in cross-reference

Syntax `%CREFREF`

Remarks **%CREFREF** causes symbols that are defined but never referenced to be omitted from the cross-reference listing. Normally when you request a cross-reference, these symbols appear in the symbol table.

See also **%CREF, %CREFALL, %CREFUREF**

Example

```
%CREF
abc EQU 4       ;will not appear in CREF listing
xyz EQU 1       ;will appear in listing
mov ax,xyz      ;makes XYZ appear in listing
END
```

%CREFUREF — Ideal, MASM

Function Lists only the unreferenced symbols in cross-reference

Syntax `%CREFUREF`

Remarks **%CREFUREF** enables the listing of unreferenced symbols in the symbol table cross-reference. When you use this directive, only unreferenced symbols appear in the symbol table. To see both referenced and unreferenced symbols, use the **%CREFALL** directive.

See also **%CREFALL, %CREFREF**

Example

```
%CREF
abc  EQU 2   ;doesn't appear in listing
%CREFUREF
```

```
def  EQU 1   ;appears in listing
END
```

%CTLS — Ideal, MASM

Function Prints listing controls

Syntax
```
%CTLS
```

Remarks **%CTLS** causes listing control directives (such as **%LIST**, **%INCL**, and so on) to be placed in the listing file; normally, they are not listed. It takes effect on all subsequent lines, so the **%CTLS** directive itself will not appear in the listing file.

See also **%NOCTLS**

Example
```
%CTLS
%NOLIST     ;this will be in listing file
```

.DATA — MASM

Function Defines the start of a data segment

Syntax
```
.DATA
```

Remarks The **.DATA** directive indicates the start of the initialized data in your module. You must first have used the **.MODEL** directive to specify a memory model.

You can place as many **.DATA** directives as you want in a source file. All the different pieces will be combined to produce one data segment, exactly as if you had entered all the data at once after a single **.DATA** directive.

The data segment is put in a group called **DGROUP**, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA?** directives. You can access data in any of these segments by making sure that one of the segment registers is pointing to **DGROUP**.

See the **.MODEL** directive for complete information on the segment attributes for the data segment.

See also **.CODE , .CONST, .DATA?, DATASEG, .FARDATA, .FARDATA?, .MODEL, .STACK**

Example
```
.DATA
ARRAY1 DB 100 DUP (0)     ;NEAR initialized data
```

.DATA? — MASM

Function Defines the start of an uninitialized data segment

Syntax `.DATA?`

Remarks The **.DATA?** directive indicates the start of the uninitialized data in your module. You must first have used the **.MODEL** directive to specify a memory model.

You create uninitialized data using the **DUP** operator with the **?** symbol. For example,

```
DB 6 DUP (?)
```

You do not have to use this directive when writing an assembler-only program. It exists so that you can write routines that interface to high-level languages and then use this directive for uninitialized data.

You can place as many **.DATA?** directives as you want in a source file. All the pieces will be combined to produce one data segment, exactly as if you had entered all the data at once after a single **.DATA?** directive.

The uninitialized data segment is put in a group called **DGROUP**, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA** directives.

See **.MODEL** for complete information on the segment attributes for the uninitialized data segment.

See also **.CODE, .CONST, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

Example

```
.DATA?
TEMP DD 4 DUP (?)     ;uninitialized data
```

DATASEG — Ideal, MASM

Function Defines the start of a data segment

Syntax `DATASEG`

Remarks **DATASEG** is the same as **.DATA**. It must be used in Ideal mode only.

See also **.CODE , .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

DB — Ideal, MASM

Function Allocates byte-size storage

Syntax `[name] DB expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

Each *expression* allocates one or more bytes and can be one of the following:

- A constant expression that has a value between –128 and 255.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- A character string of one or more characters.
- A repeated expression using the **DUP** operator.

See also **DD, DF, DP, DQ, DT, DW**

Example

```
fibs DB 1,1,2,3,5,8,13
BUF  DB 80 DUP (?)
MSG  DB "Enter value: "
```

DD — Ideal, MASM

Function Allocates doubleword-sized storage

Syntax `[name] DD [type PTR] expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

type followed by **PTR** is optional. It adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. It has no effect on the data generated by Turbo Assembler. *type* can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. For example,

```
person  STRUC
name  DB  32 DUP(?)
age   DW  ?
person  ENDS
```

```
PPTR  DD person PTR 0   ;PPTR is a far pointer
                        ; to the structure
```

Each *expression* allocates one or more doublewords (4 bytes) and may be one of the following:

- A constant expression that has a value between –2,147,483,648 and 4,294,967,295.
- A short (32-bit) floating-point number.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- An address expression, specifying a far address in a 16-bit segment (segment:offset) or a near address in a 32-bit segment (32-bit offset only).
- A repeated expression using the **DUP** operator.

See also **DB, DF, DP, DQ, DT, DW**

Example

```
Data32 SEGMENT USE32
Xarray DB  0,1,2,3
Data32 ENDS
Data   SEGMENT
Consts DD  3.141, 2.718        ;floating-point constants
DblPtr DD  Consts              ;16-bit far pointer
NrPtr  DD  Xarray              ;32-bit near pointer
BigInt DD  12345678            ;large integer
Darray DD  4 DUP (1)           ;4 integers
```

%DEPTH — Ideal, MASM

Function Sets size of depth field in listing file

Syntax `%DEPTH` *`width`*

Remarks *width* specifies how many columns to reserve for the nesting depth field in the listing file. The depth field indicates the nesting level for INCLUDE files and macro expansions. If you specify a width of 0, this field does not appear in the listing file. Usually, you won't need to specify a width of more than 2, since that would display a depth of up to 99 without truncation.

The default width for this field is 1 column.

Example

```
%DEPTH 2     ;show nesting levels > 9
```

DF — Ideal, MASM

Function Defines far 48-bit pointer (6 byte) data

Syntax `[name] DF [type PTR] expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

type followed by **PTR** is optional. It adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. It has no effect on the data generated by Turbo Assembler. *type* can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. For example,

```
person  STRUC
name  DB  32 dup(?)
age   DW  ?
person  ENDS
DATA    SEGMENT USE32
PPTR  DF person PTR 0  ;PPTR is a 32-bit far pointer
                       ; to the structure
```

Each *expression* allocates one or more 48-bit far pointers (6 bytes) and may be one of the following:

- A constant expression that has a value between –140,737,488,355,328 and 281,474,976,710,655.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- An address expression, specifying a far address in a 48-bit segment (segment:48-bit offset).
- A repeated expression using the **DUP** operator.

This directive is normally used only with the 30386 processor.

See also **DB, DD, DP, DQ, DT, DW**

Example

```
      .386
DATA  SEGMENT USE32
MSG   DB      "All done"
FmPtr DF      MSG          ;FAR pointer to MSG
DATA  ENDS
```

DISPLAY — Ideal, MASM

Function Outputs a quoted string to the screen

Syntax `DISPLAY "text"`

Remarks *text* is any message you want to display; you must surround it with quotes (""). The message is written to the standard output device, which is usually the screen. If you wish, you can use the DOS redirection facility to send screen output to a file.

Among other things, you can use this directive to inform yourself of the generation of sections of conditional assembly.

See also **%OUT**

Example

```
DISPLAY "Assembling EGA interface routines"
```

DOSSEG — Ideal, MASM

Function Enables DOS segment-ordering at link time

Syntax `DOSSEG`

Remarks You usually use **DOSSEG** in conjunction with the **.MODEL** directive, which sets up the simplified segmentation directives. **DOSSEG** tells the linker to order the segments in your program the same way high-level languages order their segments.

You should only use this directive when you are writing stand-alone assembler programs, and then you only need to use the **DOSSEG** directive once in the main module that specifies the starting address of your program.

Segments appear in the following order in the executable program:

1. All segments that have the class name 'CODE'.
2. All segments that do not have the class name 'CODE' and are not in the group named **DGROUP**.
3. All segments in **DGROUP** in the following order:
   a. All segments that have the class name 'BEGDATA'.
   b. All segments that do not have the class name 'BEGDATA', 'BSS', or 'STACK'.
   c. All segments with a class name of 'BSS'.

d. All segments with a class name of 'STACK'.

See also **.MODEL**

Example

```
DOSSEG
.MODEL medium
```

DP Ideal, MASM

Function Defines a far 48-bit pointer (6 byte) data area

See also **DB, DD, DF, DQ, DT, DW**

DQ Ideal, MASM

Function Defines a quadword (8 byte) data area

Syntax `[name] DQ expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

Each *expression* allocates one or more quadwords (8 bytes) and can be one of the following:

- A constant expression that has a value between -2^{63} and 2^{64}-1.
- A long (64-bit) floating-point number.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- A repeated expression using the **DUP** operator.

See also **DB, DD, DF, DP, DT, DW**

Example

```
HugInt DQ 314159265358979323
BigFlt DQ 1.2345678987654321
Qarray DQ 10 DUP (?)
```

DT — Ideal, MASM

Function Defines a 10-byte data area

Syntax `[name] DT expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

Each *expression* allocates one or more 10-byte values and may be one of the following:

- A constant expression that has a value between -2^{79} and 2^{80}-1.
- A packed decimal constant expression that has a value between 0 and 99,999,999,999,999,999,999.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- A 10-byte temporary real formatted floating-point number.
- A repeated expression using the **DUP** operator.

See also **DB, DD, DF, DP, DQ, DW**

Example

```
PakNum  DT 123456         ;beware--packed decimal
TempVal DT 0.0000000001   ;high precision FP
```

DW — Ideal, MASM

Function Defines a word-size (2 byte) data area

Syntax `[name] DW [type PTR] expression [,expression]...`

Remarks *name* is the symbol you'll subsequently use to refer to the data. If you don't supply a name, the data will be allocated, but you won't be able to refer to it using a symbolic name.

type followed by **PTR** is optional. It adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. It has no effect on the data generated by Turbo Assembler. *type* can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. For example,

```
Narray  DW  100  DUP (?)
NPTR    DW  WORD PTR narray   ;NPTR is a near pointer
                              ; to a word
```

Each *expression* allocates one or more words (2 bytes) and may be one of the following:

- A constant expression that has a value between –32,767 and 65,535.
- The question mark (?) indeterminate initialization symbol; this allocates storage without giving it a specific value.
- An address expression, specifying a near address in a 16-bit segment (offset only).
- A repeated expression using the **DUP** operator.

See also **DB, DD, DF, DP, DQ, DT**

Example

```
int   DW 12345        ;16-bit integer
Wbuf  DW 6 DUP (?)    ;6 word buffer
Wptr  DW Wbuf         ;offset--only pointer to WBUF
```

ELSE — Ideal, MASM

Function Starts alternative conditional assembly block

Syntax

```
IF condition
statements1
[ELSE
statements2]
ENDIF
```

Remarks The statements introduced by **ELSE** are assembled if the condition associated with the **IF** statement evaluates to false. This means that either *statements1* will be assembled or *statements2* will be assembled, but not both.

The **ELSE** directive always pairs with the nearest preceding **IF** directive that's not already paired with an **ELSE** directive.

See also **ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IF LargeModel EQ 1
  les di,ADDR
ELSE
  lea di,ADDR
ENDIF
```

ELSEIF — Ideal, MASM

Function Starts nested conditional assembly block if an expression is True

Syntax `ELSEIF` *`expression`*

Remarks *expression* must evaluate to a constant and cannot contain any forward-referenced symbol names. If *expression* evaluates to a nonzero value, the statements within the conditional block are assembled, as long as the conditional directive (**IF**, and so on) preceding the **ELSEIF** evaluated to False.

You may have any number of **ELSEIF** directives in a conditional block. As soon as an **ELSEIF** is encountered that has a true expression, that block of code is assembled, and all other parts of the conditional block defined by **ELSEIF** or **ELSE** are skipped. You can also mix the various **ELSE*xx*** directives in the same conditional block.

See also **ELSEIF1, ELSEIF2, ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFDIFI, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB, ELSEIFNDEF**

Example

```
IF ARGSIZE EQ 1
  mov  al,argname
ELSEIF ARGSIZE EQ 2
  mov  ax,argname
ELSE
  %OUT BAD ARGSIZE
ENDIF
```

EMUL — Ideal, MASM

Function Generates emulated coprocessor instructions

Syntax `EMUL`

Remarks Turbo Assembler normally generates real floating-point instructions to be executed by an 80x87 coprocessor. Use **EMUL** if your program has installed a software floating-point emulation package, and you wish to generate instructions that will use it. **EMUL** has the same effect as specifying the **/e** command-line option.

You can combine **EMUL** with the **NOEMUL** directive when you wish to generate real floating-point instructions in one portion of a file and emulated instructions in another portion.

See also **NOEMUL**

Example

```
Finit          ;real 8087 coprocessor instruction
EMUL
Fsave BUF      ;emulated instruction
```

END — Ideal, MASM

Function Marks the end of a source file

Syntax `END [`*`startaddress`*`]`

Remarks *startaddress* is an optional symbol or expression that specifies the address in your program where you want execution to begin. If your program is linked from multiple source files, only one file may specify a *startaddress*. *startaddress* may be an address within the module; it can also be an external symbol defined in another module, declared with the **EXTRN** directive.

Turbo Assembler ignores any text after the **END** directive in the source file.

Example

```
.MODEL small
.CODE
START:
;Body of program goes here
END START                     ;program entry point is "START"
THIS LINE IS IGNORED
SO IS THIS ONE
```

ENDIF — Ideal, MASM

Function Marks the end of a conditional assembly block

Syntax

```
IF condition
statements
ENDIF
```

Remarks All conditional assembly blocks started with one of the **IF***xxxx* directives must end with an **ENDIF** directive. You can nest **IF** blocks up to 255 levels deep.

See also **ELSE, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IF DebugMode      ;assemble following if debug mode not 0
  mov  ax,0
  call DebugDump
ENDIF
```

ENDM — Ideal, MASM

Function Indicates the end of a repeat block or a macro

Syntax
```
ENDM
```

Remarks The **ENDM** directive identifies the end of the macro definition or a repeat block.

See also **IRP, IRPC, MACRO, REPT**

Example
```
IRP  reg,<ax,bx,cx,dx>
push reg
ENDM
```

ENDP — Ideal, MASM

Function Indicates the end of a procedure

Syntax Ideal mode:
```
ENDP [procname]
```

MASM mode:
```
[procname] ENDP
```

Remarks If you supply the optional *procname,* it must match the procedure name specified with the **PROC** directive that started the procedure definition.

Notice that in Ideal mode, the optional *procname* comes after the **ENDP**.

ENDP does not generate a **RET** instruction to return to the procedure's caller; you must explicitly code this.

See also **ARG, LOCAL, PROC**

Example
```
LoadIt PROC
  ;Body of procedure
  ret
LoadIt ENDP
```

ENDS — Ideal, MASM

Function Marks end of current segment structure or union

Syntax Ideal mode:

```
ENDS [segmentname]
ENDS [strucname]
```

MASM mode:

```
[segmentname] ENDS
[strucname] ENDS
```

Remarks **ENDS** marks the end of either a segment, structure, or union. If you supply the optional *segmentname*, it must match the segment name specified with the matching **SEGMENT** directive. Likewise, the optional *strucname* must match the structure name specified with the matching **STRUC** or **UNION** directive.

Notice that in Ideal mode, the optional name comes after the **ENDS.**

See also **SEGMENT, STRUC, UNION**

Example

```
DATA SEGMENT            ;start of data segment
Barray   DB 10 DUP (0)
DATA ENDS               ;end of data segment,
                        ; optional "data" included
STAT STRUC
Mode     DW ?
FuncPtr  DD ?
ENDS                    ;end of structure definition
```

EQU — Ideal, MASM

Function Defines a string, alias, or numeric equate

Syntax `name EQU expression`

Remarks *name* is assigned the result of evaluating *expression. name* must be a new symbol name that has not previously been defined in a different manner. In MASM mode, you can only redefine a symbol that was defined using the **EQU** directive if it was first defined as a string equate.

In MASM mode, **EQU** can result in one of three kinds of equates being generated: *Alias, Expression,* or *String.*

Alias: Redefines keywords or instruction mnemonics, and also allows you to assign alternative names to other symbols you have defined. *Alias* **EQU**s can be redefined.

Expression: Evaluates to a constant or address, much like when using the = directive.

String: *expression* is stored as a text string to be substituted later when *name* appears in expressions. When *expression* cannot be evaluated as an alias, constant, or address, it becomes a string expression. *String* **EQU**s can be redefined.

See also =

Example

```
BlkSize EQU 512
BufBlks EQU 4
BufSize EQU BlkSize*BufBlks
BufLen  EQU BufSize                  ;alias for BUFSIZE
DoneMsg EQU <'Returning to DOS'>
```

.ERR — MASM

Function Forces an error message

Syntax `.ERR`

Remarks **.ERR** causes an error message to occur at the line it is encountered on in the source file.

You usually use this directive inside a conditional assembly block that tests whether some assemble-time condition has been satisfied.

See also **.ERR1, .ERR2, .ERRB, .ERRDEF, .ERRDIF, .ERRDIFI, .ERRE, .ERRIDN, .ERRIDNI, .ERRNB, .ERRNDEF, .ERRNZ**

Example

```
IF $ GT 400h
  .ERR                  ;segment too big
  %OUT Segment too big
ENDIF
```

ERR — Ideal, MASM

Function Forces an error message

Syntax `ERR`

Remarks Same as **.ERR**.

See also **.ERR1, .ERR2, .ERRB, .ERRDEF, .ERRDIF, .ERRDIFI, .ERRE, .ERRIDN, .ERRIDNI, .ERRNB, .ERRNDEF, .ERRNZ**

.ERR1 — MASM

Function Forces an error message on pass 1

Syntax `.ERR1`

Remarks Normally, Turbo Assembler acts as a single-pass assembler, and the fact that **.ERR1** forces a message on pass 1 means that the error message will always be generated, since there is always at least one pass. The message appears on the screen and in the listing file, if one is specified.

If you use Turbo Assembler's multi-pass capability (invoked with the **/m** command-line switch), **.ERR1** forces the error message on the first assembly pass.

See also **.ERR2**

Example
```
.ERR1   ;this will appear during pass 1
```

.ERR2 — MASM

Function Forces an error message on pass 2

Syntax `.ERR2`

Remarks Normally, Turbo Assembler acts as a single-pass assembler, and the fact that **.ERR2** forces a message on pass 2 means that no error message is actually generated. A pass-dependent warning message is generated, however, so that you are alerted to a potentially confusing report.

If you are use Turbo Assembler's multi-pass capability (invoked with the **/m** command-line switch), **.ERR2** forces the error message on the second assembly pass. If you have selected a single pass by using **/m1**, a pass-dependent warning message is forced.

See also **.ERR1**

Example

```
.ERR2  ;this appears in the listing file or on
       ;the second assembly pass, depending on the
       ;number of assembly passes performed
```

.ERRB — MASM

Function Forces an error if argument is blank

Syntax `.ERRB <`*argument*`>`

Remarks You always use this *argument* inside a macro. It tests whether the macro was called with a real argument to replace the specified dummy *argument*. If the *argument* is blank (empty), an error message occurs on the source line where the macro was invoked.

You must always surround the argument to be tested with angle brackets (< >).

See also **.ERRNB**

Example

```
DOUBLE MACRO ARG1
  .ERRB <ARG1>        ;require an argument
  shl ARG1,1          ;double the argument's value
ENDM
```

.ERRDEF — MASM

Function Forces an error if a symbol is defined

Syntax `.ERRDEF` *symbol*

Remarks **.ERRDEF** causes an error message to be generated at the current source line number if *symbol* has already been defined in your source file.

See also **.ERRNDEF**

Example

```
SetMode MACRO ModeVal
   .ERRDEF _MODE        ;error if already defined
_MODE  EQU ModeVal
   ENDM
```

.ERRDIF — MASM

Function Forces an error if arguments are different

Syntax `.ERRDIF <argument1>,<argument2>`

Remarks You always use **.ERRDIF** inside a macro. It tests whether its two arguments are identical character strings. If the two strings are not identical, an error message occurs on the source line where the macro was invoked. The two strings are compared on a character-by-character basis; case is significant. If you want case to be ignored, use the **.ERRDIFI** directive.

You must always surround each argument in angle brackets (< >); separate arguments with a comma.

See also **.ERRDIFI, .ERRIDN, .ERRIDNI**

Example

```
SegLoad   MACRO reg,val
  .ERRDIF <reg>,<es>        ;only permit ES register
  mov ax,val
  mov reg,ax
  ENDM
```

.ERRDIFI — MASM

Function Forces an error if arguments are different, ignoring case

Syntax `.ERRDIFI <argument1>,<argument2>`

Remarks You always use **.ERRDIFI** inside a macro. It tests whether its two arguments are identical character strings. If the two strings are not identical, an error message occurs on the source line where the macro was invoked. The two strings are compared on a character-by-character basis; case is insignificant. If you want case to be significant, use the **.ERRDIF** directive.

You must always surround each argument in angle brackets (< >); separate arguments with a comma.

See also **.ERRDIF, .ERRIDN, .ERRIDNI**

Example

```
SegLoad MACRO reg,val
  .ERRDIF <reg>,<es>        ;only permit ES register
  mov ax,val                ;works no matter how reg typed
  mov reg,ax
  ENDM
```

.ERRE MASM

Function Forces an error if expression is false (0)

Syntax `.ERRE expression`

Remarks *expression* must evaluate to a constant and cannot contain any forward-referenced symbol names. If the expression evaluates to 0, an error message occurs at the current source line.

See also **.ERRNZ**

Example

```
PtrLoad MACRO PTR,val
  .ERRE val          ;error if attempt 0 load to pointer
  mov si,val
  ENDM
```

.ERRIDN MASM

Function Forces an error if arguments are identical

Syntax `.ERRIDN <argument1>,<argument2>`

Remarks You always use **.ERRIDN** inside a macro. It tests whether its two arguments are identical character strings. If the two strings are identical, an error message occurs on the source line where the macro was invoked. The two strings are compared on a character-by-character basis; case is significant. If you want case to be ignored, use the **.ERRIDNI** directive.

You must always surround each argument in angle brackets (< >); separate arguments with a comma.

See also **.ERRDIF, .ERRDIFI, .ERRIDNI**

Example

```
PushSeg MACRO reg,val
  .ERRIDN <reg>,<cs>   ;CS load is illegal
  push reg
  mov  reg,val
  ENDM
```

.ERRIDNI — MASM

Function Forces an error if arguments are identical, ignoring case

Syntax `.ERRIDNI <argument1>,<argument2>`

Remarks You always use **.ERRIDNI** inside a macro. It tests whether its two arguments are identical character strings. If the two strings are identical, an error message occurs on the source line where the macro was invoked. The two strings are compared on a character-by-character basis; case is insignificant. If you want case to be significant, use the **.ERRIDN** directive.

You must always surround each argument in angle brackets (< >); separate arguments with a comma.

See also **.ERRDIF, .ERRDIFI, .ERRIDN**

Example

```
PushSeg MACRO reg,val
  .ERRIDNI <reg>,<cs>       ;CS load is illegal
  push reg                  ;takes CS or cs
  mov  reg,val
  ENDM
```

ERRIF — Ideal, MASM

Function Forces an error if expression is true (nonzero)

See also **.ERRE, .ERRNZ**

ERRIF1 — Ideal, MASM

Function Forces an error message on pass 1

See also **.ERR1**

ERRIF2 — Ideal, MASM

Function Forces an error message on pass 2

See also **.ERR2**

ERRIFB Ideal, MASM

Function	Forces an error if argument is blank
See also	**.ERRB**

ERRIFDEF Ideal, MASM

Function	Forces an error if a symbol is defined
See also	**.ERRDEF**

ERRIFDIF Ideal, MASM

Function	Forces an error if arguments are different
See also	**.ERRDIF**

ERRIFDIFI Ideal, MASM

Function	Forces an error if arguments are different, ignoring case
See also	**.ERRDIFI**

ERRIFE Ideal, MASM

Function	Forces an error if expression is false (0)
See also	**.ERRE**

ERRIFIDN Ideal, MASM

Function	Forces an error if arguments are identical
See also	**.ERRIDN**

ERRIFIDNI — Ideal, MASM

Function Forces an error if arguments are identical, ignoring case

See also **.ERRIDNI**

ERRIFNB — Ideal, MASM

Function Forces an error if argument is not blank

See also **.ERRNB**

ERRIFNDEF — Ideal, MASM

Function Forces an error if symbol is not defined

See also **.ERRNDEF**

.ERRNB — MASM

Function Forces an error if argument is not blank

Syntax `.ERRNB <`*`argument`*`>`

Remarks You always use **.ERRNB** inside a macro. It tests whether the macro was called with a real argument to replace the specified dummy *argument*. If the *argument* is not blank, an error message occurs on the source line where the macro was invoked.

You must always surround the argument to be tested with angle brackets (< >).

See also **.ERRB**

Example

```
DoIt MACRO a,b
  .ERRNB <B>          ;only need one argument
  ;
  ENDM
```

.ERRNDEF MASM

Function Forces an error if symbol is not defined

Syntax `.ERRNDEF` *symbol*

Remarks **.ERRNDEF** causes an error message to be generated at the current source line number if *symbol* has not yet been defined in your source file. The error occurs even if the symbol is defined later in the file (forward-referenced).

See also **.ERRDEF**

Example

```
  .ERRNDEF BufSize     ;no buffer size set
BUF DB BufSize
```

.ERRNZ MASM

Function Forces an error if expression is true (nonzero)

Syntax `.ERRNZ` *expression*

Remarks *expression* must evaluate to a constant and may not contain any forward-referenced symbol names. If the expression evaluates to a nonzero value, an error message occurs at the current source line.

See also **.ERRE**

Example

```
.ERRNZ $ GT 1000h   ;segment too big
```

EVEN Ideal, MASM

Function Rounds up the location counter to the next even address

Syntax `EVEN`

Remarks **EVEN** allows you to align code for efficient access by processors that use a 16-bit data bus (8086, 80186, 80286). It does not improve performance for those processors with an 8-bit data bus (8088, 80188).

You can't use this directive in a segment that has **BYTE**-alignment, as specified in the **SEGMENT** directive that opened the segment.

If the location counter is odd when an **EVEN** directive appears, a single byte of a **NOP** instruction is inserted in the segment to make the location counter even. By padding with a **NOP, EVEN** can be used in code segments without causing erroneous instructions to be executed at run time. If the location is already even, this directive has no effect. A warning is generated for the **EVEN** directive if alignment is not strict enough.

See also **ALIGN, EVENDATA**

Example

```
EVEN
@@A:  lodsb
      xor    bl,al    ;align for efficient access
      loop   @@A
```

EVENDATA — Ideal, MASM

Function Rounds up the location counter to the next even address in a data segment

Syntax `EVENDATA`

Remarks **EVENDATA** allows you to align data for efficient access by processors that use a 16-bit data bus (8086, 80186, 80286). It does not improve performance for those processors with an 8-bit data bus (8088, 80188). **EVENDATA** even-aligns by advancing the location counter without emitting data, which is useful for uninitialized segments. A warning is generated if the alignment isn't strict enough.

You can't use this directive in a segment that has **BYTE**-alignment, as specified in the **SEGMENT** directive that opened the segment.

If the location counter is odd when an **EVENDATA** directive appears, a single byte of 0 is inserted in the segment to make the location counter even. If the location is already even, this directive has no effect.

See also **ALIGN, EVEN**

Example

```
EVENDATA
VAR1   DW   0    ;align for efficient 8086 access
```

EXITM — Ideal, MASM

Function Terminates macro- or block-repeat expansion

Syntax `EXITM`

Remarks **EXITM** stops any macro expansion or repeat block expansion that's in progress. All remaining statements after the **EXITM** are ignored.

This is convenient for exiting from multiple levels of conditional assembly.

See also **ENDM, IRP, IRPC, MACRO, REPT**

Example

```
Shiftn MACRO OP,N
Count = 0
  REPT N
  shl OP,N
Count = Count + 1
  IF Count GE 8         ;no more than 8 allowed
    EXITM
  ENDIF
  ENDM
```

EXTRN — Ideal, MASM

Function Indicates a symbol is defined in another module

Syntax `EXTRN definition [,definition]...`

Remarks Each definition describes a symbol and has the following format:

```
[language] name:type [:count]
```

language is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or NOLANGUAGE and defines any language-specific conventions to be applied to the symbol name. *name* is the symbol that's defined in another module. Using a language in the **EXTRN** directive temporarily overrides the current language setting (default or one established with the **.MODEL** directive). Note that you don't need to have a **.MODEL** directive in effect to use this feature.

type must match the type of the symbol where it's defined in another module. It can be one of the following:

- **NEAR, FAR,** or **PROC. PROC** is either **NEAR** or **FAR** (depending on the memory model set using the **MODEL** directive)
- **BYTE, WORD, DWORD, DATAPTR, CODEPTR, FWORD, PWORD, QWORD, TBYTE,** or a structure name
- **ABS**

The optional *count* specifies how many items this external symbol defines. If the symbol's definition in another file uses the **DUP** directive to allocate more than one item, you can place that value in the count field. This lets the **SIZE** and **LENGTH** operators correctly determine the size of the external data item. If you do not specify a *count,* it is assumed to be one.

You can define more than one external symbol by separating each definition with a comma (,). Also, each argument of **EXTRN** accepts the same syntax as an argument of **ARG** or **LOCAL.**

name must be declared as **PUBLIC** or **PUBLICDLL** in another module in order for your program to link correctly.

You can use the **EXTRN** directive either inside or outside a segment declared with the **SEGMENT** directive. If you place **EXTRN** inside a segment, you are informing the assembler that the external variable is in another module but in the same segment. If you place the **EXTRN** directive outside of any segment, you are informing the assembler that you do not know which segment the variable is declared in.

In MASM mode, external symbols declared outside of any segment are presumed to be reachable via the DS register, which may not always be a valid assumption. Make sure that you either place the correct segment value in DS, or use an explicit segment override when referring to these variables.

In Ideal mode, Turbo Assembler correctly checks for whether the external variable is addressable using any of the current segment registers, as described with the **ASSUME** directive.

See also **COMM, GLOBAL, PUBLIC, PUBLICDLL**

Example

```
EXTRN APROC:NEAR
  call APROC          ;calls into other module
```

.FARDATA MASM

Function Defines the start of a far data segment

Syntax `.FARDATA [`*`name`*`]`

Remarks **.FARDATA** indicates the start of a far initialized data segment. If you wish to have multiple, separate far data segments, you can provide an optional *name* to override the default segment name, thereby making a new segment.

You can place as many **.FARDATA** directives as you want in a source file. All the different pieces with the same name will be combined to produce one data segment, exactly as if you had entered all the data at once after a single **.FARDATA** directive.

Far data segments are not put in a group. You must explicitly make far segments accessible by loading the address of the far segment into a segment register before accessing the data.

See the **.MODEL** directive for complete information on the segment attributes for far data segments.

See also **.CODE, .DATA, .FARDATA?, .MODEL, .STACK**

Example

```
.FARDATA
FarBuf DB 80 DUP (0)
.CODE
  mov ax,@fardata
  mov ds,ax
  ASSUME ds:@fardata
  mov al,FarBuf[0]          ;get first byte of buffer
```

.FARDATA? MASM

Function Defines the start of a far uninitialized data segment

Syntax `.FARDATA? [`*`name`*`]`

Remarks **.FARDATA?** indicates the start of a far uninitialized data segment. If you wish to have multiple separate far data segments, you can provide an optional *name* to override the default segment name, thereby making a new segment.

You can place as many **.FARDATA?** directives as you want in a source file. All the different pieces with the same name will be combined to produce one data segment, exactly as if you had entered all the data at once after a single **.FARDATA?** directive.

Far data segments are not put in a group. You must explicitly make far segments accessible by loading the address of the far segment into a segment register before accessing the data.

See the **.MODEL** directive for complete information on the segment attributes for uninitialized far data segments.

See also **.CODE , .DATA, .FARDATA, .MODEL, .STACK**

Example

```
.FARDATA?
FarBuf DB 80 DUP (?)
.CODE
mov    ax,@fardata?
mov    ds,ax
ASSUME ds:@fardata?
```

```
mov     al,FarBuf[0]      ;get first byte of buffer
```

FARDATA — Ideal, MASM

Function Defines the start of a far data segment

Syntax `FARDATA [name]`

Remarks Same as **.FARDATA**.

See also **.FARDATA**

GLOBAL — Ideal, MASM

Function Defines a global symbol

Syntax `GLOBAL definition [,definition]...`

Remarks **GLOBAL** acts as a combination of the **EXTRN** and **PUBLIC** directives. Each definition describes a symbol and has the following format (boldface items are literal):

```
[language] name [ [ count1 ] ] :type [:count2]
```

language is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or NOLANGUAGE and defines any language-specific conventions to be applied to the symbol name. Using a language in the **GLOBAL** directive temporarily overrides the current language setting (default or one established with the **.MODEL** directive). Note that you don't need to have a **.MODEL** directive in effect to use this feature.

If *name* is defined in the current source file, it is made public exactly as if used in a **PUBLIC** directive. If *name* is not defined in the current source file, it is declared as an external symbol of type *type*, as if the **EXTRN** directive had been used.

type must match the type of the symbol in the module where it is defined. It can be one of the following:

- **NEAR, FAR,** or **PROC**
- **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name
- **ABS**

The optional *count* specifies how many items this symbol defines. If the symbol's definition uses the **DUP** directive to allocate more than one item, you can place that value in the *count* field. This lets the **SIZE** and **LENGTH** operators correctly determine the size of the external data item. If you do not specify a *count*, it is assumed to be one.

The **GLOBAL** directive lets you have an INCLUDE file included by all source files; the INCLUDE file contains all shared data defined as global symbols. When you reference these data items in each module, the **GLOBAL** definition acts as an **EXTRN** directive, describing how the data is defined in another module. In the module in which you define the data item, the **GLOBAL** definition acts as a **PUBLIC** directive, making the data available to the other modules.

You can define more than one public symbol by separating each definition with a comma (,).

You must define a symbol as **GLOBAL** before you first use it elsewhere in your source file. Also note that each argument of **GLOBAL** accepts the same syntax as an argument of **EXTRN**, **ARG**, or **LOCAL**.

Note: In QUIRKS mode, the **GLOBAL** directive can be overridden. For example,

```
global DB ?
```

is a legal declaration under **QUIRKS**, though a warning will be generated.

See also **COMM, EXTRN, PUBLIC, PUBLICDLL**

Example

```
GLOBAL X:WORD,Y:BYTE
X DW 0                  ;made public for other module
  mov al,Y              ;Y is defined as external
```

GROUP — Ideal, MASM

Function Defines segments as accessible from a single segment register

Syntax Ideal mode:

```
GROUP name segmentname [,segmentname]...
```

MASM mode:

```
name GROUP segmentname [,segmentname]...
```

Remarks *name* defines the name of the group. *segmentname* can be either a segment name defined previously with the **SEGMENT** directive or an expression starting with **SEG**. You can use *name* in the **ASSUME** directive and also as a constant in expressions, where it evaluates to the starting paragraph address of the group.

All the segments in a group must fit into 64K, even though they don't have to be contiguous when linked.

In MASM mode, you must use a group override whenever you access a symbol in a segment that is part of a group. In Ideal mode, Turbo Assembler automatically generates group overrides for symbols in segments that belong to a group.

In the example shown here, even though *var1* and *var2* belong to different segments, they both belong to the group **DGROUP**. Once the DS segment register is set to the base address of **DGROUP**, *var1* and *var2* can be accessed as belonging in a single segment.

Notice that in Ideal mode, the name comes after the **GROUP** directive.

See also **ASSUME, SEGMENT**

Example

```
DGROUP GROUP SEG1,SEG2
SEG1 SEGMENT
VAR1 DW 3
SEG1 ENDS
SEG2 SEGMENT
VAR2 DW 5
SEG2 ENDS
SEG3 SEGMENT
mov  ax,DGROUP      ;get base address of group
mov  ds,ax          ;set up to access data
ASSUME DS:DGROUP    ;inform assembler of DS
mov  ax,VAR1
mul  VAR2
SEG3 ENDS
```

IDEAL — Ideal, MASM

Function Enters Ideal assembly mode

Syntax `IDEAL`

Remarks **IDEAL** makes the expression parser only accept the more rigid, type-checked syntax required by Ideal mode. See Chapter 11 of the *User's Guide* for a complete discussion of the capabilities and advantages of Ideal mode.

Ideal mode will stay in effect until it is overridden by a **MASM** or **QUIRKS** directive.

See also **MASM, QUIRKS**

Example

```
IDEAL
  mov [BYTE ds:si],1    ;Ideal operand syntax
```

IF — Ideal, MASM

Function Starts conditional assembly block; enabled if expression is true

Syntax `IF expression`

Remarks *expression* must evaluate to a constant and may not contain any forward-referenced symbol names. If the expression evaluates to a nonzero value, the statements within the conditional block are assembled.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ENDIF, ELSE, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IF DoBuffering
    mov ax,BufNum
    call ReadBuf
ENDIF
```

IF1 — Ideal, MASM

Function Starts conditional assembly block; enabled on pass 1

Syntax `IF1`

Remarks Normally, Turbo Assembler acts as a single-pass assembler, which means that the statements within the conditional block are always assembled since there is always at least one pass.

If you use Turbo Assembler's multi-pass capability (invoked with the **/m** command-line switch), **IF1** assembles the statements within the conditional block on the first assembly pass.

When using a forward-referenced operator redefinition, you can't always tell from the listing file that something has gone wrong. By the time the listing is generated, the operator has been redefined. This means that the listing will appear to be correct, but the code would not have been generated properly to the object file.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IF1
   ;This code assembled during first pass
ENDIF
```

IF2 — Ideal, MASM

Function Starts conditional assembly block; enabled on pass 2

Syntax `IF2`

Remarks Normally, Turbo Assembler acts as a single-pass assembler. This means that the statements within the conditional block are never assembled, since there is only one pass. In this case, a pass-dependent warning message is generated to alert you to a potentially hazardous code omission.

If you use Turbo Assembler's multi-pass capability (invoked with the **/m** command-line switch), **IF2** assembles the statements within the conditional block on the second assembly pass. No pass-dependent warning is generated, however.

When using a forward-referenced operator redefinition, you can't always tell from the listing file that something has gone wrong. By the time the listing is generated, the operator has been redefined. This means that the listing will appear to be correct, but the code would not have been generated properly to the object file.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IF2
   ;this code assembles on the second assembly pass,
   ;if there is one
ENDIF
```

IFB — Ideal, MASM

Function Starts conditional assembly block; enabled if argument is blank

Syntax `IFB <argument>`

Remarks If *argument* is blank (empty), the statements within the conditional block are assembled. Use **IFB** to test whether a macro was called with a real argument to replace the specified dummy argument.

You must always surround the argument to be tested with angle brackets (< >).

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
PRINT MACRO MSG
  IFB <MSG>
  mov si,DefaultMsg
  ELSE
  mov si,MSG
  ENDIF
  call ShowIt
  ENDM
```

IFDEF — Ideal, MASM

Function Starts conditional assembly block; enabled if symbol is defined

Syntax `IFDEF symbol`

Remarks If *symbol* is defined, the statements within the conditional block are assembled.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
IFDEF SaveSize
  BUF DB SaveSize DUP (?)     ;define BUFFER only if
                              ; SAVESIZE is defined
ENDIF
```

IFDIF, IFDIFI — Ideal, MASM

Function Starts conditional assembly block; enabled if arguments are different

Syntax `IFDIF <argument1>,<argument2>`

Remarks You usually use **IFDIF** inside a macro. It tests whether its two arguments are different character strings. Either of the arguments can be macro dummy arguments that will have real arguments to the macro call that was substituted before performing the comparison. If the two strings are different, the statements within the conditional block are assembled. The two strings are compared on a character-by-character basis; case is significant. If you want case to be ignored, use the **IFDIFI** directive.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
loadb MACRO source
  IFDIF <source>,<si>
    mov si,source       ;set up string pointer
  ENDIF
  lodsb                 ;read the byte
  ENDM
```

IFE — Ideal, MASM

Function Starts conditional assembly block; enabled if *expression* is false

Syntax `IFE expression`

Remarks *expression* must evaluate to a constant and may not contain any forward-referenced symbol names. If the expression evaluates to zero, the statements within the conditional block are assembled.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFIDN, IFIDNI, IFNB, IFNDEF**

Example

```
  IFE StackSize
  StackSize=1024
    DB  StackSize DUP (?)     ;allocate stack
  ENDIF
```

IFIDN, IFIDNI — Ideal, MASM

Function Starts conditional assembly block; enabled if arguments are identical

Syntax `IFIDN <argument1>,<argument2>`

Remarks You usually use **IFIDN** inside a macro. It tests whether its two arguments are identical character strings. Either of the arguments can be macro dummy arguments that will have real arguments to the macro call that was substituted before performing the comparison. If the two strings are identical, the statements within the conditional block are assembled. The two strings are compared on a character-by-character basis; case is significant. If you want case to be ignored, use the **IFIDNI** directive.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDNI, IFNB, IFNDEF**

Example

```
RDWR MACRO BUF,RWMODE
  mov ax,BUF
  IFIDN <RWMODE>,<READ>
    call ReadIt
  ENDIF
  IFIDN <RWMODE>,<WRITE>
    call WriteIt
  ENDIF
  ENDM
```

IFNB — Ideal, MASM

Function Starts conditional assembly block, enabled if argument is nonblank

Syntax `IFNB <argument>`

Remarks If *argument* is nonblank, the statements within the conditional block are assembled. Use **IFNB** to test whether a macro was called with a real argument to replace the specified dummy argument.

You must always surround the argument to be tested with angle brackets (< >).

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNDEF**

Example

```
PopRegs MACRO REG1,REG2
```

```
IFNB <REG1>
  pop REG1
ENDIF
IFNB <REG2>
  pop REG2
ENDIF
ENDM
```

IFNDEF Ideal, MASM

Function Starts conditional assembly block; enabled if *symbol* is not defined

Syntax `IFNDEF symbol`

Remarks If *symbol* has not yet been defined in the source file, the statements within the conditional block are assembled.

Use the **ENDIF** directive to terminate the conditional assembly block.

See also **ELSE, ENDIF,IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB**

Example

```
IFNDEF  BufSize
BufSize EQU 128      ;define buffer size if not defined
ENDIF
```

%INCL Ideal, MASM

Function Allows listing of include files

Syntax `%INCL`

Remarks Use **%INCL** after a **%NOINCL** directive has turned off listing of INCLUDE files. This is the default INCLUDE file listing mode.

See also **%NOINCL**

Example

```
%INCL
INCLUDE DEFS.INC       ;contents appear in listing
```

INCLUDE — Ideal, MASM

Function Includes source code from another file

Syntax Ideal mode:

```
INCLUDE "filename"
```

MASM mode:

```
INCLUDE filename
```

Remarks *filename* is a source file containing assembler statements. Turbo Assembler assembles all statements in the included file before continuing to assemble the current file.

filename uses the normal DOS file-naming conventions, where you can enter an optional drive, optional directory, file name, and optional extension. If you don't provide an extension, .ASM is presumed.

If *filename* does not include a directory or drive name, Turbo Debugger first searches for the file in any directories specified by the /I command-line option and then in the current directory.

You can nest **INCLUDE** directives as deep as you want.

Notice that in Ideal mode, you must surround the *filename* with quotes.

Example

```
;MASM mode
INCLUDE MYMACS.INC     ;include MACRO definitions
;Ideal mode
INCLUDE "DEFS.INC"     ;include EQU definitions
```

INCLUDELIB — Ideal, MASM

Function Tells the linker to include a library

Syntax Ideal mode:

```
INCLUDELIB "filename"
```

MASM mode:

```
INCLUDELIB filename
```

Remarks *filename* is the name of the library that you want the linker to include at link time. If you don't supply an extension with *filename,* the linker assumes .LIB.

Use **INCLUDELIB** when you know that the source file will always need to use routines in the specified library. That way you don't have to remember to specify the library name in the linker commands.

Notice that in Ideal mode, you must surround the *filename* with quotes.

Example
```
INCLUDELIB diskio    ;includes DISKIO.LIB
```

INSTR — Ideal, MASM51

Function Returns the position of one string inside another string

Syntax *name* INSTR [*start*,]*string1*,*string2*

Remarks *name* is assigned a value that is the position of the first instance of *string2* in *string1*. The first character in *string1* has a position of one. If *string2* does not appear anywhere within *string1*, a value of 0 is returned.

See also **CATSTR, SIZESTR, SUBSTR**

Example
```
COMMAPOS INSTR <aaa,bbb>,<,>    ;COMMAPOS = 4
```

IRP — Ideal, MASM

Function Repeats a block of statements with string substitution

Syntax
```
IRP parameter,<arg1[,arg2]...>
  statements
ENDM
```

Remarks The *statements* within the repeat block are assembled once for each argument in the list enclosed in angle brackets. The list may contain as many arguments as you want. The arguments may be any text, such as symbols, strings, numbers, and so on. Each time the block is assembled, the next argument in the list is substituted for any instance of *parameter* in the enclosed statements.

You must always surround the argument list with angle brackets (< >), and arguments must be separated by commas. Use the **ENDM** directive to end the repeat block.

You can use **IRP** both inside and outside of macros.

See also **IRPC, REPT**

Example
```
IRP  reg,<ax,bx,cx,dx>
  push reg
ENDM
```

IRPC — Ideal, MASM

Function Repeats a block of statements with character substitution

Syntax

```
IRPC parameter,string
  statements
ENDM
```

Remarks The *statements* within the repeat block are assembled once for each character in *string*. The string may contain as many characters as you want. Each time the block is assembled, the next character in the list is substituted for any instances of *parameter* in the enclosed statements.

Use the **ENDM** directive to end the repeat block.

You can use **IRPC** both inside and outside of macros.

See also **IRP, REPT**

Example

```
IRPC LUCKY,1379
  DB   LUCKY            ;allocate a lucky number
ENDM
```

This creates 4 bytes of data containing the values 1, 3, 7, and 9.

JUMPS — Ideal, MASM

Function Enables stretching of conditional jumps to near or far addresses

Syntax

```
JUMPS
```

Remarks **JUMPS** causes Turbo Assembler to look at the destination address of a conditional jump instruction, and if it is too far away to reach with the short displacement that these instructions use, it generates a conditional jump of the opposite sense around an ordinary jump instruction to the desired target address. For example,

```
jne xyz
```

becomes

```
je @@A
jmp xyz
@@a:
```

If the destination address is forward-referenced, you should use the **NEAR** or **FAR** operator to tell Turbo Assembler how much space to allocate for the jump instruction. If you don't do this, inefficient code may be generated, due to the nature of single-pass assembly. You can use the multi-pass capability of Turbo Assembler (via the **/m** command-line switch) to force the most efficient code to be generated, but it may lengthen the assembly process, depending on the number of passes required. Therefore, it's recommended that you use the **NEAR** or **FAR** operator in the first place.

This directive has the same effect as using the **/JJUMPS** command-line option.

See also **NOJUMPS**

Example

```
  JUMPS          ;enable jump stretching
  jne SHORT @A   ;can reach A
@@A:
```

LABEL — Ideal, MASM

Function Defines a symbol with a specified type

Syntax Ideal mode:
`LABEL` *`name type`*

MASM mode:
`name` `LABEL` *`type`*

Remarks *name* is a symbol that you have not previously defined in the source file. *type* describes the size of the symbol and whether it refers to code or data. It can be one of the following:

- **NEAR, FAR,** or **PROC. PROC** is the same as either **NEAR** or **FAR,** depending on the memory set using the **MODEL** directive
- **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name

The label will only be accessible from within the current source file, unless you use the **PUBLIC** directive to make it accessible from other source files.

Notice that in Ideal mode, *name* comes after the **LABEL** directive.

Use **LABEL** to access different-sized items than those in the data structure; see the example that follows.

See also **:**

Example

```
WORDS LABEL WORD      ;access "BYTES" as WORDS
BYTES DB 64 DUP (0)
  mov WORDS[2],1      ;write WORD of 1
```

.LALL MASM

Function Enables listing of macro expansions

Syntax `.LALL`

See also **%MACS**

.LFCOND MASM

Function Shows all statements in conditional blocks in the listing

Syntax `.LFCOND`

Remarks **.LFCOND** enables the listing of false conditional blocks in assembly listings. **.LFCOND** is not affected by the **/X** option.

See also **%CONDS**

%LINUM Ideal, MASM

Function Sets the width of the line-number field in listing file

Syntax `%LINUM` *size*

Remarks **%LINUM** allows you to set how many columns the line numbers take up in the listing file. *size* must be a constant. If you want to make your listing as narrow as possible, you can reduce the width of this field. Also, if your source file contains more than 9,999 lines, you can increase the width of this field so that the line numbers are not truncated.

The default width for this field is 4 columns.

Example

```
%LINUM 5     ;allows up to line 99999
```

%LIST Ideal, MASM

Function Shows source lines in the listing

Syntax `%LIST`

Remarks **%LIST** reverses the effect of a **%NOLIST** directive that caused all listing output to be suspended.

This is the default listing mode; normally, all source lines are placed in the listing output file.

See also **.LIST, %NOLIST, .XLIST**

Example

```
%LIST
  jmp xyz     ;this line always listed
```

.LIST — MASM

Function Shows source lines in the listing

Syntax `.LIST`

See also **%LIST**

Example

```
   .XLIST          ;turn off listing
INCLUDE  MORE.INC
   .LIST           ;turn on listing
```

LOCAL — Ideal, MASM

Function Defines local variables for macros and procedures

Syntax In macros:

`LOCAL` *`symbol`* `[,`*`symbol`*`]...`

In procedures:

`LOCAL` *`name`*`:`*`type`*`[:`*`count`*`] [,`*`name`*`:`*`type`*`[:`*`count`*`]]... [=`*`symbol`*`]`

Remarks **LOCAL** can be used both inside macro definitions started with the **MACRO** directive and within procedures defined with **PROC**. It behaves slightly differently depending on where it is used.

Within a macro definition, **LOCAL** defines temporary *symbol* names that are replaced by new unique symbol names each time the macro is expanded. The unique names take the form of *??number,* where *number* is hexadecimal and starts at 0000 and goes up to FFFF.

Within a procedure, **LOCAL** defines names that access stack locations as negative offsets relative to the BP register. The first local variable starts at BP (type X count). If you end the argument list with an equal sign (=) and a symbol, that symbol will be equated to the total size of the local symbol block in bytes. You can then use this value to make room on the stack for the local variables.

Each *localdef* has the following syntax:

```
localname:[ [distance] PTR]type[:count]
```

You can use this alternative syntax for each *localdef*:

```
localname[ [count] ][:[distance] PTR]type]
```

localname is the name you'll use to refer to this local symbol throughout the procedure.

type is the data type of the argument and can be one of the following: **WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE**, or a structure name. If you don't specify a type, and you're using the alternative syntax, **WORD** size is assumed.

count specifies how many elements of the specified *type* to allocate on the stack.

The optional *distance* and **PTR** lets you tell Turbo Assembler to include debugging information for Turbo Debugger, which tells it this local variable is really a pointer to another data type. See the **PROC** directive for a discussion of how this works.

Here are some examples of valid arguments:

```
LOCAL X:DWORD:4,Y:NEAR PTR WORD
```

Here are some arguments using the alternative syntax:

```
LOCAL X[4]:DWORD,Y:PTR STRUCNAME
```

The type indicates how much space should be reserved for name. It can be one of **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD**, or **TBYTE** for a data value. It can be one of **NEAR, FAR**, or **PROC** for a code pointer.

The **LOCAL** directive must come before any other statements in a macro definition. It can appear anywhere within a procedure, but should precede the first use of the symbol it defines.

See also **ARG, MACRO, PROC, USES**

Example

```
OnCarry MACRO FUNC
   LOCAL DONE
   jnc  DONE                   ;hop around if no carry
   call FUNC                   ;else call function
DONE:
   ENDM
READ PROC NEAR
   LOCAL N:WORD =LSIZE
   push bp
   mov  bp,sp
   sub  sp,LSIZE           ;make room for local var
   mov  N,0                ;actually N = [BP-2]
     ;Body of func goes here
   add  sp,LSIZE           ;adjust stack
   pop  bp
   ret
READ ENDP
```

LOCALS — Ideal, MASM

Function Enables local symbols

Syntax `LOCALS [`*`prefix`*`]`

Remarks Local symbols normally start with two at-signs (@@), which is the default, and are only visible inside blocks whose boundaries are defined by the **PROC/ENDP** pair within a procedure or by nonlocal symbols outside a procedure. You define a nonlocal symbol using **PROC, LABEL,** or the colon operator. If you use the **LOCALS** directive, any symbols between pairs of nonlocal symbols can only be accessed from within that block. This lets you reuse symbol names inside procedures and other blocks.

prefix is the two-character symbol prefix you want to use to start local symbol names. Usually, two at-signs indicate a local symbol. If you have a program that has symbols starting with two at-signs, or if you use your own convention to indicate local symbols, you can set two different characters for the start of local symbols. The two characters must be a valid start of a symbol name, for example **.?** is OK, but **..** is not. When you set the prefix, local symbols are enabled at the same time. If you turn off local symbols with the **NOLOCALS** directive, the prefix is remembered for the next time you enable local symbols with the **LOCALS** directive.

Local symbols are automatically enabled in Ideal mode. You can use the **NOLOCALS** directive to disable local symbols. Then, all subsequent symbol names will be accessible throughout your source file.

See also **IDEAL, NOLOCALS**

Example

```
LOCALS
.MODEL small
.CODE
start:
@@1:              ; unique label
  loop    @@1
```

```
one:                ; terminates visibility of @1 above
  loop    one
@@1:                ; unique label
  loop    @@1

Foo     PROC NEAR ; terminates visibility of @1 above
@@1:                ; unique label
  loop    @@1
two:                ; doesn't terminate visibility of @@1
                    ; above because in PROCs local labels
                    ; have visibility throughout the PROC
  loop    two
@@1:                ; conflicting label with @@1 above
  loop    @@1
Foo     ENDP
END     start
```

MACRO — Ideal, MASM

Function Defines a macro

Syntax Ideal mode:

`MACRO` *`name`* `[`*`parameter`* `[,`*`parameter`*`]...]`

MASM mode:

`name` `MACRO [`*`parameter`* `[,`*`parameter`*`]...]`

Remarks You use *name* later in your source file to expand the macro. *parameter* is a placeholder you can use throughout the body of the macro definition wherever you want to substitute one of the actual arguments the macro is called with.

Use the **ENDM** directive to end the macro definition.

See also **ENDM**

Example

```
SWAP MACRO a,b    ;swap two word items
  mov ax,a
  mov a,b
  mov b,ax
  ENDM
```

%MACS — Ideal, MASM

Function Enables listing of macro expansions

Syntax
```
%MACS
```

Remarks **%MACS** reverses the effect of a previous **%NOMACS** directive, so that the lines resulting from macro expansions appear in the listing. (Same as **.LALL**.)

See also **.LALL, %NOMACS, .SALL, .XALL**

Example
```
%MACS
MyMac 1,2,3     ;expansion appears in listing
```

MASM — Ideal, MASM

Function Enters MASM assembly mode

Syntax
```
MASM
```

Remarks **MASM** tells the expression parser to accept MASM's loose expression syntax. See Appendix B for a discussion of how this differs from Ideal mode.

Turbo Assembler is in MASM mode when it first starts assembling a source file.

See also **IDEAL, QUIRKS**

Example
```
MASM
mov al,es:24h          ;ghastly construct
```

MASM51 — Ideal, MASM

Function Enables assembly of some MASM 5.1 enhancements

Syntax
```
MASM51
```

Remarks **MASM51** enables the following capabilities that are not normally available with Turbo Assembler:

- **SUBSTR, CATSTR, SIZESTR,** and **INSTR** directives
- Line continuation with backslash (\)

If you also enable Quirks mode with the **QUIRKS** directive, these additional features become available:

- Local labels defined with @@ and referred to with *@F* and *@B*
- Redefinition of variables inside **PROC**s
- Extended model **PROC**s are all **PUBLIC**.

See also **NOMASM51**

Example

```
MASM51
MyStr   CATSTR <ABC>,<XYZ>   ;MYSTR = "ABCXYZ"
```

.MODEL — MASM

Function Sets the memory model for simplified segmentation directives

Syntax

```
.MODEL [model modifier] memorymodel [module name]
      [, [language modifier] language] [,model modifier]]
.MODEL TPASCAL
```

Remarks *model modifier* must be either NEARSTACK or FARSTACK if present. This prevents Turbo Assembler from making the assumption that SS is part of DGROUP (the default for all models). Note that the syntax allows two choices for the placement of the model modifier. Either can be used; the placement following *memorymodel* is provided for compatibility with MASM 5.2 (Quick Assembler).

memorymodel is a model of tiny, small, medium, compact, large, or huge. The large and huge models use the same segment definitions, but the *@DataSize* predefined equate symbol is defined differently. (See the section "Other Simplified Segment Directives" in Chapter 5 of the *User's Guide* for a description of the *@DataSize* symbol.)

When you want to write an assembler module that interfaces to Turbo Pascal, you use a special form of the **.MODEL** directive:

```
.MODEL TPASCAL
```

This informs Turbo Assembler to use the Turbo Pascal segment-naming conventions. You can only use the **.CODE** and **.DATA** simplified segmentation directives when you specify **TPASCAL**. There is no need to supply a second argument to the **.MODEL** directive, **TPASCAL** says it all. If you try and use any of the directives that are forbidden with Turbo Pascal assembler modules, you will get a warning message.

To define *memorymodel*, you must use the **.MODEL** directive before any other simplified segmentation directives such as **.CODE, .DATA, .STACK,** and so on. The code and data segments will all be 32-bit segments if you've enabled the 80386 processor with the **.386** or **.386P** directive before issuing the **.MODEL** directive. Be certain this is what you want before you implement it. Also be sure to put the **.MODEL** directive before either **.386** or **.386P** if you want 16-bit segments.

module name is used in the large code models to declare the name of the code segment. Normally, this defaults to the module name with **_TEXT** tacked onto the end. Using the optional module name parameter gives you more flexibility in building programs from different files and modules.

language modifier is either **WINDOWS, ODDNEAR, ODDFAR,** or **NORMAL.** The modifier selects automatic generation of code for stack setup and cleanup on procedure entry and exit for MSWindows, and ODDNEAR or ODDFAR overlays. NORMAL (the default) selects normal procedure entry and exit code.

language tells Turbo Assembler what language you will be calling from to access the procedures in this module. *language* can be **C, PASCAL, BASIC, FORTRAN, PROLOG,** or **NOLANGUAGE.** Turbo Assembler automatically generates the appropriate procedure entry and exit code when you use the **PROC** and **ENDP** directives.

If you specify the C language, all public and external symbol names will be prefixed with an underscore (_). This is because, by default, Turbo C starts all names with an underscore. You don't need **MASM51** or **QUIRKS** if you want to prefix all **PUBLIC** and **EXTRN** symbols with an underbar (_) for the C language.

language also tells Turbo Assembler in what order procedure arguments were pushed onto the stack by the calling module. If you set *language* to **PASCAL, BASIC,** or **FORTRAN,** Turbo Assembler presumes that the arguments were pushed from left to right, in the order they were encountered in the source statement that called the procedure. If you set *language* to **C, PROLOG,** or **NOLANGUAGE,** Turbo Assembler presumes that the arguments were pushed in reverse order, from right to left in the source statement. With **C, PROLOG,** and **NOLANGUAGE,** Turbo Assembler also presumes that the calling function will remove any pushed arguments from the stack. For other languages, Turbo Assembler generates the appropriate form of the **RET** instruction, which removes the arguments from the stack before returning to the caller.

Note that use of the **.MODEL** directive is not required to take advantage of the language-specific features mentioned. The **PROC, EXTRN, PUBLIC, COMM, GLOBAL,** and **PUBLICDLL** directives, as well as the **CALL** instruction, accept a language specifier that temporarily overrides any other language convention in affect for the symbol with which it is used. This allows you, for example, to conveniently mix and match procedures with C and Pascal calling conventions in the same program.

If you don't supply *language*, **.MODEL** simply defines how the segments will be used with the simplified segmentation directives.

The following tables show the default segment attributes for each memory model.

Table 3.1
Default segments and types for tiny memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the MODEL directive.

Table 3.2
Default segments and types for small memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the MODEL directive.

Table 3.3
Default segments and types for medium memory model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the MODEL directive.

Table 3.4
Default segments and types for compact memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the MODEL directive.

Table 3.5
Default segments and types for large or huge memory model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the MODEL directive.

Table 3.6
Default segments and types for Turbo Pascal (TPASCAL) memory model

Directive	Name	Align	Combine	Class	Group
	.CODE	CODE	BYTE	PUBLIC	
	.DATA	DATA	WORD	PUBLIC	

See also **.CODE, .DATA, .FARDATA, .FARDATA?, .STACK**

Example

```
.MODEL MEDIUM     ;set small data, large code
```

MODEL — Ideal, MASM

Function Sets the memory model for simplified segmentation directives

Syntax

```
MODEL [model modifier] memorymodel [module name]
      [, [language modifier] language [,model modifier]]
```

See also **.MODEL**

MULTERRS — Ideal, MASM

Function Allows multiple errors to be reported on a single source line

Syntax

```
MULTERRS
```

Remarks **MULTERRS** lets more than one error or warning message appear for each source line. This is sometimes helpful in locating the cause of a subtle error or when the source line contains more than one error.

Note that sometimes additional error messages can be a "chain reaction" caused by the first error condition; these "chain" error messages may desist once you correct the first error.

See also **NOMULTERRS**

Example

```
MULTERRS
mov  ax,[bp+abc        ;produces two errors:
                       ;1) Undefined symbol: abc
                       ;2) Need right square bracket
```

NAME — Ideal, MASM

Function Sets the object file's module name

Syntax `NAME modulename`

Remarks This directive has no effect in MASM mode; it only works in Ideal mode.

Turbo Assembler usually uses the source file name with any drive, directory, or extension as the module name. Use **NAME** if you wish to change this default name; *modulename* will be the new name of the module.

Example

```
NAME loader
```

%NEWPAGE — Ideal, MASM

Function Starts a new page in the listing file

Syntax `%NEWPAGE`

Remarks The source lines appearing after **%NEWPAGE** will begin at the start of a new page in the listing file. (Same as **PAGE** with no arguments.)

See also **PAGE**

Example

```
%NEWPAGE
; Appears on first line of new page
```

%NOCONDS — Ideal, MASM

Function Disables the placement of statements in false conditional blocks in the listing file

Syntax `%NOCONDS`

Remarks **%NOCONDS** overrides the listing control. When this control is in effect, the listing won't show statements within conditional blocks, even those that evaluate as false and don't result in the evaluation of enclosed statements. (Same as **.SFCOND.**)

See also **%CONDS, .LFCOND, .SFCOND, .TFCOND**

Example

```
%NOCONDS
IF 0
   mov  ax,1    ;not in listing, since "IF 0" above
ENDIF
```

%NOCREF — Ideal, MASM

Function Disables cross-reference listing (CREF)

Syntax `%NOCREF [symbol, ... ]`

Remarks **%NOCREF** stops cross-reference information from being accumulated for symbols encountered from this point forward in the source file.

If you use **%NOCREF** alone without specifying any symbols, cross-referencing is disabled completely. If you supply one or more *symbol* names, cross-referencing is disabled only for those symbols. (Same as **.XCREF**.)

See also **%CREF, .CREF, %CREFALL, %CREFREF, %CREFUREF, .XCREF**

Example

```
%XCREF xyz
WVAL DW 0     ;CREF shows WVAL defined here
xyz  DB 0     ;doesn't appear in CREF
```

%NOCTLS — Ideal, MASM

Function Disables printing of listing controls

Syntax `%NOCTLS`

Remarks **%NOCTLS** reverses the effect of a previous **%CTLS** directive, which caused all listing-control directives to be placed in the listing file. After issuing **%NOCTLS**, all subsequent listing-control directives will not appear in the listing file.

This is the default listing-control mode that's in effect when Turbo Assembler starts assembling a source file.

See also **%CTLS**

Example

```
%NOCTLS
%LIST       ;this will not appear in listing
```

NOEMUL — Ideal, MASM

Function Forces generation of real 80x87 floating-point instructions

Syntax
```
NOEMUL
```

Remarks **NOEMUL** sets Turbo Assembler to generate real floating-point instructions to be executed by an 80x87 coprocessor. You can combine this directive with the **EMUL** directive when you wish to generate real floating-point instructions in one portion of a file and emulated instructions in another portion.

NOEMUL is the normal floating-point assembly mode that's in effect when Turbo Assembler starts to assemble a file.

See also **EMUL**

Example
```
NOEMUL        ;assemble real FP instructions
finit
EMUL          ;back to emulation
```

%NOINCL — Ideal, MASM

Function Disables listing of include files

Syntax
```
%NOINCL
```

Remarks **%NOINCL** stops all subsequent INCLUDE file source lines from appearing in the listing until a **%INCL** is enabled. This is useful if you have a large INCLUDE file that contains things such as a lot of **EQU** definitions that never change.

See also **%INCL**

Example
```
%NOINCL
INCLUDE DEFS.INC       ;doesn't appear in listing
```

NOJUMPS — Ideal, MASM

Function Disables stretching of conditional jumps

Syntax
```
NOJUMPS
```

Remarks If you use **NOJUMPS** in conjunction with **JUMPS**, you can control where in your source file conditional jumps should be expanded to reach their destination addresses.

This is the default mode Turbo Assembler uses when it first starts assembling a file.

See also **JUMPS**

%NOLIST Ideal, MASM

Function Disables output to listing file

Syntax
```
%NOLIST
```

Remarks **%NOLIST** stops all output to the listing file until a subsequent **%LIST** turns the listing back on. This directive overrides all other listing controls. (Same as **.XLIST**.)

See also **%LIST, .LIST, .XLIST**

Example
```
%NOLIST
  add dx,ByteVar       ;not in listing
```

NOLOCALS Ideal, MASM

Function Disables local symbols

Syntax
```
NOLOCALS
```

Remarks If local symbols are enabled with the **LOCALS** directive, any symbol starting with two at-signs (@@) is considered to be a local symbol. If you use symbols in your program that start with two at-signs but you don't want them to be local symbols, you can use this directive where appropriate.

Local symbols start off disabled in MASM mode.

See also **LOCALS, MASM**

Example
```
NOLOCALS
abc PROC
@@$1:
  loop @@$1
abc ENDP
xyz PROC
@@1:          ;label conflict with @@1 above
  loop @@1
xyz ENDP
```

%NOMACS Ideal, MASM

Function Lists only macro expansions that generate code

Syntax `%NOMACS`

Remarks **%NOMACS** prevents the listing source lines that generate no code from being listed, for example, comments, **EQU** and = definitions, **SEGMENT** and **GROUP** directives.

This is the default listing mode for macros that's in effect when Turbo Assembler first starts assembling a source file. (Same as **.XALL.**)

See also **.LALL, %MACS,.SALL**

NOMASM51 Ideal, MASM

Function Disables assembly of certain MASM 5.1 enhancements

Syntax `NOMASM51`

Remarks Disables the MASM 5.1 features described under the **MASM51** directive. This is the default mode when Turbo Assembler first starts assembling your source file.

See also **MASM51**

Example

```
MASM51
SLEN SIZESTR <ax,bx>  ;SLEN = 5
NOMASM51
CATSTR PROC NEAR      ;CATSTR OK user symbol in
                      ; non-MASM 5.1 mode
;
CATSTR ENDP
```

NOMULTERRS Ideal, MASM

Function Allows only a single error to be reported on a source line.

Syntax `NOMULTERRS`

Remarks **NOMULTERRS** only lets one error or warning message appear for each source line. If a source line contains multiple errors, Turbo Assembler reports the most-significant error first. When you correct this error, in many cases the other error messages disappear as well. If you prefer to decide for yourself which are the most important messages, you can use the **MULTERRS** directive to see all the messages for each source line.

By default, Turbo Assembler uses this error-reporting mode when first assembling a source file.

See also **MULTERRS**

Example

```
NOMULTERRS
mov  ax,[bp+abc        ;one error:
                       ;1) Undefined symbol: abc
```

Will produce the single error message:

```
**Error** MULTERRS.ASM(6) Undefined symbol: ABC
```

%NOSYMS — Ideal, MASM

Function Disables symbol table in listing file

Syntax `%NOSYMS`

Remarks **%NOSYMS** prevents the symbol table from appearing in your file. The symbol table, which shows all the symbols you defined in your source file, usually appears at the end of the listing file.

See also **%SYMS**

Example

```
%NOSYMS     ;now we won't get a symbol table
```

%NOTRUNC — Ideal, MASM

Function Wordwraps too-long fields in listing file

Syntax `%NOTRUNC`

Remarks The object code field of the listing file has enough room to show the code emitted for most instructions and data allocations. You can adjust the width of this field with the **%BIN** directive. If a single source line emits more code than can be displayed on a single line, the rest is normally truncated and therefore not visible. Use the **%NOTRUNC** directive when you wish to see all the code that was generated.

%NOTRUNC also controls whether the source lines in the listing file are truncated or will wrap to the next line. Use the **%TEXT** directive to set the width of the source field.

See also **%BIN, %TEXT, %TRUNC**

Example

```
%NOTRUNC
  DQ 4 DUP (1.2,3.4)    ;wraps to multiple lines
```

NOWARN — Ideal, MASM

Function Disables a warning message

Syntax `NOWARN [`*`warnclass`*`]`

Remarks If you specify **NOWARN** without *warnclass,* all warnings are disabled. If you follow **NOWARN** with a warning identifier, only that warning is disabled. Each warning message has a three-letter identifier that's described under the **WARN** directive. These are the same identifiers used by the **/W** command-line option.

See also **WARN**

Example

```
NOWARN  OVF        ;disable arithmetic overflow warnings
DW 1000h * 1234h   ;doesn't warn now
```

ORG — Ideal, MASM

Function Sets the location counter in the current segment

Syntax `ORG `*`expression`*

Remarks *expression* must not contain any forward-referenced symbol names. It can either be a constant or an offset from a symbol in the current segment or from $, the current location counter.

You can back up the location counter before data or code that has already been admitted into a segment. You can use this to go back and fill in table entries whose values weren't known at the time the table was defined. Be careful when using this technique—you may accidentally overwrite something you didn't intend to.

The **ORG** directive can be used to connect a label with a specific absolute address. The **ORG** directive can also set the starting location for .COM files (**ORG** 100h).

See also **SEGMENT**

Example

```
PROG SEGMENT
  ORG 100h           ;starting offset for .COM file
```

%OUT MASM

Function Displays message to screen

Syntax `%OUT text`

Remarks *text* is any message you want to display. The message is written to the standard output device, which is usually the screen. If you wish, you can use the DOS redirection facility to send screen output to a file.

Among other things, you can use **%OUT** so you'll know that sections of conditional assembly are being generated. (Same as **DISPLAY**.)

You can use the substitute operator inside a string passed to the **%OUT** directive; for example,

```
MAKE_DATA MACRO VALUE
  %OUT initializing a byte to: &VALUE&
  DB VALUE
  ENDM
```

See also **DISPLAY**

Example `%OUT Assembling graphics driver`

P186 Ideal, MASM

Function Enables assembly of 80186 instructions

See also **.186, .8086, .286, .286C, .286P, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

P286 Ideal, MASM

Function Enables assembly of all 80286 instructions

See also **.8086, .186, .286, .286C, .286P, .386, .386C, .386P, P8086, P286N, P286P, P386, P386N, P386P**

P286N Ideal, MASM

Function Enables assembly of non-privileged 80286 instructions

See also **.8086, .186, .286C, .286P, .286, .386, .386C, .386P, P8086, P286, P286P, P386, P386N, P386P**

P286P Ideal, MASM

Function Enables assembly of privileged 80286 instructions

See also **.8086, .186, .286C, .286P, .286, .386, .386C, .386P, P8086, P286, P286N, P386, P386N, P386P**

P287 Ideal, MASM

Function Enables assembly of 80287 coprocessor instructions

See also **.8087, .287, .387, P8087, PNO87, P387**

P386 Ideal, MASM

Function Enables assembly of all 80386 instructions

See also **.8086, .186, .286C, .286, .286P, .386C, .386P, .386, P8086, P286, P286N, P286P, P386N, P386P**

P386N Ideal, MASM

Function Enables assembly of non-privileged 80386 instructions

See also **.8086, .186, .286C, .286, .286P, .386C, .386P, .386, P8086, P286, P286N, P286P, P386, P386P**

P386P — Ideal, MASM

Function Enables assembly of privileged 80386 instructions

See also **.8086, .186, .286C, .286, .286P, .386C, .386P, .386, P8086, P286, P286N, P286P, P386, P386N**

P387 — Ideal, MASM

Function Enables assembly of 80387 coprocessor instructions

See also **.8087, .287, .387, 8087, PNO87, P287**

P8086 — Ideal, MASM

Function Enables assembly of 8086 instructions only

See also **.186, .286C, .286, .286P, .386C, .386, .386P, .8086, P286, P286N, P286P, P386, P386N, P386P**

P8087 — Ideal, MASM

Function Enables assembly of 8087 coprocessor instructions

See also **.287, .387, .8087, 8087, PNO87, P287, P387**

PAGE — MASM

Function Sets the listing page height and width, starts new pages

Syntax
```
PAGE [rows] [,cols]
PAGE +
```

Remarks *rows* specifies the number of lines that will appear on each listing page. The minimum is 10 and the maximum is 255. *cols* specifies the number of columns wide the page will be. The minimum width is 59; the maximum is 255. If you omit either *rows* or *cols,* the current setting for that parameter will remain unchanged. To change only the number of columns, precede the column width with a comma; otherwise, you'll end up changing the number of rows instead.

If you follow the **PAGE** directive with a plus sign (+), a new page starts, the section number is incremented, and the page number restarts at 1.

If you use the **PAGE** directive with no arguments, the listing resumes on a new page, with no change in section number.

See also **%NEWPAGE, %PAGESIZE**

Example

```
PAGE        ;start a new page
PAGE ,80    ;set width to 80, don't change height
```

%PAGESIZE — Ideal, MASM

Function Sets the listing page height and width

Syntax `%PAGESIZE [rows] [,cols]`

Remarks *rows* specifies the number of lines that will appear on each listing page. The minimum is 10 and the maximum is 255. *cols* specifies the number of columns wide the page will be. The minimum width is 59; the maximum is 255.

If you omit either *rows* or *cols*, the current setting for that parameter will remain unchanged. If you only want to change the number of columns, make sure you precede the column width with a comma; otherwise, you will end up changing the number of rows instead.

See also **PAGE**

Example

```
%PAGESIZE 66,132  ;wide listing, normal height
%PAGESIZE ,80     ;don't change rows, cols = 80
```

%PCNT — Ideal, MASM

Function Sets segment:offset field width in listing file

Syntax `%PCNT width`

Remarks *width* is the number of columns you wish to reserve for the offset within the current segment being assembled. Turbo Assembler sets the width to 4 for ordinary 16-bit segments and sets it to 8 for 32-bit segments used by the 386 processor. **%PCNT** overrides these default widths.

See also **%BIN, %DEPTH, %LINUM**

Example

```
%PCNT 3
ORG 1234h    ;only 234 displayed
```

PNO87 Ideal, MASM

Function Prevents the assembling of coprocessor instructions

Syntax
```
PNO87
```

Remarks Normally, Borland's Turbo Assembler allows you to assemble instructions for the 80x87 coprocessor family. Use **PN087** if you want to make sure you don't accidentally use any coprocessor instructions. Also, use **PN087** if your software doesn't have a floating-point emulation package, and you know your program may be run on systems without a numeric coprocessor.

See also **.8087, .287, .387, P8087, P287, P387**

Example
```
PNO87
fadd       ;this generates an error
```

%POPLCTL Ideal, MASM

Function Recalls listing controls from stack

Syntax
```
%POPLCTL
```

Remarks **%POPLCTL** resets the listing controls to the way they were when the last **%PUSHLCTL** directive was issued. All the listing controls that you can enable or disable (such as **%MACS, %LIST, %INCL,** and so on) are restored. None of the listing controls that set field width are restored (such as **%DEPTH, %PCNT**). The listing controls are saved on a 16-level stack. This directive is particularly useful in macros and include files, where you can invoke special listing modes that disappear once the macro expansion terminates.

See also **%PUSHLCTL**

Example
```
%PUSHLCTL
%NOLIST
%NOMACS
   • • •
%POPLCTL     ;restore listings
```

PROC — Ideal, MASM

Function Defines the start of a procedure

Syntax Ideal mode:

```
PROC [language modifier] [language] name
[distance] [USES item,] [argument [,argument]
...] [RETURNS argument [, argument]...]
```

MASM mode:

```
name PROC [language modifier] [language]
[distance] [USES item,] [argument [,argument]
...] [RETURNS argument [, argument]...]
```

Remarks *name* is the name of a procedure. *language modifier* is either **WINDOWS, ODDNEAR, ODDFAR,** or **NORMAL.** The modifier selects automatic generation of code for stack setup and cleanup on procedure entry and exit for MSWindows, **ODDNEAR** or **ODDFAR** overlays, or normal procedure entry/exit code. The language modifier specified in the **PROC** directive overrides (for this procedure only) any language modifier that has been set with a previous **.MODEL** directive. *language* is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or **NOLANGUAGE,** and selects automatic generation of code for stack setup and cleanup on procedure entry and exit for the language specified. The language specified in the **PROC** directive overrides (for this procedure only) any language that has been set with previous directives. Also note that you no longer need a **.MODEL** directive in your program in order to generate entry or exit code for your procedures.

The optional *distance* can be **NEAR** or **FAR**; it defaults to the size of the default code memory model. If you are not using the simplified segmentation directives (**.MODEL**, and so on), the default size is **NEAR.** With the tiny, small, and compact models, the default size is also **NEAR**; all other models are **FAR.** *distance* determines whether any **RET** instructions encountered within the procedure generate near or far return instructions. A **FAR** procedure is expected to be called with a **FAR** (segment and offset) **CALL** instruction, and a **NEAR** procedure is expected to be called with a **NEAR** (offset only) **CALL.**

item is a list of registers and single tokens that the procedure uses. *item* is pushed on entry and popped on exit from the procedure. You can supply more than one item by separating the items with spaces. For example,

```
Mine PROC USES ax bx foo
```

would save and restore the AX and BX registers, as well as the word at location *foo.*

argument describes an argument the procedure is called with. The language specified in the **PROC** directive, or in the **.MODEL** directive if none was specified in **PROC**, determines whether the arguments are in reverse order on the stack. You must always list the arguments in the same order they appear in the high-level language function that calls the procedure. Turbo Assembler reads them in reverse order if necessary. Each *argument* has the following syntax (boldface items are literal):

```
argname[[count1]] [:distance] PTR] type] [:count2]
```

argname is the name you'll use to refer to this argument throughout the procedure. *distance* is optional and can be either **NEAR** or **FAR** to indicate that the argument is a pointer of the indicated size. *type* is the data type of the argument and can be **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE**, or a structure name. *count1* and *count2* are the number of elements of the specified type. The total count is calculated as *count1* * *count2*.

If you don't specify *type*, **WORD** is assumed.

If you add **PTR** to indicate that the argument is in fact a pointer to a data item, Turbo Assembler emits this debug information for Turbo Debugger. Using **PTR** only affects the generation of additional debug information, not the code Turbo Assembler generates. You must still write the code to access the actual data using the pointer argument.

If you use **PTR** alone, without specifying **NEAR** or **FAR** before it, Turbo Assembler sets the pointer size based on the current memory model and, for the 386 processor, the current segment address size (16 or 32 bit). The size is set to **WORD** in the tiny, small, and medium memory models, and to **DWORD** for all other memory models using 16-bit segments. If you're using the 386 and are in a 32-bit segment, the size is set to **DWORD** for tiny, small, and medium models, and to **FWORD** for compact, large, and huge models.

The optional **RETURNS** keyword introduces one or more arguments that won't be popped from the stack when the procedure returns to its caller. Normally, if you specify the language as **PASCAL** or **TPASCAL** when using the **.MODEL** directive, or as **PASCAL** in the **PROC** directive itself, all arguments are popped when the procedure returns. If you place arguments after the **RETURNS** keyword, they will be left on the stack for the caller to make use of, and then pop. In particular, you must define a Pascal string return value by placing it after the **RETURNS** keyword.

Use the **ENDP** directive to end a procedure definition. You must explicitly specify a **RET** instruction before the **ENDP** if you want the procedure to return to its caller.

Within **PROC/ENDP** blocks you may use local symbols whose names are not known outside the procedure. Local symbols start with double at-signs (@@).

You can nest **PROC/ENDP** directives if you want; if you do, the local symbols nest also.

Argument names that begin with the local symbol prefix when local symbols are enabled are limited in scope to the current procedure.

See also **ARG, ENDP, LOCAL, LOCALS, .MODEL, USES**

Example

```
ReadLine PROC NEAR
   ;body of procedure
ReadLine ENDP
   • • •
   call ReadLine
```

PUBLIC — Ideal, MASM

Function Declares symbols to be accessible from other modules

Syntax `PUBLIC [language] symbol [,[language] symbol]...`

Remarks *language* is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or **NOLANGUAGE,** and defines any language-specific conventions to be applied to the symbol name. Using a language in the **PUBLIC** directive temporarily overrides the current language setting (default or one established with the **.MODEL** directive). Note that you don't need to have a **.MODEL** directive in effect to use this feature.

symbol is published in the object file so that it can be accessed by other modules. If you do not make a symbol public, it can only be accessed from the current source file.

You can declare the following types of symbols to be public:

- data variable names
- program labels
- numeric constants defined with **EQU**

See also **COMM, EXTRN, GLOBAL**

Example

```
  PUBLIC XYPROC    ;make procedure public
XYPROC PROC NEAR
```

PUBLICDLL — Ideal, MASM

Function Declares symbols to be accessible as dynamic link entry points from other modules

Syntax `PUBLICDLL [language] symbol [,[language] symbol]...`

Remarks *symbol* is published in the object file as a dynamic link entry point so that it can be accessed by other programs under OS/2. This statement is used only to help build an OS/2 dynamic link library. If you don't make a symbol public, it can only be accessed from the current source file.

In most cases, you declare only **PROC** labels to be **PUBLICDLL.** Other program labels, data variable names, and numeric constants defined with **EQU** may also be declared to be **PUBLICDLL.**

The optional *language* specifier causes any language-specific conventions to be applied to the symbol name. For instance, using the **C** language specifier would cause the symbol name to be preceded by an underscore character when published in the object file. Valid language specifiers are **C, PASCAL, BASIC, FORTRAN, PROLOG,** and **NOLANGUAGE.**

See also **COMM, EXTRN, GLOBAL, PUBLIC**

Example

```
        PUBLICDLL XYPROC       ;make procedure XYPROC
XYPROC PROC NEAR               ;accessible as dynamic
                               ;link entry point
```

PURGE — Ideal, MASM

Function Removes a macro definition

Syntax `PURGE macroname [,macroname]...`

Remarks **PURGE** deletes the macro definition specified by the *macroname* argument. You can delete multiple macro definitions by supplying all their names, separated by commas.

You may need to use **PURGE** to restore the original meaning of an instruction or directive whose behavior you changed by defining a macro with the same name.

See also **MACRO**

Example

```
PURGE add
  add ax,4     ;behaves as normal ADD now
```

%PUSHLCTL — Ideal, MASM

Function Saves listing controls on stack

Syntax

```
%PUSHLCTL
```

Remarks **%PUSHLCTL** saves the current listing controls on a 16-level stack. Only the listing controls that can be enabled or disabled (**%INCL, %NOINCL,** and so on) are saved. The listing field widths are not saved. This directive is particularly useful in macros, where you can invoke special listing modes that disappear once the macro expansion terminates.

See also **%POPLCTL**

Example

```
%PUSHLCTL        ;save listing controls
%NOINCL
%MACS
%POPLCTL         ;back the way things were
```

QUIRKS — Ideal, MASM

Function Enables acceptance of MASM bugs

Syntax

```
QUIRKS
```

Remarks **QUIRKS** allows you to assemble a source file that makes use of one of the true MASM bugs. You should try to stay away from using this directive, since it merely perpetuates source code constructs that only work by chance. Instead, you should really correct the offending source code so that it does what you really intended.

See the section "Turbo Assembler Quirks Mode" (page 161) in Appendix B for a complete description of this mode.

See also **IDEAL, MASM**

Example

```
     QUIRKS
BVAL DB 0
mov  BVAL,es    ;load register into byte location
```

.RADIX — MASM

Function Sets the default radix for integer constants in expressions

Syntax `.RADIX` *`expression`*

Remarks *expression* must evaluate to either 2, 8, 10, or 16. Constants in *expression* are always interpreted as decimal, no matter what the current radix is set to.

Example

```
.RADIX 8      ;set to octal
  DB 77       ;63 decimal
```

RADIX — Ideal, MASM

Function Sets the default radix for integer constants in expressions

Syntax `RADIX`

See also **.RADIX**

RECORD — Ideal, MASM

Function Defines a record that contains bit fields

Syntax Ideal mode:
`RECORD` *`name field`* `[,`*`field`*`]...`

MASM mode:
`name` `RECORD` *`field`* `[,`*`field`*`]...`

Remarks *name* identifies the record so that you can use this name later when allocating memory to contain records with this format. Each *field* describes a group of bits in the record and has the following format:

```
fieldname:width[=expression]
```

fieldname is the name of a field in the record. If you use *fieldname* in an expression, it has the value of the number of bits that the field must be shifted to the right in order to place the low bit of the field in the lowest bit in the byte or word that comprises the record.

width is a constant between 1 and 16 that specifies the number of bits in the field. The total width of all fields in the record cannot exceed 32 bits. If the total number of bits in all fields is 8 or less, the record will occupy 1 byte; if it is between 9 and 16 bits, it will occupy 2 bytes; otherwise, it will occupy 4 bytes.

expression is an optional field that provides a default value for the field; it must be preceded with an equal sign (=). When *name* is used to define storage, this default value will be placed in the field if none is supplied. Any unused bits in the high portion of the record will be initialized to 0.

The first field defined by **RECORD** goes into the most-significant bits of the record with successive fields filling the lower bits. If the total width of all the fields is not exactly 8 or 16 bits, all the fields are shifted to the right so that the lowest bit of the last field occupies the lowest bit of the byte or word that comprises the record.

See also **STRUC**

Example
```
MyRec RECORD val:3=4, MODE:2, SIZE:4
```

REPT — Ideal, MASM

Function Repeats a block of statements

Syntax
```
REPT expression
  statements
ENDM
```

Remarks *expression* must evaluate to a constant and cannot contain any forward-referenced symbol names.

The *statements* within the repeat block are assembled as many times as specified by expression.

REPT can be used both inside and outside of macros.

See also **ENDM, IRP, IRPC**

Example
```
REPT 4
  shl ax,1
ENDM
```

.SALL — MASM

Function Suppresses the listing of all statements in macro expansions

Syntax
```
.SALL
```

Remarks Use **.SALL** to cut down the size of your listing file when you want to see how a macro gets expanded.

See also **.LALL, %MACS, %NOMACS, .XALL**

Example
```
.SALL
 MyMacro 4      ;invoke macro
 add ax,si      ;this line follows MYMACRO in listing
```

SEGMENT — Ideal, MASM

Function Defines a segment with full attribute control

Syntax Ideal mode:
```
SEGMENT name [align] [combine] [use] ['class']
```
MASM mode:
```
name SEGMENT [align] [combine] [use] ['class']
```

Remarks *name* defines the name of a segment. If you have already defined a segment with the same name, this segment is treated as a continuation of the previous one.

You can also use the same segment name in different source files. The linker will combine all segments with the same name into a single segment in the executable program.

align specifies the type of memory boundary where the segment must start. It can be one of the following:

BYTE	Use the next available byte address
WORD	Use the next word-aligned address
DWORD	Use the next doubleword-aligned address
PARA	Use the next paragraph address (16-byte aligned)
PAGE	Use the next page address (256-byte aligned)

PARA is the default alignment type used if you do not specify an align type.

combine specifies how segments from different modules but with the same name will be combined at link time. It can be one of the following:

- **AT** *expression*: Locates the segment at the absolute paragraph address specified by *expression*. *expression* must not contain any forward-referenced symbol names. The linker does not generate any data or code for **AT** segments. You usually use **AT** segments to allow symbolic access to fixed memory locations, such as the display screen or the ROM BIOS data area.
- **COMMON**: Locates this segment and all other segments with the same name at the same address. The length of the resulting common segment is the length of the longest segment.
- **MEMORY**: Concatenates all segments with the same name to form a single contiguous segment. This is the same as the **PUBLIC** combine type.
- **PRIVATE**: Does not combine this segment with any other segments.
- **PUBLIC**: Concatenates all segments with the same name to form a single contiguous segment. The total length of the segment is the sum of all the lengths of the segments with the same name.
- **STACK**: Concatenates all segments with the same name to form a single contiguous segment. The linker initializes the stack segment (SS) register to the beginning of this segment. It initializes the stack pointer (SP) register to the length of this segment, allowing your program to use segments with this combine type as a calling stack, without having to explicitly set the SS and SP registers. The total length of the segment is the sum of all the lengths of the segments with the same name.
- **VIRTUAL**: Defines a special kind of segment which will be treated as a common area and attached to another segment at link time. The **VIRTUAL** segment is assumed to be attached to the enclosing segment. The **VIRTUAL** segment also inherits its attributes from the enclosing segment. The **ASSUME** directive considers a **VIRTUAL** segment to be a part of its parent segment; in all other ways, a **VIRTUAL** segment is treated just like a normal segment. The linker treats virtual segments as a common area that will be combined across modules. This permits static data that comes into many modules from Include files to be shared.

PRIVATE is the default combine type used if you do not specify one.

use specifies the default word size for the segment, and can only be used after enabling the 80386 processor with the **P386** or **P386N** directive. It can be one of the following:

- **USE16:** This is the default segment type when you do not specify a use-segment attribute. A **USE16** segment can contain up to 64K of code and/or data. If you reference 32-bit segments, registers, or constants while in a **USE16** segment, additional instruction bytes will be generated to override the default 16-bit size.

- **USE32:** A **USE32** segment can contain up to 4 Gb (gigabytes) of code and/or data. If you reference 16-bit segments, registers, or constants while in a **USE32** segment, additional instruction bytes will be generated to override the default 32-bit size.

class controls the ordering of segments at link time. Segments with the same class name are loaded into memory together, regardless of the order in which they appear in the source file. You must always enclose the class name in quotes (' or ").

The **ENDS** directive closes the segment opened with the **SEGMENT** directive. You can nest segment directives, but Turbo Assembler treats them as unnested; it simply resumes adding data or code to the original segment when an **ENDS** terminates the nested segment.

See also **CODESEG, DATASEG, GROUP, MODEL**

Example

```
PROG SEGMENT PARA PUBLIC 'CODE'
   • • •
PROG ENDS
```

.SEQ MASM

Function Sets sequential segment-ordering

Syntax `.SEQ`

Remarks **.SEQ** causes the segments to be ordered in the same order in which they were encountered in the source file. By default, Turbo Assembler uses this segment-ordering when it first starts assembling a source file. The **DOSSEG** directive can also affect the ordering of segments.

.SEQ does the same thing as the **/S** command-line option. If you used the **/A** command-line option to force alphabetical segment-ordering, **.SEQ** will override it.

See also **.ALPHA, DOSSEG**

Example

```
  .SEQ
xyz SEGMENT     ;this segment will be first
xyz ENDS
abc SEGMENT
abc ENDS
```

.SFCOND MASM

Function Prevents statements in false conditional blocks from appearing in the listing file

Syntax `.SFCOND`

See also **%CONDS, .LFCOND, %NOCONDS, .TFCOND**

SIZESTR Ideal, MASM51

Function Returns the number of characters in a string

Syntax *name* SIZESTR *string*

Remarks *name* is assigned a numeric value that is the number of characters in a string. A null string <> has a length of zero.

See also **CATSTR, INSTR, SUBSTR**

Example

```
RegList EQU <si di>
RegLen SIZESTR RegList      ;RegLen = 5
```

.STACK MASM

Function Defines the start of the stack segment

Syntax `.STACK [`*size*`]`

Remarks *size* is the number of bytes to reserve for the stack. If you do not supply a size, the **.STACK** directive reserves 1 Kb (1024 bytes).

You usually only need to use **.STACK** if you are writing a standalone assembler program. If you are writing an assembler routine that will be called from a high-level language, that language will normally have set up any stack that is required.

See also **.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, STACK**

Example

```
.STACK  200h    ;allocate 512 byte stack
```

STACK — Ideal, MASM

Function Defines the start of the stack segment

See also **.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

.STARTUP — MASM

Function Generates startup code for the particular model in effect.

Syntax `.STARTUP`

Remarks **.STARTUP** causes startup code to be generated for the particular model in effect at the time. The near label **@Startup** is defined at the beginning of the startup code and the program's **END** directive becomes `END @Startup`. This directive is provided for Microsoft Quick Assembler compatibility.

See also **@Startup, STARTUPCODE**

STARTUPCODE — Ideal, MASM

Function Generates startup code for the particular model in effect.

Syntax `STARTUPCODE`

Remarks **STARTUPCODE** causes startup code to be generated for the particular model in effect at the time. The near label **@Startup** is defined at the beginning of the startup code and the program's **END** directive becomes `END @Startup`. This directive is provided for Microsoft Quick Assembler compatibility.

See also **@Startup, .STARTUP**

STRUC — Ideal, MASM

Function Defines a structure

Syntax Ideal mode:

```
STRUC name
  fields
ENDS [name]
```

MASM mode:

```
name STRUC
  fields
[name] ENDS
```

Remarks The difference in how **STRUC** is handled in Ideal and MASM mode is only in the order of the directive and the structure name on the first line of the definition, and the order of the **ENDS** directive and the optional structure name on the last line of the definition. In Turbo Assembler, you can nest the **STRUC** directive and also combine it with the **UNION** directive.

name identifies the structure, so you can use this name later when allocating memory to contain structures with this format.

fields define the fields that comprise the structure. Each field uses the normal data allocation directives (**DB**, **DW**, and so on) to define its size. *fields* may be named or remain nameless. The field names are like any other symbols in your program—they must be unique. In Ideal mode, the field names do not have to be unique.

You can supply a default value for any field by putting that value after the data allocation directive, exactly as if you were initializing an individual data item. If you do not want to supply a default value, use the **?** indeterminate initialization symbol. When you use the structure name to actually allocate data storage, any fields without a value will be initialized from the default values in the structure definition. If you don't supply a value, *and* there is no default value, **?** will be used.

Be careful when you supply strings as default values; they will be truncated if they are too long to fit in the specified field. If you specify a string that is too short for the field in MASM mode, it will be padded with spaces to fill out the field. When in Ideal mode, the rest of the string from the structure definition will be used. This lets you control how the string will be padded by placing appropriate values in the structure definition.

At any point while declaring the structure members, you may include a nested structure or union definition by using the **STRUC** or **UNION** directive instead of one of the data allocation directives, or you may use the name of a previously defined structure.

When you nest structures or unions using the **STRUC** or **UNION** directive, you still access the members as if the structure only has one level by using a single period (.) structure member operator. When you nest structures by using a previously defined structure name, you use multiple period operators to step down through the structures.

See also **ENDS, UNION**

Example

```
IDEAL
```

```
.MODEL small
DATASEG
STRUC B
  B1  DD   0
  B2  DB   ?
ENDS

STRUC A
  A1  DW    ?                       ;first field
  A2  DD    ?                       ;second field
  BINST     B   <>
  STRUC
     D     DB  "XYZ"
     E     DQ  1.0
  ENDS
ENDS

AINST     A   <>
CINST     A   ?
DINST     A

CODESEG
  mov    al,[AINST.BINST.B2]
  mov    al,[AINST.D]
  mov    ax,[WORD CINST.BINST.B1]
END
```

SUBSTR — Ideal, MASM51

Function Defines a new string as a substring of an existing string

Syntax `name SUBSTR string,position[,size]`

Remarks *name* is assigned a value consisting of characters from *string* starting at *position,* and with a length of *size.* The first character in the *string* is *position* 1. If you do not supply a *size,* all the remaining characters in *string* are returned, starting from *position.*

string may be one of the following:

- a string argument enclosed in angle brackets, like <*abc*>
- a previously defined text macro
- a numeric string substitution starting with percent (%)

See also **CATSTR, INSTR, SIZESTR**

Example

```
N = 0Ah
HEXC   SUBSTR <0123456789ABCDEF>,N + 1,1   ;HEXC = "A"
```

SUBTTL — MASM

Function Sets subtitle in listing file

Syntax `SUBTTL text`

Remarks The subtitle appears at the top of each page, after the name of the source file, and after any title set with the **TITLE** directive.

You may place as many **SUBTTL** directives in your program as you wish. Each directive changes the subtitle that will be placed at the top of the next listing page.

See also **%SUBTTL**

Example

```
SUBTTL Video driver
```

%SUBTTL — Ideal, MASM

Function Sets subtitle in listing file

Syntax `%SUBTTL "text"`

Remarks The subtitle *text* appears at the top of each page, after the name of the source file, and after any title set with the **%TITLE** directive. Make sure that you place the subtitle text between quotes ("").

You may place as many **%SUBTTL** directives in your program as you wish. Each directive changes the subtitle that will be placed at the top of the next listing page.

See also **SUBTTL**

Example

```
%SUBTTL "Output routines"
```

%SYMS — Ideal, MASM

Function Enables symbol table in listing file

Syntax `%SYMS`

Remarks Placing **%SYMS** anywhere in your source file causes the symbol table to appear at the end of the listing file. (The symbol table shows all the symbols you defined in your source file.)

This is the default symbol listing mode used by Turbo Assembler when it starts assembling a source file.

See also **%NOSYMS**

Example
```
%SYMS     ;symbols now appear in listing file
```

%TABSIZE — Ideal, MASM

Function Sets tab column width in the listing file

Syntax `%TABSIZE` *width*

Remarks *width* is the number of columns between tabs in the listing file. The default tab column width is 8 columns.

See also **%BIN, %PAGE, %PCNT, %TEXT**

Example
```
%TABSIZE 4      ;small tab columns
```

%TEXT — Ideal, MASM

Function Sets width of source field in listing file

Syntax `%TEXT` *width*

Remarks *width* is the number of columns to use for source lines in the listing file. If the source line is longer than this field, it will either be truncated or wrapped to the following line, depending on whether you've used **%TRUNC** or **%NOTRUNC**.

See also **%BIN, %DEPTH, %NOTRUNC, %PCNT, %TRUNC**

Example
```
%TEXT 80      ;show 80 columns from source file
```

.TFCOND — MASM

Function Toggles conditional block-listing mode

Syntax `.TFCOND`

Remarks Normally, conditional blocks are not listed by Turbo Assembler, and the first **.TFCOND** encountered enables a listing of conditional blocks. If you use the **/X** command-line option, conditional blocks start off being listed, and the first **.TFCOND** encountered disables listing them. Each time **.TFCOND** appears in the source file, the state of false conditional listing is reversed.

See also **%CONDS, .LFCOND, %NOCONDS, .SFCOND**

TITLE — MASM

Function Sets title in listing file

Syntax `TITLE text`

Remarks The title *text* appears at the top of each page, after the name of the source file and before any subtitle set with the **SUBTTL** directive.

You may only use the **TITLE** directive once in your program.

See also **SUBTTL, %SUBTTL, %TITLE**

Example `TITLE Sort Utility`

%TITLE — Ideal, MASM

Function Sets title in listing file

Syntax `%TITLE "text"`

Remarks The title *text* appears at the top of each page, after the name of the source file and before any subtitle set with the **%SUBTTL** directive. Make sure that you place the title *text* between quotes ("").

You may only use the **%TITLE** directive once in your program.

See also **SUBTTL, %SUBTTL, TITLE**

Example `%TITLE "I/O Library"`

%TRUNC — Ideal, MASM

Function Truncates listing fields that are too long

Syntax `%TRUNC`

Remarks **%TRUNC** reverses the effect of a previous **%NOTRUNC** directive. This directive changes the object-code field and the source-line field so that too-wide fields are truncated and excess information is lost.

See also **%NOTRUNC**

Example
```
%TRUNC
  DD 1,2,3,4,5          ;don't see all fields
```

UDATASEG — Ideal, MASM

Function Defines the start of an uninitialized data segment

See also **.CODE, .CONST, .DATA, DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

UFARDATA — Ideal, MASM

Function Defines the start of an uninitialized far data segment

See also **.CODE , .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

UNION — Ideal, MASM

Function Defines a union

Syntax Ideal mode (disabled by *Quirks*):
```
UNION NAME
  fields
ENDS [name]
```

MASM mode:
```
NAME UNION
  fields
[name] ENDS
```

Remarks The only difference in how **UNION** behaves in Ideal and MASM mode is in the order of the directive and the union name on the first line of the definition, and the order of the **ENDS** directive and the optional union name on the last line of the definition. Turbo Assembler allows you to nest **UNION** and to combine it with **STRUC**.

A **UNION** is just like a **STRUC** except that all its members have an offset of zero (0) from the start of the union. This results in a set of fields that are overlayed, allowing you to refer to the memory area defined by the union with different names and different data sizes. The length of a union is the length of its largest member, not the sum of the lengths of its members as in a structure.

name identifies the union, so you can use this name later when allocating memory to contain unions with this format.

fields define the fields that comprise the union. Each field uses the normal data allocation directives (**DB**, **DW**, etc.) to define its size. The field names are like any other symbols in your program—they must be unique.

You can supply a default value for any field by putting that value after the data allocation directive, exactly as if you were initializing an individual data item. If you don't want to supply a default value, use the **?** indeterminate initialization symbol. When you use the union name to actually allocate data storage, any fields that you don't supply a value for will be initialized from the default values in the structure definition. If you don't supply a value *and* there is no default value, ? will be used.

Be careful when you supply strings as default values; they will be truncated if they are too long to fit in the specified field. If you specify a string that is too short for the field, it will be padded with spaces to fill out the field.

At any point while declaring the union members, you may include a nested structure or union definition by using the **STRUC** or **UNION** directive instead of one of the data-allocation directives. When you nest structures and unions using the **STRUC** or **UNION** directive, you still access the members as if the structure only has one level by using a single period (.) structure member operator. When you nest unions by using a previously defined union name, you use multiple period operators to step down through the structures and unions.

See also **ENDS, UNION**

Example

```
UNION B
  BMEM1   DW  ?
  BMEM2   DB  ?
ENDS
UNION A
  B       DW  ?                      ;first field--offset 0
  C       DD  ?                      ;second field--offset 0
BUNION  B   <>                       ;starts at 0
   STRUC
   D    DB  "XYZ"                    ;at offset 0
   E    DQ   1.0                     ;at offset 1
   ENDS
ENDS
AINST   A   <>                       ;allocate a union of type A
   mov  al,[AINST.BUNION.BMEM1]      ;multiple levels
mov  al,[AINST.D]               ;single level
```

USES — Ideal, MASM

Function Indicates register/data usage for procedures

Syntax `USES item [,item]...`

Remarks **USES** appears within a **PROC/ENDP** pair and indicates which registers or data items you want to have pushed at the beginning of the procedure and which ones you want popped just before the procedure returns.

item can be any register or single-token data item that can be legally **PUSH**ed or **POP**ped. There is a limit of 8 items per procedure.

Notice that you separate *item* names with commas, not with spaces like you do when specifying the *item* as part of the **PROC** directive. You can also specify these *items* on the same line as the **PROC** directive, but this directive makes your procedure declaration easier to read and also allows you to put the **USES** directive inside a macro that you can define to set up your procedure entry and exit.

You must use this directive before the first instruction that actually generates code in your procedure.

Note: **USES** is only available when used with language extensions to a **.MODEL** statement.

See also **ARG, LOCALS, PROC**

Example

```
MyProc PROC
USES  cx,si,di,foo
      mov   cx,10
      mov   foo,cx
rep   movsb
      ret    ;this will pop CX, SI, & DI registers, and the word
MyProc ENDP  ;at location foo
```

WARN — Ideal, MASM

Function Enables a warning message

Syntax `WARN [`*`warnclass`*`]`

Remarks If you specify **WARN** without *warnclass,* all warnings are enabled. If you follow **WARN** with a warning identifier, only that warning is enabled. Each warning message has a three-letter identifier:

ALN	Segment alignment
ASS	Assumes segment is 16-bit
BRK	Brackets needed
ICG	Inefficient code generation
LCO	Location counter overflow
OPI	Open IF conditional
OPP	Open procedure
OPS	Open segment
OVF	Arithmetic overflow
PDC	Pass-dependent construction
PRO	Write-to-memory in protected mode needs CS override
PQK	Assuming constant for [const] warning
RES	Reserved word warning
TPI	Turbo Pascal illegal warning

These are the same identifiers used by the **/W** command-line option.

See also **WARN**

Example

```
NOWARN  OVF         ;disable arithmetic overflow warnings
DW  1000h * 1234h   ;doesn't warn now
```

.XALL — MASM

Function Lists only macro expansions that generate code or data

See also **.LALL, %MACS, %NOMACS, .SALL**

.XCREF MASM

Function Disables cross-reference listing (CREF)

See also **%CREF, .CREF, %NOCREF**

.XLIST MASM

Function Disables output to listing file

See also **%LIST, .LIST, %NOLIST**

A P P E N D I X

A

Turbo Assembler syntax summary

This appendix uses a modified Backus-Naur form (BNF) to summarize the syntax for Turbo Assembler expressions, both in MASM mode and in Ideal mode.

Note: In the following sections, the ellipses (...) mean the same element is repeated as many times as it is found.

Lexical grammar

valid_line

white_space valid_line
punctuation valid_line
number_string valid_line
id_string valid_line
null

white_space

space_char white_space
space_char

space_char

All control characters, characters > 128, ' '

id_string

id_char id_strng2

id_strng2

id_chr2 id_strng2
null

id_char

$, %, _, ?, alphabetic characters

id_chr2

id_chars plus numerics

number_string

num_string
str_string

num_string

digits alphanums
digits '.' digits exp
digits exp ; Only if MASM mode in a DD, DQ, or DT

digits

digit digits
digit

digit

0 through 9

alphanums

digit alphanum
alpha alphanum
null

alpha

alphabetic characters

exp

E + digits
E – digits
E digits
null

str_string

Quoted string, quote enterable by two quotes in a row

punctuation

Everything that is not a space_char, id_char, '"', "'", or digits

The period (.) character is handled differently in MASM mode and Ideal mode. This character is not required in floating-point numbers in MASM mode and also cannot be part of a symbol name in Ideal mode. In MASM mode, it is sometimes the start of a symbol name and sometimes a punctuation character used as the structure member selector.

Here are the rules for the period (.) character:

1. In Ideal mode, it is always treated as punctuation.
2. In MASM mode, it is treated as the first character of an ID in the following cases:

   a. When it is the first character on the line, or in other special cases like **EXTRN** and **PUBLIC** symbols, it gets attached to the following symbol if the character that follows it is an *id_chr2*, as defined in the previous rules.
   b. If it appears other than as the first character on the line, or if the resulting symbol would make a defined symbol, the period gets appended to the start of the symbol following it.

MASM mode expression grammar

mexpr1

'SHORT' mexpr1
'.TYPE' mexpr1
'SMALL' mexpr1 (16-bit offset cast [386 only])
'LARGE' mexpr1 (32-bit offset cast [386 only])
mexpr2

mexpr2

mexpr3 'OR' mexpr3 ...
mexpr3 'XOR' mexpr3 ...
mexpr3

mexpr3

mexpr4 'AND' mexpr4 ...
mexpr4

mexpr4

'NOT' mexpr4
mexpr5

mexpr5

mexpr6 'EQ' mexpr6 ...
mexpr6 'NE' mexpr6 ...
mexpr6 'LT' mexpr6 ...
mexpr6 'LE' mexpr6 ...
mexpr6 'GT' mexpr6 ...
mexpr6 'GE' mexpr6 ...
mexpr6

mexpr6

mexpr7 '+' mexpr7 ...
mexpr7 '-' mexpr7 ...
mexpr7

mexpr7

mexpr8 '' mexpr8 ...*
mexpr8 '/' mexpr8 ...
mexpr8 'MOD' mexpr8 ...
mexpr8 'SHR' mexpr8 ...
mexpr8 'SHL' mexpr8 ...

mexpr8

mexpr8

mexpr9 'PTR' mexpr8
mexpr9
'OFFSET' mexpr8
'SEG' mexpr8
'TYPE' mexpr8
'THIS' mexpr8

mexpr9

mexpr10 ':' mexpr10 ...
mexpr10

mexpr10

'+' mexpr10
'-' mexpr10
mexpr11

mexpr11

'HIGH' mexpr11
'LOW' mexpr11
mexpr12

mexpr12

mexpr13 mexpr13 ... (Implied addition only if '[' or '(' present)
mexpr12 mexpr13 '.' mexpr8

mexpr13

'LENGTH' id
'SIZE' id
'WIDTH' id
'MASK' id
'(' mexpr1 ')'
'[' mexpr1 ']'
id
const

Ideal mode expression grammar

pointer

'SMALL' pointer (16-bit offset cast [386 only])
'LARGE' pointer (32-bit offset cast [386 only])
type 'PTR' pointer
type 'LOW' pointer (Low caste operation)
type 'HIGH' pointer (High caste operation)
type pointer
pointer2

type

'UNKNOWN'
'BYTE'
'WORD'
'DWORD'
'QWORD'
'PWORD'
'TBYTE'
'SHORT'
'NEAR'
'FAR'
struct_id
'TYPE' pointer

pointer2

pointer3 '.' id (Structure item operation)
pointer3

pointer3

expr ':' pointer3
expr

expr

'SYMTYPE' expr (Symbol type operation)
expr2

expr2

expr3 'OR' expr3 ...

expr3 'XOR' expr3 ...
expr3

expr3

expr4 'AND' expr4 ...
expr4

expr4

'NOT' expr4
expr5

expr5

expr6 'EQ' expr6 ...
expr6 'NE' expr6 ...
expr6 'LT' expr6 ...
expr6 'LE' expr6 ...
expr6 'GT' expr6 ...
expr6 'GE' expr6 ...
expr6

expr6

expr7 '+' expr7 ...
expr7 '-' expr7 ...
expr7

expr7

expr8 '' expr8 ...*
expr8 '/' expr8 ...
expr8 'MOD' expr8 ...
expr8 'SHR' expr8 ...
expr8 'SHL' expr8 ...
expr8

expr8

'+' expr8
'-' expr8
expr9

expr9

'HIGH' expr9
'LOW' expr9
expr10

expr10

'OFFSET' pointer
'SEG' pointer
'SIZE' id
'LENGTH' id
'WIDTH' id
'MASK' id
id
const
'(' pointer ')'
'[' pointer '] (Always means 'contents-of ')

A P P E N D I X

B

Compatibility issues

Turbo Assembler in MASM mode is very compatible with MASM version 4.0, and additionally supports all the extensions provided by MASM versions 5.0 and 5.1. However, 100% compatibility is an ideal that can only be approached, since there is no formal specification for the language and different versions of MASM are not even compatible with each other.

For most programs, you will have no problem using Turbo Assembler as a direct replacement for MASM version 4.0 or 5.1. Occasionally, Turbo Assembler will issue a warning or error message where MASM would not, which usually means that MASM has not detected an erroneous statement. For example, MASM accepts

```
abc  EQU [BP+2]
     PUBLIC  abc
```

and generates a nonsense object file. Turbo Assembler correctly detects this and many other questionable constructs.

If you are having trouble assembling a program with Turbo Assembler, you might try using the **QUIRKS** directive; for example,

```
TASM /JQUIRKS MYFILE
```

which may make your program assemble properly. If it does, add **QUIRKS** to the top of your source file. Even better, review this appendix and determine which statement in your source file *needs* the **QUIRKS** directive. Then you can rewrite the line(s) of code so that you don't even have to use **QUIRKS**.

If you're using certain features of MASM version 5.1, you'll need **MASM51** in your source file. These capabilities are discussed later in this appendix.

One- versus two-pass assembly

See Chapter 3 in the User's Guide for a complete discussion of this option.

Normally, Turbo Assembler performs only one pass when assembling code, while MASM performs two. This gives Turbo Assembler a speed advantage, but can introduce minor incompatibilities when forward references and pass-dependent constructions are involved. TASM 2.0 introduces a new command-line option (**/m**) to specify the number of passes desired. For maximum compatibility with MASM, two passes (**/m2**) should be used.

Environment variables

See Chapter 3 in the User's Guide for a discussion of how to do this.

In keeping with the approach used by other Borland language products, Turbo Assembler does not use environment variables to control default options. Instead, you can place default options in a configuration file and then set up different configuration files for different projects.

If you have used the **INCLUDE** or **MASM** environment variables to configure MASM to behave as you wish, you will have to make a configuration file for Turbo Assembler. Any options that you have specified using the **MASM** variable can simply be placed in the configuration file. Any directories that you have specified using the **INCLUDE** variable should be placed in the configuration file using the **/I** command-line option.

Microsoft binary floating-point format

By default older versions of MASM generated floating-point numbers in a format incompatible with the IEEE standard floating-point format. MASM version 5.1 generates IEEE floating-point data by default and has the **.MSFLOAT** directive to specify that the older format be used.

Turbo Assembler does not support the old floating-point format, and therefore does not let you use **.MSFLOAT**.

Turbo Assembler Quirks mode

Some MASM features are so problematic in nature that they weren't included in Turbo Assembler's MASM mode. However, programmers occasionally like to take advantage of some of these "quirky" features. For that reason, Turbo Assembler includes Quirks mode, which emulates these potentially troublesome features of MASM.

You can enable Quirks mode either with the **QUIRKS** keyword in your source file or by using the **/JQUIRKS** command-line option when running Turbo Assembler.

The constructs that follow cause Turbo Assembler to generate error messages in MASM mode, but are accepted in Quirks mode.

Byte move to/ from segment register

MASM does not check the operand size when moving segment registers to and from memory. For example, the following is perfectly acceptable under MASM:

```
SEGVAL  D...  ?
USEFUL  DB    ?
   mov  SEGVAL,es       ;overwrites part of "USEFUL"!
```

This is clearly a programming error that only works because the corruption of *USEFUL* does not affect the program's behavior. Rather than using Quirks mode, redefine *SEGVAL* to be a **DW**, as was presumably intended.

Erroneous near jump to far label or procedure

With MASM, a far jump instruction to a target within the same segment generates a near or a short jump whether or not the target is overridden with **FAR PTR**:

```
CODE SEGMENT
     jmp abc
     jmp FAR PTR abc   ;doesn't generate far JMP
```

```
abc LABEL FAR
CODE ENDS
```

Turbo Assembler normally assembles a far **JMP** instruction when you tell it that the destination is a far pointer. If you want it to behave as MASM does and always treat it as a short or near jump if the destination is in the same segment, you must enable Quirks mode.

Loss of type information with = and EQU directive

Examine the following code fragment:

```
X DW 0
Y  = OFFSET X
   mov ax,Y
```

MASM will generate the instruction **MOV AX,[X]** here, where Turbo Assembler will correctly generate **MOV AX,OFFSET X**. This happens because MASM doesn't correctly save all the information that describes the expression on the right side of the = directive.

This also happens when using the **EQU** directive to define symbols for numeric expressions.

Segment-alignment checking

MASM allows the **ALIGN** directive to specify an alignment that is more stringent than that of the segment in which it is used. For example,

```
CODE SEGMENT WORD
   ALIGN     4      ;segment is only word aligned
CODE ENDS
```

This is a dangerous thing to do, since the linker may undo the effects of the **ALIGN** directive by combining this portion of segment **CODE** with other segments of the same name in other modules. Even then, you can't be guaranteed that the portion of the segment in your module will be aligned on anything better than a word boundary.

You must enable Quirks mode to accept this construct.

Signed immediate arithmetic and logical instructions

MASM version 4.0 only sign-extends immediate operands on arithmetic instructions. When Turbo Assembler is not in Quirks mode, it does sign-extension on immediate operands for logical instructions as well. This results in shorter, faster instructions, but changes the size of code segments containing these constructs. This may cause problems with self-modifying code or any code that knows approximately how long the generated instructions are. The following code shows an instruction that both MASM and Turbo Assembler generate correctly, and another that Turbo Assembler generates correctly and MASM version 4.0 does not:

```
add  ax,-1        ;MASM and Turbo do sign-extend
xor  cx,0FFFFh    ;MASM 4 uses word immediate
```

Here MASM version 4.0 generates the byte sequence 81 F1 FFFF for the **XOR** instruction, and Turbo Assembler generates the shorter but compatible 83 F1 FF.

Masm 5.1 features

Some of the new features introduced with MASM version 5.1 are always available when using Turbo Assembler. Other features must be enabled with the **MASM51** directive. Some features of MASM 5.1 and Turbo Assembler (implemented more powerfully by Turbo Assembler) are discussed in a previous section, "Turbo Assembler Quirks Mode," on page 161.

When Turbo Assembler first starts assembling your source file, it is in MASM mode with MASM 5.1 features disabled. This is like starting your program with the **MASM** and **NOMASM51** directives.

The following MASM 5.1 features are always available:

Each of the extensions listed here is detailed earlier in this book in Chapter 2 or Chapter 3.

- Parameters and local arguments to **PROC** directive
- **.TYPE** operator extensions
- **COMM** directive extension
- **.CODE** sets **CS ASSUME** to current segment
- **.MODEL** directive high-level language support
- List all command-line option (**/LA**)
- Additional debug information with **DW, DD,** and **DF** directives
- **ELSEIF** family of directives

- **@Cpu** and **@WordSize** directives
- **%** expression operator with text macros
- **DW, DD,** and **DF** debug information extensions

The following features are available when you use **MASM51**:

- **SUBSTR, CATSTR, SIZESTR,** and **INSTR** directives
- Line continuation with backslash

These features are only available when you use both **MASM51** and **QUIRKS**:

- Local labels defined with @@ and referred to with **@F** and **@B**
- Redefinition of variables inside **PROC**s; :: definitions
- Extended model **PROC**s are all **PUBLIC**

Masm 5.1/Quirks mode features

Since several features of MASM 5.1 adversely affect some of Turbo Assembler's features, we've provided an alternative through Turbo Assembler that achieves what MASM 5.1 intended. To use these features, you must enable Quirks mode with the **QUIRKS** directive and the MASM 5.1 features with the **MASM51** directive.

Here is a short summary of what is covered under the various operating modes of TASM:

QUIRKS

1. Allows far jumps to be generated as near or short if **CS** assumes agree.
2. Allows all instruction sizes to be determined in a binary operation solely by a register, if present.
3. Destroys **OFFSET,** segment override (and so on) information on = or numeric **EQU** assignments.
4. Forces **EQU** assignments to expressions with **PTR** or : in them to be text.
5. Disables **UNION** directive.
6. Allows **GLOBAL** directive to be overridden.

MASM51

1. Enables *Instr, Catstr, Substr, Sizestr,* and \ line continuations.
2. Makes **EQU**'s to keywords TEXT instead of ALIASes.
3. No longer discards leading whitespace on *%textmacro* in macro arguments.

MASM51 and QUIRKS

Everything listed under **QUIRKS** and **MASM51** in this summary, and the following:

1. Enables @@F and @@B local labels.
2. In extended models, automatically makes **PUBLIC** procedure names.
3. Makes near labels in **PROC**s redefinable in other **PROC**s.
4. Enables :: operator to define symbols that can be reached outside of current **PROC**.
5. Makes "old-style" line continuation characters work the same way as in MASM 5.1 when they appear at the end of a line.

QASM compatibility

Turbo Assembler 2.0 has new and modified directives to support source code for MASM 5.2 (QASM):

- **.STARTUP** and **STARTUPCODE** generate startup code for the particular model in effect at the time. These also define the near label **@Startup** and cause the **END** statement at the end of the module to generate the equivalent of **END @Startup**. Note: Only the **STARTUPCODE** directive is available in IDEAL mode.
- **.MODEL** and **MODEL**: It is now possible to select a third field in the **.MODEL** directive to specify the stack association with **DGROUP: NEARSTACK,** or **FARSTACK**. For example,

  ```
  .MODEL SMALL, C, FARSTACK
  ```

  would specify that the stack not be included in **DGROUP**. This capability is already provided in TASM through the model modifiers of the same name. The additional field is provided only for MASM compatibility.
- **@Model** is a predefined symbol that reflects the model currently in effect: 0 = TINY, 1 = SMALL, 2 = COMPACT, 3 = MEDIUM, 4 = LARGE, 5 = HUGE.

A P P E N D I X C

Turbo Assembler highlights

Chapter 11 in the User's Guide provides you with more details about Ideal mode.

Besides its high compatibility with MASM, Turbo Assembler has a number of enhancements that you can use simultaneously with the typical MASM-style statements. These enhancements can be used both in Ideal mode and in MASM mode.

Here we'll introduce you to each of the enhancements and point you to where more-detailed discussions of each topic can be found in the manual.

Extended command-line syntax

Turbo Assembler has a greatly improved command-line syntax that is a superset of the MASM command-line syntax. You can specify multiple files to assemble by entering each individually or by using wildcards (* and ?). You can also group files so that one set of command-line options applies to one set of files, and another set applies to a second set of files. (For a complete description of Turbo Assembler command-line options, turn to Chapter 3 of the *User's Guide.*)

GLOBAL directive

The **GLOBAL** directive lets you define variables as a cross between an **EXTRN** and a **PUBLIC**. This means you can put **GLOBAL** definitions in a header file that's included in all source modules and then define the data in just one module. This gives you greater flexibility, since you can initialize data defined with the **GLOBAL** directive and you can't with the **COMM** directive. (Chapter 6 in the *User's Guide* shows you how to use **GLOBAL;** Chapter 3 in this book defines **GLOBAL.**)

PUBLICDLL directive

The **PUBLICDLL** directive lets you define program labels and procedures to be dynamic link entry points as well as publicizing them to your other modules, which allows you to build dynamic link libraries in assembly code. (See Chapter 3 in this book for a complete definition of **PUBLICDLL**.)

COMM directive extension

The **COMM** directive has been extended to allow the array element size and the array element count to be selected independently of each other for **FAR** communal variables. (See Chapter 3 in this book for a complete discussion of the **COMM** directive.)

Local symbols

The **LOCALS** and **NOLOCALS** directives control whether symbols that start with two at-signs (@@) are local to a block of code. These two directives are also defined in Chapter 3 of this book. (For more information on local symbols, refer to the section "Local labels" in Chapter 9 of the *User's Guide*.)

Conditional jump extension

The **JUMPS** and **NOJUMPS** control whether conditional jumps get extended into the "opposite sense" condition and a near jump instruction. This lets you have a conditional jump with a destination address further away than the usual –128 to +127 bytes. ("Automatic jump sizing" in Chapter 9 of the *User's Guide* discusses how to use this feature.)

Ideal mode

Turbo Assembler's Ideal mode gives you a new and more rational way to construct expressions and instruction operands. By learning just a few simple rules, you can handle complex instruction operands in a better manner. (Chapter 11 of the *User's Guide* introduces Ideal mode.)

UNION directive/STRUC nesting

Unions are like structures defined with the **STRUC** directive except that all the members have an offset of zero (0) from the start of the structure, effectively "overlaying" all the members.

In Turbo Assembler, you can nest **STRUC** and also combine it with **UNION**. The section entitled "The STRUC directive" in Chapter 9 of the *User's Guide* shows you how to use this directive. Chapter 3 in this book provides a complete definition of both **STRUC** and **UNION**.

EMUL and NOEMUL directives

You can control whether floating-point instructions are emulated or are real coprocessor instructions with the **EMUL** and **NOEMUL** directives. Within a single source file, you can switch back and forth as many times as you wish between emulated and real floating-point instructions.

Explicit segment overrides

The Turbo Assembler lets you explicitly force a segment override to be generated on an instruction by using one of the **SEGCS, SEGDS, SEGES, SEGSS, SEGFS,** or **SEGGS** overrides. They function much like the **REP** and **LOCK** overrides.

The section entitled "Segment override prefixes" in Chapter 9 of the *User's Guide* shows you how to use these overrides.

Constant segments

The Turbo Assembler lets you use a constant value any time that a segment value should be supplied. You can also add a constant value to a segment. For example,

```
     jmp    FAR PTR 0FFFFh:0       ;jump into the ROM BIOS
LOWDATA SEGMENT AT 0
     ASSUME DS:LOWDATA+40h         ;DS points to BIOS data area
     mov    ax,DS:[3FH]            ;read word from BIOS data area
LOWDATA ENDS
```

The section entitled "The SEGMENT directive" in Chapter 9 of the *User's Guide* explains this in more detail.

Extended CALL instruction

The **CALL** instruction has been extended in Turbo Assembler to allow high-level language routines to be called in a language-independent manner. Any **CALL** instruction can now specify a language and an argument list for the routine being called. Turbo Assembler automatically generates the necessary stack setup and cleanup code required to pass the arguments to a high-level routine written in the specified language. (See chapters 7 and 8 in the *User's Guide* for examples of how to use this feature with the Turbo languages.)

Extended PUSH and POP instructions

The **PUSH** and **POP** instructions have been extended in Turbo Assembler to allow more than one argument to appear in a single **PUSH** or **POP** instruction. For example,

```
push ax dx      ;equivalent to PUSH AX then PUSH DX
pop  dx ax      ;equivalent to POP DX then POP AX
```

In addition, the PUSH instruction allows constant arguments even when generating code for the 8086 processor. Such instructions are replaced in the object code by a 10-byte sequence that simulates the 80186/286/386 PUSH immediate value instruction.

Language-specific extensions

The **CALL, COMM, EXTRN, GLOBAL, .MODEL, PROC,** and **PUBLIC** statements have been extended in Turbo Assembler to allow high-level language routines and symbols to be specified and accessed in a language-independent manner. These extensions allow you to write generic code that is automatically modified by Turbo Assembler, according to the rules of the language you specify in a particular statement. For example, you could write a routine that takes several parameters, manipulates them, and then calls a routine in a high-level language before returning to its caller. Normally you would have to tailor such a routine to a particular language's calling conventions, including language-specific code to perform stack setup on procedure entry, correctly access parameters on the stack, set up and clean up the stack when calling the high-level routine, and clean up the stack when the procedure returns to its caller. Any symbol-naming conventions (such as C's requirement that underscores precede all symbol names) would have to be hard-coded into your routine as well. (Chapters 7 and 8 in the *User's Guide* provide you with examples of how to use this feature with the Turbo languages.)

Now, using the extended statements, Turbo Assembler performs all these chores for you. You only need to specify which language conventions to apply when using each statement. This allows you to mix and match calling conventions in each module on a procedure-by-procedure basis. The **PROC** and **.MODEL** statements include further extensions to ease language-independent programming.

Extended LOOP instruction in 386 mode

When you are writing code for the 80386, Turbo Assembler lets you determine explicitly whether the **LOOP** instruction should use the CX or the ECX register as its counter.

The section entitled "New versions of LOOP and JCXZ" in Chapter 10 of the *User's Guide* shows you how to use this instruction.

Extended listing controls

You have much greater control over the format and the content of the listing file with Turbo Assembler. You can control whether **INCLUDE** files are listed, push and pop the listing control state, and control the width of all the fields in the listing, including whether they get truncated or wrap to the next line.

Chapter 5 of the *User's Guide* provides a description of all the options you can use.

Alternate directives

The Turbo Assembler provides alternative keywords for a number of directives, in particular those that start with a period (.). All alternative listing control directives start with a percent sign (%), and all alternative processor control directives start with a *P*.

Refer to Chapter 3 in this book for a complete list of all the directives that Turbo Assembler supports.

Predefined variables

The Turbo Assembler defines a number of variables that have a value you can access from your source files. These include **??date**, **??time**, **??filename**, and **??version**, in addition to the predefined variables supported for MASM 5.0 compatibility.

Take a look at Chapter 1, "Predefined symbols," of this book for a definition of these variables.

Masm 5.0 and 5.1 enhancements

Turbo Assembler has all the extensions Masm 5.0 and 5.1 have over MASM 4.0. If you are not familiar with these extensions, here's a list of where to look for some of the more important topics:

- **80386 Support**: See "The 80386" in Chapter 10 of the *User's Guide.*
- **Simplified Segmentation Directives**: See "Simplified segment directives and 80386 segment types" in Chapter 10 of the *User'Guide.*
- **String Equates**: See "Using equate substitutions" in Chapter 6 of the *User's Guide.*
- **RETF and RETN Instructions**: See "How subroutines work" in Chapter 5 of the *User's Guide.*
- **Communal Variables**: See the **COMM** directive in Chapter 3 in this book.

- **Explicitly Including Library Files:** See the **INCLUDELIB** directive in Chapter 3 of this book.
- **Predefined Variables:** See "Simplified segment directives" in Chapter 5 and also Chapter 9 of the *User's Guide.*

Improved SHL and SHR handling

When you use **SHL** and **SHR** as part of an arithmetic expression, MASM does not permit the shift count to be negative. Turbo Assembler accepts negative shift counts and performs the opposite type of shift. For example, 16 **SHL** –2 is equivalent to 16 **SHR** 2.

Multi-pass capability

Turbo Assembler 2.0 can pass over your source code more than once either for compatibility with some of MASM's pass-dependent constructions or to remove **NOP** instructions that were added to the code because of forward references. This feature is enabled by the command-line switch **/m#**, where # is the maximum number of passes allowed. Turbo Assembler automatically assesses the need to perform extra passes up to the maximum that you specify. (See Chapter 3 in the *User's Guide* for a complete description of this option.)

A P P E N D I X

D
Utilities

Turbo Assembler provides six powerful stand-alone utilities. You can use these stand-alone utilities with your Turbo Assembler files, as well as with your other modules.

These highly useful adjuncts to Turbo Assembler are

- MAKE (including the TOUCH utility; the stand-alone program manager MAKE)
- TLINK (the Turbo Linker)
- TLIB (the Turbo Librarian)
- GREP (a file-search utility)
- OBJXREF (an object module cross-referencer)
- TCREF (a cross-reference utility)

This appendix documents MAKE, TLINK, and TLIB; GREP, OBJXREF, and TCREF are documented in text files available to you on disk.

MAKE: The program manager

Borland's command-line MAKE, derived from the UNIX program of the same name, helps you keep the executable versions of your programs current. Many programs consist of many source files, each of which may need to pass through preprocessors, assemblers, compilers, and other utilities before being combined with the rest of the program. Forgetting to recompile a module that has been changed—or that depends on something you've changed—can lead to frustrating bugs. On the other hand, recompiling *everything* just to be safe can be a tremendous waste of time.

MAKE solves this problem. You provide MAKE with a description of how the source and object files of your program are processed to produced the finished product. MAKE looks at that description and at the date stamps on your files, then does what's necessary to create an up-to-date version. During this process, MAKE may invoke many different compilers, assemblers, linkers, and utilities, but it never does more than is necessary to update the finished program.

MAKE's usefulness extends beyond programming applications. You can use MAKE to control any process that involves selecting files by name and processing them to produce a finished product. Some common uses include text processing, automatic backups, sorting files by extension into other directories, and cleaning temporary files out of your directory.

How MAKE works

MAKE keeps your program up-to-date by performing the following tasks:

- Reads a special file (called a makefile) that you have created. This file tells MAKE which .OBJ and library files have to be linked in order to create your executable file, and which source and header files have to be compiled to create each .OBJ file.
- Checks the time and date of each .OBJ file against the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the source file must be recompiled.
- Calls the compiler to recompile the source file.
- Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of your executable file.
- If any of the .OBJ files is later than the .EXE file, calls the linker to recreate the .EXE file.

Caution! MAKE relies completely upon the timestamp DOS places on each file. This means that, in order for MAKE to do its job, your system's time and date *must* be set correctly. If you own an AT or a PS/2, make sure that the battery is in good repair. Weak batteries can cause your system's clock to lose track of the date and time, and MAKE will no longer work as it should.

The original IBM PC and most compatibles didn't come with a built-in clock or calendar. If your system falls into this category, and you haven't added a clock, be sure to set the system time and date correctly (using the DOS DATE and TIME commands) each time you start your machine.

Starting MAKE

To use MAKE, type `make` at the DOS prompt. MAKE then looks for a file specifically named MAKEFILE. If MAKE can't find MAKEFILE, it looks for MAKEFILE.MAK; if it can't find that or BUILTINS.MAK (described later), it halts with an error message.

What if you want to use a file with a name other than MAKEFILE or MAKEFILE.MAK? You give MAKE the file (**-f**) option, like this:

```
make -fmyfile.mak
```

The general syntax for MAKE is

make *option option ... target target ...*

where *option* is a MAKE option (discussed later), and *target* is the name of a target file to be handled by explicit rules.

Here are the MAKE syntax rules:

MAKE stops if any command it has executed is aborted via a Control-Break. Thus, a Control-Break stops the currently executing command and MAKE as well.

- The word *make* is followed by a space, then a list of make options.
- Each make option must be separated from its adjacent options by a space. Options can be placed in any order, and any number of these options can be entered (as long as there is room in the command line). All options that do not specify a string (-*s* or –*a*, for example) can have an optional – or + after them. This specifies whether you wish to turn the option off () or on (+).
- After the list of make options comes a space, then an optional list of targets.
- Each target must also be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, recompiling their constituents as necessary.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they will be built as necessary.

The BUILTINS.MAK file

You will often find that there are MAKE macros and rules that you use again and again. There are three ways of handling them.

- First, you can put them in every makefile you create.
- Second, you can put them all in one file and use the **!include** directive in each makefile you create. (See page 196 for more on directives.)
- Third, you can put them all in a BUILTINS.MAK file.

Each time you run MAKE, it looks for a BUILTINS.MAK file; however, there is no requirement that any BUILTINS.MAK file exist. If MAKE finds a BUILTINS.MAK file, it interprets that file first. If MAKE cannot find a BUILTINS.MAK file, it proceeds directly to interpreting MAKEFILE (or whatever makefile you specify).

The first place MAKE searches for BUILTINS.MAK is the current directory. If it's not there, *and* if you're running under DOS 3.0 or higher, MAKE then searches the directory from which MAKE.EXE was invoked. You should place the BUILTINS.MAK file in the same directory as the MAKE.EXE file.

MAKE always searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. The two files have identical syntax rules.

MAKE also searches for any **!include** files (see page 197 for more on this MAKE directive) in the current directory. If you use the **–I** (include) option, it will also search in the directory specified with the **–I** option.

Command-line options

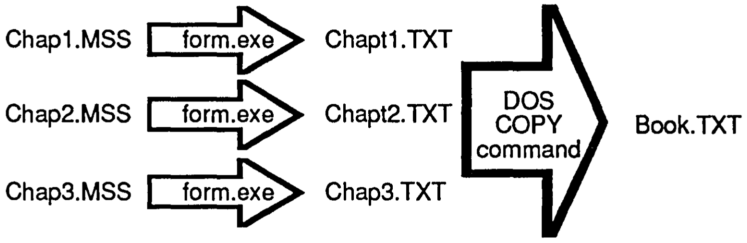
Here's a complete list of MAKE's command-line options. Note that case (upper or lower) *is* significant; the option **–d** is not a valid substitute for **–D**.

Option	What it does
–? or **–h**	Prints a help message.
–a	Causes an automatic dependency check on .OBJ files.
–B	Builds all targets regardless of file dates.
–D*identifier*	Defines the named identifier to the string consisting of the single character 1 (one).
–D*iden=string*	Defines the named identifier *iden* to the string after the equal sign. The string cannot contain any spaces or tabs.
–f*filename*	Uses *filename* as the MAKE file. If *filename* does not exist and no extension is given, tries FILENAME.MAK.
–i	Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. This is equivalent to putting '–' in front of all commands in the MAKEFILE (described below).
–I*directory*	Searches for include files in the indicated directory (as well as in the current directory).
–K	Keeps (does not erase) temporary files created by MAKE. All temporary files have the form MAKE####.$$$, where #### ranges from 0000 to 9999.
–n	Prints the commands but does not actually perform them. This is useful for debugging a makefile.
–s	Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed.
–S	Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules.
–U*identifier*	Undefines any previous definitions of the named identifier.
–W	Makes the current specified non-string options (like **–s** and **–a**) to be the default (writes them to MAKE.EXE). The default options are displayed by **–?** or **–h** with plus signs following.

A simple use of MAKE

For our first example, let's look at a simple use of MAKE that doesn't involve programming. Suppose you're writing a book, and decide to keep each chapter of the manuscript in a separate file. (Let's assume, for the purposes of this example, that your book is quite short: It has three chapters, in the files CHAP1.MSS, CHAP2.MSS, and CHAP3.MSS.) To produce a current draft of the book, you run each chapter through a formatting program, called FORM.EXE, then use the DOS COPY command to concatenate the outputs to make a single file containing the draft, like this:

MAKE can also backup files, pull files out of different subdirectories, and even automatically run your programs should the data files they use be modified.



Like programming, writing a book requires a lot of concentration. As you write, you might modify one or more of the manuscript files, but you don't want to break your concentration by noting which ones you've changed. On the other hand, you don't want to forget to pass any of the files you've changed through the formatter before combining it with the others, or you won't have a fully updated draft of your book!

One inelegant and time-consuming way to solve this problem is to create a batch file that reformats every one of the manuscript files. It might contain the following commands:

```
FORM CHAP1.MSS
FORM CHAP2.MSS
FORM CHAP3.MSS
COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

Running this batch file would always produce an updated version of your book. However, suppose that, over time, your book got bigger and one day contained 15 chapters. The process of reformatting the entire book might become intolerably long.

MAKE can come to the rescue in this sort of situation. All you need to do is create a file, usually named MAKEFILE, which tells MAKE what files BOOK.TXT depends on and how to process them. This file will contain rules that explain how to rebuild BOOK.TXT when some of the files it depends on have been changed.

In this example, the first rule in your makefile might be

```
book.txt: chap1.txt chap2.txt chap3.txt
          copy /a chap1.txt+chap2.txt+chap3.txt book.txt
```

What does this mean? The first line (the one that begins with `book.txt:`) says that BOOK.TXT depends on the formatted text of each of the three chapters. If any of the files that BOOK.TXT depends on are newer than BOOK.TXT itself, MAKE must rebuild BOOK.TXT by executing the COPY command on the subsequent line.

This one rule doesn't tell the whole story, though. Each of the chapter files depends on a manuscript (.MSS) file. If any of the CHAP?.TXT files is newer than the corresponding .MSS file, the .MSS file must be recreated. Thus, you need to add more rules to the makefile as follows:

```
chap1.txt: chap1.mss
           form chap1.mss

chap2.txt: chap2.mss
           form chap2.mss

chap3.txt: chap3.mss
           form chap3.mss
```

Each of these rules shows how to format one of the chapters, if necessary, from the original manuscript file.

MAKE understands that it must update the files that another file depends on before it attempts to update that file. Thus, if you change CHAP3.MSS, MAKE is smart enough to reformat Chapter 3 before combining the .TXT files to create BOOK.TXT.

We can add one more refinement to this simple example. The three rules look very much the same—in fact, they're identical except for the last character of each file name. And, it's pretty easy to forget to add a new rule each time you start a new chapter. To solve these problems, MAKE allows you to create something called an *implicit rule*, which shows how to make one type of file from another, based on the files' extensions. In this case, you can replace the three rules for the chapters with one implicit rule:

```
.mss.txt:
          form $*.mss
```

This rule says, in effect, "If you need to make a file out of an .MSS file to make things current, here's how to do it." (You'll still have to update the first rule—the one that makes BOOK.TXT, so that MAKE knows to concatenate the new chapters into the output file. This rule, and others following, make use of a *macro*. See page 191 for an in-depth discussion of macros.)

Once you have the makefile in place, all you need to do to create an up-to-date draft of the book is type a single command at the DOS prompt: MAKE.

Creating makefiles

Creating a program from an assortment of program files, include files, header files, object files, and so on, is very similar to the text-processing example you just looked at. The main difference is that the commands you'll use at each step of the process will invoke preprocessors, compilers, assemblers, and linkers instead of a text formatter and the DOS COPY command. Let's explore how to create makefiles—the files that tell MAKE how to do these things—in greater depth.

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up-to-date. You can create as many makefiles as you want and name them whatever you want; MAKEFILE is just the default name that MAKE looks for if you don't specify a makefile when you run MAKE.

You create a makefile with any ASCII text editor, such as Sprint, MicroStar, SideKick, or your favorite editor. All rules, definitions, and directives end at the end of a line. If a line is too long, you can continue it to the next line by placing a backslash (\) as the last character on the line.

Use whitespace (blanks and tabs) to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Components of a makefile

Creating a makefile is almost like writing a program, with definitions, commands, and directives. These are the constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules
- macro definitions
- directives:
  - file inclusion directives
  - conditional execution directives
  - error detection directives

- macro undefinition directives

Let's look at each of these in more detail.

Comments

Comments begin with a pound sign (#) character; the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere; they don't have to start in a particular column.

A backslash will *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. If the backslash precedes the #, it is no longer the last character on the line; if it follows the #, then it is part of the comment itself.

Here are some examples of comments in a makefile:

Explicit and implicit rules are discussed following the section on commands.

```
# Makefile for my book

# This file updates the file BOOK.TXT each time I
# change one of the .MSS files

# Explicit rule to make BOOK.TXT from six chapters. Note the
# continuation lines.

book.txt: chap1.txt chap2.txt chap3.txt\
          chap4.txt chap5.txt chap6.txt
          copy /a chap1.txt+chap2.txt+chap3.txt+chap4.txt+\
                  chap5.txt+chap6.txt book.txt

# Implicit rule to format individual chapters
.mss.txt:
          form $*.mss
```

Command lists

Both explicit and implicit rules (discussed later) can have lists of commands. This section describes how these commands are processed by MAKE.

Commands in a command list take the form

[prefix ...] command_body

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

Prefixes

The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at-sign (@) or a hyphen (–) followed immediately by a number.

Prefix	What it does
@	Prevents MAKE from displaying the command before executing it. The display is hidden even if the **–s** option is not given on the MAKE command line. This prefix applies only to the command on which it appears.
–num	Affects how MAKE treats exit codes. If a number (*num*) is provided, then MAKE aborts processing only if the exit status exceeds the number given. In this example, MAKE aborts only if the exit status exceeds 4: `-4 myprog sample.x` If no *–num* prefix is given and the status is nonzero, MAKE stops and deletes the current target file.
–	With a hyphen but no number, MAKE will not check the exit status at all. Regardless of the exit status, MAKE continues.

Command body

The command body is treated exactly as if it were entered as a line to COMMAND.COM, with the exception that pipes (|) are not supported.

In addition to the **<**, **>**, and **>>** redirection operators, MAKE adds the **<<** and **&&** operators. These operators create a file on the fly for input to a command. The **<<** operator creates a temporary file and redirects the command's standard input so that it comes from the created file. If you have a program that accepted input from *stdin*, the command

```
   myprog <<!
 This is a test
 !
```

would create a temporary file containing the string "This is a test \n", redirecting it to be the sole input to *myprog*. The exclamation point (!) is a delimiter in this example; you can use any character except # or \ as a delimiter for the file. The first line containing the delimiter character as its first character ends the file. The rest of the line following the delimiter character (in this case, an exclamation point) is considered part of the preceding command.

The **&&** operator is similar to **<<**. It creates a temporary file, but instead of making the file the standard input to the command, the **&&** operator is replaced with the temporary file's name. This is useful when you want MAKE to create a file that's going to be used as input to a program. The following example creates a "response file" for TLINK:

Macros are covered starting on page 191.

```
MYPROG.EXE: $(MYOBJS)
    tlink /c @&&!
COS $(MYOBJS)
$*
$*
$(MYLIBS) EMU.LIB MATHS.LIB CS.LIB
!
```

Note that macros (indicated by **$** signs) are expanded when the file is created. The **$*** is replaced with the name of the file being built, without the extension, and **$**(MYOBJS) and **$**(MYLIBS) are replaced with the values of the macros MYOBJS and MYLIBS. Thus, TLINK might see a file that looks like this:

```
COS a.obj b.obj c.obj d.obj
MYPROG
MYPROG
w.lib x.lib y.lib z.lib EMU.LIB MATHS.LIB CS.LIB
```

All temporary files are deleted unless you use the **–K** command-line option. Use the **–K** option to "debug" your temporary files if they don't appear to be working correctly.

Batching programs

MAKE allows utilities that can operate on a list of files to be batched. Suppose, for example, that MAKE needs to submit several assembler files to Turbo Assembler for processing. MAKE could run TASM.EXE once for each file, but it's much more efficient to invoke TASM.EXE with a list of all the files to be assembled on the command line. This saves the overhead of reloading Turbo Assembler each time.

MAKE's batching feature lets you accumulate the names of files to be processed by a command, combine them into a list, and invoke that command only once for the whole list.

To cause MAKE to batch commands, you use braces in the command line:

```
command command-line... { batch-item } ...rest-of-command
```

This command syntax delays the execution of the command until MAKE determines what command (if any) it has to invoke next. If the next command is identical except for what's in the braces ({...}), the two commands will be combined by appending the parts of the commands that appeared inside the braces.

Here's an example that shows how batching works. Suppose MAKE decides to invoke the following three commands in succession:

```
tasm {file1.asm }
tasm {file2.asm }
tasm {file3.asm}
```

Rather than invoking Turbo Assembler three times, MAKE issues the single command

```
tasm file1.asm file2.asm file3.asm
```

Note that the spaces at the ends of the file names in braces are essential to keep them apart, since the contents of the braces in each command are concatenated exactly as-is.

Here's an example that uses an implicit rule. Suppose your makefile had an implicit rule to compile assembler programs to .OBJ files:

```
.asm.obj:
        TASM -c {$< }
```

As MAKE uses the implicit rule on each assembler file, it expands the macro **$<** into the actual name of the file and adds that name to the list of files to compile. (Again, note the space inside the braces to keep the names separate.) The list grows until MAKE discovers that it has to run a program other than TASM, or if there are no more commands to process, or if MAKE runs out of room on the command line. If this happens, MAKE puts as much as it can on one command line and puts the rest on the next command line. When the list is done, MAKE invokes TASM (with the **–c** option) on the whole list of files at once.

Executing DOS commands

MAKE executes the DOS "internal" commands listed here by invoking a copy of COMMAND.COM to perform them:

break	del	prompt	time
cd	dir	rd	type
chdir	echo	rem	ver
cls	erase	ren	verify
copy	md	rename	vol
ctty	mkdir	rmdir	
date	path	set	

MAKE searches for any other command name using the DOS search algorithm:

1. MAKE first searches for the file in the current directory, then searches each directory in the path.
2. In each directory, MAKE first searches for a file of the specified name with the extension .COM. If it doesn't find it, it searches for the same file name with an .EXE extension. Failing that, MAKE searches for a file by the specified name with a .BAT extension.
3. If MAKE finds a .BAT file, it invokes a copy of COMMAND.COM to execute the batch file.

If you supply a file-name extension in the command line, MAKE searches only for that extension. Here are some examples:

- This command causes COMMAND.COM to change the current directory to C:\include:

  ```
  cd c:\include
  ```

- MAKE uses the full search algorithm in searching for the appropriate files to perform this command:

  ```
  tlink lib\c0s x y,z,z,lib\cs
  ```

- MAKE searches for this file using only the .COM extension:

  ```
  myprog.com geo.xyz
  ```

- MAKE executes this command using the explicit file name provided:

  ```
  c:\myprogs\fil.exe -r
  ```

Explicit rules

The first rule in the last example is an explicit rule—a rule that specifies complete file names explicitly. Explicit rules take the form

target [*target*] ...: [*source source* ...]
    [*command*]
    [*command*]
    ...

where *target* is the file to be updated, *source* is a file on which *target* depends, and *command* is any valid DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more source files, and an optional list of commands to be performed. Target and source file names listed in explicit rules can contain normal DOS drive and directory specifications; they can also contain wildcards.

➡ *Syntax here is important.*

- *target* must be at the start of a line (in column 1).
- The *source* file(s) must be preceded by at least one space or tab, after the colon.
- Each *command* must be indented, (must be preceded by at least one blank or tab). As mentioned before, the backslash can be used as a continuation character if the list of source files or a given command is too long for one line.

Both the source files and the commands are optional; it is possible to have an explicit rule consisting only of *target [target ...]* followed by a colon.

The idea behind an explicit rule is that the command or commands listed will create or update *target,* usually using the *source* files. When MAKE encounters an explicit rule, it first checks to see if any of the *source* files are themselves target files elsewhere in the makefile. If so, MAKE evaluates that rule first.

Once all the *source* files have been created or updated based on other rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *source.* If any *source* has been modified more recently than *target,* the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or to the end of the file. Blank lines are ignored.

Special considerations

An explicit rule with no command lines following it is treated a little differently than an explicit rule with command lines.

- If an explicit rule includes commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on two sets of files: the files given in the explicit rule, and any file that matches an implicit rule for the target(s). This lets you specify a dependency to be handled by an implicit rule.

Examples

Here's one example of an explicit rule:

```
prog.exe: myprog.asm prog2.asm include\stdio.inc
  tasm /t myprog.asm   # Recompile myprog using Turbo Assembler
  tasm /t prog2.asm # Recompile prog2 using Turbo Assembler
  tlink myprog prog2, prog
```

Here are some better examples:

1.
   ```
   prog.exe: myprog.obj prog2.obj
   tlink myprog prog2, prog
   ```

2.
   ```
   myprog.obj: myprog.asm include\stdio.inc
   tasm myprog.asm
   ```

3.
   ```
   prog2.obj: prog2.c include\stdio.inc
   tasm prog2.asm
   ```

The first rule states that

1. PROG.EXE depends on MYPROG.ASM, PROG2.ASM, and STDIO.INC.

   and

2. If any of the three change, PROG.EXE can be rebuilt by the series of commands given.

However, this may create unnecessary work because, even if only MYPROG.ASM changes, PROG2.ASM will still be recompiled. This is because all of the commands under a rule will be executed as soon as that rule's target is out-of-date.

The three examples show three good ways to build the same assembler program. Only the modules affected by a change are rebuilt. If PROG2.ASM is changed, it's the only one recompiled; the same holds true for MYPROG.ASM. But if the include file STDIO.INC is changed, both are recompiled. (The link step is always done if any of the source is changed.)

Automatic dependency checking

Turbo Assembler works with Borland's MAKE to provide automatic dependency checking for include files. TASM produces .OBJ files that tell MAKE what include files were used to create those .OBJ files. MAKE's **–a** command-line option checks this information and makes sure that everything is up-to-date.

When MAKE does an automatic dependency check, it reads the include files' names, times, and dates from the .OBJ file. If any include files have been modified, MAKE causes the .OBJ file to be recompiled. For example, consider the following makefile:

```
.asm.obj:
   tasm -c $*
```

Now assume that the following source file, called MYFILE.ASM, has been compiled with TASM:

```
#include <stdio.inc>
#include "dcl.inc"

void myfile() {}
```

If you then invoke MAKE with the following command line

```
make  -a  myfile.obj
```

it checks the time and date of MYFILE.ASM, and also of STDIO.INC and DCL.INC.

Implicit rules

MAKE allows you to define *implicit* rules as well as explicit ones. Implicit rules are generalizations of explicit rules; they apply to all files that have certain identifying extensions.

Here's an example that illustrates the relationship between the two rules. Consider this explicit rule from the preceding example:

```
myprog.obj: myprog.asm include\stdio.inc
        tasm -c myprog.asm
```

This rule is typical because it follows a general principle: An .OBJ file is dependent on the .asm file with the same file name and is created by executing TASM. In fact, you might have a makefile where you have several (or even several dozen) explicit rules following this same format.

By rewriting the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.asm.obj:
   tasm -c $<
```

This rule means "Any file ending with .OBJ depends on the file with the same name that ends in .asm." The .OBJ file is created with the second line of the rule, where `$<` represents the file's name with the source (.asm) extension. (The symbol `$<` is a special macro. Macros are discussed starting on page 191. The `$<` macro will be replaced by the full name of the appropriate .asm source file each time the command executes.)

Here's the syntax for an implicit rule:

.source_extension.target_extension:
   [*command*]
   [*command*]
   ...

As before, the commands are optional and must be indented.

source_extension (which must begin with its period in column 1) is the extension of the source file; that is, it applies to any file having the format

   fname.source_extension

Likewise, the *target_extension* refers to the file

   fname.target_extension

where *fname* is the same for both files. In other words, this implicit rule replaces all explicit rules having the format

fname.target_extension: *fname.source_extension*
    [*command*]
    [*command*]
    ...)

for any *fname*.

Note MAKE uses implicit rules if it can't find any explicit rules for a given target, or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is found with the same name as the target, but with the mentioned source extension.

For example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.asm.obj:
   tasm -c $<
```

If you had an assembler program named RATIO.ASM that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
tasm -c ratio.asm
```

which, of course, does the compile step necessary to create RATIO.OBJ.

MAKE also uses implicit rules if you give it an explicit rule with no commands. Suppose you had the following implicit rule at the start of your makefile:

```
.asm.obj:
   tasm -c $<
```

You could then remove the command from the rule:

```
myprog.obj: myprog.asm include\stdio.inc
            tasm -c myprog.asm
```

and it would execute exactly as before.

If you're using Turbo Assembler and you enable automatic dependency checking in MAKE, you can remove all the rules that have .OBJ files as targets. With automatic dependency checking enabled and implicit rules, the three-rule assembler example shown in the section on explicit rules becomes

```
.asm.obj:
        tasm -c $<

prog.exe: myprog.obj prog2.obj
          tlink lib\c0s myprog prog2, prog, , lib\cs
```

You can write several implicit rules with the same target extension. If more than one implicit rule exists for a given target extension, the rules are checked in the order in which they appear in the makefile, until a match is found for the source extension, or until MAKE has checked all applicable rules.

MAKE uses the first implicit rule that involves a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule, up to the next line that begins without whitespace or to the end of the file, are considered to be part of the command list for the rule.

Macros

Often, you'll find yourself using certain commands, file names, or options again and again in your makefile. For instance, if you're writing an assembler program that uses the medium memory model, all your TASM commands will use the switch **–mm**, which means to compile to the medium memory model. But, suppose you wanted to switch to the large memory model? You could go through and change all the **–mm** options to **–ml**. Or, you could define a macro.

A *macro* is a name that represents some string of characters. A macro definition gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
MODEL = m
```

This line defines the macro `MODEL`, which is now equivalent to the string `m`. Using this macro, you could write each command to invoke TASM to look something like this:

```
tasm -c -m$(MODEL) myprog.c
```

When you run MAKE, each macro (in this case, `$(MODEL)`) is replaced with its expansion text (here, **m**). The command that's actually executed would be

```
tasm -c -mm myprog.asm
```

Now, changing memory models is easy. If you change the first line to

```
MODEL = l
```

you've changed all the commands to use the large memory model. In fact, if you leave out the first line altogether, you can specify which memory model you want each time you run MAKE, using the **–D** (define) command-line option:

```
make -DMODEL = l
```

This tells MAKE to treat **MODEL** as a macro with the expansion text *l*.

Defining macros

Macro definitions take the form

macro_name = expansion text

where *macro_name* is the name of the macro. *macro_name* should be a string of letters and digits with no whitespace in it, although you can have whitespace between *macro_name* and the equal sign (=). The *expansion text* is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by newline.

If *macro_name* has previously been defined, either by a macro definition in the makefile or by the **–D** option on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macro names **model**, **Model**, and **MODEL** are all different.

Using macros

You invoke macros in your makefile using this format

$(*macro_name*)

You need the parentheses for all invocations, even if the macro name is just one character long (with the exception of the predefined macros). This construct—`$(macro_name)`—is known as a *macro invocation.*

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Special considerations

Macros in macros: Macros cannot be invoked on the left side (*macro_name*) of a macro definition. They can be used on the right side (*expansion text*), but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

See page 196 for information on directives.

Macros in directives: Macro invocations are expanded immediately in **!if** and **!elif** directives. If the macro being invoked in an **!if** or **!elif** directive is not currently defined, it is expanded to the value 0 (FALSE).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

Predefined macros

MAKE comes with several special macros built in: **$d, $*, $<, $:, $.,** and **$&**. The first is a test to see if a macro name is defined; it's used in the conditional directives **!if** and **!elif**. The others are file name macros, used in explicit and implicit rules. In addition, the current DOS environment strings (the strings you can view and set using the DOS SET command) are automatically loaded as macros. Finally, MAKE defines two macros: **__MSDOS__**, defined to be 1 (one); and **__MAKE__**, defined to be MAKE's version in hexadecimal (for this version, 0x0300).

Macro	What it does
$d	Defined as a test macro
$*	Base file name macro with path
$<	Full file name macro with path
$:	Path only macro
$.	Full file name macro, no path
$&	Base file name macro, no path

Defined Test Macro ($d) The defined test macro (**$d**) expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in **!if** and **!elif** directives.

For example, suppose you want to modify your makefile so that if you don't specify a memory model, it'll use the medium one. You could put this at the start of your makefile:

```
!if !$d(MODEL)    # if MODEL is not defined
MODEL=m           # define it to m (MEDIUM)
!endif
```

If you then invoke MAKE with the command line

```
make -DMODEL=l
```

then **MODEL** is defined as *l*. If, however, you just invoke MAKE by itself,

```
make
```

then **MODEL** is defined as *m*, your "default" memory model.

File name macros

The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built.

Base file name macro ($*): The base file name macro is allowed in the commands for an explicit or an implicit rule. This macro (**$***) expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.ASM
$* expands to A:\P\TESTFILE
```

For example, you could modify this explicit rule

```
prog.exe: myprog.obj prog2.obj
          tlink lib\c0s myprog prog2, prog, , lib\cs
```

to look like this:

```
prog.exe: myprog.obj prog2.obj
          tlink lib\c0s myprog prog2, $*, , lib\cs
```

When the command in this rule is executed, the macro **$*** is replaced by the target file name (without extension), *prog*. For implicit rules, this macro is very useful.

For example, an implicit rule for TASM might look like this:

```
.asm.obj:
        tasm -c $*
```

Full file name macro ($<): The full file name macro (**$<**) is also used in the commands for an explicit or implicit rule. In an explicit rule, **$<** expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.ASM
$< expands to A:\P\TESTFILE.ASM
```

For example, the rule

```
mylib.obj: mylib.asm
           copy $< \oldobjs
           tasm -c $*
```

copies MYLIB.OBJ to the directory \OLDOBJS before compiling MYLIB.ASM.

In an implicit rule, **$<** takes on the file name plus the source extension. For example, the implicit rule

```
.asm.obj:
        tasm -c $*.asm
```

produces exactly the same result as

```
.asm.obj:
        tasm -c $<
```

because the extension of the target file name *must* be .asm.

File-name path macro ($:): This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.ASM
$: expands to A:\P\
```

File-name and extension macro ($.): This macro expands to the file name, with an extension but without the path name, like this:

```
File name is A:\P\TESTFILE.ASM
$. expands to TESTFILE.ASM
```

File name only macro ($&): This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.ASM
$& expands to TESTFILE
```

Directives

Turbo Assembler's MAKE allows something that other versions of MAKE don't: directives similar to those allowed in C, assembler, and Turbo Pascal. You can use these directives to perform a variety of useful and powerful actions. Some directives in a makefile begin with an exclamation point (!) as the first character of the line. Others begin with a period. Here is the complete list of MAKE directives:

Table D.1 MAKE directives

.AUTODEPEND	Turns on autodependency checking.
!ELIF	Conditional execution.
!ELSE	Conditional execution.
!ENDIF	Conditional execution.
!ERROR	Causes MAKE to stop and print an error message.
!IF	Conditional execution.
.IGNORE	Tells MAKE to ignore return value of a command.
!INCLUDE	Specifies a file to include in the makefile.
.NOAUTODEPEND	Turns off autodependency checking.
.NOIGNORE	Turns off **.ignore.**
.NOSILENT	Tells MAKE to print commands before executing them.
.NOSWAP	Tells MAKE to not swap itself in and out of memory.
.PATH.EXT	Gives MAKE a path to search for files with extension .EXT.
.SILENT	Tells MAKE to not print commands before executing them.
.SWAP	Tells MAKE to swap itself in and out of memory.
!UNDEF	Causes the definition for a specified macro to be forgotten.

Dot directives

Each of the following directives has a corresponding command-line option, but takes precedence over that option. For example, if you invoke MAKE like this:

```
make -a
```

but the makefile has a .NOAUTODEPEND directive, then autodependency checking will be off.

.AUTODEPEND and .NOAUTODEPEND turn on or off autodependency checking. It corresponds with the –a command-line option.

.IGNORE and .NOIGNORE tell MAKE to ignore the return value of a command, much like placing the prefix – in front of it (described earlier). They corresponds with the **–i** command-line option.

.SILENT and .NOSILENT tell MAKE whether or not to print commands before executing them. They corresponds with the **–s** command-line option.

.SWAP and .NOSWAP tell MAKE to swap itself out of memory. They corresponds with the **–S** option.

.PATH.extension

This directive, placed in a makefile, tells MAKE where to look for files of the given extension. For example, if the following is in a makefile:

```
.PATH.asm = C:\TASMCODE

.asm.obj:
     tasm $*

tmp.exe: tmp.obj
  tasm tmp.obj
```

MAKE will look for TMP.ASM, the implied source file for TMP.OBJ, in C:\TASMCODE instead of the current directory.

The .PATH is also a macro that has the value of the path. The following is an example of the use of .PATH. The source files are contained in one directory, the .OBJ files in another, and all the .EXE files in the current directory.

```
.PATH.asm   = C:\TASMCODE
.PATH.obj = C:\OBJS

.asm.obj:
     tasm -o$(.PATH.obj)\$& $<

.obj.exe:
     tasm -e$&.exe $<

tmp.exe: tmp.obj
```

File-inclusion directive

A file-inclusion directive (**!include**) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

!include *"filename"*

You can nest these directives to any depth. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file MODEL.MAC that contained the following:

```
!if !$d(MODEL)
MODEL=m
!endif
```

You could use this conditional macro definition in any makefile by including the directive

```
!include "MODEL.MAC"
```

When MAKE encounters **!include**, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional execution directives

Conditional execution directives (**!if**, **!elif**, **!else**, and **!endif**) give you a measure of flexibility in constructing makefiles. Rules and macros can be made conditional, so that a command-line macro definition (using the **–D** option) can enable or disable sections of the makefile.

The format of these directives parallels those in C, assembly language, and Turbo Pascal:

```
!if expression
[ lines ]
!endif

!if expression
[ lines ]
!else
[ lines ]
!endif

!if expression
[ lines ]
!elif expression
[ lines ]
!endif
```

Note

[lines] can be any of the following statement types:

- macro_definition
- explicit_rule
- implicit_rule
- include_directive
- if_group
- error_directive
- undef_directive

The conditional directives form a group, with at least an **!if** directive beginning the group and an **!endif** directive closing the group.

- One **!else** directive can appear in the group.
- **!elif** directives can appear between the **!if** and any **!else** directives.
- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested to any depth.

Any rules, commands, or directives must be complete within a single source file.

All **!if** directives must have matching **!endif** directives within the same source file. Thus the following include file is illegal, regardless of what's in any file that might include it, because it doesn't have a matching **!endif** directive:

```
!if $(FILE_COUNT) > 5
   some rules
!else
   other rules
<end-of-file>
```

Expressions allowed in conditional directives

Expressions are allowed in an **!if** or an **!elif** directive; they use an assembler-like syntax. The expression is evaluated as a simple 32-bit signed integer.

You can enter numbers as decimal, octal, or hexadecimal constants. For example, these are legal constants in a MAKE expression:

```
4536   # decimal constant
0677   # octal constant (distinguished by leading 0)
0x23aF # hexadecimal constant (distinguished by leading 0x)
```

An expression can use any of the following operators:

Operator	Operation
Unary operators	
-	negation (unary minus)
~	bit complement (inverts all bits)

!	logical NOT (yields 0 if operand is nonzero, 1 otherwise)

Binary operators

+	addition
–	subtraction
*	multiplication
/	division
%	remainder
>>	right shift
<<	left shift
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR (XOR)
&&	logical AND
||	logical OR
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
==	equality
!=	inequality

Ternary operator

? :	The operand before the ? is treated as a test. If the value of the first operand is nonzero, then the second operand (the part between the ? and :) is the result. If the value of the first operand is zero, the value of the result is the value of the third operand (the part after the :).

Parentheses can be used to group operands in an expression. In the absence of parentheses, all the unary operators take precedence over binary operators. The binary operators have the same precedences as they do in assembler, and are listed here in order of decreasing precedence.

* / %	multiplicative operators
+ –	additive operators
<< >>	bitwise shift operators
<= >=	relational operators
< >	relational operators
= !=	relational operators
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND

|| logical OR

Operators of equal precedence are executed from left to right, except for nested ternary operators (**?:**), which are executed right to left.

Since this many layers of operator precedence can be confusing, we recommend that you use parentheses liberally in your expressions.

You can invoke macros within an expression; the special macro **$d()** is recognized. After all macros have been expanded, the expression must have proper syntax.

Error directive

The error directive (**!error**) causes MAKE to stop and print a fatal diagnostic containing the text after **!error**. It takes the format

!error *any_text*

This directive is designed to be included in conditional directives to allow a user-defined error condition to abort MAKE. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(MODEL)
# if MODEL is not defined
!error MODEL not defined
!endif
```

If you reach this spot without having defined **MODEL**, then MAKE stops with this error message:

```
Fatal makefile 4: Error directive: MODEL not defined
```

Macro undefinition directive

The macro "undefinition" directive (**!undef**) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is

!undef *macro_name*

MAKE error messages

MAKE diagnostic messages fall into two classes: errors and fatal errors.

- When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart the compilation.

- Errors indicate some sort of syntax or semantic error in the source makefile.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

In manual	What you'll see onscreen
argument(s)	The command-line or other argument
expression	An expression
filename	A file name (with or without extension)
line number	A line number
message	A message string

The error messages are listed in ASCII alphabetic order; messages beginning with symbols come first. Since messages that begin with one of the variables just listed cannot be alphabetized by what you will actually see when you receive such a message, all such messages have been placed at the beginning of each error message list.

For example, if you have tried to link a file named NOEXIT.ASM, you might receive the following actual message:

```
noexit does not exist--don't know how to make it
```

In order to look this error message up, you would need to find

***filename* does not exist—don't know how to make it**

at the beginning of the list of error messages.

If the variable occurs later in the text of the error message (for example, "Illegal character in constant expression: *expression*"), you can find the explanation of the message in correct alphabetical order; in this case, under *I*.

Fatal error messages

***filename* does not exist – don't know how to make it**

There's a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.

Circular dependency exists in makefile

The makefile indicates that a file needs to be up-to-date BEFORE it can be built. Take, for example, the explicit rules:

file*a*: file*b*

file*b*: file*c*
file*c*: file*a*

This implies that file*a* depends on file*b*, which depends on file*c*, and file*c* depends on file*a*. This is illegal, since a file cannot depend on itself, indirectly or directly.

Error directive: *message*
MAKE has processed an **#error** directive in the source file, and the text of the directive is displayed in the message.

Incorrect command-line argument: *argument*
You've used incorrect command-line arguments.

No terminator specified for in-line file operator
The makefile contains either the **&&** or **<<** command-line operators to start an in-line file, but the file is not terminated.

Not enough memory
All your working storage has been exhausted. You should perform your make on a machine with more memory. If you already have 640K in your machine, you may have to simplify the source file, or unload some memory-resident programs.

Unable to execute command
A command failed to execute; this may be because the command file could not be found, or because it was misspelled, or (less likely) because the command itself exists but has been corrupted.

Unable to open makefile
The current directory does not contain a file named MAKEFILE, and there is no MAKEFILE.MAK.

Unable to redirect input or output
MAKE was unable to open the temporary files necessary to redirect input or output. If you are on a network, make sure you have rights to the current directory.

Errors

Bad file name format in include statement
Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.

Bad undef statement syntax

An **!undef** statement must contain a single identifier and nothing else as the body of the statement.

Character constant too long

Character constants can be only one or two characters long.

Command arguments too long

The arguments to a command were more than the 127-character limit imposed by DOS.

Command syntax error

This message occurs if

- The first rule line of the makefile contained any leading whitespace.
- An implicit rule did not consist of *.ext.ext:*.
- An explicit rule did not contain a name before the : character.
- A macro definition did not contain a name before the = character.

Command too long

The length of a command has exceeded 128 characters. You might wish to use a response file.

Division by zero

A divide or remainder in an **!if** statement has a zero divisor.

Expression syntax error in !if statement

The expression in an **!if** statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

File name too long

The file name in an **!include** directive is too long for the compiler to process. File names in DOS can be no longer than 64 characters.

If statement too long

An **If** statement has exceeded 4,096 characters.

Illegal character in constant expression X

MAKE encountered some character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.

Illegal octal digit
An octal constant was found containing a digit of 8 or 9.

Macro expansion too long
A macro cannot expand to more than 4,096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

Misplaced elif statement
An **!elif** directive is missing a matching **!if** directive.

Misplaced else statement
There's an **!else** directive without any matching **!if** directive.

Misplaced endif statement
There's an **!endif** directive without any matching **!if** directive.

No file name ending
The file name in an include statement is missing the correct closing quote or angle bracket.

Redefinition of target *filename*
The named file occurs on the left side of more than one explicit rule.

Rule line too long
An implicit or explicit rule was longer than 4,096 characters.

Unable to open include file *filename*
The named file cannot be found. This can also be caused if an include file included itself. Check whether the named file exists.

Unexpected end of file in conditional started on line *line number*
The source file ended before MAKE encountered an **!endif**. The **!endif** was either missing or misspelled.

Unknown preprocessor statement
A ! character was encountered at the beginning of a line, and the statement name following was not **error, undef, if, elif, include, else,** or **endif.**

TLIB: The Turbo Librarian

TLIB is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

The libraries included with Turbo Assembler were built with TLIB. You can use TLIB to build your own libraries, or to modify your own libraries, libraries furnished by other programmers, or commercial libraries you have purchased. You can use TLIB to

- *create* a new library from a group of object modules
- *add* object modules or other libraries to an existing library
- *remove* object modules from an existing library
- *replace* object modules from an existing library
- *extract* object modules from an existing library
- *list* the contents of a new or existing library

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension.

TLIB can also create (and include in the library file) an extended dictionary, which may be used to speed up linking. See the section on the **/E** option (page 210) for details.

Although TLIB is not essential to creating executable programs with Turbo Assembler, it is a useful programmer's productivity tool. You will find TLIB indispensable for large development projects. If you work with object module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

The advantages of using object module libraries

When you program in assembler, you often create a collection of useful assembler directives. You are likely to split those directives into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. If you always include all the source files, on the other hand, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of assembler directives. When you link your program with a library, the linker scans the library and automatically selects only those modules needed for the current program. In addition, a library consumes less disk space than a collection of object module files, especially if each of the object files is small. A library also speeds up the action of the linker, because it only opens a single file, instead of one file for each object module.

The components of a TLIB command line

Run TLIB by typing a TLIB command line at the DOS prompt. To get a summary of TLIB's usage, just type `TLIB` and press *Enter.*

The TLIB command line takes the following general form, where items listed in square brackets ([*like this*]) are optional:

tlib *libname* [/C] [/E] [/Psize] [*operations*] [, *listfile*]

Component	Description
tlib	The command name that invokes TLIB. **Note:** If the named library does not exist and there are *add* operations, TLIB creates the library.
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a *libname*. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both TASM and TC's project-make facility require the .LIB extension in order to recognize library files.
/C	The case-sensitive flag. This option is not normally used; see page 211 for a detailed explanation.
/E	Create extended dictionary; see page 210 for a detailed explanation.
/Psize	Set the library page size to *size*; see page 210 for a detailed explanation.
operations	The list of operations TLIB performs. Operations may appear in any order. If you only want to examine the contents of the library, don't give any operations.
listfile	The name of the file listing library contents. The *listfile* name (if given) must be preceded by a comma. If you do not give a file name, no listing is produced. The listing is an alphabetical list of each module, followed by an alphabetical list of each public symbol defined in that module. The default extension for the *listfile* is .LST.

You can direct the listing to the screen by using the *listfile* name CON, or to the printer by using the name PRN.

The following sections provide details about using TLIB. For examples of how to use TLIB, refer to the "Examples" section on page 211.

The operation list

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character *action symbol* followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to the DOS-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

Replacing a module means first removing it, then adding the replacement module.

File and module names

When TLIB adds an object file to a library, the file is simply called a *module*. TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you only need to supply the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB operations

TLIB recognizes three action symbols (–, +, *), which you can use singly or combined in pairs for a total of five distinct operations. For operations that use a pair of characters, the order of the characters in not important. The action symbols and what they do are listed here:

To create a library, add modules to a library that does not yet exist.

You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.

Action symbol	Name	Description
+	**Add**	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	**Remove**	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation only needs a module name. TLIB allows you to enter a full path name with drive and extension included, but ignores everything except the module name.
*	**Extract**	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.
-* *-	**Extract & Remove**	TLIB copies the named module to the corresponding file name and then removes it from the library. This is just a shorthand for an *extract* followed by a *remove* operation.
-+ +-	**Replace**	TLIB replaces the named module with the corresponding file. This is just shorthand for a *remove* followed by an *add* operation.

Using response files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using *response files*. A response file is simply an ASCII text file that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one DOS command line.

To use a response file *pathname*, specify `@pathname` at any position on the TLIB command line.

- More than one line of text can make up a response file; you use the "and" character (**&**) at the end of a line to indicate that another line follows.
- You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest.
- You can use more than one response file in a single TLIB command line.

See "Examples" for a sample response file and a TLIB command line incorporating it.

Creating an extended dictionary: The /E option

To speed up linking with large library files (such as the standard Cx.LIB library), you can direct TLIB to create an *extended dictionary* and append it to the library file. This dictionary contains, in a very compact form, information that is not included in the standard library dictionary. This information enables TLINK to process library files faster, especially when they are located on a floppy disk or a slow hard disk. All the libraries on the Turbo Assembler distribution disks contain the extended dictionary.

To create an extended dictionary for a library that is being modified, use the **/E** option when you invoke TLIB to add, remove, or replace modules in the library. To create an extended dictionary for an existing library that you don't want to modify, use the **/E** option and ask TLIB to remove a nonexistent module from the library. TLIB will display a warning that the specified module was not found in the library, but it will also create an extended dictionary for the specified library. For example, enter

```
tlib /E mylib -bogus
```

Setting the page size: The /P option

Every DOS library file contains a dictionary (which appears at the end of the .LIB file, following all of the object modules). For each module in the library, this dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library—it cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, the page size must be increased using the **/P** option; the page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), on the average 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use **/P** with the next available higher page size.

Advanced operation: The /C option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add to the library a module that would cause a duplicate symbol, TLIB displays a message and won't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates.

If you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should ***not*** *use the* ***/C*** *option.*

It may seem odd that, without the **/C** option, TLIB rejects symbols that differ only in case. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case. Such linkers, for example, will treat *stars*, *Stars*, and *STARS* as the same identifier. TLINK, on the other hand, has no problem distinguishing uppercase and lowercase symbols, and it will properly accept a library containing symbols that differ only in case. In this example, then, Turbo Assembler would treat *stars*, *Stars*, and *STARS* as three separate identifiers. As long as you use the library only with TLINK, you can use the TLIB **/C** option without any problems.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB.

1. To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

   ```
   tlib mylib +x +y +z
   ```

2. To create a library as in #1 and get a listing in MYLIB.LST too, type

   ```
   tlib mylib +x +y +z, mylib.lst
   ```

3. To get a listing in CS.LST of an existing library CS.LIB, type

   ```
   tlib cs, cs.lst
   ```

4. To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

   ```
   tlib mylib -+x +a -z
   ```

5. To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type

   ```
   tlib mylib *y, mylib.lst
   ```

6. To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

   First create a text file, ALPHA.RSP, with

   ```
   +a.obj +b.obj +c.obj &
         +d.obj +e.obj +f.obj &
         +g.obj
   ```

   Then use the TLIB command, which produces a listing file named ALPHA.LST:

   ```
   tlib alpha @alpha.rsp, alpha.lst
   ```

Turbo Link

The new TLINK has more features, handles much larger programs, and is quite fast.

Turbo Link (TLINK) is an extremely fast and compact linker; it's invoked as a separate program and can also be used as a standalone linker. This section describes how to use TLINK as a standalone linker.

Invoking TLINK

You can invoke TLINK at the DOS command line by typing `tlink` with or without parameters. When it is invoked without parameters, TLINK displays a summary of parameters and options that looks like this:

In addition to the slash, you can also use a hyphen to precede TLINK's commands. So, for example, /m and -m are equivalent.

```
Turbo Link  Version 3.0 Copyright (c) 1987, 1990 Borland International
Syntax: TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m = map file with publics
         /x = no map file at all
         /i = initialize all segments
         /l = include source line numbers
         /s = detailed map of segments
         /n = no default libraries
         /d = warn if duplicate symbols in libraries
         /c = lower case significant in symbols
         /3 = enable 32-bit processing
         /v = include full symbolic debug information
         /e = ignore Extended Dictionary
         /t = create COM file
```

In TLINK's summary display, the line

```
The syntax is: TLINK objfiles, exefile, mapfile, libfiles
```

specifies that you supply file names *in the given order*, separating the file *types* with commas.

For example, if you supply the command line

```
tlink /c mainline wd ln tx,fin,mfin,lib\comm lib\support
```

TLINK will interpret it to mean that

- case is significant during linking (**/c**).
- the .OBJ files to be linked are MAINLINE.OBJ, WD.OBJ, LN.OBJ, and TX.OBJ.
- the executable program name will be FIN.EXE.
- the map file is MFIN.MAP.
- the library files to be linked in are COMM.LIB and SUPPORT.LIB, both of which are in subdirectory LIB.

TLINK appends extensions to file names that have none:

- .OBJ for object files
- .EXE for executable files
- .MAP for map files
- .LIB for library files

If no .EXE file name is specified, TLINK derives the name of the executable file by appending .EXE to the first object file name listed. If, for example, you had not specified FIN as the .EXE file name in the previous example, TLINK would have created MAINLINE.EXE as your executable file.

When you use the **/t** option, the executable file extension defaults to .COM rather than .EXE.

TLINK always generates a map file, unless you explicitly direct it not to by including the **/x** option on the command line.

- If you give the **/m** option, the map file will include a list of public symbols.
- If you give the **/s** option, the map file will include a detailed segment map.

These are the rules TLINK follows when determining the name of the map file.

- If you don't specify any .MAP files, TLINK derives the map file name by adding a .MAP extension to the .EXE file name. (You can give the .EXE file name on the command line or in the response file; if no .EXE name is given, TLINK will derive it from the name of the first .OBJ file.)
- If you specify a map file name in the command line (or in the response file), TLINK adds the .MAP extension to the given name.

Even if you specify a map file name, if you use the **/x** option, TLINK won't create any map files at all.

Using response files

TLINK lets you supply the various parameters on the command line, in a response file, or in any combination of the two.

A response file is just a text file that contains the options and/or file names that you would usually type in after the name TLINK on your command line.

Unlike the command line, however, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the previous command-line example as a response file, FINRESP, like this:

```
/c mainline wd+
   ln tx,fin
   mfin
   lib\comm lib\support
```

You would then enter your TLINK command as

```
tlink @finresp
```

Note that you must precede the file name with an "at" character (@) to indicate that the next name is a response file.

Alternately, you might break your link command into multiple response files. For example, you can break the previous command line into the following two response files:

File name	Contents

LISTOBJS	`mainline+`
	`wd+`
	`ln tx`

LISTLIBS	`lib\comm+` `lib\support`

You would then enter the TLINK command as

```
tlink /c @listobjs,fin,mfin,@listlibs
```

TLINK options

TLINK options can occur anywhere on the command line. The options consist of a slash (/), a hyphen (-), or the DOS switch character, followed by the option-specifying character (*m, x, i, l, s, n, d, c, 3, v, e,* or *t*). (The DOS switch character is / by default. You can change it by using an INT 21H call.)

If you have more than one option, spaces are not significant (**/m/c** is the same as **/m /c**), and you can have them appear in different places on the command line. The following sections describe each of the options.

/x, /m, /s options

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error messages produced during the link.

If you want to create a more complete map, the **/m** option will add a list of public symbols to the map file, sorted alphabetically as well as in increasing address order. This kind of map file is useful in debugging. Many debuggers can use the list of public symbols to allow you to refer to symbolic addresses when you are debugging.

The **/s** option creates a map file with segments, public symbols and the program start address just like the **/m** option did, but also adds a detailed segment map. The following is an example of a detailed segment map:

Figure D.1
Detailed map of segments

```
Address    Length  Class   Segment Name  Group      Module   Alignment/
           (Bytes)                                           Combining

0000:0000  0E5B   C=CODE  S=SYMB_TEXT   G=(none)   M=SYMB.C  ACBP=28
00E5:000B  2735   C=CODE  S=QUAL_TEXT   G=(none)   M=QUAL.C  ACBP=28
0359:0000  002B   C=CODE  S=SCOPY_TEXT  G=(none)   M=SCOPY   ACBP=28
035B:000B  003A   C=CODE  S=LRSH_TEXT   G=(none)   M=LRSH    ACBP=20
035F:0005  0083   C=CODE  S=PADA_TEXT   G=(none)   M=PADA    ACBP=20
0367:0008  005B   C=CODE  S=PADD_TEXT   G=(none)   M=PADD    ACBP=20
036D:0003  0025   C=CODE  S=PSBP_TEXT   G=(none)   M=PSBP    ACBP=20
036F:0008  05CE   C=CODE  S=BRK_TEXT    G=(none)   M=BRK     ACBP=28
03CC:0006  066F   C=CODE  S=FLOAT_TEXT  G=(none)   M=FLOAT   ACBP=20
0433:0006  000B   C=DATA  S=_DATA       G=DGROUP   M=SYMB.C  ACBP=48
0433:0012  00D3   C=DATA  S=_DATA       G=DGROUP   M=QUAL.C  ACBP=48
0433:00E6  000E   C=DATA  S=_DATA       G=DGROUP   M=BRK     ACBP=48
0442:0004  0004   C=BSS   S=_BSS        G=DGROUP   M=SYMB.C  ACBP=48
0442:0008  0002   C=BSS   S=_BSS        G=DGROUP   M=QUAL.C  ACBP=48
0442:000A  000E   C=BSS   S=_BSS        G=DGROUP   M=BRK     ACBP=48
```

For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Most of the information in the detailed segment map is self-explanatory, except for the ACBP field.

The ACBP field encodes the A (*alignment*), C (*combining*), and B (*big*) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (alignment)	00	An absolute segment
	20	A byte-aligned segment
	40	A word-aligned segment
	60	A paragraph-aligned segment
	80	A page-aligned segment
	A0	An unnamed absolute portion of storage
The C field (combination)	00	May not be combined
	08	A public combining segment
The B field (big)	00	Segment less than 64K
	02	Segment exactly 64K

/l (line numbers)

The **/l** option creates a section in the .MAP file for source code line numbers. To use it, you must have created the .OBJ files by compiling with the **-y** (Line numbers...On) or -v (Debug information) option. If you tell TLINK to create no map at all (using the **/x** option), this option will have no effect.

/i (uninitialized trailing segments)

The **/i** option causes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This option is not normally necessary.

/n (ignore default libraries)

The **/n** option causes the linker to ignore default libraries specified by some compilers. You'll need this option if the default libraries are in another directory, because TLINK does not support searching for libraries. You may want to use this option when linking modules written in another language.

/c (case sensitivity)

The **/c** option forces the case to be significant in public and external symbols. For example, by default, TLINK regards *cloud, Cloud,* and *CLOUD* as equal; the **/c** option makes them different.

/d (duplicate symbols)

Normally, TLINK will not warn you if a symbol appears in more than one library file. If the symbol must be included in the program, TLINK will use the copy of that symbol in the first file on the command line in which it is found. Since this is a commonly used feature, TLINK does not normally warn about the duplicate symbols. The following hypothetical situation illustrates how you might want to use this feature.

Suppose you have two libraries: one called SUPPORT.LIB, and a supplemental one called DEBUGSUP.LIB. Suppose also that DEBUGSUP.LIB contains duplicates of some of the routines in SUPPORT.LIB (but the duplicate routines in DEBUGSUP.LIB include slightly different functionality, such as debugging versions of the routines). If you include DEBUGSUP.LIB *first* in the link command, you will get the debugging routines and *not* the routines in SUPPORT.LIB.

If you are not using this feature or are not sure which routines are duplicated, you may include the **/d** option. TLINK will list all symbols duplicated in libraries, even if those symbols are not going to be used in the program.

Given this option, TLINK will also warn about symbols that appear both in an .OBJ and a .LIB file. In this case, since the symbol that appears in the first (left-most) file listed on the command line is the one linked in, the symbol in the .OBJ file is the one that will be used.

With Turbo Assembler, the distributed libraries you would use in any given link command do not contain any duplicated symbols. So while EMU.LIB and FP87.LIB (or CS.LIB and CL.LIB) obviously have duplicate symbols, they would never rightfully be used together in a single link. There are no symbols duplicated between EMU.LIB, MATHS.LIB, and CS.LIB, for example.

/e (extended dictionary)

The library files that are shipped with Turbo Assembler all contain an *extended dictionary* with information that enables TLINK to link faster with those libraries. This extended dictionary can also be added to any other library file using the **/E** option with TLIB (see the section on TLIB starting on page 206). The **/e** option disables the use of this dictionary.

Although linking with libraries that contain an extended dictionary is faster, you might want to use the **/e** switch if you have a program that needs slightly more memory to link when an extended dictionary is used.

Unless you use **/e**, TLINK will ignore any debugging information contained in a library that has an extended dictionary.

/t (tiny model .COM file)

If you compile your file in the tiny memory model and link it with this option toggled on, TLINK will generate a .COM file instead of the usual .EXE file. Also, when you use **/t**, the default extension for the executable file is .COM.

Note: .COM files may not exceed 64K in size, cannot have any segment-relative fix ups, cannot define a stack segment, and must have a starting address equal to 0:100H. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100H.

/v option

The **/v** option directs TLINK to include debugging information in the executable file. If this option is found anywhere on the command line, debugging information will be included for all modules that contain debugging information. You can use the **/v+** and **/v–** options to selectively enable or disable inclusion of debugging information on a module-by-module basis. For example, this command

```
tlink mod1 /v+ mod2 mod3 /v- mod4
```

includes debugging information for modules *mod2* and *mod3*, but not for *mod1* and *mod4*.

/3 (80386 32-bit code)

The **/3** option should be used when one or more of the object modules linked has been produced by TASM or a compatible assembler, and contains 32-bit code for the 80386 processor. This option increases the memory requirements of TLINK and slows down linking, so it should be used only when necessary.

Restrictions

There is only one serious restriction to TLINK; TLINK does not generate Windows or OS/2 .EXE files.

Previous restrictions that no longer apply:

- Common variables are now supported.
- Segments that are of the same name and class that are uncombinable are now accepted. They aren't combined, and they appear separately in the map file.
- Any Microsoft code can now be linked with TLINK.

TLINK can of course be used with Turbo C (both the integrated environment and command-line versions), TASM, Turbo Prolog, and other compilers.

Error messages

TLINK has three types of errors: fatal errors, nonfatal errors, and warnings.

- A fatal error causes TLINK to stop immediately; the .EXE file is deleted.
- A nonfatal error does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

In manual	What you'll see on screen
filename	A file name (with or without extension)
group	A group name
module	A module name
segment	A segment name
symbol	A symbol name
XXXXh	A 4-digit hexadecimal number, followed by *h*

The error messages are listed in ASCII alphabetic order; messages beginning with symbols come first. Since messages that begin with one of the variables just listed cannot be alphabetized by what you will actually see when you receive such a message, all such messages have been placed at the beginning of each error message list.

For example, if you have tried to link a file named NOEXIT.OBJ, you might receive the following actual message:

```
noexit.obj: bad object file
```

In order to look this error message up, you would need to find

***filename*: bad object file**

at the beginning of the list of error messages.

If the variable occurs later in the text of the error message (for example, "Invalid segment definition in module *module*"), you can find the message in correct alphabetical order; in this case, under *I*.

Fatal errors

When fatal errors happen, TLINK stops and deletes the .EXE file.

***filename*: bad object file**
An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was pressed.

***filename*: unable to open file**
This occurs if the named file does not exist or is misspelled.

***group*: group exceeds 64K**
This message will occur if a group exceeds 64K bytes when the segments of the group are combined.

***module*: bad .obj file, virtual LEDATA with no VirDef**

This message indicates an error in the debug information in an object file. Either the compiler generated a bad object file or the object file has been damaged.

***module*: bad .obj file, virtual reference with no VirDef**

This message indicates an error in the debug information in an object file. Either the compiler generated a bad object file or the object file has been damaged.

***segment*: segment exceeds 64K**

This message will occur if too much data was defined for a given data or code segment, when segments of the same name in different source files are combined.

symbol* in module *module1* conflicts with module *module2

This error message can result from a conflict between two symbols (either public or communal). This usually means that a symbol with different attributes is defined in two modules.

Bad character in parameters

One of the following characters was encountered in the command line or in a response file:

```
"  *  <  =  >  ?  [  ]  |
```

or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.

Cannot generate COM file : data below initial CS:IP defined

This error results from trying to generate data or code below the starting address (usually 100) of a .COM file. Be sure that the starting address is set to 100 by using the (ORG 100H) instruction. This error message should not occur for programs written in a high-level language. If it does, ensure that the correct startup (C0) object modules are being linked in.

Cannot generate COM file : invalid initial entry point address

You used the **/t** option, but the program starting address is not equal to 100H, which is required with .COM files.

Cannot generate COM file : program exceeds 64K

You used the **/t** option, but the total program size exceeds the .COM file limit.

Cannot generate COM file : segment-relocatable items present

You used the **/t** option, but the program contains segment-relative fixups, which are not allowed with .COM files.

Cannot generate COM file : stack segment present

You used the /t option, but the program declares a stack segment, which is not allowed with .COM files.

Invalid entry point offset

This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.

Invalid group definition in module *module*

This error can occur if an attempt was made to assign a segment to more than one group. It can also result from a malformed GRPDEF record in an .OBJ file. This latter case could result from custom-built .OBJ files or a bug in the translator used to generate the .OBJ file.

Invalid initial stack offset

This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.

Invalid segment definition in module *module*

This message will generally occur only if a compiler produced a flawed object file. If this occurs in a file created by Turbo C, recompile the file. If the problem persists, contact Borland.

Invalid switch in parameter block

This results from a logic error in TLINK or in the integrated development environment.

msdos error, ax = XXXXh

This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes where this error could occur are read, write, seek, and close.

Not enough memory

There was not enough memory to complete the link process. Try removing any terminate-and-stay-resident applications currently loaded, or reduce the size of any RAM disk currently active. Then run TLINK again.

Not enough memory to link

TLINK requires at least 145K free memory in order to run. Try to free up some memory by releasing terminate-and-stay-resident programs.

Relocation offset overflow in module *module*

This error only occurs for 32-bit object modules and indicates a relocation (segment fixup) offset greater than the DOS limit of 64K.

Relocation table full

The file being linked contains more base fixups than the standard DOS relocation table can hold (base fixups are created mostly by calls to far functions).

Symbol limit exceeded

This message results from the linker's internal symbol table overflowing. This usually means that the programs being linked have exceeded the linker's capacity for public or external symbols.

Table limit exceeded

This error message results from exceeding some internal limitation of TLINK. Try reducing the size of your application before retrying a link. Getting this error message usually means you've exceeded the linker's capacity for public or external symbols. Too many distinct segments can also cause this error message.

32-bit record encountered in module *module* : use "/3" option

This message occurs when an object file that contains 80386 32-bit records is encountered, and the **/3** option has not been used. Simply restart TLINK with the **/3** option.

Unknown option

A forward slash character (/) or hyphen (-) was encountered on the command line or in a response file without being followed by one of the allowed options.

Write failed, disk full?

This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.

Nonfatal errors

TLINK has three nonfatal errors. As mentioned, when a nonfatal error occurs, the .EXE and .MAP files are not deleted. *These errors are treated as fatal errors under the integrated environment.*

Fixup overflow in module *module*, at *segname*:xxxxh, target = *symbol*

This indicates an incorrect data or code reference in an object file that TLINK must fix up at link time.

This message is most often caused by a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named module, so look in the source file of that module for the offending reference.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or a high-level language besides Turbo C, there may be other possible causes for this message. Even in Turbo C, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Out of memory

This error is a catchall for running into a TLINK limit on memory usage. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together.

Undefined symbol <symbol> in module <module>

The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check to make sure the symbol is spelled correctly. You will usually see this error from TLINK for Turbo C symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs.

Warnings

TLINK has five warning messages.

Warning: *symbol* defined in module *module* is duplicated in module *module*

The named symbol is defined in each of the named modules. This could happen if a given object file is named twice in the command line.

Warning: no stack

This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Turbo C, or for any application program that will be converted to a .COM file. For other programs, this indicates an error.

Warning: no stub for fixup in *module* at *segment*:xxxxh
This error occurs when the target for a fixup is in an overlay segment, but no stub segment is found for the segment. This is usually the result of not making public a symbol in an overlay that is referenced from the same module.

Warning: segment *segment* is in two groups: *group1* and *group2*
The linker found conflicting claims by the two named groups.

TOUCH

There are times when you want to force a particular target file to be recompiled or rebuilt, even though no changes have been made to its sources. One way to do this is to use the TOUCH utility. TOUCH changes the date and time of one or more files to the current date and time, making it "newer" than the files that depend on it.

You can force MAKE to rebuild a target file by *touching* one of the files that target depends on. To touch a file (or files), type

*You can use the DOS wildcards * and ? with TOUCH.*

touch *filename* [*filename* ...]

at the DOS prompt. TOUCH will then update the file's creation date(s). Once you do this, you can invoke MAKE to rebuild the touched target file(s).

Important!

Before you use the TOUCH utility, it's vitally important to set your system's internal clock to the proper date and time. If you're using an IBM PC, XT, or compatible that doesn't have a battery-powered clock, don't forget to set the time and date using the DOS "time" and "date" commands. Failing to do this will keep both TOUCH and MAKE from working properly.

A P P E N D I X

E

Error messages

This chapter describes all the messages that Turbo Assembler generates. Messages usually appear on the screen, but you can redirect them to a file or printer using the standard DOS redirection mechanism of putting the device or file name on the command line, preceded by the greater than (>) symbol. For example,

```
TASM MYFILE >ERRORS
```

Turbo Assembler generates several types of messages:

- information messages
- warning messages
- error messages
- fatal error messages

Information messages

Turbo Assembler displays two information messages: one when it starts assembling your source file(s) and another when it has finished assembling each file. Here's a sample startup display:

```
Turbo Assembler Version 1.00 Copyright (C) 1988 Borland International
Assembling file: TEST.ASM
```

When Turbo Assembler finishes assembling your source file, it displays a message that summarizes the assembly process; the message looks like this:

```
Error messages: None
Warning messages: None
Remaining memory: 279k
```

You can suppress all information messages by using the /T command-line option. This only suppresses the information messages if no errors occur during assembly. If there are any errors, the /T option has no effect and the normal startup and ending messages appear.

Warning and error messages

Warning messages let you know that something undesirable may have happened while assembling a source statement. This might be something such as the Turbo Assembler making an assumption that is usually valid, but might not always be correct. You should always examine the cause of warning messages to see if the generated code is what you wanted. Warning messages *won't* stop Turbo Assembler from generating an object file. These messages are displayed using the following format:

```
**Warning** filename(line) message
```

If the warning occurs while expanding a macro or repeat block, the warning message contains additional information, naming the macro and the line within it where the warning occurred:

```
**Warning** filename(line) macroname(macroline) message
```

Error messages, on the other hand, *will* prohibit Turbo Assembler from generating an object file, but assembly will continue to the end of the file. Here's a typical error message format:

```
**Error** filename(line) message
```

If the error occurs while expanding a macro or repeat block, the error message contains additional information, naming the macro and the line within it where the error occurred:

```
**Error** filename(line) macroname(macroline) message
```

The following warning and error messages are arranged in alphabetical order:

Argument needs type override

The expression needs to have a specific size or type supplied, since its size can't be determined from the context. For example,

```
mov  [bx],1
```

You can usually correct this error by using the **PTR** operator to set the size of the operand:

```
mov  WORD PTR[bx],1
```

Argument to operation or instruction has illegal size

An operation was attempted on something that could not support the required operation. For example,

```
Q LABEL QWORD
QNOT = not Q            ;can't negate a qword
```

Arithmetic overflow

A loss of arithmetic precision occurred somewhere in the expression. For example,

```
X = 20000h * 20000h    ;overflows 32 bits
```

All calculations are performed using 32-bit arithmetic.

ASSUME must be segment register

You have used something other than a segment register in an **ASSUME** statement. For example,

```
ASSUME ax:CODE
```

You can only use segment registers with the **ASSUME** directive.

Assuming segment is 32 bit

You have started a segment using the **SEGMENT** directive after having enabled 80386 instructions, but you have not specified whether this is a 16- or 32-bit segment with either the **USE16** or **USE32** keyword.

In this case, Turbo Assembler presumes that you want a 32-bit segment. Since that type of code segment won't execute properly under DOS (without you taking special measures to ensure that the 80386 processor is executing instructions in a 32-bit segment), the warning is issued as **USE32**.

You can remove this warning by explicitly specifying **USE16** as an argument to the **SEGMENT** directive.

Bad keyword in SEGMENT statement

One of the align/combine/use arguments to the **SEGMENT** directive is invalid. For example,

```
DATA SEGMENT PAFA PUBLIC    ;PAFA should be PARA
```

Can't add relative quantities

You have specified an expression that attempts to add together two addresses, which is a meaningless operation. For example,

```
ABC   DB  ?
DEF = ABC + ABC             ;error, can't add two relatives
```

You can subtract two relative addresses, or you can add a constant to a relative address, as in:

```
XYZ    DB  5 DUP (0)
XYZEND EQU $
XYZLEN = SYZEND - XYZ       ;perfectly legal
XYZ2 = XYZ + 2              ;legal also
```

Can't address with currently ASSUMEd segment registers

An expression contains a reference to a variable for which you have not specified the segment register needed to reach it. For example,

```
DSEG SEGMENT
     ASSUME ds:DSEG
     mov  si,MPTR           ;no segment register to reach XSEG
DSEG ENDS
XSEG SEGMENT
MPTR DW   ?
XSEG ENDS
```

Can't convert to pointer

Part of the expression could not be converted to a memory pointer, for example, by using the **PTR** operator,

```
mov cl,[BYTE PTR al]        ;can't make AL into pointer
```

Can't emulate 8087 instruction

The Turbo Assembler is set to generate emulated floating-point instructions, either via the **/E** command-line option or by using the **EMUL** directive, but the current instruction can't be emulated. For example,

```
EMUL
FNSAVE [WPTR]               ;can't emulate this
```

The following instructions are not supported by floating-point emulators: **FNSAVE, FNSTCW, FNSTENV,** and **FNSTSW.**

Can't make variable public

The variable is already declared in such a way that it can't be made public. For example,

```
EXTRN  ABC:NEAR
PUBLIC ABC                    ;error, already EXTRN
```

Can't override ES segment

The current statement specifies an override that can't be used with that instruction. For example,

```
stos  DS:BYTE PTR[di]
```

Here, the **STOS** instruction can only use the **ES** register to access the destination address.

Can't subtract dissimilar relative quantities

An expression subtracts two addresses that can't be subtracted from each other, such as when they are each in a different segment:

```
SEG1 SEGMENT
A:
SEG1 ENDS
SEG2 SEGMENT
B:
     mov  ax,B-A      ;illegal, A and B in different
segments
SEG2 ENDS
```

Can't use macro name in expression

A macro name was encountered as part of an expression. For example,

```
MyMac    MACRO
     ENDM
     mov ax,MyMac     ;wrong!
```

Can't use this outside macro

You have used a directive outside a macro definition that can only be used inside a macro definition. This includes directives like **ENDM** and **EXITM**. For example,

```
DATA SEGMENT
     ENDM             ;error, not inside macro
```

Code or data emission to undeclared segment

A statement that generated code or data is outside of any segment declared with the **SEGMENT** directive. For example,

```
;First line of file
    inc  bx          ;error, no segment
    END
```

You can only emit code or data from within a segment.

Constant assumed to mean immmediate constant

This warning appears if you use an expression such as [0], which under MASM is interpreted as simply 0. For example,

```
mov  ax[0]   ;means mov ax,0 NOT mov ax,DS:[0]
```

Constant too large

You have entered a constant value that is properly formatted, but is too large. For example, you can only use numbers larger than 0ffffh when you have enabled 80386 instructions with the **.386** or **.386P** directive.

CS not correctly assumed

A near **CALL** or **JMP** instruction can't have as its target an address in a different segment. For example,

```
SEG1  SEGMENT
LAB1  LABEL NEAR
SEG1  ENDS
SEG2  SEGMENT
      jmp LAB1        ;error, wrong segment
SEG2  ENDS
```

This error only occurs in MASM mode. Ideal mode correctly handles this situation.

CS override in protected mode

The current instruction requires a CS override, and you are assembling instructions for the 286 or 386 in protected mode (**P286P** or **P386P** directives). For example,

```
     P286P
     .CODE
CVAL DW   ?
     mov  CVAL,1      ;generates CS override
```

The **/P** command-line option enables this warning. When running in protected mode, instructions with CS overrides won't work without you taking special measures.

CS unreachable from current segment

When defining a code label using colon (:), **LABEL** or **PROC**, the **CS** register is not assumed to either the current code segment or to a group that contains the current code segment. For example,

```
PROG1    SEGMENT
     ASSUME cs:PROG2
START:                 ;error, bad CS assume
```

This error only occurs in MASM mode. Ideal mode correctly handles this situation.

Declaration needs name

You have used a directive that needs a symbol name, but none has been supplied. For example,

```
PROC                   ;error, PROC needs a name
     ret
ENDP
```

You must always supply a name as part of a **SEGMENT, PROC,** or **STRUC** declaration. In MASM mode, the name precedes the directive; in Ideal mode, the name comes after the directive.

Directive ignored in Turbo Pascal model

You have tried to use one of the directives that can't be used when writing an assembler module to interface with Turbo Pascal. Read about the **.MODEL** directive that specifies Turbo Pascal in Chapter 3 of this manual. Refer to Chapter 8 of the *User's Guide* for information about interfacing to Turbo Pascal.

Directive not allowed inside structure definition

You have used a directive inside a **STRUC** definition block that can't be used there. For example,

```
X STRUC
MEM1 DB  ?
     ORG $+4           ;error, can't use ORG inside STRUC
MEM2 DW  ?
ENDS
```

Also, when declaring nested structures, you cannot give a name to any that are nested. For example,

```
FOO STRUC
    FOO2 STRUC         ;can't name inside
    ENDS
ENDS
```

If you want to use a named structure inside another structure, you must first define the structure and then use that structure name inside the second structure.

Duplicate dummy argument: __

A macro defined with the **MACRO** directive has more than one dummy parameter with the same name. For example,

```
XYZ  MACRO A,A        ;error, duplicate dummy name
     DB A
     ENDM
```

Each dummy parameter in a macro definition must have a different name.

ELSE or ENDIF without IF

An **ELSE** or **ENDIF** directive has no matching **IF** directive to start a conditional assembly block. For example,

```
BUF  DB 10 DUP (?)
     ENDIF            ;error, no matching IFxxx
```

Expecting offset quantity

An expression expected an operand that referred to an offset within a segment, but did not encounter the right sort of operand. For example,

```
CODE SEGMENT
     mov  ax,LOW CODE
CODE ENDS
```

Expecting offset or pointer quantity

An expression expected an operand that referred to an offset within a specific segment, but did not encounter the right sort of operand. For example,

```
CODE SEGMENT
     mov  ax,SEG CODE    ;error, code is a segment not
                         ; a location within a segment
CODE ENDS
```

Expecting pointer type

The current instruction expected an operand that referenced memory. For example,

```
les di,4        ;no good, 4 is a constant
```

Expecting scalar type

An instruction operand or operator expects a constant value. For example,

```
BB   DB   4
     rol  ax,BB         ;ROL needs constant
```

Expecting segment or group quantity

A statement required a segment or group name, but did not find one. For example,

```
DATA SEGMENT
     ASSUME  ds:FOO ;error, FOO is not group or segment name
FOO  DW  0
DATA ENDS
```

Extra characters on line

A valid expression was encountered, but there are still characters left on the line. For example,

```
ABC = 4 shl 3 3   ;missing operator between 3 and 3
```

This error often happens in conjunction with another error that caused the expression parser to lose track of what you intended to do.

Forward reference needs override

An expression containing a forward-referenced variable resulted in more code being required than Turbo Assembler anticipated. This can happen either when the variable is unexpectedly a far address for a **JMP** or **CALL** or when the variable requires a segment override in order to access it. For example,

```
       ASSUME cs:DATA
       call A            ;presume near call
A PROC FAR               ;oops, it's far
       mov ax,MEMVAR     ;doesn't know it needs override
DATA   SEGMENT
MEMVAR DW  ?             ;oops, needs override
```

Correct this by explicitly supplying the segment override or **FAR** override.

Global type doesn't match symbol type

This warning is given when a symbol is declared using the **GLOBAL** statement and is also defined in the same module, but the type specified in the **GLOBAL** and the actual type of the symbol don't agree.

ID not member of structure

In Ideal mode, you have specified a symbol that is not a structure member name after the period (.) structure member operator. For example,

```
      IDEAL
STRUC DEMO
      DB  ?
ENDS
COUNT     DW 0
      mov ax,[(DEMO bx).COUNT] ;COUNT not part of structure
```

You must follow the period with the name of a member that belongs to the structure name that precedes the period.

This error often happens in conjunction with another error that caused the expression parser to lose track of what you intended to do.

Illegal forward reference

A symbol has been referred to that has not yet been defined, and a directive or operator requires that its argument not be forward-referenced. For example,

```
IF MYSYM       ;error, MYSYM not defined yet
      ;
ENDIF
MYSYM EQU 1
```

Forward references may not be used in the argument to any of the **IFxxx** directives, nor as the count in a **DUP** expression.

Illegal immediate

An instruction has an immediate (constant) operand where one is not allowed. For example,

```
mov  4,al
```

Illegal indexing mode

An instruction has an operand that specifies an illegal combination of registers. For example,

```
mov  al,[si+ax]
```

On all processors except the 80386, the only valid combinations of index registers are: BX, BP, SI, DI, BX+SI, BX+DI, BP+SI, BP+DI.

Illegal instruction

A source line starts with a symbol that is neither one of the known directives nor a valid instruction mnemonic.

```
move ax,4        ;should be "MOV"
```

Illegal instruction for currently selected processor(s)

A source line specifies an instruction that can't be assembled for the current processor. For example,

```
.8086
push  1234h      ;no immediate push on 8086
```

When Turbo Assembler first starts assembling a source file, it generates instructions for the 8086 processor, unless told to do otherwise.

If you wish to use the extended instruction mnemonics available on the 186/286/386 processors, you must use one of the directives that enables those instructions (**P186, P286, P386**).

Illegal local argument

The **LOCAL** directive inside a macro definition has an argument that is not a valid symbol name. For example,

```
X    MACRO
     LOCAL 123      ;not a symbol
     ENDM
```

Illegal local symbol prefix

The argument to the **LOCALS** directive specifies an invalid start for local symbols. For example,

```
LOCALS  XYZ         ;error, not 2 characters
```

The local symbol prefix must be exactly two characters that themselves are a valid symbol name, such as __, @@, and so on (the default is @@).

Illegal macro argument

A macro defined with the **MACRO** directive has a dummy argument that is not a valid symbol name. For example,

```
X    MACRO 123      ;invalid dummy argument
     ENDM
```

Illegal memory reference

An instruction has an operand that refers to a memory location, but a memory location is not allowed for that operand. For example,

```
mov  [bx],BYTE PTR A    ;error, can't move from MEM to MEM
```

Here, both operands refer to a memory location, which is not a legal form of the **MOV** instruction. On the 80x86 family of processors, only one of the operands to an instruction can refer to a memory location.

Illegal number

A number contains one or more characters that are not valid for that type of number. For example,

```
Z = 0ABCGh
```

Here, *G* is not a valid letter in a hexadecimal number.

Illegal origin address

You have entered an invalid address to set the current segment location ($). You can enter either a constant or an expression using the location counter ($), or a symbol in the current segment.

Illegal override in structure

You have attempted to initialize a structure member that was defined using the **DUP** operator. You can only initialize structure members that were declared without **DUP**.

Illegal override register

A register other than a segment register (CS, DS, ES, SS, and on the 80386, FS and GS) was used as a segment override, preceding the colon (:) operator. For example,

```
mov dx:XYZ,1          ;DX not a segment register
```

Illegal radix

The number supplied to the **.RADIX** directive that sets the default number radix is invalid. For example,

```
.RADIX  7             ;no good
```

The radix can only be set to one of 2, 8, 10, or 16. The number is interpreted as decimal no matter what the current default radix is.

Illegal register multiplier

You have attempted to multiply a register by a value, which is not a legal operation; for example,

```
mov ax*3,1
```

The only context where you can multiply a register by a constant expression is when specifying a scaled index operand on the 80386 processor.

Illegal segment address

This error appears if an address greater than 65,535 is specified as a constant segment address; for example,

```
FOO SEGMENT AT 12345h
```

Illegal use of constant

A constant appears as part of an expression where constants can't be used. For example,

```
mov bx+4,5
```

Illegal use of register

A register name appeared in an expression where it can't be used. For example,

```
X = 4 shl ax  ;can't use register with SHL operator
```

Illegal use of segment register

A segment register name appears as part of an instruction or expression where segment registers cannot be used. For example,

```
add  SS,4          ;ADD can't use segment regs
```

Illegal USES register

You have entered an invalid register to push and pop as part of entering and leaving a procedure. The valid registers follow:

AX	CX	DS	ES
BX	DI	DX	SI

If you have enable the 80386 processor with the **.386** or **.386P** directive, you can use the 32-bit equivalents for these registers.

Illegal warning ID

You have entered an invalid three-character warning identifier. See the options discussed in Chapter 3 of the *User's Guide* for a complete list of the allowed warning identifiers.

Instruction can be compacted with override

The code generated contains **NOP** padding, due to some forward-referenced symbol. You can either remove the forward reference or explicitly provide the type information as part of the expression. For example,

```
    jmp X         ;warning here
    jmp SHORT X   ;no warning
X:
```

Invalid model type

The model directive has an invalid memory model keyword. For example,

```
.MODEL GIGANTIC
```

Valid memory models are tiny, small, compact, medium, large, and huge.

Invalid operand(s) to instruction

The instruction has a combination of operands that are not permitted. For example,

```
fadd  ST(2),ST(3)
```

Here, *FADD* can only refer to one stack register by name; the other must be the stack top.

Labels can't start with numeric characters

You have entered a symbol that is neither a valid number nor a valid symbol name, such as *123XYZ*.

Line too long—truncating

The current line in the source file is longer than 255 characters. The excess characters will be ignored.

Location counter overflow

The current segment has filled up, and subsequent code or data will overwrite the beginning of the segment. For example,

```
ORG 0FFF0h
ARRAY  DW 20 DUP (0)    ;overflow
```

Missing argument list

An **IRP** or **IRPC** repeat block directive does not have an argument to substitute for the dummy parameter. For example,

```
IRP X           ;no argument list
      DB X
ENDM
```

IRP and **IRPC** must always have both a dummy parameter and an argument list.

Missing argument or <

You forgot the angle brackets or the entire expression in an expression that requires them. For example,

```
ifb    ;needs an argument in <>s
```

Missing argument size variable

An **ARG** or **LOCAL** directive does not have a symbol name following the optional = at the end of the statement. For example,

```
ARG   A:WORD,B:DWORD=        ;error, no name after =
LOCAL X:TBYTE=               ;same error here
```

ARG and **LOCAL** must always have a symbol name if you have used the optional equal sign (=) to indicate that you want to define a size variable.

Missing COMM ID

A **COMM** directive does not have a symbol name before the type specifier. For example,

```
COMM  NEAR  ;error, no symbol name before "NEAR"
```

COMM must always have a symbol name before the type specifier, followed by a colon (:) and then the type specifier.

Missing dummy argument

An **IRP** or **IRPC** repeat block directive does not have a dummy parameter. For example,

```
RP              ;no dummy parameter
    DB X
ENDM
```

IRP and **IRPC** must always have both a dummy parameter and an argument list.

Missing end quote

A string or character constant did not end with a quote character. For example,

```
DB   "abc       ;missing " at end of ABC
mov  al,'X      ;missing ' after X
```

You should always end a character or string constant with a quote character matching the one that started it.

Missing macro ID

A macro defined with the **MACRO** directive has not been given a name. For example,

```
MACRO               ;error, no name
      DB A
      ENDM
```

Macros must always be given a name when they are defined.

Missing module name

You have used the **NAME** directive but you haven't supplied a module name after the directive. Remember that the **NAME** directive only has an effect in Ideal mode.

Missing or illegal language ID

You have entered something other than one of the allowed language identifiers after the **.MODEL** directive. See Chapter 3 of this book for a complete description of the **.MODEL** directive.

Missing or illegal type specifier

A statement that needed a type specifier (like **BYTE, WORD,** and so on) did not find one where expected. For example,

```
RED  LABEL XXX      ;error, "XXX" is not a type specifier
```

Missing term in list

In Ideal mode, a directive that can accept multiple arguments (**EXTRN, PUBLIC,** and so on) separated by commas does not have an argument after one of the commas in the list. For example,

```
EXTRN XXX:BYTE,,YYY:WORD
```

In Ideal mode, all argument lists must have their elements separated by precisely one comma, with no comma at the end of the list.

Missing text macro

You have not supplied a text macro argument to a directive that requires one. For example,

```
NEWSTR  SUBSTR          ;ERROR - SUBSTR NEEDS ARGUMENTS
```

Model must be specified first

You used one of the simplified segmentation directives without first specifying a memory model. For example,

```
.CODE       ;error, no .MODEL first
```

You must always specify a memory model using the **.MODEL** directive before using any of the other simplified segmentation directives.

Module is pass-dependent—compatibility pass was done

This warning occurs if a pass-dependent construction was encountered and the **/m** command-line switch was specified. A MASM-compatible pass was done.

name must come first

You put a symbol name after a directive, and the symbol name should come first. For example,

```
STRUC ABC          ;error, ABC must come before STRUC
```

Since Ideal mode expects the name to come after the directive, you will encounter this error if you try to assemble Ideal mode programs in MASM mode.

Near jump or call to different CS

This error occurs if the user attempts to perform a **NEAR CALL** or **JMP** to a symbol that's defined in an area where CS is assumed to a different segment.

Need address or register

An instruction does not have a second operand supplied, even though there is a comma present to separate two operands; for example,

```
mov ax,            ;no second operand
```

Need angle brackets for structure fill

A statement that allocates storage for a structure does not specify an initializer list. For example,

```
STR1 STRUC
M1   DW   ?
M2   DD   ?
     ENDS
     STR1        ;no initializer list
```

Need colon

An **EXTRN, GLOBAL, ARG,** or **LOCAL** statement is missing the colon after the type specifier (**BYTE, WORD,** and so on). For example,

```
EXTRN X BYTE,Y:WORD      ;X has no colon
```

Need expression

An expression has an operator that is missing an operand. For example,

```
X = 4 + * 6
```

Need file name after INCLUDE

An **INCLUDE** directive did not have a file name after it. For example,

```
INCLUDE      ;include what?
```

In Ideal mode, the file name must be enclosed in quotes.

Need left parenthesis

A left parenthesis was omitted that is required in the expression syntax. For example,

```
DB 4 DUP 7
```

You must always enclose the expression after the **DUP** operator in parentheses.

Need pointer expression

This error only occurs in Ideal mode and indicates that the expression between brackets ([ ]) does not evaluate to a memory pointer. For example,

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

In Ideal mode, you must always supply a memory-referencing expression between the brackets.

Need quoted string

You have entered something other than a string of characters between quotes where it is required. In Ideal mode, several directives require their argument to be a quoted string. For example,

```
IDEAL
DISPLAY "ALL DONE"
```

Need register in expression

You have entered an expression that does not contain a register name where one is required.

Need right angle bracket

An expression that initializes a structure, union, or record does not end with a > to match the < that started the initializer list. For example,

```
MYSTRUC STRUCNAME <1,2,3
```

Need right parenthesis

An expression contains a left parenthesis, but no matching right parenthesis. For example,

```
X = 5 * (4 + 3
```

You must always use left and right parentheses in matching pairs.

Need right square bracket

An expression that references a memory location does not end with a] to match the [that started the expression. For example,

```
mov  ax,[si      ;error, no closing ] after SI
```

You must always use square brackets in matching pairs.

Need stack argument

A floating-point instruction does not have a second operand supplied, even though there is a comma present to separate two operands. For example,

```
fadd ST,
```

Need structure member name

In Ideal mode, the period (.) structure member operator was followed by something that was not a structure member name. For example,

```
      IDEAL
STRUC DEMO
      DB   ?
ENDS
COUNT      DW 0
      mov  ax,[(DEMO bx).]
```

You must always follow the period operator with the name of a member in the structure to its left.

Not expecting group or segment quantity

You have used a group or segment name where it can't be used. For example,

```
CODE SEGMENT
     rol  ax,CODE      ;error, can't use segment name here
```

One non-null field allowed per union expansion

When initializing a union defined with the **UNION** directive, more than one value was supplied. For example,

```
U     UNION
      DW   ?
      DD   ?
ENDS
UINST U <1,2>      ;error, should be <?,2> or <1,?>
```

A union can only be initialized to one value.

Only one startup sequence allowed

This error appears if you have more than one **.STARTUP** or **STARTUPCODE** statement in a module.

Open conditional

The end of the source file has been reached as defined with the **END** directive, but a conditional assembly block started with one of the **IFxxx** directives has not been ended with the **ENDIF** directive. For example,

```
IF BIGBUF
END          ;no ENDIF before END
```

This usually happens when you type **END** instead of **ENDIF** to end a conditional block.

Open procedure

The end of the source file has been reached as defined with the **END** directive, but a procedure block started with the **PROC** directive has not been ended with the **ENDP** directive. For example,

```
MYFUNC PROC
   END          ;no ENDIF before ENDP
```

This usually happens when you type **END** instead of **ENDP** to end a procedure block.

Open segment

The end of the source file has been reached as defined with the **END** directive, but a segment started with the **SEGMENT** directive has not been ended with the **ENDS** directive. For example,

```
DATA SEGMENT
     END          ;no ENDS before END
```

This usually happens when you type **END** instead of **ENDS** to end a segment.

Open structure definition

The end of the source file has been reached as defined with the **END** directive, but a structure started with the **STRUC** directive has not been ended with the **ENDS** directive. For example,

```
X    STRUC
VAL1 DW   ?
     END        ;no ENDS before it
```

This usually happens when you type **END** instead of **ENDS** to end a structure definition.

Operand types do not match

The size of an instruction operand does not match either the other operand or one valid for the instruction; for example,

```
ABC  DB   5
     • • •
     mov  ax,ABC
```

Pass-dependent construction encountered

The statement may not behave as you expect, due to the one-pass nature of Turbo Assembler. For example,

```
IF1
              ;Happens on assembly pass
ENDIF
IF2
              ;Happens on listing pass
ENDIF
```

Most constructs that generate this error can be re-coded to avoid it, often by removing forward references.

Pointer expression needs brackets

In Ideal mode, the operand contained a memory-referencing symbol that was not surrounded by brackets to indicate that it references a memory location. For example,

```
B  DB   0
   mov  al,B      ;warning, Ideal mode needs [B]
```

Since MASM mode does not require the brackets, this is only a warning.

Positive count expected

A **DUP** expression has a repeat count less than zero. For example,

```
BUF  -1 DUP (?)      ;error, count < 0
```

The count preceding a **DUP** must always be 1 or greater.

Record field too large

When you defined a record, the sum total of all the field widths exceeded 32 bits. For example,

```
AREC RECORD   RANGE:12,TOP:12,BOTTOM:12
```

Recursive definition not allowed for EQU

An **EQU** definition contained the same name that you are defining within the definition itself. For example,

```
ABC EQU TWOTIMES ABC
```

Register must be AL or AX

An instruction which requires one operand to be the AL or AX register has been given an invalid operand. For example,

```
IN  CL,dx      ;error, "IN" must be to AL or AX
```

Register must be DX

An instruction which requires one operand to be the DX register has been given an invalid operand. For example,

```
IN  AL,cx      ;error, must be DX register instead of CX
```

Relative jump out of range by __ bytes

A conditional jump tried to reference an address that was greater than 128 bytes before or 127 bytes after the current location. If this is in a **USE32** segment, the conditional jump can reference between 32,768 bytes before and 32,767 bytes after the current location.

Relative quantity illegal

An instruction or directive has an operand that refers to a memory address in a way that can't be known at assembly time, and this is not allowed. For example,

```
DATA SEGMENT PUBLIC
X    DB   0
     IF   OFFSET X GT 127    ;not known at assemble time
```

Reserved word used as symbol

You have created a symbol name in your program that Turbo Assembler reserves for its own use. Your program will assemble properly, but it is good practice not to use reserved words for your own symbol names.

Rotate count must be constant or CL

A shift or rotate instruction has been given an operand that is neither a constant nor the CL register. For example,

```
rol  ax,DL      ;error, can't use DL as count
```

You can only use a constant value or the CL register as the second operand to a rotate or shift instruction.

Rotate count out of range

A shift or rotate instruction has been given a second operand that is too large. For example,

```
.8086
shl   DL,3      ;error, 8086 can only shift by 1
.286
ror   ax,40     ;error, max shift is 31
```

The 8086 processor only allows a shift count of 1, but the other processors allow a shift count up to 31.

Segment alignment not strict enough

The align boundary value supplied is invalid. Either it is not a power of 2, or it specifies an alignment stricter than that of the align type in the **SEGMENT** directive. For example,

```
DATA SEGMENT PARA
     ALIGN 32         ;error, PARA is only 16
     ALIGN 3          ;error, not power of 2
```

Segment attributes illegally redefined

A **SEGMENT** directive re-opens a segment that has been previously defined, and tries to give it different attributes. For example,

```
DATA SEGMENT BYTE PUBLIC
DATA ENDS
DATA SEGMENT PARA ;error, previously had byte alignment
DATA ENDS
```

If you re-open a segment, the attributes you supply must either match exactly or be omitted entirely. If you don't supply any attributes when re-opening a segment, the old attributes will be used.

Segment name is superfluous

This warning appears with a **.CODE** *xxx* statement, where the model specified doesn't allow more than code segment.

Smart code generation must be enabled

Certain special features of code generation require **SMART** code generation to be enabled. These include **PUSH** of a pointer, **POP** of a pointer, and **PUSH** of a constant (8086 only).

String too long

You have built a quoted string that is longer than the maximum allowed length of 255.

Symbol already defined: __

The indicated symbol has previously been declared with the same type. For example,

```
BB  DB  1,2,3
BB  DB  ?          ;error, BB already defined
```

Symbol already different kind

The indicated symbol has already been declared before with a different type. For example,

```
BB  DB  1,2,3
BB  DW  ?          ;error, BB already a byte
```

Symbol has no width or mask

The operand of a **WIDTH** or **MASK** operator is not the name of a record or record field. For example,

```
B  DB  0
   mov ax,MASK B      ;B is not a record field
```

Symbol is not a segment or already part of a group

The symbol has either already been placed in a group or it is not a segment name. For example,

```
DATA    SEGMENT
DATA    ENDS
DGROUP    GROUP DATA
DGROUP2   GROUP DATA  ;error, DATA already belongs to DGROUP
```

Text macro expansion exceeds maximum line length

This error occurs when expansion of a text macro causes the maximum allowable line length to be exceeded.

Too few operands to instruction

The instruction statement requires more operands than were supplied. For example,

```
add  ax      ;missing second arg
```

Too many errors or warnings

No more error messages will be displayed. The maximum number of errors that will be displayed is 100; this number has been exceeded. Turbo Assembler continues to assemble and prints warnings rather than error messages.

Too many initial values

You have supplied too many values in a structure or union initialization. For example,

```
XYZ   STRUC
A1 DB ?
A2 DD ?
XYZ   ENDS
```

```
ANXYZ XYZ   <1,2,3>      ;error, only 2 members in XYZ
```

You can supply fewer initializers than there are members in a structure or union, but never more.

Too many register multipliers in expression

An 80386 scaled index operand had a scale factor on more than one register. For example,

```
mov EAX,[2*EBX+4*EDX]      ;too many scales
```

Too many registers in expression

The expression has more than one index and one base register. For example,

```
mov ax,[BP+SI+DI]      ;can't have SI and DI
```

Too many USES registers

You specified more than 8 **USES** registers for the current procedure.

Trailing null value assumed

A data statement like **DB**, **DW**, and so on, ends with a comma. TASM treats this as a null value. For example,

```
db 'hello',13,10   ;same as ...,13,10?
```

Undefined symbol

The statement contains a symbol that wasn't defined anywhere in the source file.

Unexpected end of file (no END directive)

The source file does not have an **END** directive as its last statement.

All source files must have an **END** statement.

Unknown character

The current source line contains a character that is not part of the set of characters that make up Turbo Assembler symbol names or expressions. For example,

```
add  ax,!1      ;error, exclamation is illegal character
```

Unmatched ENDP: __

The **ENDP** directive has a name that does not match the **PROC** directive that opened the procedure block. For example,

```
ABC PROC
XYZ ENDP      ;error, XYZ should be ABC
```

Unmatched ENDS: __

The **ENDS** directive has a name that does not match either the **SEGMENT** directive that opened a segment or the **STRUC** or **UNION** directive that started a structure or union definition. For example,

```
ABC STRUC
XYZ ENDS      ;error, XYZ should be ABC
DATA SEGMENT
CODE ENDS     ;error, code should be DATA
```

USE32 not allowed without .386

You have attempted to define a 32-bit segment, but you have not specified the 80386 processor first. You can only define 32-bit segments after you have used the **.386** or **.386P** directives to set the processor to be 80386.

User-generated error

An error has been forced by one of the directives, which then forces an error. For example,

```
    .ERR      ;shouldn't get here
```

USES has no effect without *language*

This warning appears if you specify a **USES** statement when no *language* is in effect.

Value out of range

The constant is a valid number, but it is too large to be used where it appears. For example,

```
DB 400
```

Fatal error messages

Fatal error messages cause Turbo Assembler to immediately stop assembling your file. Whatever caused the error prohibited the assembler from being able to continue. Here's a list of possible fatal error messages.

Bad switch __

You have used an invalid command-line option. See Chapter 3 of the *User's Guide* for a description of the command-line options.

Can't find @file __

You have specified an indirect command file name that does not exist. Make sure that you supply the complete file name. Turbo Assembler does not presume any default extension for the file name. You've probably run out of space on the disk where you asked the cross-reference file to be written.

Can't locate file ___

You have specified a file name with the *INCLUDE* directive that can't be found. Read about the *INCLUDE* directive in Chapter 3 in this book to learn where Turbo Assembler searches for included files.

An INCLUDE file could not be located. Make sure that the name contains any necessary disk letter or directory path.

Error writing to listing file

You've probably run out of space on the disk where you asked the listing file to be written.

Error writing to object file

You've probably run out of space on the disk where you asked the object file to be written.

File not found

The source file name you specified on the command line does not exist. Make sure you typed the name correctly, and that you included any necessary drive or path information if the file is not in the current directory.

File was changed or deleted while assembly in progress

Another program, such as a pop-up utility, has changed or deleted the file after Turbo Assembler opened it. Turbo Assembler can't re-open a file that was previously opened successfully.

Insufficient memory to process command line

You have specified a command line that is either longer than 64K or can't be expanded in the available memory. Either simplify the command line or run Turbo Assembler with more memory free.

Internal error

This message should never happen during normal operation of Turbo Assembler. Save the file(s) that caused the error and report it to Borland's Technical Support department.

Invalid command line

The command line that you used to start Turbo Assembler is badly formed. For example,

```
TASM ,MYFILE
```

does not specify a source file to assemble. See Chapter 3 of the *User's Guide* for a complete description of the Turbo Assembler command line.

Invalid number after __

You have specified a valid command-line switch (option), but have not supplied a valid numeric argument following the switch. See Chapter 3 of the *User's Guide* for a discussion of the command-line options.

Maximum macro expansion size exceeded

A macro expanded into more text than would fit in the macro expansion area. Since this area is up to 64 Kb long, you will usually only see this message if you have a macro with a bug in it, causing it to expand indefinitely.

Out of hash space

The hash space has one entry for each symbol you define in your program. It starts out allowing 16,384 symbols to be defined, as long as Turbo Assembler is running with enough free memory. If your program has more than this many symbols, use the **/KH** command-line option to set the number of symbol entries you need in the hash table.

Out of memory

You don't have enough free memory for Turbo Assembler to assemble your file.

If you have any TSR (RAM-resident) programs installed, you can try removing them from memory and try assembling your file again. You may have to reboot your system in order for memory to be properly freed.

Another solution is to split the source file into two or more source files, or rewrite portions of it so that it requires less memory to assemble. You can also use shorter symbol names, reduce the number of comments in macros, and reduce the number of forward references in your program.

Out of string space

You don't have enough free memory for symbol names, file names, forward-reference tracking information, and macro text. You can use the **/KS** command-line option to allocate more memory to the string space. Normally, half of the free memory is assigned for use as string space.

Too many errors found

Turbo Assembler has stopped assembling your file because it contained so many errors. You may have made a few errors that have snowballed. For example, failing to define a symbol that you use on many lines is really a single error (failing to define the symbol), but you will get an error message for each line that referred to the symbol.

Turbo Assembler will stop assembling your file if it encounters a total of 100 errors or warnings.

Unexpected end of file (no END directive)

Your source file ended without a line containing the **END** directive. All source files must end with an **END** directive.

I N D E X

2.0

REFERENCE GUIDE

TURBO ASSEMBLER®

BORLAND

1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300 ■ PART # 15MN-ASD03-20 ■ BOR 1491
UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK, TWYFORD, BERKSHIRE RG10 9NN, ENGLAND
43 AVENUE DE L'EUROPE—BP 6, 78141 VELIZY VILLACOUBLAY CEDEX FRANCE