CHAPTER 9

Using Macros

A “macro” is a symbolic name you give to a series of characters (a text macro) or to one or more statements (a macro procedure or function). As the assembler evaluates each line of your program, it scans the source code for names of previously defined macros. When it finds one, it substitutes the macro text for the macro name. In this way, you can avoid writing the same code several places in your program.

This chapter describes the following types of macros:

- Text macros, which expand to text within a source statement.
- Macro procedures, which expand to one or more complete statements and can optionally take parameters.
- Repeat blocks, which generate a group of statements a specified number of times or until a specified condition becomes true.
- Macro functions, which look like macro procedures and can be used like text macros but which also return a value.
- Predefined macro functions and string directives, which perform string operations.

This chapter explains how to use macros for simple code substitutions and how to write sophisticated macros with parameter lists and repeat loops. It also describes how to use these features in conjunction with local symbols, macro operators, and predefined macro functions.

Text Macros

You can give a sequence of characters a symbolic name and then use the name in place of the text later in the source code. The named text is called a text macro.

The **TEXTEQU** directive defines a text macro, as these examples show:

name **TEXTEQU** *<text>*
name **TEXTEQU** *macroId | textmacro*
name **TEXTEQU** *%constExpr*

In the previous lines, *text* is a sequence of characters enclosed in angle brackets, *macroId* is a previously defined macro function, *textmacro* is a previously defined text macro, and *%constExpr* is an expression that evaluates to text.

Here are some examples:

```
msg     TEXTEQU <Some text>         ; Text assigned to symbol
string  TEXTEQU msg                 ; Text macro assigned to symbol
msg     TEXTEQU <Some other text>   ; New text assigned to symbol
value   TEXTEQU %(3 + num)          ; Text representation of resolved
                                    ;   expression assigned to symbol
```

The first line assigns text to the symbol `msg`. The second line equates the text of the `msg` text macro with a new text macro called `string`. The third line assigns new text to `msg`. Although `msg` has new text, `string` retains its original text value. The fourth line assigns 7 to `value` if `num` equals 4. If a text macro expands to another text macro (or macro function, as discussed on page 248), the resulting text macro will expand recursively.

Text macros are useful for naming strings of text that do not evaluate to integers. For example, you might use a text macro to name a floating-point constant or a bracketed expression. Here are some practical examples:

```
pi      TEXTEQU <3.1416>            ; Floating point constant
WPT     TEXTEQU <WORD PTR>          ; Sequence of key words
arg1    TEXTEQU <[bp+4]>            ; Bracketed expression
```

Macro Procedures

If your program must perform the same task many times, you can avoid repeatedly typing the same statements each time by writing a macro procedure. Think of macro procedures (commonly called macros) as text-processing mechanisms that automatically generate repeated text.

This section uses the term “macro procedure” rather than “macro” when necessary to distinguish between a macro procedure and a macro function. Macro functions are described in “Returning Values with Macro Functions.”

Conforming to common usage, this chapter occasionally speaks of “calling” a macro, a term that deserves further scrutiny. It’s natural to think of a program calling a macro procedure in the same way it calls a normal subroutine procedure, because they seem to perform identically. However, a macro is simply a representative for real code. Wherever a macro name appears in your program, so in reality does all the code the macro represents. A macro does not cause the processor to vector off to a new location as does a normal procedure. Thus, the expression “calling a macro” may imply the effect, but does not accurately describe what actually occurs.

Creating Macro Procedures

You can define a macro procedure without parameters by placing the desired statements between the **MACRO** and **ENDM** directives:

name **MACRO**
statements
ENDM

For example, suppose you want a program to beep when it encounters certain errors. You could define a `beep` macro as follows:

```
beep    MACRO
    mov  ah, 2          ;; Select DOS Print Char function
    mov  dl, 7          ;; Select ASCII 7 (bell)
    int  21h            ;; Call DOS
ENDM
```

The double semicolons mark the beginning of macro comments. Macro comments appear in a listing file only at the macro’s initial definition, not at the point where the macro is referenced and expanded. Listings are usually easier to read if the comments aren’t repeatedly expanded. However, regular comments (those with a single semicolon) are listed in macro expansions. See Appendix C for listing files and examples of how macros are expanded in listings.

Once you define a macro, you can call it anywhere in the program by using the macro’s name as a statement. The following example calls the `beep` macro two times if an error flag has been set.

```
        .IF     error   ; If error flag is true
        beep            ;   execute macro two times
        beep
        .ENDIF
```

During assembly, the instructions in the macro replace the macro reference. The listing file shows:

```
                          .IF                         error
0017  80 3E 0000 R 00   *         cmp     error, 000h
001C  74 0C             *         je      @C0001
                          beep
001E  B4 02               1         mov     ah, 2
0020  B2 07               1         mov     dl, 7
0022  CD 21               1         int     21h
                          beep
0024  B4 02               1         mov     ah, 2
0026  B2 07               1         mov     dl, 7
0028  CD 21               1         int     21h
                          .ENDIF
002A                  *@C0001:
```

Contrast this with the results of defining `beep` as a procedure using the **PROC** directive and then calling it with the **CALL** instruction.

Many such tasks can be handled as either a macro or a procedure. In deciding which method to use, you must choose between speed and size. For repetitive tasks, a procedure produces smaller code, because the instructions physically appear only once in the assembled program. However, each call to the procedure involves the additional overhead of a **CALL** and **RET** instruction. Macros do not require a change in program flow and so execute faster, but generate the same code multiple times rather than just once.

Passing Arguments to Macros

By defining parameters for macros, you can define a general task and then execute variations of it by passing different arguments each time you call the macro. The complete syntax for a macro procedure includes a parameter list:

name **MACRO** *parameterlist*
statements
ENDM

The *parameterlist* can contain any number of parameters. Use commas to separate each parameter in the list. You cannot use reserved words as parameter names unless you disable the keyword with **OPTION NOKEYWORD**. You must also set the compatibility mode with **OPTION M510** or the /Zm command-line option.

To pass arguments to a macro, place the arguments after the macro name when you call the macro:

macroname arglist

The assembler treats as one item all text between matching quotation marks in an *arglist*.

The `beep` macro introduced in the previous section used the MS-DOS interrupt to write only the bell character (ASCII 7). We can rewrite the macro with a parameter that accepts any character:

```
writechar MACRO char
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

Whenever it expands the macro, the assembler replaces each instance of `char` with the given argument value. The rewritten macro now writes any character to the screen, not just ASCII 7:

```
        writechar 7             ; Causes computer to beep
        writechar 'A'           ; Writes A to screen
```

If you pass more arguments than there are parameters, the additional arguments generate a warning (unless you use the **VARARG** keyword; see page 242). If you pass fewer arguments than the macro procedure expects, the assembler assigns empty strings to the remaining parameters (unless you have specified default values). This may cause errors. For example, a reference to the `writechar` macro with no argument results in the following line:

```
        mov     dl,
```

The assembler generates an error for the expanded statement but not for the macro definition or the macro call.

You can make macros more flexible by leaving off arguments or adding additional arguments. The next section tells some of the ways your macros can handle missing or extra arguments.

Specifying Required and Default Parameters

Macro parameters can have special attributes to make them more flexible and improve error handling. You can make parameters required, give them default values, or vary their number. Variable parameters are used almost exclusively with the **FOR** directive, so are covered in "FOR Loops and Variable-Length Parameters," later in this chapter.

The syntax for a required parameter is:

parameter:**REQ**

For example, you can rewrite the `writechar` macro to require the `char` parameter:

```
writechar MACRO char:REQ
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

If the call does not include a matching argument, the assembler reports the error in the line that contains the macro reference. **REQ** can thus improve error reporting.

You can also accommodate missing parameters by specifying a default value, like this:

parameter:=*textvalue*

Suppose that you often use `writechar` to beep by printing ASCII 7. The following macro definition uses an equal sign to tell the assembler to assume the parameter `char` is 7 unless you specify otherwise:

```
writechar  MACRO char:=<7>
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

If a reference to this macro does not include the argument `char`, the assembler fills in the blank with the default value of 7 and the macro beeps when called.

Enclose the default parameter value in angle brackets so the assembler recognizes the supplied value as a text value. This is explained in detail in "Text Delimiters and the Literal-Character Operator," later in this chapter.

Missing arguments can also be handled with the **IFB**, **IFNB**, **.ERRB**, and **.ERRNB** directives. They are described in the section "Conditional Directives" in chapter 1 and in Help. Here is a slightly more complex macro that uses some of these techniques:

```
Scroll MACRO distance:REQ, attrib:=<7>, tcol, trow, bcol, brow
    IFNB <tcol>             ;; Ignore arguments if blank
        mov   cl, tcol
    ENDIF
    IFNB <trow>
        mov   ch, trow
    ENDIF
    IFNB <bcol>
        mov   dl, bcol
    ENDIF
    IFNB <brow>
        mov   dh, brow
    ENDIF
    IFDIFI <attrib>, <bh>   ;; Don't move BH onto itself
        mov   bh, attrib
    ENDIF
    IF distance LE 0        ;; Negative scrolls up, positive down
        mov   ax, 0600h + (-(distance) AND 0FFh)
    ELSE
        mov   ax, 0700h + (distance AND 0FFh)
    ENDIF
    int   10h
ENDM
```

In this macro, the `distance` parameter is required. The `attrib` parameter has a default value of 7 (white on black), but the macro also tests to make sure the corresponding argument isn't BH, since it would be inefficient (though legal) to load a register onto itself. The **IFNB** directive is used to test for blank arguments. These are ignored to allow the user to manipulate rows and columns directly in registers CX and DX at run time.

The following shows two valid ways to call the macro:

```
        ; Assume DL and CL already loaded
        dec     dh                  ; Decrement top row
        inc     ch                  ; Increment bottom row
        Scroll -3                   ; Scroll white on black dynamic
                                    ;   window up three lines
        Scroll 5, 17h, 2, 2, 14, 12 ; Scroll white on blue constant
                                    ;   window down five lines
```

This macro can generate completely different code, depending on its arguments. In this sense, it is not comparable to a procedure, which always has the same code regardless of arguments.

Defining Local Symbols in Macros

You can make a symbol local to a macro by identifying it at the start of the macro with the **LOCAL** directive. Any identifier may be declared local.

You can choose whether you want numeric equates and text macros to be local or global. If a symbol will be used only inside a particular macro, you can declare it local so that the name will be available for other declarations outside the macro.

You must declare as local any labels within a macro, since a label can occur only once in the source. The **LOCAL** directive makes a special instance of the label each time the macro appears. This prevents redefinition of the label when expanding the macro. It also allows you to reuse the label elsewhere in your code.

You must declare all local symbols immediately following the **MACRO** statement (although blank lines and comments may precede the local symbol). Separate each symbol with a comma. You can attach comments to the **LOCAL** statement and list multiple **LOCAL** statements in the macro. Here is an example macro that declares local labels:

```
power   MACRO   factor:REQ, exponent:REQ
    LOCAL   again, gotzero      ;; Local symbols
    sub     dx, dx              ;; Clear top
    mov     ax, 1               ;; Multiply by one on first loop
    mov     cx, exponent        ;; Load count
    jcxz    gotzero             ;; Done if zero exponent
    mov     bx, factor          ;; Load factor
again:
    mul     bx                  ;; Multiply factor times exponent
    loop    again               ;; Result in AX
gotzero:
ENDM
```

If the labels `again` and `gotzero` were not declared local, the macro would work the first time it is called, but it would generate redefinition errors on subsequent calls. MASM implements local labels by generating different names for them each time the macro is called. You can see this in listing files. The labels in the `power` macro might be expanded to `??0000` and `??0001` on the first call and to `??0002` and `??0003` on the second.

You should avoid using anonymous labels in macros (see "Anonymous Labels" in Chapter 7). Although legal, they can produce unwanted results if you expand a macro near another anonymous label. For example, consider what happens in the following:

```
Update  MACRO arg1
@@: .
    .
    .
    loop @B
ENDM
        .
        .
        .
        jcxz    @F
        Update  ax
@@:
```

Expanding `Update` places another anonymous label between the jump and its target. The line

```
        jcxz    @F
```

consequently jumps to the start of the loop rather than over the loop—exactly the opposite of what the programmer intended.

Assembly-Time Variables and Macro Operators

In writing macros, you will often assign and modify values assigned to symbols. Think of these symbols as assembly-time variables. Like memory variables, they are symbols that represent values. But since macros are processed at assembly time, any symbol modified in a macro must be resolved as a constant by the end of assembly.

The three kinds of assembly-time variables are:

- Macro parameters
- Text macros
- Macro functions

When the assembler expands a macro, it processes the symbols in the order shown here. MASM first replaces macro parameters with the text of their actual arguments, then expands text macros.

Macro parameters are similar to procedure parameters in some ways, but they also have important differences. In a procedure, a parameter has a type and a memory location. Its value can be modified within the procedure. In a macro, a parameter is a placeholder for the argument text. The value can only be assigned to another symbol or used directly; it cannot be modified. The macro may interpret the argument text it receives either as a numeric value or as a text value.

It is important to understand the difference between text values and numeric values. Numeric values can be processed with arithmetic operators and assigned to numeric equates. Text values can be processed with macro functions and assigned to text macros.

Macro operators are often helpful when processing assembly-time variables. Table 9.1 shows the macro operators that MASM provides.

Table 9.1 MASM Macro Operators

Symbol	Name	Description
< >	Text Delimiters	Opens and closes a literal string.
!	Literal-Character Operator	Treats the next character as a literal character, even if it would normally have another meaning.
%	Expansion Operator	Causes the assembler to expand a constant expression or text macro.
&	Substitution Operator	Tells the assembler to replace a macro parameter or text macro name with its actual value.

The next sections explain these operators in detail.

Text Delimiters and the Literal-Character Operator

The angle brackets (< >) are text delimiters. A text value is usually delimited when assigning a text macro. You can do this with **TEXTEQU**, as previously shown, or with the **SUBSTR** and **CATSTR** directives discussed in "String Directives and Predefined Functions," later in this chapter.

By delimiting the text of macro arguments, you can pass text that includes spaces, commas, semicolons, and other special characters. The following example expands a macro called `work` in two different ways:

```
work    <1, 2, 3, 4, 5> ; Passes one argument with 13 chars,
                        ;   including commas and spaces
work    1, 2, 3, 4, 5   ; Passes five arguments, each
                        ;   with 1 character
```

The literal-character operator (**!**) lets you include angle brackets as part of a delimited text value, so the assembler does not interpret them as delimiters. The assembler treats the character following **!** literally rather than as a special character, like this:

```
errstr  TEXTEQU <Expression !> 255>  ; errstr = "Expression > 255"
```

Text delimiters also have a special use with the **FOR** directive, as explained in "FOR Loops and Variable-Length Parameters," later in this chapter.

Expansion Operator

The expansion operator (**%**) expands text macros or converts constant expressions into their text representations. It performs these tasks differently in different contexts, as discussed in the following.

Converting Numeric Expressions to Text

The expansion operator can convert numbers to text. The operator forces immediate evaluation of a constant expression and replaces it with a text value consisting of the digits of the result. The digits are generated in the current radix (default decimal).

This application of the expansion operator is useful when defining a text macro, as the following lines show. Notice how you can enclose expressions with parentheses to make them more readable:

```
a       TEXTEQU <3 + 4>         ; a = "3 + 4"
b       TEXTEQU %3 + 4          ; b = "7"
c       TEXTEQU %(3 + 4)        ; c = "7"
```

When assigning text macros, you can use numeric equates in the constant expressions, but not text macros:

```
num     EQU     4               ; num = 4
numstr  TEXTEQU <4>             ; numstr = <4>
a       TEXTEQU %3 + num        ; a = <7>
b       TEXTEQU %3 + numstr     ; b = <7>
```

The expansion operator gives you flexibility when passing arguments to macros. It lets you pass a computed value rather than the literal text of an expression. The following example illustrates by defining a macro

```
work    MACRO   arg
    mov ax, arg * 4
ENDM
```

which accepts different arguments:

```
        work    2 + 3           ; Passes "2 + 3"
                                ; Code: mov ax, 2 + (3 * 4)
        work    %2 + 3          ; Passes 5
                                ; Code: mov ax, 5 * 4
        work    2 + num         ; Passes "2 + num"
        work    %2 + num        ; Passes "6"
        work    2 + numstr      ; Passes "2 + numstr"
        work    %2 + numstr     ; Passes "6"
```

You must consider operator precedence when using the expansion operator. Parentheses inside the macro can force evaluation in a desired order:

```
work    MACRO   arg
    mov ax, (arg) * 4
ENDM

        work    2 + 3           ; Code: mov ax, (2 + 3) * 4
        work    %2 + 3          ; Code: mov ax, (5) * 4
```

Several other uses for the expansion operator are reviewed in "Returning Values with Macro Functions," later in this chapter.

Expansion Operator as First Character on a Line

The expansion operator has a different meaning when used as the first character on a line. In this case, it instructs the assembler to expand any text macros and macro functions it finds on the rest of the line.

This feature makes it possible to use text macros with directives such as **ECHO**, **TITLE**, and **SUBTITLE**, which take an argument consisting of a single text value. For instance, **ECHO** displays its argument to the standard output device during assembly. Such expansion can be useful for debugging macros and expressions, but the requirement that its argument be a single text value may have unexpected results. Consider this example:

```
    ECHO    Bytes per element: %(SIZEOF array / LENGTHOF array)
```

Instead of evaluating the expression, this line echoes it:

```
Bytes per element: %(SIZEOF array / LENGTHOF array)
```

However, you can achieve the desired result by assigning the text of the expression to a text macro and then using the expansion operator at the beginning of the line to force expansion of the text macro.

```
temp    TEXTEQU %(SIZEOF array / LENGTHOF array)
%       ECHO    Bytes per element: temp
```

Note that you cannot get the same results simply by putting the **%** at the beginning of the first echo line, because **%** expands only text macros, not numeric equates or constant expressions.

Here are more examples of the expansion operator at the start of a line:

```
; Assume memmod, lang, and os specified with /D option
%   SUBTITLE  Model: memmod  Language: lang  Operating System: os

; Assume num defined earlier
tnum    TEXTEQU %num
%       .ERRE   num LE 255, <Failed because tnum !> 255>
```

Substitution Operator

References to a parameter within a macro can sometimes be ambiguous. In such cases, the assembler may not expand the argument as you intend. The substitution operator (**&**) lets you identify unambiguously any parameter within a macro.

As an example, consider the following macro:

```
errgen  MACRO   num, msg
    PUBLIC  errnum
    errnum BYTE    “Error num: msg”
ENDM
```

This macro is open to several interpretations:

- Is `errnum` a distinct word or the word `err` next to the parameter `num`?
- Should `num` and `msg` within the string be treated literally as part of the string or as arguments?

In each case, the assembler chooses the most literal interpretation. That is, it treats `errnum` as a distinct word, and `num` and `msg` as literal parts of the string.

The substitution operator can force different interpretations. If we rewrite the macro with the **&** operator, it looks like this:

```
errgen  MACRO   num, msg
    PUBLIC  err&num
    err&num BYTE    “Error &num: &msg”
ENDM
```

When called with the following arguments,

```
errgen  5, <Unreadable disk>
```

the macro now generates this code:

```
        PUBLIC  err5
err5    BYTE    "Error 5: Unreadable disk"
```

When it encounters the **&** operator, the assembler interprets subsequent text as a parameter name until the next **&** or until the next separator character (such as a space, tab, or comma). Thus, the assembler correctly parses the expression `err&num` because `num` is delimited by **&** and a space. The expression could also be written as `err&num&`, which again unambiguously identifies `num` as a parameter.

The rule also works in reverse. You can delimit a parameter reference with **&** at the end rather than at the beginning. For example, if `num` is 5, the expression `num&12` resolves to "512."

The assembler processes substitution operators from left to right. This can have unexpected results when you are pasting together two macro parameters. For example, if `arg1` has the value `var` and `arg2` has the value `3`, you could paste them together with this statement:

```
&arg1&&arg2&    BYTE    "Text"
```

Eliminating extra substitution operators, you might expect the following to be equivalent:

```
&arg1&arg2      BYTE    "Text"
```

However, this actually produces the symbol `vararg2,` because in processing from left to right, the assembler associates both the first and the second **&** symbols with the first parameter. The assembler replaces `&arg1&` by `var`, producing `vararg2`. The `arg2` is never evaluated. The correct abbreviation is:

```
arg1&&arg2      BYTE    "Text"
```

which produces the desired symbol `var3`. The symbol `arg1&&arg2` is replaced by `var&arg2`, which is replaced by `var3`.

The substitution operator is also necessary if you want to substitute a text macro inside quotes. For example,

```
arg     TEXTEQU <hello>
%echo   This is a string “&arg” ; Produces: This is a string “hello”
%echo   This is a string “arg”  ; Produces: This is a string “arg”
```

You can also use the substitution operator in lines beginning with the expansion operator (**%**) symbol, even outside macros (see page 236). It may be necessary to use the substitution operator to paste text macro names to adjacent characters or symbol names, as shown here:

```
text    TEXTEQU <var>
value   TEXTEQU %5
%       ECHO    textvalue is text&&value
```

This echoes the message

```
textvalue is var5
```

Macro substitution always occurs before evaluation of the high-level control structures. The assembler may therefore mistake a bit-test operator (**&**) in your macro for a substitution operator. You can guarantee the assembler correctly recognizes a bit-test operator by enclosing its operands in parentheses, as shown here:

```
test    MACRO   x
    .IF ax==&x      ; &x substituted with parameter value
    mov     ax, 10
    .ELSEIF ax&(x)  ; & is bitwise AND
    mov ax, 20
    .ENDIF
ENDM
```

The rules for using the substitution operator have changed significantly since MASM 5.1, making macro behavior more consistent and flexible. If you have macros written for MASM 5.1 or earlier, you can specify the old behavior by using **OLDMACROS** or **M510** with the **OPTION** directive (see page 24).

Defining Repeat Blocks with Loop Directives

A “repeat block” is an unnamed macro defined with a loop directive. The loop directive generates the statements inside the repeat block a specified number of times or until a given condition becomes true.

MASM provides several loop directives, which let you specify the number of loop iterations in different ways. Some loop directives can also accept arguments for each iteration. Although the number of iterations is usually specified in the directive, you can use the **EXITM** directive to exit the loop early.

Repeat blocks can be used outside macros, but they frequently appear inside macro definitions to perform some repeated operation in the macro. Since repeat blocks are macros themselves, they end with the **ENDM** directive.

This section explains the following four loop directives: **REPEAT**, **WHILE**, **FOR**, and **FORC**. In versions of MASM prior to 6.0, **REPEAT** was called **REPT**, **FOR** was called **IRP**, and **FORC** was called **IRPC**. MASM 6.1 recognizes the old names.

The assembler evaluates repeat blocks on the first pass only. You should therefore avoid using address spans as loop counters, as in this example:

```
REPEAT  (OFFSET label1 - OFFSET label2) ; Don't do this!
```

Since the distance between two labels may change on subsequent assembly passes as the assembler optimizes code, you should not assume that address spans remain constant between passes.

Note The **REPEAT** and **WHILE** directives should not be confused with the **REPEAT** and **WHILE** directives (see "Loop-Generating Directives" in Chapter 7), which generate loop and jump instructions for run-time program control.

REPEAT Loops

REPEAT is the simplest loop directive. It specifies the number of times to generate the statements inside the macro. The syntax is:

REPEAT *constexpr*
statements
ENDM

The *constexpr* can be a constant or a constant expression, and must contain no forward references. Since the repeat block expands at assembly time, the number of iterations must be known then.

Here is an example of a repeat block used to generate data. It initializes an array containing sequential ASCII values for all uppercase letters.

```
alpha   LABEL   BYTE            ;  Name the data generated
letter  =       'A'             ;  Initialize counter
REPEAT  26                      ;; Repeat for each letter
    BYTE    letter              ;; Allocate ASCII code for letter
    letter  = letter + 1        ;; Increment counter
ENDM
```

Here is another use of **REPEAT**, this time inside a macro:

```
beep    MACRO   iter:=<3>
    mov ah, 2                   ;; Character output function
    mov dl, 7                   ;; Bell character
    REPEAT iter                 ;; Repeat number specified by macro
        int 21h                 ;; Call DOS
    ENDM
ENDM
```

WHILE Loops

The **WHILE** directive is similar to **REPEAT**, but the loop continues as long as a given condition is true. The syntax is:

WHILE *expression*
statements
ENDM

The *expression* must be a value that can be calculated at assembly time. Normally, the expression uses relational operators, but it can be any expression that evaluates to zero (false) or nonzero (true). Usually, the condition changes during the evaluation of the macro so that the loop won't attempt to generate an infinite amount of code. However, you can use the **EXITM** directive to break out of the loop.

The following repeat block uses the **WHILE** directive to allocate variables initialized to calculated values. This is a common technique for generating lookup tables. (A lookup table is any list of precalculated results, such as a table of interest payments or trigonometric values or logarithms. Programs optimized for speed often use lookup tables, since calculating a value often takes more time than looking it up in a table.)

```
cubes   LABEL   BYTE            ;; Name the data generated
root    =   1                   ;; Initialize root
cube    =   root * root * root  ;; Calculate first cube
WHILE   cube LE 32767           ;; Repeat until result too large
    WORD   cube                 ;; Allocate cube
    root   =    root + 1        ;; Calculate next root and cube
    cube   =    root * root * root
ENDM
```

FOR Loops and Variable-Length Parameters

With the **FOR** directive you can iterate through a list of arguments, working on each of them in turn. It has the following syntax:

FOR *parameter*, *<argumentlist>*
statements
ENDM

The *parameter* is a placeholder that represents the name of each argument inside the **FOR** block. The argument list must contain comma-separated arguments and must always be enclosed in angle brackets. Here's an example of a **FOR** block:

```
series  LABEL   BYTE
FOR     arg, <1,2,3,4,5,6,7,8,9,10>
    BYTE  arg DUP (arg)
ENDM
```

On the first iteration, the `arg` parameter is replaced with the first argument, the value 1. On the second iteration, `arg` is replaced with 2. The result is an array with the first byte initialized to 1, the next 2 bytes initialized to 2, the next 3 bytes initialized to 3, and so on.

The argument list is given specifically in this example, but in some cases the list must be generated as a text macro. The value of the text macro must include the angle brackets.

```
arglist TEXTEQU <!<3,6,9!>>     ; Generate list as text macro
%FOR  arg, arglist
    .                           ; Do something to arg
    .
    .
ENDM
```

Note the use of the literal character operator (**!**) to identify angle brackets as characters, not delimiters. See "Text Delimiters (<>) and the Literal-Character Operator," earlier in this chapter.

The **FOR** directive also provides a convenient way to process macros with a variable number of arguments. To do this, add **VARARG** to the last parameter to indicate that a single named parameter will have the actual value of all additional arguments. For example, the following macro definition includes the three possible parameter attributes—required, default, and variable.

```
work    MACRO   rarg:REQ, darg:=<5>, varg:VARARG
```

The variable argument must always be last. If this macro is called with the statement

```
work 4, , 6, 7, a, b
```

the first argument is received as the value 4, the second is replaced by the default value 5, and the last four are received as the single argument `<6, 7, a, b>`. This is the same format expected by the **FOR** directive. The **FOR** directive discards leading spaces but recognizes trailing spaces.

The following macro illustrates variable arguments:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

When called with

```
show  'O', 'K', 13, 10
```

the macro displays each of the specified characters one at a time.

The parameter in a **FOR** loop can have the required or default attribute. You can modify the `show` macro to make blank arguments generate errors:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg:REQ, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

The macro now generates an error if called with

```
show  'O',, 'K', 13, 10
```

Another approach would be to use a default argument:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg:=<‘ ’>, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

Now calling the macro with

```
        show  ‘O’,, ‘K’, 13, 10
```

inserts the default character, a space, for the blank argument.

FORC Loops

The **FORC** directive is similar to **FOR,** but takes a string of text rather than a list of arguments. The statements are assembled once for each character (including spaces) in the string, substituting a different character for the parameter each time through.

The syntax looks like this:

FORC *parameter*, < *text*>
statements
ENDM

The *text* must be enclosed in angle brackets. The following example illustrates **FORC**:

```
FORC arg, <ABCDEFGHIJKLMNOPQRSTUVWXYZ>
    BYTE  ‘&arg’             ;; Allocate uppercase letter
    BYTE  ‘&arg’ + 20h       ;; Allocate lowercase letter
    BYTE  ‘&arg’ - 40h       ;; Allocate ordinal of letter
ENDM
```

Notice that the substitution operator must be used inside the quotation marks to make sure that `arg` is expanded to a character rather than treated as a literal string.

With versions of MASM earlier than 6.0, **FORC** is often used for complex parsing tasks. A long sentence can be examined character by character. Each character is then either thrown away or pasted onto a token string, depending on whether it is a separator character. The new predefined macro functions and string processing directives discussed in the following section are usually more efficient for these tasks.

String Directives and Predefined Functions

The assembler provides four directives for manipulating text:

Directive	Description
SUBSTR	Assigns part of string to a new symbol.
INSTR	Searches for one string within another.
SIZESTR	Determines the size of a string.
CATSTR	Concatenates one or more strings to a single string.

These directives assign a processed value to a text macro or numeric equate. For example, the following lines

```
num     =       7
newstr  CATSTR  <3 + >, %num, < = > , %3 + num ; "3 + 7 = 10"
```

assign the string `"3 + 7 = 10"` to `newstr`. **CATSTR** and **SUBSTR** assign text in the same way as the **TEXTEQU** directive. **SIZESTR** and **INSTR** assign a number in the same way as the **=** operator. The four string directives take only text values as arguments. Use the expansion operator (**%**) when you need to make sure that constants and numeric equates expand to text, as shown in the preceding lines.

Each of the string directives has a corresponding predefined macro function version: **@SubStr**, **@InStr**, **@SizeStr**, and **@CatStr**. Macro functions are similar to the string directives, but you must enclose their arguments in parentheses. Macro functions return text values and can appear in any context where text is expected. The following section, "Returning Values with Macro Functions," tells how to write your own macro functions. The following example is equivalent to the previous **CATSTR** example:

```
num     =       7
newstr  TEXTEQU @CatStr( <3 + >, %num, < = > , %3 + num )
```

Macro functions are often more convenient than their directive counterparts because you can use a macro function as an argument to a string directive or to another macro function. Unlike string directives, predefined macro function names are case sensitive when you use the /Cp command-line option.

Each string directive and predefined function acts on a string, which can be any *textItem*. The *textItem* can be text enclosed in angle brackets (< >), the name of a text macro, or a constant expression preceded by **%** (as in **%constExpr**). Refer to Appendix B, "BNF Grammar," for a list of types that *textItem* can represent.

The following sections summarize the syntax for each of the string directives and functions. The explanations focus on the directives, but the functions work the same except where noted.

SUBSTR

name **SUBSTR** *string*, *start*[[, length]]
@SubStr(*string*, *start*[[, *length*]])

The **SUBSTR** directive assigns a substring from a given *string* to the symbol *name*. The *start* parameter specifies the position in *string*, beginning with 1, to start the substring. The *length* gives the length of the substring. If you do not specify *length*, **SUBSTR** returns the remainder of the string, including the *start* character.

INSTR

name **INSTR** [[*start*,]] *string*, *substring*
@InStr([[*start*]], *string*, *substring* **)**

The **INSTR** directive searches a specified *string* for an occurrence of *substring* and assigns its position number to *name*. The search is case sensitive. The *start* parameter is the position in *string* to start the search for *substring*. If you do not specify *start*, it is assumed to be position 1, the start of the string. If **INSTR** does not find *substring*, it assigns position 0 to *name*.

The **INSTR** directive assigns the position value *name* as if it were a numeric equate. In contrast, the **@InStr** returns the value as a string of digits in the current radix.

The **@InStr** function has a slightly different syntax than the **INSTR** directive. You can omit the first argument and its associated comma from the directive. You can leave the first argument blank with the function, but a blank function argument must still have a comma. For example,

```
pos     INSTR   <person>, <son>
```

is the same as

```
pos     = @InStr( , <person>, <son> )
```

You can also assign the return value to a text macro, like this:

```
strpos  TEXTEQU @InStr( , <person>, <son> )
```

SIZESTR

name **SIZESTR** *string*
@SizeStr(*string* **)**

The **SIZESTR** directive assigns the number of characters in *string* to *name*. An empty string returns a length of zero. The **SIZESTR** directive assigns the size value to a name as if it were a numeric equate. The **@SizeStr** function returns the value as a string of digits in the current radix.

CATSTR

name CATSTR *string*[, *string*]...
@CatStr(*string*[, *string*]... **)**

The **CATSTR** directive concatenates a list of text values into a single text value and assigns it to *name*. **TEXTEQU** is technically a synonym for **CATSTR**. **TEXTEQU** is normally used for single-string assignments, while **CATSTR** is used for multistring concatenations.

The following example pushes and pops one set of registers, illustrating several uses of string directives and functions:

```
; SaveRegs - Macro to generate a push instruction for each
; register in argument list. Saves each register name in the
; regpushed text macro.
regpushed TEXTEQU <>                    ;; Initialize empty string

SaveRegs MACRO regs:VARARG
    LOCAL reg
    FOR reg, <regs>                     ;; Push each register
        push reg                        ;;   and add it to the list
        regpushed CATSTR   <reg>, <,>, regpushed
    ENDM                                ;; Strip off last comma
    regpushed CATSTR <!<>, regpushed    ;; Mark start of list with <
    regpushed SUBSTR regpushed, 1, @SizeStr( regpushed )
    regpushed CATSTR regpushed, <!>>    ;; Mark end with >
ENDM

; RestoreRegs - Macro to generate a pop instruction for registers
; saved by the SaveRegs macro. Restores one group of registers.

RestoreRegs MACRO
    LOCAL reg
    %FOR reg, regpushed                  ;; Pop each register
    pop reg
    ENDM
ENDM
```

Notice how the `SaveRegs` macro saves its result in the `regpushed` text macro for later use by the `RestoreRegs` macro. In this case, a text macro is used as a global variable. By contrast, the `reg` text macro is used only in `RestoreRegs`. It is declared **LOCAL** so it won't take the name `reg` from the global name space. The MACROS.INC file provided with MASM 6.1 includes expanded versions of these same two macros.

Returning Values with Macro Functions

A macro function is a named group of statements that returns a value. When calling a macro function, you must enclose its argument list in parentheses, even if the list is empty. The function always returns text.

MASM 6.1 provides several predefined macro functions for common tasks. The predefined macros include **@Environ** (see page 10) and the string functions **@SizeStr**, **@CatStr**, **@SubStr**, and **@InStr** (discussed in the preceding section).

You define macro functions in exactly the same way as macro procedures, except that a macro function always returns a value through the **EXITM** directive. Here is an example:

```
DEFINED MACRO   symbol:REQ
    IFDEF symbol
        EXITM <-1>              ;; True
    ELSE
        EXITM <0>               ;; False
    ENDIF
ENDM
```

This macro works like the **defined** operator in the C language. You can use it to test the defined state of several different symbols with a single statement, as shown here:

```
IF DEFINED( DOS ) AND NOT DEFINED( XENIX )
    ;; Do something
ENDIF
```

Notice that the macro returns integer values as strings of digits, but the **IF** statement evaluates numeric values or expressions. There is no conflict because the assembler sees the value returned by the macro function exactly as if the user had typed the values directly into the program:

```
IF -1 AND NOT 0
```

Returning Values with EXITM

The return value must be text, a text equate name, or the result of another macro function. A macro function must first convert a numeric value—such as a constant, a numeric equate, or the result of a numeric expression—before returning it. The macro function can use angle brackets or the expansion operator (**%**) to convert numbers to text. The `DEFINED` macro, for instance, could have returned its value as

```
        EXITM   %-1
```

Here is another example of a macro function that uses the **WHILE** directive to calculate factorials:

```
factorial   MACRO   num:REQ
    LOCAL   i, factor
    factor  =   num
    i       =   1
    WHILE   factor GT 1
        i       =   i * factor
        factor  =   factor - 1
    ENDM
    EXITM   %i
ENDM
```

The integer result of the calculation is changed to a text string with the expansion operator (**%**). The `factorial` macro can define data, as shown here:

```
var     WORD    factorial( 4 )
```

This statement initializes `var` with the number 24 (the factorial of 4).

Using Macro Functions with Variable-Length Parameter Lists

You can use the **FOR** directive to handle macro parameters with the **VARARG** attribute. “FOR Loops and Variable-Length Parameters,” page 242, explains how to do this in simple cases where the variable parameters are handled sequentially, from first to last. However, you may sometimes need to process the parameters in reverse order or nonsequentially. Macro functions make these techniques possible.

For example, the following macro function determines the number of arguments in a **VARARG** parameter:

```
@ArgCount MACRO arglist:VARARG
    LOCAL count
    count = 0
    FOR arg, <arglist>
        count = count + 1       ;; Count the arguments
    ENDM
    EXITM %count
ENDM
```

You can use `@ArgCount` inside a macro that has a **VARARG** parameter, as shown here:

```
work    MACRO args:VARARG
%   ECHO Number of arguments is: @ArgCount( args )
ENDM
```

Another useful task might be to select an item from an argument list using an index to indicate the item. The following macro simplifies this.

```
@ArgI MACRO index:REQ, arglist:VARARG
    LOCAL count, retstr
    retstr TEXTEQU <>           ;; Initialize count
    count  = 0                  ;; Initialize return string
    FOR arg, <arglist>
        count = count + 1
        IF count EQ index       ;; Item is found
            retstr TEXTEQU <arg> ;; Set return string
            EXITM               ;;   and exit IF
        ENDIF
    ENDM
    EXITM retstr                ;; Exit function
ENDM
```

You can use `@ArgI` like this:

```
work    MACRO args:VARARG
%   ECHO Third argument is: @ArgI( 3, args )
ENDM
```

Finally, you might need to process arguments in reverse order. The following macro returns a new argument list in reverse order.

```
@ArgRev MACRO arglist:REQ
    LOCAL txt, arg
    txt TEXTEQU <>
%   FOR arg, <arglist>
        txt CATSTR <arg>, <,>, txt      ;; Paste each onto list
    ENDM
                                        ;; Remove terminating comma
    txt SUBSTR  txt, 1, @SizeStr( %txt ) - 1
    txt CATSTR  <!<>, txt, <!>>         ;; Add angle brackets
    EXITM txt
ENDM
```

Here is an example showing `@ArgRev` in use:

```
work    MACRO   args:VARARG
%   FOR  arg, @ArgRev( <args> )   ;; Process in reverse order
        ECHO    arg
    ENDM
ENDM
```

These three macro functions appear in the MACROS.INC include file, located on one of the MASM distribution disks.

Expansion Operator in Macro Functions

This list summarizes the behavior of the expansion operator (**%**) with macro functions.

- If a macro function is preceded by a **%**, it will be expanded. However, if it expands to a text macro or a macro function call, it will not expand further.
- If you use a macro function call as an argument for another macro function call, a **%** is not needed.
- If a macro function is called inside angle brackets and is preceded by **%**, it will be expanded.

Advanced Macro Techniques

The concept of replacing macro names with predefined macro text is simple in theory, but it has many implications and complications. Here is a brief summary of some advanced techniques you can use in macros.

Defining Macros within Macros

Macros can define other macros, a technique called “nesting macros.” MASM expands macros as it encounters them, so nested macros are always processed in nesting order. You cannot reference a nested macro directly in your program, since the assembler begins expansion from the outer macro. In effect, a nested macro is local to the macro that defines it. Only the amount of available memory limits the number of macros a program can nest.

The following example demonstrates how one macro can define another. The macro takes as an argument the name of a shift or rotate instruction, then creates another macro that simplifies the instruction for 8088/86 processors.

```
shifts  MACRO   opname                  ;; Macro generates macros
    opname&s    MACRO operand:REQ, rotates:=<1>
        IF rotates LE 2                ;; One at a time is faster
            REPEAT rotate              ;;   for 2 or less
                opname  operand, 1
            ENDM
        ELSE                           ;; Using CL is faster for
            mov     cl, rotates        ;;   more than 2
            opname  operand, cl
        ENDIF
    ENDM
ENDM
```

Recall that the 8086 processor allows only 1 or CL as an operand for shift and rotate instructions. Expanding `shifts` generates a macro for the shift instruction that uses whichever operand is more efficient. You create the entire series of macros, one for each shift instruction, like this:

```
        ; Call macro repeatedly to make new macros
        shifts  ror                    ; Generates rors
        shifts  rol                    ; Generates rols
        shifts  shr                    ; Generates shrs
        shifts  shl                    ; Generates shls
        shifts  rcl                    ; Generates rcls
        shifts  rcr                    ; Generates rcrs
        shifts  sal                    ; Generates sals
        shifts  sar                    ; Generates sars
```

Then use the new macros as replacements for shift instructions, like this:

```
        shrs    ax, 5
        rols    bx, 3
```

Testing for Argument Type and Environment

Macros can expand conditional blocks of code by testing for argument type with the **OPATTR** operator. **OPATTR** returns a single word constant that indicates the type and scope of an expression, like this:

OPATTR *expression*

If *expression* is not valid or is forward-referenced, **OPATTR** returns a 0. Otherwise, the return value incorporates the bit flags shown in the table below.

OPATTR serves as an enhanced version of the **.TYPE** operator, which returns only the low byte (bits 0–7) shown in the table. Bits 11–15 of the return value are undefined.

Bit	Set If expression
0	References a code label
1	Is a memory variable or has a relocatable data label
2	Is an immediate value
3	Uses direct memory addressing
4	Is a register value
5	References no undefined symbols and is without error
6	Is relative to SS
7	References an external label
8–10	Has the following language type: • 000—No language type • 001—C • 010—SYSCALL • 011—STDCALL • 100—Pascal • 101—FORTRAN • 110—Basic

A macro can use **OPATTR** to determine if an argument is a constant, a register, or a memory operand. With this information, the macro can conditionally generate the most efficient code depending on argument type.

For example, given a constant argument, a macro can test it for 0. Depending on the argument's value, the code can select the most effective method to load the value into a register:

```
IF CONST
mov     bx, CONST    ; If CONST > 0, move into BX
ELSE
sub     bx, bx       ; More efficient if CONST = 0
ENDIF
```

The second method is faster than the first, yet has the same result (with the byproduct of changing the processor flags).

The following macro illustrates some techniques using **OPATTR** by loading an address into a specified offset register:

```
load    MACRO reg:REQ, adr:REQ
    IF (OPATTR (adr)) AND 00010000y    ;; Register
        IFDIFI reg, adr                ;; Don't load register
            mov     reg, adr           ;;   onto itself
        ENDIF
    ELSEIF (OPATTR (adr)) AND 00000100y
        mov     reg, adr               ;; Constant
    ELSEIF (TYPE (adr) EQ BYTE) OR (TYPE (adr) EQ SBYTE)
        mov    reg, OFFSET adr         ;; Bytes
    ELSEIF (SIZE (TYPE (adr)) EQ 2
        mov    reg, adr                ;; Near pointer
    ELSEIF (SIZE (TYPE (adr)) EQ 4
        mov    reg, WORD PTR adr[0]    ;; Far pointer
        mov    ds,  WORD PTR adr[2]
    ELSE
        .ERR <Illegal argument>
    ENDIF
ENDM
```

A macro also can generate different code depending on the assembly environment. The predefined text macro **@Cpu** returns a flag for processor type. The following example uses the more efficient constant variation of the **PUSH** instruction if the processor is an 80186 or higher.

```
IF  @Cpu AND 00000010y
    pushc  MACRO op             ;; 80186 or higher
        push op
    ENDM
ELSE
    pushc  MACRO op             ;; 8088/8086
        mov  ax, op
        push ax
    ENDM
ENDIF
```

Another macro can now use `pushc` rather than conditionally testing for processor type itself. Although either case produces the same code, using `pushc` assembles faster because the environment is checked only once.

You can test the language and operating system using the **@Interface** text macro. The memory model can be tested with the **@Model**, **@DataSize**, or **@CodeSize** text macros.

You can save the contexts inside macros with **PUSHCONTEXT** and **POPCONTEXT**. The options for these keywords are:

Option	Description
ASSUMES	Saves segment register information
RADIX	Saves current default radix
LISTING	Saves listing and CREF information
CPU	Saves current CPU and processor
ALL	All of the above

Using Recursive Macros

Macros can call themselves. In MASM 5.1 and earlier, recursion is an important technique for handling variable arguments. MASM 6.1 handles variable arguments much more cleanly with the **FOR** directive and the **VARARG** attribute, as described in "FOR Loops and Variable-Length Parameters," earlier in this chapter. However, recursion is still available and may be useful for some macros.