1

ADVANCED MACROS

Macro programming is typically considered an advanced operation in assembly language programming. However, there's basic macro programming, and then there's *advanced macro programming*. Basic macro programming, as covered in *Art of 64-Bit Assembly*, Volume 1 (*Ao64Av1*), simply uses macros as instruction replacements to reduce repetitive code in an assembly language program. Advanced macro programming goes beyond that to change the very syntax of the assembly language itself.

In this chapter, you'll explore the power of Microsoft Macro Assembler (MASM) macros as a foundation for building domain-specific languages in assembly. Starting with a refresher on macro syntax and structure, you'll dive into advanced macro operators and text-processing directives that enable flexible, readable, and maintainable code. You'll learn how to

simulate MASM directives, create function-like macros, and manipulate strings and symbols at compile time.

Along the way, you'll tackle practical techniques like macro-based namespaces, structured data declarations, and context-free macro constructs—all while keeping performance implications in mind. By the end, you'll be equipped to create expressive, high-level abstractions within MASM's low-level environment.

1.1 An Introduction to Domain-Specific Languages

A *domain-specific language (DSL)* is a special-purpose language created specifically for a certain problem domain. DSLs are often written in *very high-level languages (VHLLs)* such as spreadsheet programs, Haskell, lex/yacc, macro preprocessors (m4), and so on. The last example, macro preprocessors, is a key concept here. In this chapter, you'll use MASM macro capabilities to create some simple DSLs.

DSLs are often called *small languages*. Although you could write a relatively complete DSL to provide all the programming facilities required by that problem domain, most DSLs implement only the language features needed for the specific domain and fall back to another programming language for generic programming requirements. For example, the lex and yacc programming languages contain their own constructs to handle regular expressions and context-free grammars but fall back to C/C++ for normal programming operations.

MASM's macro language also lets you create small languages. When you're invoking one of your macros, you're using one of your DSL statements. When you drop back into assembly language (between the macro invocations), you're in the fallback situation. In theory, it might be possible to create an extremely comprehensive macro package that allows you to write complete programs by using nothing more than macro invocations. However, this situation is rare when using MASM macros.

1.2 A Review of the MASM Macro Declaration

Although *Ao64Av1* covered the MASM macro declaration in considerable detail, a brief review is in order. The standard MASM macro declaration takes the form

```
macroName   macro  zero_or_more_arguments
             .
             .
             .
            endm
```

where *macroName* is a user-specified (unique) identifier and *zero_or_more_arguments* is a list (possibly empty) of identifiers separated by commas. The argument names need not be unique throughout the source file, only within the macro itself.

Here is an example of a simple macro definition:

```
pushregs   macro
           push   rax
           push   rcx
           push   rdx
           endm
```

A macro invocation tells MASM to expand the body of the macro in place of the macro name appearing in the source file. For example, here is a macro invocation for the `pushregs` macro:

```
           pushregs
```

MASM will substitute the following lines of text for the macro invocation:

```
           push   rax
           push   rcx
           push   rdx
```

If arguments are present in the macro declaration, MASM allows the macro invocation to include arguments as well:

```
movem32    macro dest, src
           push  rax
           mov   eax, src
           mov   dest, eax
           pop   rax
           endm
            .
            .
            .
           movem32 i, j
```

When expanding the macro containing arguments, MASM substitutes the text for each argument in place of the corresponding formal macro parameter identifier wherever that identifier appears in the macro's body. For example, the macro invocation at the end of the previous example expands to the following:

```
           push  rax
           mov   eax, j
           mov   i, eax
           pop   rax
```

It's important to understand that MASM converts macro invocation arguments to text values before doing the macro expansion. In many ways, the macro invocation `movem32 i, j` is equivalent to the following:

```
movem32a   macro
           push  rax
```

```
            mov   eax, src
            mov   dest, eax
            pop   rax
            endm
             .
             .
             .
dest texteq <i>
src  texteq <j>
     movem32a
```

The macro invocation effectively creates text equates out of the macro arguments, and the macro body expands those text equates whenever it encounters them.

Note that the following is also a legal invocation of `movem32`:

```
     movem32 <i>, <j>
```

Enclosing `i` and `j` in the angle brackets explicitly specifies them as text literals. MASM still expands the text to `i` and `j` within the macro body's expansion.

1.2.1 MASM Text Literals

A MASM *text literal* is a sequence of characters delimited by the angle brackets < and >. Note that text literals are completely different from string literals, which you delimit using quotation marks (") or apostrophes ('). MASM text/string processing at compile time is always done on text values; string literals are character strings to be used by the runtime application.

Text literals can be useful if a particular macro argument must contain special symbols (such as a comma). For example, consider the following macro definition:

```
printm   macro arg
         call  print
         byte  arg
         byte  0
         endm
          .
          .
          .
         printm "Hello, World!", nl
```

Whoops! This produces a warning message complaining about too many arguments. However, you can correct this problem by using the angle-bracket text-literal delimiters around the `printm` operand, as follows:

```
         printm <"Hello, World!", nl>
```

This expands to

```
        call  print
        byte  "Hello, World!", nl
        byte  0
```

which gives what you want.

1.2.2 The Macro Text Expansion Operator (&)

Although text and string literals are fundamentally different entities to MASM, sometimes it would be nice to convert between the two. A convenient way to convert a text object to a string is to use `&`, the MASM *text expansion operator*. MASM supports two forms of the text expansion operator

```
&identifier
&identifier&
```

where *identifier* is a valid MASM identifier that's a macro argument. MASM will replace the identifier with the text associated with that identifier. Note that the `&` operator is valid only within a macro body.

The expansion of the argument text via `&` happens regardless of the context. Therefore, the following expansion takes place even though the expansion operator is inside a string literal:

```
tostr   macro  arg
        byte   "&arg"
        endm
         .
         .
         .
        tostr <x=y>  ; Expands to byte "x=y"
```

Placing the expansion operator before and after the identifier is useful when MASM might not be able to differentiate the identifier from any text immediately following it. For example, consider the following macro definition:

```
useName   macro  id
          mov    eax, &id&_i
          mov    &id&_j, eax
          endm
```

Invoking `useName x` emits the following code (taken directly from a MASM output listing):

```
                                        useName x
00000000  8B 05 00000000 R  1           mov    eax, x_i
00000006  89 05 00000004 R  1           mov    x_j, eax
```

The MASM program had to define the two symbols `x_i` and `x_j` prior to assembling this file. MASM expanded `&id&` to `x`, which was concatenated with `_i` or `_j` to produce the `x_i` and `x_j` symbols. Had this macro not used the `&` operator before and after `x`, MASM would have thought that `id_i` and `id_j` were the macro arguments, not `id` alone; MASM would have reported an error in that case.

If you pass a macro argument as an argument to another macro, MASM will automatically expand the invoking argument before passing it on to the second macro. For example:

```
mac3    macro  arg1
        tostr  arg1
        byte   0
        endm

tostr   macro  arg2
        byte   "&arg2"
        endm
```

A macro invocation of the form `mac3 <Hello>` produces the following expansion (as expected):

```
        byte   "Hello"
        byte   0
```

If you attempt to force the expansion of `arg1` in the invocation of `tostr` inside `mac3` (by specifying `&arg1`), MASM will effectively ignore the expansion operator (as it expands the text anyway). You can also specify a text-expansion-to-text operation as follows:

```
mac3    macro  arg1
        tostr  <&arg1>
        byte   0
        endm
```

This expands the text contained within `arg1` to a text string, which is inside a text literal definition, converting it back to text. This is redundant in this situation, but still legal.

Sadly, the text expansion operator works only inside a macro (and only for macros and local macro symbols). You cannot expand text equates defined inside or outside of a macro by using this operator; I'll provide a work-around for this later in this chapter.

1.2.3 The Text Escape Character (!)

The text literal operators < and > don't solve all your problems. For example, suppose you want to pass the text literal constant `x < y` as a macro argument:

```
mac   macro  arg
      byte   "&arg", 0
```

```
        endm
          .
          .
          .
        mac   <x < y>
```

MASM will be confused by the `x < y` literal constant because it treats the less-than sign as a text literal left angle bracket.

To resolve this issue, use `!`, the MASM *text escape operator*. If you place the exclamation mark in front of another special MASM symbol, MASM will treat that special character as a normal character. For example, you can recode the previous macro expansion as follows:

```
        mac   <x !< y>
```

Use `!!` to encode an exclamation mark in a text literal (rather than using it as the escape character).

The MASM escape character is one of the more problematic symbols in macro expansions. In theory, the `!!` sequence should let you insert an exclamation mark within a text literal. In practice, if that expansion occurs inside a macro argument, MASM will likely treat the `!` in the new text literal as its own escape sequence, thus you wind up losing the exclamation mark in several situations. The straightforward `mac` example expands as you want. However, consider this:

```
mac2    macro  arg
        tobyte arg
        byte   0
        endm

tobyte  macro  arg
        byte   "&arg"
        endm
          .
          .
          .
        mac2   <x !< y>
```

If you assemble this, MASM will (again) complain that it found an unexpected `<` symbol. That's because `mac2` expands to this:

```
        tobyte x < y
        byte   0
```

Now you have an unescaped `<` symbol as the `tobyte` argument, which confuses MASM. You can overcome this issue with the following:

```
        mac2   <x !!!< y>
```

This expands to

```
        tobyte x !< y
        byte   0
```

which can now expand (without issue) to this:

```
        byte   "x < y"
        byte   0
```

While this hack job ultimately delivers the result you want, expecting the macro's user to understand the internal operation of the `mac2` macro (specifically, that it passes the argument on to another macro) is a very poor design.

Ultimately, using exclamation marks as actual exclamation marks in text literals is problematic, and you want to avoid it as much as possible. If you must use escaped exclamation marks in a text literal, either make sure that text literal never expands more than once or take into account that you must handle it specially (and comment the code, as appropriate).

1.2.4 The Value Expansion Operator (%)

The MASM *value expansion operator,* `%`, expands a MASM numeric symbol to decimal text form. If you have defined a symbol such as `x = 5`, then `%x` is converted to `<5>`.

Unfortunately, the `%` operator converts only numeric values to decimal strings. Consider the following:

```
txt1    textequ %41h    ; txt1 becomes <65>.
txt2    textequ %'A'    ; txt2 becomes <65>.
```

On the plus side, the `%` operator allows an arbitrary numeric constant expression to appear after the `%`. Therefore, you can also use the `%` operator thusly:

```
txt3    textequ %'A'-1  ; txt3 becomes 64.
```

One of the more beneficial use cases for the value expansion operator is with the `echo` directive. If you place a `%` operator in column 1 on the source line containing an `echo` directive, MASM will expand text identifiers in the `echo` operand field to their textual equivalent (unfortunately, it will not expand numeric constant identifiers to their numeric value, which seems kind of strange).

1.2.5 Local Symbols in Macros

If you define a symbol in the body of a macro, MASM will emit that same symbol every time you expand the macro. Consider the following:

```
shft1s  macro
lbl:    shr eax, 1
```

```
        jc  lbl
        endm
```

This macro will shift all the 1 bits out of the low-order (LO) bit positions of EAX. Now consider the following macro invocations:

```
        shft1s
        shft1s
```

The first macro will expand and assemble without any problems. The second, however, reports a problem (redefined symbol). This is because these two macros expand to the following code:

```
lbl:    shr  eax, 1
        jc   lbl
lbl:    shr  eax, 1
        jc   lbl
```

MASM sees two definitions of the symbol `lbl` and balks at this. To solve this issue, MASM provides a `local` directive with the syntax

```
local list_of_1_or_more_identifiers
```

where *`list_of_1_or_more_identifiers`* is a list of one or more valid MASM identifiers, separated by commas if there are more than one. They must be unique among themselves (and within a given macro), but they do not have to be unique throughout the program. For example, two different macros could reuse local macro symbols.

The `local` directive must be the first nonblank/noncomment line after the `macro` statement. You can have multiple `local` statements in a macro, but they must all appear together. All the symbols defined in the `local` directives within the macro must be unique.

MASM converts each of these identifiers to a text value and initializes it with a string of the form `??`*`xxxx`* where *`xxxx`* is a four-digit hexadecimal number. MASM generates a sequence of these *`xxxx`* values (starting from 0000 and incrementing for each local symbol defined via a macro expansion).

The `local` directive can fix the `shft1s` macro as follows:

```
shft1s  macro
        local  lbl
lbl:    shr    eax, 1
        jc     lbl
        endm
```

With this directive present, MASM expands the former pair of invocations to something like the following MASM assembly listing output:

```
                                        shft1s
00000000  D1 E8             1  ??0000:   shr    eax, 1
00000002  72 FC             1         jc     ??0000
```

```
                                        shft1s
00000004  D1 E8             1  ??0001:   shr    eax, 1
00000006  72 FC             1         jc     ??0001
```

For statement labels such as `lbl` appearing in `shft1s`, the value of the local symbol is not an issue. However, consider the following modification to the `shft1s` macro:

```
shft1s macro
       local  lbl, const1
const1 =      1
lbl:   shr    eax, const1
       jc     lbl
       endm
```

This expands to the following code (with possibly different `??`*xxxx* values):

```
                                        shft1s
00000000  D1 E8             1  ??0000:   shr    eax, ??0001
00000002  72 FC             1         jc     ??0000
                                        shft1s
00000004  D1 E8             1  ??0002:   shr    eax, ??0003
00000006  72 FC             1         jc     ??0002
```

If you look at the symbol table produced by MASM in the listing, you'll see the following entry for `??0001`:

```
??0001 . . . . . . . . . . . .        Number   00000001h
```

Therefore, `const1` (a text symbol) expands to `??0001`, which is the number 1.

You get into trouble when passing one of these local symbols where MASM wants a nontext value rather than a text value. Consider the following slight modification to `shft1s`:

```
shft1s macro
       local  lbl, const1
const1 =      1
lbl:   shr    eax, const1
%      echo   const=const1
       jc     lbl
       endm
```

When the `%` operator appears in column 1 of a line containing an `echo` directive, MASM will expand all the symbols on the line to their actual values. In this example, you would normally expect to see MASM print `const=1` each time it expands this macro (after all, the macro has just associated the value 1 with the symbol `const1`). Unfortunately, the `const1` symbol does not have the numeric value 1. Instead, it has the text value `<??0001>` (at least for the first expansion in these examples). The text symbol `??0001` eventually

expands to 1, but `echo` doesn't do a deep expansion of the text symbols it encounters. It expands the values to one level and prints that value. Therefore, the macro invocations in the current example will print the following at assembly time:

```
const=??0001
const=??0003
```

This can be very frustrating at times.

One solution is to use a temporary text equate to receive the expansion of the constant symbol and then expand that temporary value in the `echo` directive:

```
shft1s  macro
        local   lbl, const1, txtCnst
const1  =       1
lbl:    shr     eax, const1
txtCnst texteq  %const1
%       echo    const=txtCnst
        jc      lbl
        endm
```

This is painful and kludgy, but it has the desired effect with

```
const=1
const=1
```

writing `1` for both outputs.

1.2.6 Macro Functions

MASM provides an `exitm` (exit macro) directive. The original intent of this directive was to allow a macro to prematurely terminate, generally used in conjunction with conditional assembly directives. However, later versions of MASM (starting around MASM v6) expanded the syntax of macro definitions to include a functional syntax. The `exitm` directive enables this functional syntax.

In normal MASM syntax, the `exitm` directive is a statement by itself:

```
exitm
```

To enable the functional macro syntax, the macro must have at least one `exitm` directive, and all `exitm` directives must take the form

```
exitm text_constant
```

where *`text_constant`* is either a string literal surrounded by angle brackets or a text symbol (a macro argument, local symbol, or `texteq`u).

When at least one `exitm` directive takes this form, MASM macro invocation syntax changes to

```
macroName(args)
```

where *args* is a list of zero or more argument expressions, each separated by a comma (if there are two or more arguments).

More importantly, functional macro invocations can appear anywhere a text symbol can appear, not simply replacing a MASM statement. For example, consider the following relatively trivial functional macro definition and invocations:

```
two  macro
     exitm  <2>
     endm
      .
      .
      .
     mov    eax, two()  ; Expands to mov eax, 2
      .
      .
      .
     byte   two()       ; Expands to byte 2
```

A macro function invocation isn't required to expand to a portion of an assembly language statement. You could, for example, create macros to emit whole instructions that use a functional syntax:

```
; Macro to create an HLA-like syntax in MASM:

_mov        macro src, dest
            exitm <mov dest,src>
            endm
             .
             .
             .
            mov   eax, 0    ; Identical to below
            _mov( 0, eax )  ; Source operand is first in HLA.

            mov   ah, al    ; Identical to below
            _mov( al, ah )
This expands to the following:
00000000  B8 00000000                     mov   eax, 0
00000005  B8 00000000                     _mov( 0, eax )

0000000A  8A E0                           mov   ah, al
0000000C  8A E0                           _mov( al, ah )
```

Notice that the object code for the `mov`/`_mov` pairs is identical.

1.3 Macros as Directives

Unfortunately, MASM's standard macros don't allow you to do the following (expecting the invocation line to be equivalent to `lbl byte 2`):

```
drctv    macro  arg
         byte   arg
         endm
          .
          .
          .
lbl      drctv  2
```

There are no tricks you can use to get this to work. (I've searched fruitlessly for a solution to this problem. Perhaps someone can point out how to do this. The world would love you for that.) The commonly accepted solution is to do something like the following:

```
drctv2   macro  bLbl, arg
blbl     byte   arg
         endm
```

To invoke this macro, use the syntax

```
        drctv2  lbl, 2
```

which expands to this:

```
lbl     byte    2
```

This kludge has three problems. First, the syntax is different from standard directives like `byte`, `word`, and `dword`. Ideally, you should be able to use the same syntax for your user-defined directives as MASM uses for the built-in directives. So while this solution works, it's ugly.

Second, it requires that you supply a label for the macro invocation. You can invoke the original `drctv` macro with

```
        drctv  2
```

and MASM will correctly expand this as follows:

```
        byte   2
```

Note that MASM complains about `drctv` only when you attempt to place a label definition before the macro name; invoking `drctv` by itself is fine. You must supply two operands to `drctv2`: a label and a data argument. If you attempt to invoke `drctv2` as follows, MASM will report an error because it thinks `"Hello"` is a statement label:

```
        drctv2 "Hello"
```

To work around this, supply a blank first argument by using one of the following two invocation forms:

```
        drctv2 <>,"Hello"
        drctv2 ,"Hello"
```

This, however, is just another kludge that doesn't give you the desired syntax.

The third problem with this macro is that it allows only a single data argument. You cannot, for example, write

```
        drctv2  lbl, "Hello", nl, 0
```

and expect it to emit this:

```
lbl     byte    "Hello", nl, 0
```

Instead, MASM complains about the extra arguments on the line. Fortunately, this last issue is quick to correct—just modify the definition of `drctv2` to the following:

```
drctv2  macro   bLbl,arg:vararg
blbl    byte    arg
        endm
```

Do not attach `:req` to `bLbl`. Doing so will cause MASM to complain if you supply an empty first argument, which brings back issue 2.

There's a *sneaky* way you can *almost* achieve the desired syntax: Use functional macros. Consider the following macro definition and invocation for `drctv3`:

```
            .code
main        proc

drctv3      macro   arg
            exitm   <byte arg>
            endm

lbl         drctv3  ("Hello")
            drctv3  ("There")

main        endp
            end
```

Except for the minor syntactical issue of having to put parentheses around the operands, this is very close to the directive syntax for normal MASM directives such as `byte`, `word`, and `dword`. Here's the MASM listing you get by compiling this short snippet:

```
00000000 48 65 6C 6C 6F         lbl         drctv3  ("Hello")
00000005  54 68 65 72 65                    drctv3  ("There")
```

Ho! Ho! Looks like a winner! Sadly, it is not to be.

Because of what can be described only as a defect in MASM, the way functional macros handle operands is quite a bit different from the way MASM handles operands in standard macros. Standard macro operands respect the text between quotation marks and apostrophes (that is, string literal constants). Macro function operands do not. Consider the following invocation for `drctv3`

```
someLbl     drctv3  ("Hello World!")
```

and the output:

```
00000000 48 65 6C 6C 6F         someLbl     drctv3  ("Hello World!")
         20 57 6F 72 6C
         64
```

This string is missing a character—the `!` (American Standard Code for Information Interchange, or ASCII, code `21H`). Because MASM believes that the exclamation mark is an escape character sequence (applying to the quote that immediately follows), it removes this character from the string. You can solve this particular problem by doubling up the `!` character—for example, `("Hello World!!")`. However, that solves only the current problem. Consider the following invocation of `drctv3`:

```
            drctv3  ("1) List item number one.")
```

As I've stated, MASM does not respect the string delimiters in this operand. Therefore, when seeing the first right parenthesis (after the `1`), MASM thinks that the `drctv3` operand is complete (that is, the macro's argument consists of `("1)`, ending with that first right parenthesis). Inside the macro, the lack of a closing quotation mark generates a syntax error. Unfortunately, there's no way around this issue; even putting an exclamation mark in front of the `)` won't solve the problem. This is a bug in MASM, and not much can be done about it.

However, several enterprising MASM macro programmers have come up with one solution: creating your own escape-character sequences to insert special characters into strings. The most popular approach is to use the C/C++ backslash escape character (\) with other characters to represent characters that misbehave in MASM operands. In Section 1.5, "Processing Escape Character Sequences in MASM Macros," on page XX, I'll provide an example macro that translates strings containing the following escape character sequences:

- \| → !
- \{ → (

- \} →)
- \[→ <
- \] → >

With that special macro code added to `drctv3` to process these escape sequences, you could write the following invocations:

```
lbl1   drctv3 ("Hello World\|")
list1  drctv3 ("1\}List item number one.")
```

The macro would handle translating the \| to ! and the \} to).

While this is clearly not an ideal solution (differing considerably from what you can do with standard MASM directives), it makes it possible to create strings to process arbitrary characters. I'll return to the subject of processing escape sequences after discussing text processing in MASM macros.

1.4 Text Processing in MASM Macros

The previous section was largely a complete review of the discussion of macros and some of the macro operators that appeared in *Ao64Av1*. In some respects, this section also reviews several of the MASM text-processing functions including `instr`, `substr`, `sizestr`, and `catstr`. However, whereas *AoA64v1* concentrated on explaining what these functions do, this section focuses on using these functions in your macros.

1.4.1 The sizestr Directive

The `sizestr` directive computes the length of a textual object. The syntax for the directive is

```
identifier  sizestr  string
```

where *`identifier`* is the symbol into which MASM will store the string's length, and *`string`* is the text literal whose length this directive computes. As an example, the following defines the symbol `hwLen` as a numeric constant symbol and sets it to the value 11:

```
hwLen  sizestr  <Hello World>
```

Normally, you wouldn't use the `sizestr` directive with a text literal as I have done; the `sizestr` operand would be a `textequ` symbol, macro argument symbol, or other text object. Here are some examples that better represent the use of this directive:

```
Txt         textequ <Hello>

testSize    macro   arg
            local   size1, size2
            local   txt1, txt2
```

```
size1       sizestr arg     ; Expands to <Hello>
size2       sizestr <&arg>  ; Expands to <Txt>
txt1        texteqU %size1
txt2        texteqU %size2
%echo sz1=txt1, sz2=txt2
            endm
             .
             .
             .
            testSize Txt
```

This produces the following output when you assemble it:

```
sz1 = 5, sz2 = 3
```

If you're having trouble seeing why you get this output, do a manual expansion of `textSize`:

```
size1       sizestr Txt    ; Expands to Txt, which expands to <Hello>
size2       sizestr <Txt>  ; Expands to <Txt>, then no further
txt1        texteqU %size1 ; size1 is <Hello>, so this is 5.
txt2        texteqU %size2 ; size2 is <Txt>, so this is 3.
%echo sz1 = 5, sz2 = 3
```

When `size1 sizestr arg` gets processed, MASM first replaces `arg` with `Txt`. However, `Txt` is a `texteqU`, so MASM further replaces this with the text value of `Txt` (`Hello`). The length of this text object is `5`, hence MASM assigns the value `5` to `size1`.

When `size2 sizestr <&arg>` gets processed, MASM first replaces `arg` by `Txt`. Unlike the `size1` expansion, this version of `Txt` appears inside the text literal delimiters (`<` and `>`) so MASM doesn't do any additional expansion. Hence the `size2` constant gets set to `3`, the length of `<Txt>`. By the way, note that without the `&` expansion operator, MASM would never have expanded the `arg` argument inside the text delimiters. In that case, `size2` would still have been set to `3`, but only because the text value `<arg>`'s length is also `3`.

To reiterate, `sizestr` produces a numeric value, not a text value. If you want to convert this result to text form (perhaps to print its value by using the `echo` directive as I've done in these examples), you need to use the `%` operator on the result to convert it to text (see the `texteqU` directives in this example).

1.4.2 The MASM instr Directive

The `instr` directive and function search for the presence of one string within another. This is similar, in usage, to the C/C++ `strstr` function. The syntax for the directive is

```
identifier  instr  start, source, search
```

where *`identifier`* is a symbol that will receive the offset of the *`search`* string within the *`source`* string. The search begins at position *`start`* within *`source`*.

Note that the first character in *source* has the position 1 (not 0). The following example searches for World within the string Hello World (starting at character position 1, which is the index of the H character):

```
WorldPosn   instr  1, <Hello World>, <World>
```

This particular example defines WorldPosn as a number with the value 7 (as the string World is at position 7 in Hello World if you start counting from position 1).

You can also use the instr directive to simulate many other C/C++-style string functions. Simulating strchr is relatively straightforward; simply supply a *search* operand that's one character long rather than a multicharacter string.

You can also use instr to see whether a character is a member of a particular character set. Consider the following invocation:

```
isVowel     instr  1, <aAeEiIoOuU>, <someChar>
```

If *someChar* is a text object containing a single character, this statement will set isVowel to a nonzero value if it's an (English) vowel or to 0 otherwise (if it is a vowel, isVowel's value will tell you exactly which vowel it is).

Similarly, you can use instr to simulate C/C++'s strcspn function, which skips over characters in a string that do not belong to a set of characters. Simulating strspn (skipping over characters that *are* in a set) is a bit more challenging. You have to create a character set with the complement (inverse) of the characters you want to skip over. That is, you search for a character that's not in the set (its complement) rather than characters in a set.

Another use for instr is determining whether a string is a particular word. Consider the following example:

```
whichWord   instr  1, <and or  xor not>, <someWord>
```

This example assumes *someWord* does not contain any space characters (the word delimiter, in this case). The instr directive checks whether *someWord* is the word and, or, xor, or not and returns 1, 5, 9, or 13 if it is (respectively), or 0 if it's not equal to any of these words. If, after checking for 0, you divide the result by 4, you get 0, 1, 2, or 3 to easily identify the word that *someWord* matches.

1.4.3 The MASM substr Directive

The MASM substr directive extracts a substring from a text string. The syntax for this directive is

```
destTxt  substr  source, start, length
```

where *destTxt* is the symbol that will receive the substring text, *source* is the text from which substr will extract the substring, *start* is the starting position in *source* where the substring will begin, and *length* is the length of the

substring to extract. The *length* operand is optional; if it's absent, MASM will assume you want to use the remainder of the string (starting at position *start*) for the substring.

Beyond extracting substrings, you can also use the `substr` directive to randomly access elements of a text array. To do so, supply a `length` value of `1` and specify the index by using the *start* operand. Consider the following example:

```
index       =       1
hw          textequ <"Hello World!">
len         sizestr hw
            while   index le len
chr         substr  hw, index, 1
%echo w=chr
index       =       index + 1
            endm
```

This example is more concisely coded as

```
            forc    chr, <"Hello World!">
%echo f=chr
            endm
```

though at times you might want to modify `index` differently (other than by adding `1` to it).

In addition to indexing into an array of text characters, you can use the `substr` directive to transform values. For example, to convert a numeric value in the range `0` to `0Fh` into the corresponding hexadecimal character, do the following:

```
hexRslt substr <00h01h02h03h...0dh0eh0fh>, (toHex*3)+1, 3
```

This yields the string `00h` through `0fh` depending on the value of `toHex` (filling in all the values between `03h` and `0dh` appropriately).

You can also use `substr` to generate a string of *n* copies of the same character. Just create a large text literal containing a long sequence of the character you want to repeat, using a starting index of `1`, and then specify the repeat count as the *length* value (*n*):

```
spaces  substr  <large number of spaces>, 1, n
```

Note that *n* must not exceed the number of characters you supply as the `substr` source string, and the whole line is limited to around 240 characters.

1.4.4 The MASM catstr Directive

The `catstr` function has the following syntax:

```
identifier  catstr  string1, string2, ...
```

The *identifier* is (up to this point) an undefined symbol. The *string1* and *string2* operands are text surrounded by < and > symbols. This statement stores the concatenation of the two strings into *identifier*.

The catstr statement allows two or more operands separated by commas.

The catstr directive will concatenate the text values in the order they appear in the operand field. The following statement generates Hello, World!:

```
helloWorld  catstr  <Hello>, <, >, <World!!>
```

Note that two exclamation marks are necessary in this example. Remember, ! is an operator telling MASM to treat the next symbol as text rather than as an operator. With only one ! symbol, MASM thinks that you're attempting to include a > symbol as part of the string and reports an error because there's no closing >.

A nonobvious use of catstr is to convert a text literal to a string literal. Consider the following:

```
asString  catstr  <">, <asText>, <">
```

This statement surrounds the text held in *asText* by quotation marks, converting it to a string literal. For this to work properly, *asText* must not already contain any quotes (you can check for this with instr). If it does, use apostrophes to delimit the string literal rather than quotes. MASM string literals cannot contain both quotes and apostrophes. If *asText* contains both, this code won't work.

The online MASM documentation claims that catstr is a synonym for textequ. Therefore, you can also concatenate strings by using the textequ directive:

```
helloWorld  textequ  <Hello>, <, >, <World!!>
```

I prefer using catstr as it better describes the activity taking place.

1.4.5 Functional String Directives

MASM supports functional-like syntax for the existing string directives. You can use the following macro functions within constant expressions:

- @SizeStr(*text*)
- @CatStr(*text*, *text*, ...)
- @SubStr(*text*, *start*, *length*)
- @InStr(*start*, *text*, *search_text*)

These function names are case sensitive, even if you've specified option casemap:none in your source file. This is a bug in MASM (at least in the version I'm using in this book) that you must work around by always specifying the exact case for these names.

The only problem is that MASM handles function macro arguments differently from standard macro arguments (and, worse still, it handles

arguments for these functions differently from user-written macro functions). Consider the following example:

```
someText    textequ <Hello World!!>
len         =       @SizeStr(someText)
tt          textequ %len
% echo lenst=tt
```

Assembling this sequence should display `lenst=12`. In reality, it displays `lenst=8`, because MASM thinks the text to compute the length is *someText*, rather than the expansion of *someText*. Sticking `&` in front of *someText* doesn't fix this problem; the change produces `lenst=9`, counting the `&` as another character instead of expanding the text. The only work-around I've found for this bug is the following:

```
expand      macro   arg
            exitm   arg
            endm
len         =       @SizeStr(expand(someText))
tt          textequ %len
% echo lenst=tt
```

This produces the output you'd expect (`lenst=12`). However, with the amount of work required, this approach borders on not being worthwhile.

You run into similar problems with the other string functions. Consider the following use of `@CatStr`:

```
h           textequ <Hello,>
w           textequ <World!!>
hwstr       =       @CatStr( h, w )
tt          textequ %hwstr
% echo h=tt
```

This generates two errors (on the version of MASM I'm using):

```
listing.asm(13) : error A2006:undefined symbol : hwstr
listing.asm(12) : error A2006:undefined symbol : hw
```

First, where's the symbol `hw` that MASM is complaining about? Well, MASM treats the `h, w` operands to `@Catstr` as text, not symbols to be expanded. The `@CatStr` concatenates these two characters and emits them as text, producing the following statement:

```
hwstr       =       hw
```

However, the symbol `hw` is undefined in this program fragment, hence the first error. This leaves `hwstr` undefined, which produces the second error. Unfortunately, this can't be corrected using the `expand` macro.

The bottom line is that the MASM functional macros for the string directives are basically worthless. Please feel free to email me if you figure out a

way to get them working (I'm using MASM version 14.41.34120.0 for these examples). Perhaps at some point Microsoft will correct the defects in MASM that prevent their practical use. Until then, I recommend ignoring them.

1.4.6 An In-Depth Look at Text Equates

The MASM `texteq`u directive is the fundamental tool for dealing with text strings in a MASM source file. Though *Ao64Av1* provided a decent description of the `texteq`u directive, it has several undocumented properties; taking a closer look at how this directive operates is worthwhile.

As a refresher, the basic syntax for the `texteq`u directive is

```
identifier  texteq  textual_data
```

where *`identifier`* is a MASM identifier and *`textual_data`* is a MASM expression yielding text data. Normally, *`identifier`* is an undefined (up to this point in the source file) symbol, and *`textual_data`* is a text literal, for instance:

```
helloWorld  texteq  <Hello, World>
```

After this directive, MASM will substitute the literal text `Hello, World` for the identifier `helloWorld` in the source file (notice that MASM does not include the angle brackets in the text expansion).

Consider the following code snippet:

```
abc   texteq <def>
def   =      123
       .
       .
       .
      mov    eax, abc
```

MASM will substitute `def` in place of `abc`, yielding the instruction `mov eax, 123`; nothing surprising here. Now, consider the following modification to this code:

```
abc   texteq <def>
def   texteq <ghi>
ghi   texteq <123>
       .
       .
       .
      mov    eax, abc
```

This example still produces the instruction `mov eax, 123`. When expanding a text equate, MASM recursively expands any text symbols in the expansion until there are no more text symbols. Therefore, MASM does the following with the instruction `mov eax, abc`:

- `mov eax, abc` is expanded to `mov eax, def`.
- `mov eax, def` is expanded to `mov eax, ghi`.
- `mov eax, ghi` is expanded to `mov eax, 123`.

With one exception that I will get to in a moment, MASM always expands text symbols repeatedly until it can no longer expand any text in the result.

Now consider the following code:

```
abc     textequ <def>
         .
         .
         .
abc     =       123
         .
         .
         .
        mov     eax, abc
```

This is a legal sequence of MASM instructions, and it produces the instruction `mov eax, 123`. The point of interest here is that MASM expands this text to produce `mov eax, 123`.

You might wonder why this is even legal in the first place. After all, it looks like this code is attempting to redefine the symbol `abc` from a text equate to a constant equate (which normally is illegal). However, once the code defines `abc` as a text equate, MASM will expand that identifier wherever it finds it, including in the label field of another instruction or directive. Therefore, the previous code snippet actually expands to the following:

```
abc     textequ <def>
         .
         .
         .
def     =       123
         .
         .
         .
        mov     eax, def
```

I mentioned that this expansion operation has one exception. MASM allows you to redefine text equate symbols. That is, the following is perfectly legitimate:

```
n       textequ   <125>
         .
         .
         .
        mov       al, n  ; mov al, 125
         .
         .
         .
```

```
n       textequ   <52>
         .
         .
         .
        mov       al, n  ; mov al, 52
```

Under normal circumstances, you would expect MASM to convert the second text equate to `125 textequ 52` as MASM should expand that text, even in the label field of the directive. However, MASM specially handles labels appearing in a `textequ` directive and allows you to redefine such symbols.

Sometimes this symbol redefinition gets in the way of what you want to do. Consider the following code fragment:

```
abc     textequ <def>
abc     textequ <125>   ; Really wanted to do "def textequ <125>"
        mov     al, def ; Undefined symbol error
```

This results in an error because MASM redefines `abc` as `125` rather than expanding `abc` to `def` and defining `def` to `125`. As a result, the symbol `def` is undefined when MASM processes the `mov` instruction. If you really want to expand the first `abc` to `def`, you can employ a kludge as a work-around:

```
expand      macro   symbol
            exitm   symbol
            endm

abc         textequ <def>
expand(abc) textequ <125>
            mov     al, def
```

MASM expands `abc` to `def` before invoking the macro function `expand`; that is, the macro call is actually `expand(def)`. The `expand` macro simply returns the text you pass it (`def`, in this case). Therefore, the second text equate in this example expands to `def textequ <125>`.

One issue with MASM's expansion of text equate symbols is that you have very little control over the expansion process (short of resorting to kludges like the `expand` macro). Sometimes you might not want a text equate to expand as far as possible. The previous examples generated a symbol to use in a text equate definition (`def`). Consider the following:

```
expand      macro   symbol
            exitm   symbol
            endm

abc         textequ <def>
expand(abc) textequ <125>
expand(abc) textequ <34>
            mov     al, def
```

This code is trying to redefine `def` to `<34>` after already defining it as `<125>`. Unfortunately, MASM will reject this with a mysterious error about an unexpected integer because the text equates expand as follows:

```
abc         textequ <def>
expand(abc) textequ <125>  ; Becomes "def textequ <125>"
expand(abc) textequ <34>   ; Becomes "125 textequ <34>"
```

The problem here is that MASM allows you to redefine a text symbol only if a simple identifier appears in the label field of a text equate directive. As `expand(abc)` is not a simple identifier, MASM expands this text as far as it can, yielding the string `125` (which explains the unexpected integer error).

A bit of a kludge will help you get around this problem. Consider the following code snippet:

```
mkID        macro   a, b
            exitm   <&a&&b&>
            endm

def         textequ <000>
mkID(de,f)  textequ <123>
mkID(de,f)  textequ <45>
```

The `mkID` macro builds an identifier by concatenating two text strings together (note that this trick won't work if you use the `catstr` directive; it works only when using the `&` macro expansion operator). In this example, the macro concatenates `de` and `f` to produce the identifier `def`. Because the string `def` doesn't appear on the last two lines, MASM doesn't expand the text. I can't explain why MASM doesn't expand the `def` that `mkID` produces, but I'm not going to complain too loudly about this idiosyncrasy, as it's quite useful.

Keep in mind that the `catstr` and `substr` directives also produce text symbols that behave in a manner identical to text equate symbols (in fact, `catstr` is a synonym for `textequ`). Therefore, you can use these directives to synthesize label names during assembly. Here's an example:

```
i             =       0
              while   i lt 10
              local   curID
curID         catstr  <id>, %i  ; Create the text "idn" where n
expand(curID) textequ %i        ;   is the textual representation
i             =       i+1       ;   of i; assign it the numeric
              endm              ;   string for i.
```

This code has to use the `expand` macro, given earlier, so that it doesn't simply redefine the `curID` symbol on each execution of the loop.

This code snippet creates the following symbols from a MASM symbol table listing:

```
id0  . . . . . .       Text     0
id1  . . . . . .       Text     1
id2  . . . . . .       Text     2
```

```
id3  . . . . . .  Text    3
id4  . . . . . .  Text    4
id5  . . . . . .  Text    5
id6  . . . . . .  Text    6
id7  . . . . . .  Text    7
id8  . . . . . .  Text    8
id9  . . . . . .  Text    9
```

This trick can be quite useful for creating compile-time array constants (see *Ao64Av1* (Section 4.9.1, "Declaring Arrays in Your MASM Programs") for a discussion of how to create arrays of constants at assembly time).

You can use the `mkID` macro given earlier if you need to redefine these symbols. For example:

```
expand        macro    symbol
              exitm    symbol
              endm

mkID          macro    a, b
              exitm    <&a&&b&>
              endm

; The following code generates:
;
; id0 = <0>
; id1 = <1>
; etc.

i             =        0
              while    i lt 10
              local    curID
curID         catstr   <id>, %i  ; Create the text "idn" where n
expand(curID) textequ  %i        ;   is the textual representation
i             =        i+1       ;   of i; assign it the numeric
              endm               ;   string for i.

; The following code redefines the above
; symbols as:
;
; id0 = <0>
; id1 = <2>
; id2 = <4>
; etc.

i             =        0
              while    i lt 10
              local    curID
mkID(<id>,%i) textequ  %i*2      ; This is the textual representation
i             =        i+1       ; of i; assign it the numeric
              endm               ; string for i.
```

After the assembly completes, the symbols have the following values (taken from the MASM listing output):

```
id0  . . . . . .  Text   0
id1  . . . . . .  Text   2
id2  . . . . . .  Text   4
id3  . . . . . .  Text   6
id4  . . . . . .  Text   8
id5  . . . . . .  Text   10
id6  . . . . . .  Text   12
id7  . . . . . .  Text   14
id8  . . . . . .  Text   16
id9  . . . . . .  Text   18
```

Sometimes macro parameters and text equates don't expand completely. For example, you might find that a local symbol is expanding to the text MASM associates with that local symbol (such as `??1234`) rather than to whatever text you've assigned to that local label. In these situations, you can use the MASM expression evaluation operator `%` to force an expansion of the text associated with the symbols.

If you have trouble understanding how MASM handles all these text operations and expansions, know you are not alone in your frustrations. I've been using MASM since the early 1980s (back when it was just a 16-bit assembler for the original 8088 CPU in the IBM PC), and MASM's text-processing behaviors remain a mystery to me. I often try arbitrary sequences of text expansions and macro operations just to see if I can get something to work. On the plus side, I've been able to achieve almost everything I've wanted to do with MASM macros; sometimes getting to that point takes a lot of effort, but with diligence you can usually find a solution.

1.5 Processing Escape Character Sequences in MASM Macros

In Section 1.3, "Macros as Directives," on page XX, I mentioned that one solution to dealing with special characters in MASM macro expansions is to use a user-defined escape-character sequence. I proposed using the backslash character like so:

- \| → !
- \{ → (
- \} →)
- \[→ <
- \] → >

Before describing how to implement a macro that will process these escape sequences, let's look at why they are necessary in the first place. Consider the following macro and invocation:

```
a           macro   arg
            forc    chr, <&arg>
tt          textequ <&chr>
```

```
%echo t=tt
            endm
            endm

            a       <Hello, World!!>
```

If you attempt to assemble this sequence, you get the following error (among others):

```
listing.asm(13) : error A2045:missing angle bracket or brace in literal
 a(1): Macro Called From
  listing.asm(13): Main Line Code
```

You get this error on line 1 in the macro because MASM expands the `forc` statement to the following:

```
            forc    chr, <Hello, World!>
```

Note that there's only one `!` at the end of the text string. You get a single `!` rather than `!!` because the exclamation mark is MASM's text escape sequence, and two of them in a row (in the macro invocation) produce a single `!` in the actual text string. So when MASM expands `arg` inside the macro (in the statement `forc chr, <&arg>`), you get a single `!` at the end of the text string. However, when processing this line, MASM thinks the `!` character is escaping the `>` that immediately follows (that is, MASM thinks the text string is `Hello, World>`). As a result, the text literal has no closing `>`, hence the error.

To work around this, use the following macro invocation for `a`:

```
            a  <Hello, World!!!!>
```

When MASM sees the `forc` statement inside macro `a`, it expands `arg` to

```
            forc    chr, <Hello, World!!>
```

and properly assembles this statement, because the `!!` sequence produces a single `!` and does not affect the `>` character. However, the assembler still produces the same error, albeit on a different statement. This is because the statement

```
tt          textequ <&chr>
```

expands to

```
tt          textequ <!>
```

when MASM processes the `!` at the end of the string. Unfortunately, as MASM processes individual characters at a time, there's no working around this problem.

You wind up with various issues when supplying any of the (,), <, >, or ! characters. No amount of escaping these symbols will work, as it's possible they will be passed as an argument to another macro and require re-escapement. As the number of nested macros could go on indefinitely, you'll need another solution. In this section, you'll use the backslash \ as an escape character to generate the special characters.

I've yet to find any issues with the vertical bar |, square brackets [], or braces {} during macro expansion, hence why I chose their use in these escape sequences.

The idea is to use a macro invocation such as the following:

```
            a    <Hello, World\|>
```

Because the backslash (assuming it doesn't appear at the physical end of a line) and the vertical bar have no special meaning to MASM, the a macro passes and processes this string with no problems. Consider the following macro, b (a slight modification of a that handles the special case of a backslash appearing at the end of a line), and its invocation:

```
b           macro   arg
            forc    chr, <&arg>
tt          textequ <&chr>
%echo t='&tt'      ; \ would appear at the end of a line here.
            endm
            endm

            b       <Hello, World\|>
```

Assembling this snippet produces the following output during assembly:

```
t='H'
t='e'
t='l'
t='l'
t='o'
t=','
t=' '
t='W'
t='o'
t='r'
t='l'
t='d'
t='\'
t='|'
```

The \| sequence has not been converted to an exclamation mark, as you've yet to write a macro that processes the escape sequences, but at least the macro has processed all the characters sent to it without any problems.

Consider the following macro, `bsEscape` (backslash escape), which processes the current set of backslash escape sequences:

```
; bsEscape-
;
; Backslash escape processing
;
; Handles the following escape sequences:
;
; \| becomes !
; \{ becomes (
; \} becomes )
; \[ becomes <
; \] becomes >
;
; \x (x any other character) becomes \x
; so the resulting text value can be
; further processed by a different
; escape processing macro.

bsEscape    macro   arg
            local   result, ndx, chr, len

; Get the length of the argument string
; and process the characters in the string
; from position 1 to length:

len         sizestr <&arg>
result      textequ <>
ndx         =       1
            while   ndx le len

; Extract the current character (1..len)
; from the string:

rmndr substr <&arg>, ndx, len-ndx

chr         substr  <&arg>, ndx, 1

; Check for a "\" escape sequence:

            ifidn   chr,<\>

; Get the next char after the "\":

ndx         =       ndx + 1
chr         substr  <&arg>, ndx, 1

; Check for \| and translate to "!":

            ifidn   chr, <|>

result      catstr  result, <!!>
```

```
            else    ; dif <&arg>, <|>

; Check for \{ and translate it to "(":

            ifidn   chr, <{>

result      catstr  result, <(>

            else    ; dif <&arg>, <{>

; Check for \} and translate it to ")":

            ifidn   chr, <}>

result      catstr  result, <)>

            else    ; dif <&arg>, <}>

; Check for \[ and translate it to ">":

            ifidn   chr, <[>

result      catstr  result, <!<>

            else    ; dif <&arg>, <[>

; Check for \] and translate it to ">":

            ifidn   chr, <]>

result      catstr  result, <!>>

            else    ; dif <&arg>, <]>

; Check for \\ and translate it to "\":

            ifidn   chr, <!\>

result      catstr  result, <!\>

            else    ; dif <&arg>, <\>

; Backslash in front of an arbitrary
; character, return "x" (where x is
; the character):

result      catstr  result, chr

            endif   ; <&arg>, <]>
            endif   ; <&arg>, <[>
            endif   ; <&arg>, <}>
            endif   ; <&arg>, <{>
            endif   ; <&arg>, <|>
            endif   ; <&arg>, <\>
```

```
            else    ; dif <&>,<\>

; If an unescaped character, append
; it to the result:

result      catstr  result, chr

            endif   ; <&arg>, <\>

ndx         =       ndx + 1
            endm    ; while loop

            exitm   result
            endm    ; bsEscape
```

You can find the source code for this macro in *aoaMacros.inc* at *https://artofasm.randallhyde.com*. Click the **Art of Assembly v2** link.

A simple invocation of this macro

```
rslt        textequ bsEscape ("\{\[Hello, World\|\]\}\\")
%echo r2=rslt
```

yields the following output:

```
r2="(<Hello, World!>)\"
```

Note that the `bsEscape` macro leaves `\n` as is and doesn't convert it to a newline character. The reason is simple: `bsEscape` is a macro function that can be called from almost any expression operand. Different calls to `bsEscape` may expect a different representation for the newline character. To handle escape sequences such as `\n`, `\r`, `\b`, and `\t`, write a separate escape sequence handler.

If you decide to write a separate escape character processor, make sure to call it before calling `bsEscape`. You should call the `bsEscape` expansion last, because calling other expansions after `bsEscape` will treat the resulting `!<>()` characters as special characters, not text, leading you back to the original problem.

In Chapter 2, I provide a different version of `bsEscape` (`utf16`) that also processes additional escape sequences including `\n`, `\b`, `\r`, and `\t`.

1.6 Simulating Namespaces in MASM

Though MASM provides a fair number of features like those in high-level languages, or HLLs (such as structs, unions, and procedure declarations), one nifty feature MASM is missing is the namespace declaration. A *namespace* is a collection of symbol definitions (procedures, variables, constants, macros, and so on) that are embedded in a user-defined namespace. By default, MASM offers three namespaces: external, global, and local.

External names are visible across multiple files. You can use the `extern`, `externdef`, and `public` directives to declare and use external names.

Global names are visible throughout a program source file (unless overridden by a local symbol). Normal data declarations, equates (`equ`, `textequ`, and `=`), and MASM `proc` symbols belong to the global namespace, as do all statement labels you define in a procedure that end with a pair of colons (`::`).

Local symbols are those you explicitly declare in a `local` directive in a macro or procedure, and all statement labels (with a single colon) you define within a procedure.

Sometimes, especially when including lots of libraries, various types of name conflicts occur. For example, if you're writing a set of procedures to write data to the standard output device, you might choose a name like `puti32` to convert a 32-bit integer to a string of decimal characters and output the result to the standard output. Later on, you might be writing a set of procedures to send output to the standard error device—specifically, one that converts a 32-bit integer to a string and outputs its value. Naming this procedure `puti32` as well makes perfect sense. However, when you use both libraries in an application (one that wants to write data to both the standard output and standard error output), MASM rejects the input, complaining about a duplicate defined symbol (`puti32`). This type of name conflict is known as *namespace pollution*.

To avoid namespace pollution, put these symbol definitions (`puti32` in the standard output library and `puti32` in the standard error library) into two separate namespaces. The High-Level Assembler (HLA) assembly language, for example, supports namespace definitions like the following:

```
namespace stdout;

    procedure puti32; @external;

end stdout;

namespace stderr;

    procedure puti32; @external;

end stderr;
```

You can then use the following syntax to refer to the two versions of `puti32`:

```
        call stdout.puti32
        call stderr.puti32
```

Unfortunately, MASM doesn't support a namespace syntax. However, programmers typically simulate namespaces by using prefixes or suffixes on names like the following:

```
stdout_puti32 proc
                .
```

```
                .
                .
stdout_puti32 endp

stderr_puti32 proc
                .
                .
                .
stderr_puti32 endp

                .
                .
                .
              call stdout_puti32
              call stderr_puti32
```

This scheme works reasonably well for all types of symbols including procedure names, statement labels, variables, constants, and macros. The main drawback is that this approach removes your ability to use underscores as arbitrary characters in symbols. That is, something like

```
less_than     byte 0
```

is ambiguous: Is `less_than` referring to the namespace `less` or a symbol indicating <? To overcome this ambiguity, I suggest using a completely different syntax that makes it clear you're using a name from a specific namespace.

After several attempts to devise namespace declarations for MASM, I finally came up with a scheme that works. The syntax is slightly kludgy, but it's clean and handles data, macros, and procedure calls as part of the namespace.

This version of the namespace macro uses functional macro syntax, which allows you to use the namespace macro invocation anywhere in a statement (wherever an identifier is legal). The drawback to this approach is that you must surround the namespace-local identifier in parentheses. For example:

```
        stdout(puti32)
```

Beyond this bit of syntactical ugliness, the functional macro does everything you want: It allows you to use an HLL-like syntax, specify the name directly, access data and constants, and perform all the other features you'd like:

```
        call stdout(puti32)     ; If you prefer this,
        stdout(puti32)          ;   or this (one or the other, but not both
                                ;   in the same program)
        mov eax, stdout(handle) ; Can access variables, too
        stdout(puti32) value32  ; Even allows arguments to calls
```

The trick is that a sequence like `stdout(xxxx)` invokes a macro (which you must supply, typically named `stdout_xxxx`) that handles the symbol's actual access and returns a text value that expands to do whatever is needed at that point. Consider the following definitions:

```
stdout          macro arg
                ifidn <&arg>,<puti32>
                exitm <stdout_puti32>
                endif
                endm

stdout_puti32   macro arg
                ifnb  <&arg>
                mov   eax, arg
                endif
                call  stdout_p_puti32
                endm

stdout_p_puti32 proc
                 .
                 .
                 .
stdout_p_puti32 endp
```

The `stdout` macro handles all the symbols that will belong to the `stdout` namespace. The current version contains only code that will handle the `stdout(puti32)` function call, but you can easily add additional symbols by inserting more `ifidn` sequences to the macro. For example:

```
stdout          macro arg
                ifidn <&arg>,<puti32>
                exitm <stdout_puti32>
                endif

                ifidn <&arg>,<puti64>
                exitm <stdout_puti64>
                endif
                 .
                 .
                 .
                echo  ************************************
                echo  Undefined symbol in stdout namespace:
%               echo  Symbol=arg
                echo  ************************************
                exitm <.err>
                endm
```

This version of `stdout`, including any additional symbols handled by `ifidn` clauses after `puti64`, will either exit when it finds a match or print an error. The whole `stdout` namespace is defined by this macro.

The existing example assumes that it's invoking a macro (for instance, `stdout_puti32` or `stdout_puti64`) whenever it matches one of the `stdout`

namespace symbols (`puti32` or `puti64` in the current example). The `stdout_puti32` macro handles an optional function argument. The invocation

```
        stdout(puti32) value
```

expands to this:

```
        stdout_puti32 value
```

Then the `stdout_puti32` macro expands to the following:

```
        mov  eax, value
        call stdout_p_puti32
```

Likewise, a macro invocation of the form

```
        stdout(puti32)
```

expands to

```
        stdout_puti32
```

which then expands to the following (because the `stdout_puti32` macro argument is blank, it doesn't output the `mov` instruction):

```
        call stdout_p_puti32
```

If you prefer forcing the programmer to explicitly use the `call` instruction, you can simplify the `stdout` macro a bit and use the following definition:

```
stdout          macro arg
                ifidn <&arg>,<puti32>
                exitm <stdout_puti32>
                endif
                endm
                 .
                 .
                 .
stdout_puti32   proc
                 .
                 .
                 .
stdout_puti32   endp
```

Now, the `puti32` procedure call takes the following form:

```
                call stdout(puti32)
```

This eliminates the intermediary macro that handled the HLL-like call sequence (and HLL-like function arguments).

You define and access namespace data fields (variables) and constants in an identical fashion to this last `puti32` example; simply have the `stdout` macro return the appropriate identifier:

```
stdout          macro arg
                ifidn <&arg>,<handle>
                exitm <stdout_handle>
                endif

                ifidn <&arg>,<sConst>
                exitm <stdout_sConst>
                endif
                endm
                 .
                 .
                 .
                .data
stdout_handle   dword   0
stdout_sConst   =       -11
                 .
                 .
                 .
                mov    eax, stdout(sConst)
                call   qword ptr __imp_GetStdHandle
                mov    stdout(handle), eax
```

This is a considerable amount of work to allow you to type `stdout(puti32)` rather than `call stdout_puti32`, but it makes the programmer's intent a little clearer to the reader. The one issue with the namespace macro declaration is the amount of repetitive code. Repetitive code often means that you can refactor the code by using macros to reduce the duplicated effort. Consider this simple `namespace` macro definition:

```
namespace   macro   arg, subname, fullname
            ifidn   <&arg>,<&subname>
            exitm   <exitm <&fullname>>
            endif
            endm
```

This small macro can save you a considerable amount of coding if you use numerous namespaces in your programs (particularly if they are large namespaces with lots of data, constant, macro, and procedure members). You can rewrite the `stdout` macro with the `namespace` macro like so:

```
stdout  macro      symbol
        namespace(&symbol, <puti32>, <stdout_puti32>)
        namespace(&symbol, <puti64>, <stdout_puti64>)
        namespace(&symbol, <handle>, <stdout_handle>)
        namespace(&symbol, <sConst>, <stdout_sConst>)

; Additional stdout symbols defined here...
```

```
        echo  ****************************************
%       echo  Undefined stdout identifier: symbol
        echo  ****************************************
        exitm <.err>
        endm
```

Though you'll still have to define the procedure, macro, variable, and constant identifiers (`stdout_puti32`, `stdout_puti64`, `stdout_handle`, `stdout_sConst`, and so on), the `namespace` macro streamlines the definition of the `stdout` macro.

When working with namespaces, it's best practice to create a separate header (`include`) file to contain the namespace macro definition, any macros the namespace macro invokes, all the constants used by the namespace macro, and all the necessary `externdef` statements for the variable and procedures referenced by the namespace and other macros in the header file. For example, the *stdout.inc* header file might look like the following:

stdout.inc

```
; Header file for the stdout namespace.

                ifndef stdout_inc
stdout_inc      =      1

                include    aoaMacros.inc   ; Get namespace macro.

stdout          macro      symbol
                namespace(&symbol, <puti32>, <stdout_puti32>)
                namespace(&symbol, <puti64>, <stdout_puti64>)
                namespace(&symbol, <handle>, <stdout_handle>)
                namespace(&symbol, <sConst>, <stdout_sConst>)

                echo  ****************************************
%               echo  Undefined stdout identifier: symbol
                echo  ****************************************
                exitm <.err>
                endm

stdout_puti32 macro  arg
                ifnb  <&arg>
                mov   eax, arg
                endif
                call  stdout_p_puti32
                endm

stdout_puti64 macro  arg
                ifnb  <&arg>
                mov   rax, arg
                endif
                call  stdout_p_puti64
                endm

                externdef stdout_p_puti32:proc
                externdef stdout_p_puti64:proc
                externdef stdout_handle:dword
```

```
stdout_sConst =         -11
              endif     ; ifndef stdout_inc
```

A separate source file (probably *stdout.asm*) would contain the actual procedure and data declarations to match the external symbols in the header file.

This book uses namespaces for macros, library code, and other definitions you might want to include in your own programs. The generic `aoa` namespace will encapsulate several example data types, macros, and other symbols I create throughout this book. The `aoa` namespace entries appear in the *aoaMacros.inc* header file, which you can find at *https://artofasm.randallhyde.com*.

1.7 Creating Macro Data Declaration Directives

With all the preparation done, you're finally able to create data declaration directives that behave similarly (though certainly not identically) to MASM's built-in data directives.

This section presents a data declaration for IP4 addresses consisting of a 32-bit IP address and a 16-bit (socket) port number. The syntax for this directive is

```
        ip  (ipadrs:port {,ipadrs:port ...})
```

where `{,ipadrs:port ...}` means you can have zero or more additional entries on the line after at least one IP address/port value appears. The `ipadrs` component will consist of

```
        xxx.xxx.xxx.xxx
```

where *xxx* is a decimal number in the range 0 to 255.

The `port` number is a 16-bit unsigned integer in the range 0 to 65535. The `ip` directive will store these 6 bytes in memory in network (big-endian) order as follows:

- Byte 0: High-order (HO) byte of the IP address (for `123.231.12.234`, the value `123` will appear in byte 0)
- Byte 1: Next-to-HO byte of the IP address (for instance, `231` from `123.231.12.234`)
- Byte 2: Next-to-LO byte of the IP address (`012` from `123.231.12.234`)
- Byte 3: LO byte of the IP address (`234` from `123.231.12.234`)
- Byte 4: HO byte of the port number (for example, if the port is `258` `[0102h]`, this byte will contain `01`)
- Byte 5: LO byte of the port number (if the port is `258` `[0102h]`, this byte will contain `02`)

Given the ip declaration

```
        ip (123.231.12.234:258)
```

the ip macro will emit the following 6 bytes:

```
        byte 123, 231, 12, 234, 1, 2  ; 0x7B, 0xE7, 0x0C, 0xEA, 0x01, 0x02
```

The ip macro will appear in a header file named *ip.inc* and is relatively simple (with all the heavy lifting done by helper macros in *ip.inc*). The code for the ip macro is the following:

```
; ip-
;
; Data declaration to handle IP addresses
;
; Syntax:
;
;   {label}   ip      (xxx.yyy.zzz.ttt:pppp {,...})
;
; Given an argument of the form
;
;   xxx.yyy.zzz.ttt:ppp
;
; this macro produces a string of the form:
;
;   byte xxx, yyy, zzz, ttt, HO(pppp), LO(pppp)

ip          macro   args:vararg
            local   curIP, exip, output, port
output      textequ <>

            for     curIP, <&args>
exip        textequ extractIP(&curIP)
            ifnb    output
output      catstr  output, <,>
            endif
output      catstr  output, exip
port        textequ getPort(&curIP)
output      catstr  output, <,>, port
            endm
            exitm   <byte &output>
            endm
```

Ultimately (after defining the extractIP and getPort helper macros), this macro produces a text string of the form byte $value_1$, $value_2$, ..., $value_n$, where the $value_i$ values appear in groups of 6 bytes for each IP address the invoker specifies in the ip macro parameter list. This macro executes a loop that processes each of the IP arguments, concatenates the results, and then emits the byte statement to reserve storage for the IP addresses.

The real work is done by the two macro functions extractIP and getPort, which appear as textequ arguments. The extractIP function takes an

IP address of the form *xxx.yyy.zzz.ttt:pppp* (where *xxx.yyy.zzz.ttt* is the IP address and *pppp* is the port number) and returns a string of the form *xxx, yyy, zzz, ttt*. When appended to a `byte` directive, this emits the IP address in network order form.

The `getPort` function extracts the text after the `:` character, converts it to a 16-bit integer, and then emits the big-endian form of the 16-bit value. Concatenating these 2 bytes to the end of the (big-endian) 4-byte IP address produces the 6-byte result.

The `extractIP` function is also relatively simple, as it calls two helper functions to do the real work. The source code for `extractIP` is the following:

```
; extractIP-
;
; Extracts the 4 bytes from the dotted IP address
; and returns them as a big-endian 32-bit value

extractIP   macro   ipAdrs
            local   byte0, byte1, byte2, byte3
            local   rem0, rem1, rem2
            local   ipVal

; Get the first byte of the IP address:

byte0       getByte (<&ipAdrs>, <.>)
rem0        getRem  (<&ipAdrs>, <.>)

byte1       getByte (rem0, <.>)
rem1        getRem  (rem0, <.>)

byte2       getByte (rem1, <.>)
rem2        getRem  (rem1, <.>)

byte3       getByte (rem2, <:>)
ipVal       =       byte0 + (byte1 shl 8) + \
                    (byte2 shl 16) + (byte3 shl 24)

            exitm   %ipVal
            endm
```

The two helper macros are `getByte` and `getRem`. The `getByte` macro extracts the first byte value at the beginning of the string passed to it. The second `getByte` argument is a separator character; for the first three bytes, the separator character is `.`, and for the last byte of the IP address, the separator character is `:` (which separates the last address byte from the port value). Here's the source code for `getByte`:

```
; getByte-
;
; Extract characters up to delimiter (sep) character
; and convert to a byte value.
```

```
getByte     macro   adrs, sep
            local   sepPosn, theByte, byteVal
            local   tAdrs

; Have to convert macro argument to a test string.

tAdrs       textequ <&adrs>

; Locate the "." or ":" character that ends the
; current 8-bit value in the dotted IP address:

sepPosn     instr   1, tAdrs, <&sep>

; If a separator was present, extract the
; preceding byte:

            if      sepPosn gt 0
theByte     substr  tAdrs, 1, sepPosn-1
            else

; Syntax error if separator not found:

            echo    Syntax error in IP address (getByte)
            .err
            exitm   <>
            endif

; Convert the byte value to a number and ensure it
; is less than 256:

byteVal     =       theByte
            if      byteVal gt 255
            echo    IP address contains a value that
%           echo    is greater than 255 (&byte0)
            .err
            exitm   <>
            endif

; Return a textequ directive for this byte value:

            exitm   <textequ &theByte>
            endm
```

In addition to extracting the leading byte from the IP address, the `getByte` macro checks the value of that number to ensure it's in the range 0 to 255.

After extracting a byte from the front of the address string, `extractIP` has to delete that byte so further calls to `getByte` will fetch successive address bytes.

The `getRem` (get remainder) macro handles deleting the leading byte of the address. This macro accepts a string, deletes the leading address byte and separator character, and then returns the resulting string. Here's the source code for `getRem`:

```
; getRem-
;
; Extracts the remainder of the string after the
; IP address byte at the beginning of the string.
; The sep argument is either "." or ":".

getRem      macro   adrs, sep
            local   sepPosn, rem, tAdrs

; Have to convert macro argument to a test string.

tAdrs       textequ <&adrs>

; Determine the location of the "." or ":"
; delimiter:

sepPosn     instr   1, tAdrs, <&sep>

; If the delimiter is present, extract all the text
; beyond the delimiter character:

            if      sepPosn gt 0
rem         substr  tAdrs, sepPosn+1
            else

; If no delimiter, you have a syntax error:

            echo    Syntax error in IP address (getRem)
            .err
            exitm   <>
            endif

; Return the remainder after removing the first
; byte prefix:

            exitm   <textequ &rem>
            endm
```

The current set of macros (`extractIP`, `getRem`, `getByte`, and `ip`) process the 4-byte IP address. All that remains is the `getPort` macro, which extracts the port number of the end of the string. Here's the source code for that macro:

```
; getPort-
;
; Extracts the port value from the end of the IP
; address (after the ":" separator), verifies that
; it's less than 65,536, and returns a big-endian
; 16-bit value:
;
;    <HObyte, LObyte>

getPort     macro   ipAdrs
            local   tAdrs, sepPosn, port
            local   portVal, pOutput
```

```
; Have to convert macro argument to a test string.

tAdrs       textequ <&ipAdrs>

; Determine the location of the ":" delimiter:

sepPosn     instr   1, tAdrs, <:>

; No need to check for ":", extractIP would have
; already complained if it was not present. If it
; is not present, this code may generate some
; additional errors, but that's okay.
;
; Get the port value:

port        substr  tAdrs, sepPosn+1

; Convert port to a numeric value and verify
; that it's less than 65,536:

portVal     =       port
            if      portVal gt 65535
            echo    Port value is out of range.
%           echo    Should be 0..65535, is (&port)
            .err
            exitm   <>
            endif

; Convert portVal to network byte order (big endian):

pOutput     catstr  %(portVal shr 8)+((portVal and 0ffh) shl 8)
            exitm   pOutput
            endm
```

The `getPort` macro checks the result (after extracting it) to ensure its value is in the range 0 to 65,535 (16 bits). Because port numbers are 16-bit values and the `ip` macro emits the data via the `word` directive, the `getPort` macro converts the 16-bit value into big-endian form.

NOTE *All these macros appear in the* ip.inc *header file at* https://artofasm.randallhyde.com.

Listing 1-1 is a trivial little main program that demonstrates using the `ip` macro but doesn't actually do anything when run.

Listing1-1.asm

```
; A program that demonstrates the ip macro that
; provides a data declaration for IP addresses

            option     casemap:none
            include    aoalib.inc   ; AoA library + constants
            includelib aoalib.lib   ; Link in aoalib library
```

```
            include    ip.inc       ; ip macro declaration

            .const

; Program title:

            align   word
ttlStr      byte    "Listing 1-1", 0

            .data

; Demonstrate the use of the ip macro to create some
; IP addresses:

TwoAdrses   ip      (123.222.122.23:258, 111.222.0.121:555)
LocalAdrs   ip      (192.168.2.80:20560)
Watchdog    ip      (192.168.2.19:20651)
GitLab      ip      (10.0.1.41:8080)

            .code

; Here is the main assembly language function:

            public  asmMain
asmMain     proc

; Create a fake display here:

            push    rbp
            mov     rbp, rsp
            sub     rsp, 56
            push    rbx

allDone:    pop     rbx
            leave
            ret     ; Returns to caller

asmMain     endp
            end
```

Listing 1-1: A demonstration of the `ip` macro

As this isn't a runnable program, I simply assembled this source file with the following command to produce a listing file:

```
C:\>ml64 /c /FlListing1-1.lst listing1-1.asm
```

I extracted the following lines from the listing file so you could see the output from the macros:

```
 00000000                       .data

                ; Demonstrate the use of the ip macro to
                ; create some IP addresses:
```

```
 00000000 177ADE7B 0201     TwoAdrses   ip      (123.222.122.23:258,
111.222.000.121:555)
       7900DE6F 2B02
 = 6                szTA    textequ %size TwoAdrses
                %   echo    size TwoAdrses=szTA

 = 12               szofTA  textequ %sizeof TwoAdrses
                %   echo    sizeof TwoAdrses=szofTA

 = 1                lenTA   textequ %length TwoAdrses
                %   echo    length TwoAdrses=lenTA

 = 2                lenofTA textequ %lengthof TwoAdrses
                %   echo    lengthof TwoAdrses=lenofTA

 0000000C 5002A8C0 5050     LocalAdrs   ip      (192.168.2.80:20560)
 00000012 1302A8C0 AB50     Watchdog    ip      (192.168.2.19:20651)
 00000018 2901000A 901F     GitLab      ip      (10.0.1.41:8080)
```

The `ip` macro emits a 6-byte IP address/port value to the object file. Having the macro emit 8 bytes (with 2 bytes of padding at the end) might be more convenient, because it's easier to index into an array of 8 bytes rather than 6. The change is trivial; I will leave that up to you.

The MASM `size(of)` and `length(of)` functions will return expected values when applied to the `ip_adrs` type (or variables of this type). Listing 1-2 is the entire *ip.inc* header file that uses the `ip_adrs` structure (as well as defining other useful structures).

ip.inc

```
; An include file that provides the ip macro
; for creating IP address declarations

            ifndef  ip_inc
ip_inc      =       1

            option     casemap:none

; Some useful structures for holding
; socket addresses (IP address + port number):

ipBytes     struct
b3          byte    ?
b2          byte    ?
b1          byte    ?
b0          byte    ?
ipBytes     ends

portBytes   struct
HO          byte    ?
LO          byte    ?
portBytes   ends

❶ bipadrs     struct
adrs        ipBytes   {}
```

```
port        portBytes {}
bipadrs     ends

; ip_adrs is the actual type that the
; ip macro uses to declare initialized
; IP address variables:

❷ ip_adrs     struct
adrs        dword   ?
port        word    ?
ip_adrs     ends

;-----------------------------------------------------
;
; getByte-
;
; Extracts a single byte from the beginning of an IP
; address string (xxx.yyy.zzz.ttt:pppp). The sep
; argument is the separator (either "." or ":").

❸ getByte     macro   adrs, sep
            local   sepPosn, theByte, byteVal
            local   tAdrs

; Have to convert macro argument to a test string.

tAdrs       textequ <&adrs>

; Locate the "." or ":" character that ends the
; current 8-bit value in the dotted IP address:

❹ sepPosn     instr   1, tAdrs, <&sep>

; If a separator was present, extract the
; preceding byte:

            if      sepPosn gt 0
theByte     substr  tAdrs, 1, sepPosn-1
            else

; Syntax error if separator not found:

            echo    Syntax error in IP address (getByte)
            .err
            exitm   <>
            endif

; Convert the byte value to a number and ensure it
; is less than 256:

byteVal     =       theByte
          ❺ if      byteVal gt 255
            echo    IP address contains a value that
%           echo    is greater than 255 (&byte0)
```

```
                .err
                exitm   <>
                endif

    ; Return a textequ directive for this byte value:

                exitm   <textequ &theByte>
                endm

    ;--------------------------------------------------
    ;
    ; getRem-
    ;
    ; Extracts the remainder of the string after the
    ; IP address byte at the beginning of the string.
    ; The sep argument is either "." or ":".

❻ getRem        macro   adrs, sep
                local   sepPosn, rem, tAdrs

    ; Have to convert macro argument to a test string.

    tAdrs       textequ <&adrs>

    ; Determine the location of the "." or ":"
    ; delimiter:

    sepPosn     instr   1, tAdrs, <&sep>

    ; If the delimiter is present, extract all the text
    ; beyond the delimiter character:

                if      sepPosn gt 0
❼ rem           substr  tAdrs, sepPosn+1
                else

    ; If no delimiter, you have a syntax error:

                echo    Syntax error in IP address (getRem)
                .err
                exitm   <>
                endif

    ; Return the remainder after removing the first
    ; byte prefix:

                exitm   <textequ &rem>
                endm

    ;--------------------------------------------------
    ;
    ; extractIP-
    ;
```

```
; Extracts the four bytes from the dotted IP address
; and returns them as a big-endian 32-bit value:

❽ extractIP   macro   ipAdrs
              local   byte0, byte1, byte2, byte3
              local   rem0, rem1, rem2
              local   ipVal

; Get the first byte of the IP address:

byte0         getByte (<&ipAdrs>, <.>)
rem0          getRem  (<&ipAdrs>, <.>)

byte1         getByte (rem0, <.>)
rem1          getRem  (rem0, <.>)

byte2         getByte (rem1, <.>)
rem2          getRem  (rem1, <.>)

byte3         getByte (rem2, <:>)
ipVal         =       byte0 + (byte1 shl 8) + \
                      (byte2 shl 16) + (byte3 shl 24)

              exitm   %ipVal
              endm

;-----------------------------------------------------
;
; getPort-
;
; Extracts the port value from the end of the IP
; address (after the ":" separator), verifies that
; it's less than 65,536, and returns a big-endian
; 16-bit value:

❾ getPort     macro   ipAdrs
              local   tAdrs, sepPosn, port
              local   portVal, pOutput

; Have to convert macro argument to a test string.

tAdrs         textequ <&ipAdrs>

; Determine the location of the ":" delimiter:

sepPosn       instr   1, tAdrs, <:>

; No need to check for ":", extractIP would have
; already complained if it was not present. If it
; is not present, this code may generate some
; additional errors, but that's okay.
;
; Get the port value:
```

```
port        substr  tAdrs, sepPosn+1

; Convert port to a numeric value and verify
; that it's less than 65,536:

portVal     =       port
            if      portVal gt 65535
            echo    Port value is out of range.
%           echo    Should be 0..65535, is (&port)
            .err
            exitm   <>
            endif

; Convert portVal to network byte order (big endian):

❶ pOutput     catstr  %(portVal shr 8)+((portVal and 0ffh) shl 8)
            exitm   pOutput
            endm

;------------------------------------------------
;
; ip-
;
; Data declaration to handle IP addresses.
;
; Syntax:
;
;   {label}   ip      (xxx.yyy.zzz.ttt:pppp {,...})
;
; Given an argument of the form
;
;   xxx.yyy.zzz.ttt:ppp
;
; this macro produces a string of the form:
;
;   ip_adrs {xxx + (yyy shl 8) + (zzz shl 16) +
;               (ttt shl 24)}, (pppp shr 8) +
;                    ((pppp and 0ffh) shl 8)}

ip          macro   args:vararg
            local   curIP, exip, output, port

output      textequ <>

            for     curIP, <&args>
exip        textequ extractIP(&curIP)
            ifnb    output
output      catstr  output, <,>
            endif
output      catstr  output, <{>, exip
port        textequ getPort(&curIP)
output      catstr  output, <,>, port, <}>
% echo prt=port
            endm
```

```
            exitm   <ip_adrs &output>
            endm

            endif   ; ip_inc
```

Listing 1-2: The ip.inc *header file*

The `bipadrs` structure ❶ holds the IP address and port values as two arrays of bytes, while the `ip_adrs` structure ❷ maps to this same 6-byte structure via the `dword` and `word` fields. This provides two views of the data and makes it easier to access based on how you are using it.

The `getByte` macro ❸ extracts a leading byte value from the *xxx.yyy.zzz.aaa* IP address string and returns the byte value as the (macro function) result. The macro then grabs all the characters up to the separator character (`.` or `:`) ❹, converts this to a small decimal integer value ❺, and checks to ensure it's in the range 0 to 255.

The `getRem` macro ❻ extracts all the text after the leading integer value ❼ and then returns this text as the macro function result.

The `extractIP` ❽ macro extracts the 4 bytes of the IP address and creates a `dword` value for output by the `ip` macro. The `getPort` macro ❾ extracts the port value and creates a word value for use by the `ip` macro ❿.

Listing 1-3 is a simple main program, similar to Listing 1-1, that demonstrates the operation (at assembly time) of the `ip` macro.

Listing1-3.asm

```
; A program demonstrating the (new) ip macro that
; provides a data declaration for IP addresses with
; the ip_adrs structure

            option     casemap:none
            include    aoalib.inc   ; AoA library + constants
            includelib aoalib.lib   ; Link in aoalib library

            include    ip.inc       ; ip macro declaration

            .const

; Program title:

            align   word
ttlStr      byte    "Listing 1-3", 0

            .data

; Demonstrate the use of the ip macro to create some
; IP addresses:

TwoAdrses   ip      (123.222.122.23:258, 111.222.000.121:555)
szTA        textequ %size TwoAdrses
%           echo    size TwoAdrses=szTA

szofTA      textequ %sizeof TwoAdrses
%           echo    sizeof TwoAdrses=szofTA
```

```
lenTA       textequ %length TwoAdrses
%           echo    length TwoAdrses=lenTA

lenofTA     textequ %lengthof TwoAdrses
%           echo    lengthof TwoAdrses=lenofTA

LocalAdrs   ip      (192.168.2.80:20560)
Watchdog    ip      (192.168.2.19:20651)
GitLab      ip      (10.0.1.41:8080)

myIP        ip_adrs {01020304h, 0102h}
            .code

; Here is the main assembly language function:

            public  asmMain
asmMain     proc

; Create a fake display here:

            push    rbp
            mov     rbp, rsp
            sub     rsp, 56
            push    rbx

allDone:    pop     rbx
            leave
            ret     ; Returns to caller

asmMain     endp
            end
```

Listing 1-3: A demonstration of the new version of the ip macro

During assembly, MASM produces the following output from the `echo` statements in the main program:

```
size TwoAdrses=6
sizeof TwoAdrses=12
length TwoAdrses=1
lengthof TwoAdrses=2
```

This is the output you would expect for an IP address data structure.

As a data declaration, the `ip` macro isn't perfect. While standard MASM data declarations allow you to create arrays such as

```
byteArray  byte     256 dup (0)
```

the `ip` macro doesn't (currently) allow the use of the `dup` operator. Modifying the macro to allow this is probably possible; I will leave doing so up to you. However, I find that it's just as easy to do something like

```
ipArray    ip_adrs  256 dup ({adrs,port})
```

and dispense with all the effort needed to extend the `ip` macro. When you initialize an `ip_adrs` structure in this fashion, you have to manually convert the `dword` and `word` values to network byte order, which is easy enough to do by writing a couple of macros, as Listing 1-4 shows.

Listing1-4.asm

```
_bswap32    macro   value
            local   txtValue
txtValue    textequ %((value shr 24) and 0ffh) + \
                      ((value shr 8) and 0ff00h) + \
                      ((value shl 8) and 0ff0000h) + \
                      (value and 0ffh) shl 24
            exitm   txtValue
            endm

_bswap16    macro   value
            local   txtValue
txtValue    textequ %((value shr 8) and 0ffh) + \
                      (value and 0ffh) shl 8
            exitm   txtValue
            endm

tt textequ %_bswap32(01020304h)
%echo tt

tt textequ %_bswap16(0102h)
%echo tt
            end
```

Listing 1-4: The `_bswap32` and `_bswap16` macros

If you compile Listing 1-4, MASM produces the following output:

```
67305985
513
```

Note that the 32-bit byte swap of `01020304h` is `04030201h`, which is 67305985_{10}. Likewise, the 16-bit byte swap of `0102h` is `0201h`, which is 513_{10}. Having a `_bswap64` macro at one point or another might be useful. I'll leave it up to you to write that macro.

Given the `_bswap32` and `_bswap16` macros, you could initialize an `ip_adrs` structure thusly:

```
ipArray    ip_adrs  256 dup ({_bswap32(adrs),_bswap16(port)})
```

Rather than specifying the actual IP address as a 32-bit number, it might be better to write a `_dotted` macro function that converts an IP4 *dotted address* into a 32-bit big-endian value. For example:

```
ipArray    ip_adrs  256 dup ({_dotted(x.y.z.t),_bswap16(port)})
```

This macro is a slight modification of the `extractIP` macro appearing in Listing 1-2. I'll leave it up to you to create the `_dotted` macro from `extractIP`.

1.8 Context-Free Macros

Standard MASM macros are like regular languages. A *regular language* is a set of strings (with all those strings belonging to that language). A *regular expression* is a mechanism that generates all possible strings in a particular regular language. Consider the following regular expression:

```
[A-Za-z][A-Za-z0-9_]*
```

This generates strings that begin with an ASCII alphabetic character and are followed by zero or more alphabetic, numeric, or underscore characters (that is, identifiers in a programming language like C/C++).

> **REGULAR EXPRESSIONS AND STRINGS**
>
> Those who've not taken an automata theory course may think that regular expressions recognize or match strings. Technically, finite-state automata *recognize* strings, and regular expressions *generate* strings. Mathematically, however, you can show that regular expressions and finite-state automata are equivalent; that is, you can convert a regular expression to a finite-state automata (and vice versa). So it's not entirely incorrect to say that you can use regular expressions to recognize certain classes of strings.

Now consider the following macro and invocations:

```
genID   macro
        local       anID
        exitm       <&anID>
        endm

id1     texteq      genID()
% echo &id1
id1     texteq      genID()
% echo &id1
```

Assuming you assemble this code in an assembly source file with little else, you should get output like the following:

```
??0000
??0001
```

The `genID` macro generates a new (typically unique) identifier as its text output on every invocation.

This is why I claim standard MASM macros are like regular expressions. Invoking a macro generates a string (which could be empty) at the

point of invocation. Note that macro invocations typically don't have any memory; the string that one macro invocation produces does not affect the string produced by a later macro invocation.

One macro invocation could affect a later invocation of that same macro. By storing some information in a global value (a MASM constant you define outside the macro), you can pass data from one macro to another. Here's a simple example:

```
macConstant = 0

incConst    macro
macConstant = macConstant+1
            exitm <macConstant-1>
            endm

x           =       incConst()
tx          textequ %x
% echo x=tx
x           =       incConst()
tx          textequ %x
% echo x=tx
```

This produces the following output:

```
x=0
x=1
```

By manipulating constants outside the macro (or, more correctly, constants that are not local to the macro), it's possible to have one macro invocation affect the operation of a different macro invocation.

In the previous example, the invocation of `incConst` affected later invocations of that same macro. However, nothing is stopping you from having an invocation of one macro affect the operation of a different macro. This section is going to take advantage of that fact to implement some interesting macros.

Suppose, for a moment, that you want to write a macro to simulate an `if-endif` statement from an HLL, using syntax such as the following (the underscore prefixes are present to avoid conflicts with the MASM conditional assembly directives):

```
        _if  register
          .
          .
          .
        _endif
```

This `if` statement takes a very simple expression: a single register. If the register contains `0`, control transfers to the statement following the `_endif` clause; if the register contains a nonzero value, control falls through to the

first statement following the _if clause. In assembly language, this sequence is relatively easy to encode:

```
;       _if  register

        test register, register
        jz   skipIf
         .
         .
         .
;       _endif
skipIf:
```

Writing a macro to handle this is complicated by the _if macro needing to communicate the name of the label to the _endif macro, which emits the target label that's the destination of the jz instruction that _if emits. As you've seen already, the _if macro can store this destination information in a global (to the macro) constant that the _endif macro can reference. Consider the following implementation of these two macros:

```
; _if and _endif macros-
;
; Simulate an HLL if/endif statement.
; Expression must be a register. Body
; of _if statement executes if the
; register contains a nonzero value.

_if         macro   register
            local   destination
_ifdest     textequ <&destination>
            test    register, register
            jz      _ifdest
            endm

_endif      macro
_ifdest:
            endm
```

Here is the sample output from a relatively trivial invocation of the _if and _endif macros:

```
                            _if     eax
 00000004  85 C0          1 test    eax, eax
 00000006  74 10          1 jz      _ifdest

 00000008  FF 15 00000000 E call    print
 0000000E  45 41 58 20 21   byte    "EAX != 0", nl, 0
       3D 20 30 0A 00

                            _endif
 00000018                 1 _ifdest:
```

Note that _ifdest is a textequ that just happens to contain the string ??0000 in the simple version of the program I compiled. Therefore, _ifdest expands to ??0000 in both places in the program (after the jz instruction and as the statement label at the end of this example).

Unfortunately, these two macros have a severe limitation: They might not work properly if multiple invocations appear in your program, and they don't always fail. Consider the following example:

```
            _if     eax

            call    print
            byte    "EAX != 0", nl, 0

            _endif

            _if     ecx

            call    print
            byte    "ECX != 0", nl, 0

            _endif
```

Here's the (reasonable) code that MASM emits for this example:

```
                              _if     eax
00000004  85 C0             1 test    eax, eax
00000006  74 10             1 jz      _ifdest
00000008  FF 15 00000000 E  1 call    print
0000000E  45 41 58 20 21      byte    "EAX != 0", nl, 0
      3D 20 30 0A 00
                              _endif
00000018                    1 _ifdest:

                              _if     ecx
00000018  85 C9             1 test    ecx, ecx
0000001A  74 10             1 jz      _ifdest
0000001C  FF 15 00000000 E  1 call    print
00000022  45 43 58 20 21      byte    "ECX != 0", nl, 0
      3D 20 30 0A 00
                              _endif
0000002C                    1 _ifdest:
```

Note that MASM substitutes ??0000 for the first two occurrences of _ifdest, and ??0001 for the second two occurrences of _ifdest (or comparable values). This is because the _if macro generates a new label to attach to _ifdest whenever you invoke it.

The problem occurs when you attempt to nest the _if-_endif macros. Consider the following macro invocations:

```
            _if     eax

            _if     ebx
            call    print
            byte    "EAX and EBX != 0", nl, 0

            _endif

            call    print
            byte    "At least EAX != 0", nl, 0

            _endif
```

MASM chokes on this, generating the following error messages:

```
 Assembling: listing.asm
listing.asm(67) : error A2005:symbol redefinition : ??0001
 _endif(1): Macro Called From
  listing.asm(67): Main Line Code
listing.asm(56) : error A2006:undefined symbol : ??0000
 _if(4): Macro Called From
  listing.asm(56): Main Line Code
```

Here's what has gone wrong:

1. The first invocation of _if assigns the text value ??0000 to the global textequ _ifdest.
2. The next invocation of _if (note this occurs before encountering an _endif macro) replaces the value of _ifdest with ??0001.
3. When MASM processes the first _endif macro invocation, it emits the text held in _ifdest (currently ??0001) as a statement label.
4. When MASM processes the second _endif macro invocation, it emits the text held in _ifdest (which is still ??0001) as a statement label. As label ??0001 already exists, this results in the symbol redefinition error.
5. In the meantime, the first _if macro generated a jz ??0000 instruction. The label ??0000 has never been defined (the second invocation of _endif emitted ??0001 rather than ??0000). As a result, symbol ??0000 is undefined, and MASM complains about this.

The problem is that the nested _if overwrites the value held in the global _ifdest constant. For this macro to work properly, the _if and _endif macros need to preserve and restore the existing _ifdest value (specifically, _if needs to push the original value onto a stack, and _endif needs to pop that value off the stack).

Because this scheme needs a stack to work properly, the computational model is that of a *push-down automata (PDA)*. Just as finite-state machines

are equivalent to regular expressions, PDAs are equivalent to context-free grammars. Hence this section's name: You're creating macros that operate as PDAs and, therefore, generate strings in a context-free language (which is what a context-free grammar does).

To resolve the existing issue with the `_if` and `_endif` macros, you have to substitute a stack (with `push` and `pop` operations) in place of the single global value held in `_ifdest`. Although there's no compile-time `push` and `pop` operations for constant values, you can write macros to do that work for you.

1.8.1 The _push and _pop Macros (Version 1)

There are two ways to write the `push` and `pop` macros (I'll refer to these macros as `_push` and `_pop` going forward). The first version uses a single compile-time variable (a `textequ` text constant) to hold all the stack data. This approach has the advantage of being easy to encode and understand, though a size limitation (of roughly less than 240 characters) is associated with all the text appearing in the `textequ` constant. If you don't need an especially deep stack (say, fewer than 20 entries), this is probably the best solution to use.

The first version creates a global (compile-time) `textequ` constant value to hold the stack data. When (properly) implementing the `_if` and `_endif` macros with a `textequ` stack, you'll need one stack level for each nested pair at the deepest nesting levels. As it's rare to see an `if` statement nested more than about five or six levels deep, this variant of the stack will typically suffice.

Initially, the stack is empty, so you can originally define the stack by using a statement such as the following:

```
ifStk    textequ  <>
```

Text that's pushed onto this stack will be *prepended* to the beginning of the string. To easily pop data from this text string, the `push` operation will insert a special separator character between items on the stack. It doesn't matter what the special character is, only that it must not appear in normal stack data. For the `_push` macro I'm about to present, I've chosen to use the `:` character as the separator. The `_push` macro needs two arguments: the global constant that will hold the stack data and the data value to push onto the stack. The macro invocation will take the following form:

```
        _push  (stackConst, <text to push>)
```

The `_push` macro uses the MASM functional macro syntax (hence the parentheses around the arguments). Technically speaking, this isn't necessary. Writing the macro with the following syntax is easy enough:

```
        _push  stackConst, <text to push>
```

You will soon see, however, that the `_pop` macro will require the functional syntax. I chose to implement `_push` with the functional syntax simply to ensure that both macros use the same syntax.

As noted previously, the _push macro's first argument is the stack to use, and the second argument is the value to push on that stack. Consider the following two invocations of _push:

```
        _push(ifStk, <World>)
        _push(ifStk, <Hello >)
```

The first invocation results in the ifStk text string containing the data World:. After the execution of the second invocation, the ifStk string will contain the following:

```
        Hello :World:
```

Notice that successive macro invocations insert the data at the beginning of the string (that is, they prepend the data to the string).

Here's the implementation of this _push macro:

```
; _push macro-
;
; First argument is the name of a compile-time stack
; object (textequ constant, typically <> when the
; stack is empty). Second argument is a value (text)
; to "push" onto the stack. Inserts "value:" at the
; beginning of the specified stack object.
;
; Returns the new stack value including the pushed
; value (which should be assigned to the original
; stack variable).
;
; Note: Technically, this could be a normal macro,
; not a "functional" macro. The "exitm <>" statement
; appears only to force this macro to have parentheses
; around the arguments so that the syntax matches that
; of the "_pop" macro.
;
; Calling sequence:
;
;           _push   (stackVar, <text to push>)
;
; The angle brackets ("<" and ">") aren't necessary if
; the value being pushed is a single word.

_push       macro   theStack, theValue
            local   result
result      catstr  <&theValue>, <:>, <&theStack>
theStack    textequ result
            exitm   <>
            endm
```

This macro simply returns the empty string as its result. Therefore, when you invoke the macro as in the previous examples, it replaces the

whole statement with the empty string (it's as though you'd inserted a blank line at that point).

The _pop macro is considerably more complex. First, a couple of things can go wrong when you attempt to pop data from a stack:

- The stack could be empty, which is an error.
- The stack could contain text but no special separator character (: in the current implementation).

Unlike the _push macro, _pop is a functional macro that returns a value—the value popped from the stack. Therefore, the calling sequence is slightly different insofar as all _pop invocations must have a label in the label field of the instruction:

```
result     textequ _pop(theStack)
```

As you can see, another difference is that the _pop macro has only a single operand (the stack from which to pop data). Like the _push macro, the _pop macro also adjusts the stack object you pass it after the operation is complete, removing the first item and the following separator character.

Here's the implementation of the _pop macro:

```
; _pop macro-
;
; Passed a single argument that's the name of a global
; compile-time constant containing stack information.
; This macro pops the first item off the stack (all
; text up to the first ":" character) and returns that
; as the "function" result. This macro also removes
; the popped item from the stack specified as this
; macro's argument.
;
; Usage syntax:
;
; pData     _pop(stackVar)
;
; Call will put popped text into "pData" and
; replace the value of "stackVar" with its string
; minus the popped data.

_pop        macro   theStack
            local   result, rPosn, slen

; First, check for an empty stack:

            ifb     theStack
            echo    Attempt to pop an empty stack!
            .err
            exitm   <>
            endif

; Find the first ":" character in the stack variable:
```

```
rPosn       instr   1, theStack, <:>
            if      rPosn le 0
            echo    Malformed stack data in pop.
            .err
            exitm   <>
            endif

; Okay, this macro has found the ":" character,
; extract all the data up to, but not including,
; the ":" character:

result      substr  theStack, 1, rPosn-1

; Remove the popped item (including the ":" char)
; from the beginning of the stack variable:

slen        sizestr theStack
            if      slen gt rPosn
theStack    substr  theStack, rPosn+1
            else
theStack    textequ <>
            endif

; Return the popped text as part of a "textequ"
; directive.

            exitm   <!<&result!>>
            endm
```

Look to the comments for more information regarding what's going on in this macro.

Listing 1-5 assembles and demonstrates the use of these two macros. The program itself does nothing at runtime; all the exciting stuff happens during assembly.

Listing1-5.asm

```
; A program that demonstrates the _push and _pop
; macros (version 1 macros)

            include    alalib.inc
            includelib alalib.lib
            option     casemap:none

use1var      =         1
            include    ctPushPop.inc

            .const

; Program title:

            align   word
ttlStr      byte    "Listing 1-5", 0
```

```
            .code

; Here is the main assembly language function:

            public  asmMain
asmMain     proc

; Create a fake display here:

            push    rbp
            mov     rbp, rsp
            sub     rsp, 56
            push    rbx

; Must create an ifStk for the _push and _pop macros to use:

ifStk       textequ <>
%           echo    Before push #1, the stack is <<ifStk>>
            echo

            _push(ifStk, <Hello >)
%           echo    After push #1, the stack is <<ifStk>>
            echo

            _push(ifStk, <Hello >)
%           echo    After push #2, the stack is <<&ifStk>>
            echo

tos         textequ _pop(ifStk)
%           echo After 1st pop, item popped is <<&tos>>
%           echo and stack contains <<&ifStk>>
            echo

nos         textequ _pop(ifStk)
%           echo After 2nd pop, item popped is <<&nos>>
%           echo and stack contains <<&ifStk>>

allDone:    pop     rbx
            leave
            ret     ; Returns to caller

asmMain     endp
            end
```

Listing 1-5: A demonstration of the _push and _pop macros

During assembly, here's the output from Listing 1-5:

```
C:\>nmake /s /c /nologo /f listing1-5.mak
 Assembling: listing1-5.asm
Before push #1, the stack is <<>>

After push #1, the stack is <<Hello :>>
```

```
After push #2, the stack is <<Hello :Hello :>>

After 1st pop, item popped is <<Hello >>
and stack contains <<Hello :>>

After 2nd pop, item popped is <<Hello >>
and stack contains <<>>
```

As you can see, this form of the _push and _pop macros works properly.

1.8.2 The _push and _pop Macros (Version 2)

The second version of the _push and _pop macros uses a slightly different approach. Rather than using a single textequ constant to hold all the stack data, this version uses a separate textequ constant to hold each element of the stack. This implementation doesn't suffer as badly from the MASM line-length limitation of 240 characters; while each item you push onto the stack is limited to 240 characters, there's no limit on the total number of stack entries or their cumulative size (unlike the first implementation).

Although this implementation uses a separate textequ variable for each element of the stack, you do not have to guess at the maximum stack depth beforehand (and define that many stack elements). Instead, the _push macro will automatically create these stack entries as it needs them. If the stack already contains n entries and you try to push $n + m$ entries onto the stack, the _push will create m new entries on the stack. This is completely transparent to the user of the _push macro. Here's the implementation of these variants of the _push and _pop macros:

```
; As for version (1) of the _push macro, this variant
; requires two arguments: a stack and a value to push.
; However, there is a big difference between this
; implementation and the version (1) implementation
; with respect to the stack argument. In version (1),
; the stack was a text constant that held all the
; stack data. In version (2), the stack is simply
; a numeric constant the specifies the number of
; stack elements currently present. Initially, this
; stack object should be assigned the value 0,
; as follows:
;
; someStk   =       0
;
; This macro will automatically create text constant
; names taking the form "stkName_n" where "stkName"
; is the name of the stack variable (e.g., "someStk")
; and "n" is the current stack element (e.g., "0"
; through the current depth of the stack). For
; example, if you push two items onto the stack,
; the _push macro will set the stack variable to
; the value "2" and create the following two
; text constants:
;
```

```
; stkName_0 textequ <1st value pushed>
; stkName_1 textequ <2nd value pushed>
;
; The important thing to keep in mind is that you
; must not use names of the form "stkName_n" (where
; stkName is a stack identifier and "n" is a
; decimal value) in your programs as this may create
; duplicate symbol errors in your code.

_push       macro   theStack, theValue
            local   lclName

; Create a symbol of the form "_xxx_nnn" where xxx is
; the text passed in as theStack and nnn is the current
; value of theStack (initially 0).
;
; Set this symbol to the value specified by theValue.
; The stack consists of a sequence of symbols of the
; form "_xxx_nnn" where "nnn" is the position on the
; stack where the value is stored and theValue is the
; value held in that stack entry.

lclName    catstr    <_>,<&theStack>,<_>,%theStack
lclName    =         theValue

; Bump up the stack pointer so the next
; push operation will store its value in
; the next greater stack element:

theStack    =       theStack + 1
            exitm   <>
            endm

; _pop macro-
;
; "theStack" is a global numeric variable
; whose value is the current size of the stack.
; Fetch the item at element(theStack - 1) and
; return this as the function result. Also,
; decrement the stack pointer.

_pop        macro   theStack
            local   suffix

; First, check for an empty stack:

            if      theStack le 0
            echo    Attempt to pop an empty stack!
            .err
            exitm   <>
            endif

; Get the result from the "top of stack." This is
; the value at element "theStack - 1" in the array
; of stack elements.
```

```
theStack    =       theStack - 1

; Generate an identifier that corresponds to
; the element on the top of the stack array (this will
; be an identifier of the form "theStack_nnn" where
; theStack is the name of the stack object and "nnn"
; is the index to the current top of stack):

suffix  textequ  %@CatStr(<_>,<&theStack>,<_>,%theStack)

; Return the popped text:

            exitm   %suffix
            endm

; _peek macro-
;
; "theStack" is a global numeric variable
; whose value is the current size of the
; stack. Fetch the item at element(theStack - 1)
; and return this as the function result.

_peek       macro   theStack
            local   suffix, identifier

; First, check for an empty stack:

            if      theStack le 0
            echo    Attempt to pop an empty stack!
            .err
            exitm   <>
            endif

; Extract the item from the top of stack:

theStack    =       theStack - 1

; Generate an identifier that corresponds to
; the element on the top of the stack array (this will
; be an identifier of the form "theStack_nnn" where
; theStack is the name of the stack object and "nnn"
; is the index to the current top of stack):

suffix  textequ  %@CatStr(<_>,<&theStack>,<_>,%theStack)

; Restore the stack pointer:

theStack    =       theStack + 1

; Return the popped text:

            exitm   %suffix
            endm
            endm
```

Listing 1-6 is a minor modification to Listing 1-5 that demonstrates calls to version 2 of the _push and _pop macros.

Listing1-6.asm

```
; A program that demonstrates version 2 of
; the _push and _pop macros

            option     casemap:none
            include    ctPushPop.inc

            .const

; Program title:

            align   word
ttlStr      byte    "Listing 1-6", 0

            .code

; Here is the main assembly language function:

            public  asmMain
asmMain     proc

; Create a fake display here:

            push    rbp
            mov     rbp, rsp
            sub     rsp, 56
            push    rbx

ifStk       =       0
            _push(ifStk, 1234)
            _push(ifStk, 5678)

tos         =       _pop(ifStk)
tosTxt      textequ %tos
%           echo    After 1st pop, item popped is &tosTxt

nos         =       _pop(ifStk)
nosTxt      textequ %nos
%           echo After 2nd pop, item popped is &nosTxt

allDone:    pop     rbx
            leave
            ret     ; Returns to caller

asmMain     endp
            end
```

Listing 1-6: A test program for version 2 of the _push and _pop macros

Here's the sample output you get when you assemble the source code in Listing 1-6:

```
C:\>nmake /s /c /nologo /f listing1-6.mak
 Assembling: Listing1-6.asm
After 1st pop, item popped is 5678
After 2nd pop, item popped is 1234
```

As you can see, this code properly pushed, then popped, `5678` and `1234`.

1.8.3 The _if and _endif Macros

Now that the stack operations are available, it's possible to implement context-free versions of the `_if` and `_endif` macros from earlier that allow full nesting of the `if` statements. The solution to the nesting issue is to push the `jz` target address on a stack when processing the `_if` macro and pop this target address from the stack when processing the `_endif` macro. Whenever nested `if-endif` statements appear, the nested `if` statement preserves the jump target for the enclosing `if-endif` and restores it (back to the top of stack) when the nested `if-endif` is complete.

The implementation of the context-free `_if` macro is relatively straightforward:

```
_ifStk      =       0       ; Use version 2 of _push/_pop.
_ifDest     =       0
_if         macro   register
            local   destination

            _push(_ifStk, %_ifDest)
            test    register, register
            jz      @CatStr( <_ifDest_>, %_ifDest )
_ifDest     =       _ifDest + 1
            endm
```

MASM converts the destination local symbol in the `_if` macro to a text string such as `??0000`. The code pushes the destination label onto the stack for later retrieval by the `_pop` macro in `_endif`. Then the `_if` macro emits the `test` instruction and the `jz destination` instruction (substituting `??0000` for `destination` to produce the instruction `jz ??0000`).

The implementation of the `_endif` macro is also straightforward:

```
_endif      macro
            local   popVal, theDest
popVal      =       _pop(_ifStk)
theDest     textequ  <_ifDest_>, %popVal
theDest:
            endm
```

This macro simply pops the current label off the top of the stack and emits that label into the code stream. This forms the target address of the corresponding `jz` instruction from the `_if` macro.

Listing 1-7 contains a small demonstration program for the _if and _endif macros. This isn't (exactly) a runnable program. The corresponding make file (*listing1-7.mak*) produces a listing file that you can observe to see how well these macros behave.

Listing1-7.asm

```
; A program that demonstrates _if and _endif
; context-free macros using the _push and _pop
; macros (either implementation, version 1 or 2
; of _push and _pop will work fine here)

            option     casemap:none
            include    aoalib.inc   ; AoA library + constants
            includelib aoalib.lib   ; Link in aoalib library
            include    ctPushPop.inc

            .const

; Program title:

            align   word
ttlStr      byte    "Listing 1.7", 0

; _if and _endif macros-
;
; Simulate an HLL if/endif statement.
; Expression must be a register. Body
; of _if statement executes if the
; register contains a nonzero value.
;
; Context-free version that saves _if target
; address on the stack for use by _endif.

_ifStk      =       0       ; Use version 2 of _push/_pop.
_ifDest     =       0

_if         macro   register
            local   destination

            _push(_ifStk, %_ifDest)
            test    register, register
            jz      @CatStr( <_ifDest_>, %_ifDest )
_ifDest     =       _ifDest + 1
            endm

_endif      macro
            local   popVal, theDest
popVal      =       _pop(_ifStk)
theDest     textequ  <_ifDest_>, %popVal
theDest:
            endm

            .data

            .code
```

```
; Here is the main assembly language function:

            public  asmMain
asmMain     proc

            push    rbp
            mov     rbp, rsp
            sub     rsp, 56
            push    rbx

            mov     eax, 1 ; Initialize EAX and EBX
            mov     ebx, 1 ;   to hold nonzero values.
; Outermost _if statement invoked here:

            _if     eax
            call    print
            byte    "EAX != 0", nl, 0

; Nested _if statement:

            _if     ebx
            call    print
            byte    "EBX is also non-zero", nl, 0
            _endif  ; ebx _if statement

; Single nop so that the target locations for the nested
; and nesting _if statements are different:

            nop
            _endif  ; eax _if statement

allDone:    pop     rbx
            leave
            ret     ; Returns to caller

asmMain     endp
            end
```

Listing 1-7: An example program that demonstrates the use of nested `_if` and `_endif` macros

Rather than run this program to test the results, let's just look at the assembly listing to see how well these macros have worked. Here's the macro expansion for the `_if`-`_endif` invocations:

```
; Outermost _if statement invoked here:

                                    _if     eax
  09  85 C0             1   test    eax, eax
❶ 0B  74 31             1   jz      ??0000
  0D  FF 15 00000000 E      call    print
  13  45 41 58 20 21        byte    "EAX != 0", nl, 0
      3D 20 30 0A 00

; Nested _if statement:
```

```
                                _if     ebx
  1D  85 DB                 1   test    ebx, ebx
❷ 1F  74 1C                 1   jz      ??0003
  21  FF 15 00000000 E          call    print
  27  45 42 58 20 69            byte    "EBX is also "
      73 20 61 6C 73             byte    "non-zero", nl, 0
      6F 20 6E 6F 6E
      2D 7A 65 72 6F
      0A 00
                                _endif  ; ebx _if statement
❸ 3D                        1   ??0006:

; Single nop so that the target locations for
; the nested and nesting _if statements are
; different:

  3D  90                        nop
                                _endif  ; eax _if statement
❹ 3E                        1   ??0009:
```

If you look closely, you'll notice what seems to be a problem. The first `_if` macro's `jz` instruction transfers control to the label `??0000` ❶, and the second `_if` macro's `jz` instruction transfers control to the label `??0003` ❷. Neither of those labels appear in this code. In fact, where you would expect `??0003` to appear, you see `??0006` ❸, and where you'd expect `??0000`, you see `??0009` ❹. What has gone wrong?

Actually, nothing is wrong with this code. If you look in the symbol table listing (at the end of the listing file produced during assembly), you'll find the following definition for symbol `??0006`:

```
??0006 . . . . . .          Text     ??0008
```

This means that `??0006` is a text object that expands to `??0008`. That's still not the correct target label (`??0003`). However, looking up `??0008` in the symbol table shows

```
??0008 . . . . . .          Text     ifStk_1
```

and looking up `ifStk_1` gives us this:

```
ifStk_1  . . . . . .        Text     ??0003
```

This is where the correct target label appears. Several layers of textual substitution are taking place here. However, as long as MASM finds a text-type symbol, it will keep substituting the text for that symbol; if the result is also a text symbol, MASM will recursively replace the symbol with its text until the result is not a text identifier. In this particular case, MASM performs the following substitutions:

```
??0006 -> ??0008 -> ifStk_1 -> ??0003
```

Using this same process, MASM produces ??0000 from ??0009 with the following substitution sequence:

```
??0009 -> ??000B -> ifStk_0 -> ??0000
```

As the final conclusive proof, consider the object (numeric machine) code that MASM produces for the two jz instructions. For the first jz, you have the following:

```
0B  74 31             1   jz      ??0000
0D  FF 15 00000000 E      call    print
```

The first hexadecimal value on the line (0B) is the offset into the (current) code section where the machine code for the jz instruction appears. The second hexadecimal value on the line (74) is the opcode byte for the jz instruction. The third hexadecimal value on the line is the offset, from the next instruction, to the target/destination address. The address of the next instruction is 0Dh (the first hexadecimal value at the beginning of the next line). Therefore, the jz instruction will transfer control to offset 31h + 0Dh, or 3Eh, if the zero flag is set. If you look at the code at offset 3Eh in the code section, you find the following:

```
                            _endif  ; eax _if statement
0000003E            1   ??0009:
```

Lo and behold, this is the target label ??0009.

If you apply this same process to the second _if statement, you'll find that its jz instruction transfers control to location 1Ch + 21h (3Dh). Looking at the code at offset 3Dh in the code section, you find this:

```
                            _endif  ; ebx _if statement
0000003D            1   ??0006:
```

So, ??0006 is the correct target label for this jz instruction.

1.8.4 More on Context-Free Macros

This section has presented a simple macro for an if statement. Naturally, you could make lots of improvements to this macro. For example, why not support _elseif and _else clauses? For that matter, why not add in _while-_endwhile, _for- _endfor, _switch- _case-_endcase, and other HLL-like statements while you're at it? Then, of course, there's this macro's abysmal support for Boolean expressions. Full arithmetic expressions are another example of a context-free grammar that you could implement using a stack and context-free macros.

Sadly, that full macro implementation is beyond the scope of this book. (I cannot, for example, assume that readers are sufficiently grounded in compiler theory to understand how such code would work.) But this book doesn't have to present such macros; they already exist. The MASM32

software development kit, or SDK (which, despite its name, supports 64-bit code), includes a macro library named *vasily.inc* that provides a set of macros for HLL-like statements in assembly language. Check out the downloads at *https://masm32.com*.

1.9 Performance Impact of Macro Programming

Although you can do very powerful things with MASM's macros, keep in mind that this power and abstraction come at a cost. If you go crazy simulating HLLs in your assembly language code, your programs will likely be larger and slower than a comparable program written in an HLL. This is because those HLL compilers have optimizations that don't exist in macro-driven assembly code. While you can write assembly code that looks similar to some HLLs with appropriate macros, why go through the pain of writing the code in assembly if you could have written it in an HLL in the first place?

Some macros, such as invoking procedures in an HLL-like fashion, won't harm your assembly language source code. But if the larger percentage of the statements in your assembly language source code are macros simulating HLL statements, you might want to rethink your language choice.

Another performance factor to consider when using macros is the performance of MASM itself. MASM's macro processor is not particularly fast (it is, after all, a *pure text* interpreter system). If you start using lots of large macros and invoke them frequently in your assembly source files, *very slow* assembly times will result. This can be really annoying as your projects get larger.

1.10 Summary

This chapter explored the power and flexibility of MASM's macro system, focusing on advanced techniques that enable the creation of DSLs within assembly language programs. The chapter began with a conceptual overview of DSLs and explained how MASM macros can serve as a lightweight mechanism for defining specialized language constructs.

The chapter then reviewed MASM's macro syntax and explored key macro operators, including these:

- `&` (text expansion)
- `!` (escape character)
- `%` (value expansion)
- `textequ` (text equates)

These features allow developers to perform sophisticated compile-time text processing, generate symbolic names dynamically, and build macros that behave like functions.

A major emphasis is placed on using macros as directive-like constructs, allowing programmers to simulate MASM's built-in syntax for data declarations and other statements. Several sections demonstrated how to

implement custom data declaration macros, such as those for IP addresses, and to address the syntactic limitations and bugs in MASM through practical work-arounds.

The chapter introduced namespace simulation using macros to avoid name collisions in large programs. By leveraging functional macro syntax and macro-generated identifiers, it becomes possible to build modular and scalable libraries with clear symbolic boundaries.

Finally, you learned how to build context-free macros that allow assembly language programmers to simulate HLL-like constructs.

In sum, this chapter equips you with the tools and techniques to transform MASM from a basic assembler into a customizable, high-level macro system, capable of expressing complex ideas with clarity and precision.

1.11 For More Information

Source code for the programs appearing in this chapter can be found at *https://artofasm.randallhyde.com*. Click the link to **The Art of 64-Bit Assembly, Volume 2**.

For a huge set of macros that simulate HLL control statements and functions, be sure to check out *https://www.masm32.com*. Download the MASM 64-bit SDK, which contains lots of useful macros (and other definitions you'll want when writing Microsoft Windows applications in assembly). These macros are not only directly useful in your own programs but also a good source of tricks you can use when writing macros.

If you can still find it (online or in treeware), Microsoft's MASM 6.*x* manual is a handy book to have around when working on your own macro libraries. Though this manual is old (covering the early 32-bit versions of the assembler), MASM's macro facilities haven't changed much in the past 20 to 30 years.

Though it's out of print at this point, if you can dig up a copy of *The Waite Group's Microsoft Macro Assembler Bible* (SAMS, 1992) by Nabajyoti Barkakati and myself, you'll find that it also has a good description of MASM's macro facilities.