7.1 Introduction

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

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

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

7.2 Conditional-Assembly Directives

The conditional-assembly directives include the following:

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

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

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

IF
statements
[**ELSE**
statements]
ENDIF

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

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

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

7.2.1 IF and IFE Directives

Syntax

IF *expression*
IFE *expression*

The **IF** and **IFE** directives test the value of an *expression*. The **IF** directive grants assembly if the value of *expression* is true (nonzero). The **IFE** directive grants assembly if the value of *expression* is false (0). The *expression* must resolve to an absolute value and must not contain forward references.

Example

```
IF      debug
        EXTRN dump:FAR
        EXTRN trace:FAR
        EXTRN breakpoint:FAR
ENDIF
```

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

7.2.2 IF1 and IF2 Directives

Syntax

IF1
IF2

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

Example

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

7.2.3 IFDEF and IFNDEF Directives

Syntax

IFDEF *name*
IFNDEF *name*

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

The name can be any valid name. Note that if *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

Example

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

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

```
MASM test /Dbuffer;
```

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

```
MASM test;
```

7.2.4 IFB and IFNB Directives

Syntax

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

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

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

Example

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

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

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

7.2.5 IFIDN and IFDIF Directives

Syntax

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

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

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

Example

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

divide    6,%test
```

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

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

```
test        EQU     0
```

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

```
test        DW      0
```

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

7.3 Conditional Error Directives

Conditional error directives can be used to debug programs and check for assembly-time errors. By inserting a conditional error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional error directives to test for boundary conditions in macros.

The conditional error directives, and the errors they produce, are listed in Table 7.1.

Table 7.1

Conditional Error Directives

Directive	Number	Message
.ERR1	87	`Forced error - pass1`
.ERR2	88	`Forced error - pass2`
.ERR	89	`Forced error`
.ERRE	90	`Forced error - expression equals 0`
.ERRNZ	91	`Forced error - expression not equal 0`
.ERRNDEF	92	`Forced error - symbol not defined`
.ERRDEF	93	`Forced error - symbol defined`
.ERRB	94	`Forced error - string blank`
.ERRNB	95	`Forced error - string not blank`
.ERRIDN	96	`Forced error - strings identical`
.ERRDIF	97	`Forced error - strings different`

Like other fatal assembler errors, those generated by conditional error directives cause the assembler to return exit code 7. If a fatal error is encountered during assembly, **MASM** will delete the object module. All conditional error directives except **ERR1** generate fatal errors.

7.3.1 .ERR, .ERR1, and .ERR2 Directives

Syntax

.ERR
.ERR1
.ERR2

The **.ERR**, **.ERR1**, and **.ERR2** directives force an error at the points at which they occur in the source file. The **.ERR** directive forces an error regardless of the pass, while the **.ERR1** and **.ERR2** directives force the error only on their respective passes. The **.ERR1** directive only appears on the screen or in the listing file if you use the **/D** option to request a Pass 1 listing. Unlike other conditional error directives, it is not a fatal error.

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

Example

```
IFDEF dos
      .
      .
      .
ELSE
      IFDEF  xenix
      .
      .
      .
      ELSE
      .ERR
      ENDIF
ENDIF
```

This example makes sure that either the symbol `dos` or the symbol `xenix` is defined. If neither is defined, the nested **ELSE** condition is assembled and an error message is generated. Since the **.ERR** directive is used, an error would be generated on each pass. You could use the **.ERR2** directive if you wanted only a fatal error, or you could use the **.ERR1** directive if you wanted only a warning error.

7.3.2 .ERRE and .ERRNZ Directives

Syntax

.ERRE *expression*
.ERRNZ *expression*

The **.ERRE** and **.ERRNZ** directives test the value of an *expression*. The **.ERRE** directive generates an error if the *expression* is false (0). The **.ERRNZ** directive generates an error if the *expression* is true (nonzero). The *expression* must resolve to an absolute value and must not contain forward references.

Example

```
buffer  MACRO  count,bname
        .ERRE  count LE 128      ;; Allocate memory, but
        bname  DB    count DUP(0);;   no more than 128 bytes
        ENDM

buffer  128,buf1        ; Data allocated - no error
buffer  129,buf2        ; Error generated
```

In this example, the **.ERRE** directive is used to check the boundaries of a parameter passed to the macro `buffer`. If `count` is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If `count` is greater than 128, the expression will be false (0) and the error will be generated.

7.3.3 .ERRDEF and .ERRNDEF Directives

Syntax

.ERRDEF *name*
.ERRNDEF *name*

The **.ERRDEF** and **.ERRNDEF** directives test whether or not *name* has been defined. The **.ERRDEF** directive produces an error if *name* is defined as a label, variable, or symbol. The **.ERRNDEF** directive produces an error if *name* has not yet been defined. If *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

Example

```
.ERRDEF  symbol
IFDEF    config1
         .
         .symbol  EQU  0
         .
ENDIF
IFDEF    config2
         .
         .symbol  EQU  1
         .
ENDIF
.ERRNDEF symbol
```

In this example, the **.ERRDEF** directive at the beginning of the conditional blocks makes sure that `symbol` has not been defined before entering the blocks. The **.ERRNDEF** directive at the end ensures that `symbol` was defined somewhere within the blocks.

7.3.4 .ERRB and .ERRNB Directives

Syntax

.ERRB <*string*>
.ERRNB <*string*>

The **.ERRB** and **.ERRNB** directives test the given *string*. The **.ERRB** directive generates an error if *string* is blank. The **.ERRNB** directive generates an error if *string* is not blank. The string can be any name, number, or expression. The angle brackets (< >) are required.

These conditional error directives can be used within macros to test for the existence of parameters.

Example

```
work  MACRO  realarg,testarg
      .ERRB  <realarg>      ;; Error if no parameters
      .ERRNB <testarg>      ;; Error if more than one parameter
      .
      .
      .
      ENDM
```

In this example, error directives are used to make sure that one, and only one, argument is passed to the macro. The **.ERRB** directive generates an error if no argument is passed to the macro. The **.ERRNB** directive generates an error if more than one argument is passed to the macro.

7.3.5 .ERRIDN and .ERRDIF Directives

Syntax

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

The **.ERRIDN** and **.ERRDIF** directives test whether two strings are identical. The **.ERRIDN** directive generates an error if the strings are identical. The **.ERRDIF** generates an error if the strings are different. The strings can be names, numbers, or expressions. To be identical, each character in *string1* must match the corresponding character in *string2*. String checks are case-sensitive. The angle brackets (< >) are required.

Example

```
addem    MACRO ad1,ad2,sum
         .ERRIDN <ax>,<ad2> ;; Error if ad2 is 'ax'
         .ERRIDN <AX>,<ad2> ;; Error if ad2 is 'AX'
         mov   ax,ad1       ;; Would overwrite if ad2 were AX
         add   ax,ad2
         mov   sum,ax       ;; Sum must be register or memory
         ENDM
```

In this example, the **.ERRIDN** directive is used to protect against passing the **AX** register as the second parameter, because the macro won't work if the **AX** register is passed as the second parameter. Note that the directive is used twice to protect against the two most likely spellings.

Chapter 8
Macro Directives

8.1 Introduction 117
8.2 Macro Directives 117
8.2.1 MACRO and ENDM Directives 118
8.2.2 Macro Calls 121
8.2.3 LOCAL Directive 122
8.2.4 PURGE Directive 123
8.2.5 REPT and ENDM Directives 124
8.2.6 IRP and ENDM Directives 125
8.2.7 IRPC and ENDM Directives 126
8.2.8 EXITM Directive 127
8.3 Macro Operators 128
8.3.1 Substitute Operator 129
8.3.2 Literal-Text Operator 130
8.3.3 Literal-Character Operator 131
8.3.4 Expression Operator 131
8.3.5 Macro Comment 132

8.1 Introduction

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

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

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

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

8.2 Macro Directives

The macro directives are listed below:

MACRO
ENDM
LOCAL
PURGE
REPT

IRP

IRPC

EXITM

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

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

8.2.1 MACRO and ENDM Directives

Syntax

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

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

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

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

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

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

Example

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

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

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

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

```
TITLE Macro File
```

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

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

Note

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

Note

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

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

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

8.2.2 Macro Calls

Syntax

name [*actualparameter*,,,]

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

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

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

Examples

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

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

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

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

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

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

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

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

8.2.3 LOCAL Directive

Syntax

LOCAL *dummyname,,,*

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

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

??number

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

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

Note

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

Example

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

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

8.2.4 PURGE Directive

Syntax

PURGE *macroname,,,*

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

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

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

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

Examples

```
PURGE    addup
```

The first example deletes the macro named `addup`.

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

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

8.2.5 REPT and ENDM Directives

Syntax

REPT *expression*
statements
ENDM

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

Example

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

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

8.2.6 IRP and ENDM Directives

Syntax

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

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

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

Example

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

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

Notes

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

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

If this macro is called with

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

the macro expansion becomes

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

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

8.2.7 IRPC and ENDM Directives

Syntax

IRPC *dummyname,string*
statements
ENDM

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

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

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

Example

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

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

8.2.8 EXITM Directive

Syntax

EXITM

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

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

Example

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

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

8.3 Macro Operators

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

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

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

8.3.1 Substitute Operator

Syntax

&dummyparameter

or

dummyparameter&

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

Example

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

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

```
          errgen  1,wait
```

the macro is expanded to

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

Note

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

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

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

In this example, the dummy parameter x is replaced immediately when the macro is called. The dummy parameter z, however, is not replaced until the **IRP** directive is processed. This means the parameter is replaced once for each number in the **IRP** parameter list. If the macro is called with

```
        alloc     var
```

the expanded macro will be

```
var1    DB        1
var2    DB        2
var3    DB        3
```

8.3.2 Literal-Text Operator

Syntax

<text>

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

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

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

8.3.3 Literal-Character Operator

Syntax

!*character*

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

8.3.4 Expression Operator

Syntax

%*text*

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

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

Example

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

sym1    EQU     100
sym2    EQU     200

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

In this example, the macro call

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

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

8.3.5 Macro Comment

Syntax

;;*text*

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

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