CHAPTER 7

Controlling Program Flow

Very few programs execute all lines sequentially from **.STARTUP** to **.EXIT**. Rather, complex program logic and efficiency dictate that you control the flow of your program—jumping from one point to another, repeating an action until a condition is reached, and passing control to and from procedures. This chapter describes various ways for controlling program flow and several features that simplify coding program-control constructs.

The first section covers jumps from one point in the program to another. It explains how MASM 6.1 optimizes both unconditional and conditional jumps under certain circumstances, so that you do not have to specify every attribute. The section also describes instructions you can use to test conditional jumps.

The next section describes loop structures that repeat actions or evaluate conditions. It discusses MASM directives, such as **.WHILE** and **.REPEAT**, that generate appropriate compare, loop, and jump instructions for you, and the **.IF**, **.ELSE**, and **.ELSEIF** directives that generate jump instructions.

The "Procedures" section in this chapter explains how to write an assembly-language procedure. It covers the extended functionality for **PROC**, a **PROTO** directive that lets you write procedure prototypes similar to those used in C, an **INVOKE** directive that automates parameter passing, and options for the stack-frame setup inside procedures.

The last section explains how to pass program control to an interrupt routine.

Jumps

Jumps are the most direct way to change program control from one location to another. At the processor level, jumps work by changing the value of the IP (Instruction Pointer) register to a target offset and, for far jumps, by changing the CS register to a new segment address. Jump instructions fall into only two categories: conditional and unconditional.

Unconditional Jumps

The **JMP** instruction transfers control unconditionally to another instruction. **JMP**'s single operand contains the address of the target instruction.

Unconditional jumps skip over code that should not be executed, as shown here:

```
; Handle one case
label1: .
        .
        .
        jmp continue

; Handle second case
label2: .
        .
        .
        jmp continue
        .
        .
        .
continue:
```

The distance of the target from the jump instruction and the size of the operand determine the assembler's encoding of the instruction. The longer the distance, the more bytes the assembler uses to code the instruction. In versions of MASM prior to 6.0, unconditional **NEAR** jumps sometimes generated inefficient code, but MASM can now optimize unconditional jumps.

Jump Optimizing

The assembler determines the smallest encoding possible for the direct unconditional jump. MASM does not require a distance operator, so you do not have to determine the correct distance of the jump. If you specify a distance, it overrides any assembler optimization. If the specified distance falls short of the target address, the assembler generates an error. If the specified distance is longer than the jump requires, the assembler encodes the given distance and does not optimize it.

The assembler optimizes jumps when the following conditions are met:

- You do not specify **SHORT**, **NEAR**, **FAR**, **NEAR16**, **NEAR32**, **FAR16**, **FAR32**, or **PROC** as the distance of the target.
- The target of the jump is not external and is in the same segment as the jump instruction. If the target is in a different segment (but in the same group), it is treated as though it were external.

If these two conditions are met, MASM uses the instruction, distance, and size of the operand to determine how to optimize the encoding for the jump. No syntax changes are necessary.

Note This information about jump optimizing also applies to conditional jumps on the 80386/486.

Indirect Operands

An indirect operand provides a pointer to the target address, rather than the address itself. A pointer is a variable that contains an address. The processor distinguishes indirect (pointer) operands from direct (address) operands by the instruction's context.

You can specify the pointer's size with the **WORD**, **DWORD**, or **FWORD** attributes. Default sizes are based on **.MODEL** and the default segment size.

```
        jmp     [bx]          ; Uses .MODEL and segment size defaults
        jmp     WORD PTR [bx] ; A NEAR16 indirect call
```

If the indirect operand is a register, the jump is always a **NEAR16** jump for a 16-bit register, and **NEAR32** for a 32-bit register:

```
        jmp     bx            ; NEAR16 jump
        jmp     ebx           ; NEAR32  jump
```

A **DWORD** indirect operand, however, is ambiguous to the assembler.

```
jmp     DWORD PTR [var]       ; A NEAR32 jump in a 32-bit segment;
                              ; a FAR16 jump in a 16-bit segment
```

In this case, your code must clear the ambiguity with the **NEAR32** or **FAR16** keywords. The following example shows how to use **TYPEDEF** to define **NEAR32** and **FAR16** pointer types.

```
NFP     TYPEDEF PTR NEAR32
FFP     TYPEDEF PTR FAR16
        jmp     NFP PTR [var] ; NEAR32 indirect jump
        jmp     FFP PTR [var] ; FAR16  indirect jump
```

You can use an unconditional jump as a form of conditional jump by specifying the address in a register or indirect memory operand. Also, you can use indirect memory operands to construct jump tables that work like C **switch** statements, Pascal **CASE** statements, or Basic **ON GOTO**, **ON GOSUB**, or **SELECT CASE** statements, as shown in the following example.

```
NPVOID  TYPEDEF NEAR PTR
        .DATA
ctl_tbl NPVOID  extended,    ; Null key (extended code)
                ctrla,       ; Address of CONTROL-A key routine
                ctrlb        ; Address of CONTROL-B key routine
        .CODE
        .
        .
        .
        mov     ah, 8h       ; Get a key
        int     21h
        cbw                  ; Stretch AL into AX
        mov     bx, ax       ; Copy
        shl     bx, 1        ; Convert to address
        jmp     ctl_tbl[bx]  ; Jump to key routine

extended:
        mov     ah, 8h       ; Get second key of extended key
        int     21h
        .                    ; Use another jump table
        .                    ;   for extended keys
        .
        jmp     next
ctrla:  .                    ; CONTROL-A code here
        .
        .
        jmp     next
ctrlb:  .                    ; CONTROL-B code here
        .
        .
        jmp     next
        .
        .
next:   .                    ; Continue
```

In this instance, the indirect memory operands point to addresses of routines for handling different keystrokes.

Conditional Jumps

The most common way to transfer control in assembly language is to use a conditional jump. This is a two-step process:

1. First test the condition.
2. Then jump if the condition is true or continue if it is false.

All conditional jumps except two (**JCXZ** and **JECXZ**) use the processor flags for their criteria. Thus, any statement that sets or clears a flag can serve as a test basis for a conditional jump. The jump statement can be any one of 30 conditional-jump instructions. A conditional-jump instruction takes a single operand containing the target address. You cannot use a pointer value as a target as you can with unconditional jumps.

Jumping Based on the CX Register

JCXZ and **JECXZ** are special conditional jumps that do not consult the processor flags. Instead, as their names imply, these instructions cause a jump only if the CX or ECX register is zero. The use of **JCXZ** and **JECXZ** with program loops is covered in the next section, "Loops."

Jumping Based on the Processor Flags

The remaining conditional jumps in the processor's repertoire all depend on the status of the flags register. As the following list shows, several conditional jumps have two or three names—**JE** (Jump if Equal) and **JZ** (Jump if Zero), for example. Shared names assemble to exactly the same machine instruction, so you may choose whichever mnemonic seems more appropriate. Jumps that depend on the status of the flags register include:

Instruction	Jumps if
JC/JB/JNAE	Carry flag is set
JNC/JNB/JAE	Carry flag is clear
JBE/JNA	Either carry or zero flag is set
JA/JNBE	Carry and zero flag are clear
JE/JZ	Zero flag is set
JNE/JNZ	Zero flag is clear
JL/JNGE	Sign flag ≠ overflow flag
JGE/JNL	Sign flag = overflow flag
JLE/JNG	Zero flag is set or sign ≠ overflow
JG/JNLE	Zero flag is clear and sign = overflow
JS	Sign flag is set
JNS	Sign flag is clear
JO	Overflow flag is set
JNO	Overflow flag is clear
JP/JPE	Parity flag is set (even parity)
JNP/JPO	Parity flag is clear (odd parity)

The last two jumps in the list, **JPE** (Jump if Parity Even) and **JPO** (Jump if Parity Odd), are useful only for communications programs. The processor sets

the parity flag if an operation produces a result with an even number of set bits. A communications program can compare the flag against the parity bit received through the serial port to test for transmission errors.

The conditional jumps in the preceding list can follow any instruction that changes the processor flags, as these examples show:

```
; Uses JO to handle overflow condition
        add     ax, bx          ; Add two values
        jo      overflow        ; If value too large, adjust

; Uses JNZ to check for zero as the result of subtraction
        sub     ax, bx          ; Subtract
        mov     cx, Count       ; First, initialize CX
        jnz     skip            ; If the result is not zero, continue
        call    zhandler        ; Else do special case
```

As the second example shows, the jump does not have to immediately follow the instruction that alters the flags. Since **MOV** does not change the flags, it can appear between the **SUB** instruction and the dependent jump.

There are three categories of conditional jumps:

- Comparison of two values
- Individual bit settings in a value
- Whether a value is zero or nonzero

Jumps Based on Comparison of Two Values

The **CMP** instruction is the most common way to test for conditional jumps. It compares two values without changing either, then sets or clears the processor flags according to the results of the comparison.

Internally, the **CMP** instruction is the same as the **SUB** instruction, except that **CMP** does not change the destination operand. Both set flags according to the result of the subtraction.

You can compare signed or unsigned values, but you must choose the subsequent conditional jump to reflect the correct value type. For example, **JL** (Jump if Less Than) and **JB** (Jump if Below) may seem conceptually similar, but a failure to understand the difference between them can result in program bugs. Table 7.1 shows the correct conditional jumps for comparisons of signed and unsigned values. The table shows the zero, carry, sign, and overflow flags as ZF, CF, SF, and OF, respectively.

Table 7.1 Conditional Jumps Based on Comparisons of Two Values

Signed Comparisons		Unsigned Comparisons	
Instruction	**Jump if True**	**Instruction**	**Jump if True**
JE	ZF = 1	**JE**	ZF = 1
JNE	ZF = 0	**JNE**	ZF = 0
JG/JNLE	ZF = 0 and SF = OF	**JA/JNBE**	CF = 0 and ZF = 0
JLE/JNG	ZF = 1 or SF ≠ OF	**JBE/JNA**	CF = 1 or ZF = 1
JL/JNGE	SF ≠ OF	**JB/JNAE**	CF = 1
JGE/JNL	SF = OF	**JAE/JNB**	CF = 0

The mnemonic names of jumps always refer to the comparison of **CMP**'s first operand (destination) with the second operand (source). For instance, in this example, **JG** tests whether the first operand is greater than the second.

```
cmp     ax, bx ; Compare AX and BX
jg      next1  ; Equivalent to: If ( AX > BX ) goto next1
jl      next2  ; Equivalent to: If ( AX < BX ) goto next2
```

Jumps Based on Bit Settings

The individual bit settings in a single value can also serve as the criteria for a conditional jump. The **TEST** instruction tests whether specific bits in an operand are on or off (set or clear), and sets the zero flag accordingly.

The **TEST** instruction is the same as the **AND** instruction, except that **TEST** changes neither operand. The following example shows an application of **TEST**.

```
        .DATA
bits    BYTE    ?
        .CODE
        .
        .
        .
; If bit 2 or bit 4 is set, then call task_a
                                ; Assume "bits" is 0D3h    11010011
        test    bits, 10100y    ; If 2 or 4 is set     AND 00010100
        jz      skip1           ;                          --------
        call    task_a          ; Then call task_a         00010000
skip1:                          ; Jump taken
        .
        .
        .
; If bits 2 and 4 are clear, then call task_b
                                ; Assume "bits" is 0E9h    11101001
        test    bits, 10100y    ; If 2 and 4 are clear AND 00010100
        jnz     skip2           ;                          --------
        call    task_b          ; Then call task_b         00000000
skip2:                          ; Jump taken
```

The source operand for **TEST** is often a mask in which the test bits are the only bits set. The destination operand contains the value to be tested. If all the bits set in the mask are clear in the destination operand, **TEST** sets the zero flag. If any of the flags set in the mask are also set in the destination operand, **TEST** clears the zero flag.

The 80386/486 processors provide additional bit-testing instructions. The **BT** (Bit Test) series of instructions copy a specified bit from the destination operand to the carry flag. A **JC** or **JNC** can then route program flow depending on the result. For variations on the **BT** instruction, see the *Reference*.

Jumps Based on a Value of Zero

A program often needs to jump based on whether a particular register contains a value of zero. We've seen how the **JCXZ** instruction jumps depending on the value in the CX register. You can test for zero in other data registers nearly as efficiently with the **OR** instruction. A program can **OR** a register with itself without changing the register's contents, then act on the resulting flags status. For example, the following example tests whether BX is zero:

```
        or      bx, bx          ; Is BX = 0?
        jz      is_zero         ; Jump if so
```

This code is functionally equivalent to:

```
        cmp     bx, 0           ; Is BX = 0?
        je      is_zero         ; Jump if so
```

but produces smaller and faster code, since it does not use an immediate number as an operand. The same technique also lets you test a register's sign bit:

```
        or      dx, dx          ; Is DX sign bit set?
        js      sign_set        ; Jump if so
```

Jump Extending

Unlike an unconditional jump, a conditional jump cannot reference a label more than 128 bytes away. For example, the following statement is valid as long as `target` is within a distance of 128 bytes:

```
; Jump to target less than 128 bytes away
        jz      target          ; If previous operation resulted
                                ;   in zero, jump to target
```

However, if `target` is too distant, the following sequence is necessary to enable a longer jump. Note this sequence is logically equivalent to the preceding example:

```
; Jumps to distant targets previously required two steps
        jnz     skip            ; If previous operation result is
                                ;   NOT zero, jump to "skip"
        jmp     target          ;   Otherwise, jump to target
skip:
```

MASM can automate jump-extending for you. If you target a conditional jump to a label farther than 128 bytes away, MASM rewrites the instruction with an unconditional jump, which ensures that the jump can reach its target. If `target` lies within a 128-byte range, the assembler encodes the instruction `jz target` as is. Otherwise, MASM generates two substitute instructions:

```
jne $ + 2 + (length in bytes of the next instruction)
jmp NEAR PTR target
```

The assembler generates this same code sequence if you specify the distance with **NEAR PTR**, **FAR PTR**, or **SHORT**. Therefore,

```
jz      NEAR PTR target
```

becomes

```
jne     $ + 5
jmp     NEAR PTR target
```

even if `target` is less than 128 bytes away.

MASM enables automatic jump expansion by default, but you can turn it off with the **NOLJMP** form of the **OPTION** directive. For information about the **OPTION** directive, see page 24.

If the assembler generates code to extend a conditional jump, it issues a level 3 warning saying that the conditional jump has been lengthened. You can set the warning level to 1 for development and to level 3 for a final optimizing pass to see if you can shorten jumps by reorganizing.

If you specify the distance for the jump and the target is out of range for that distance, a "Jump out of Range" error results.

Since the **JCXZ** and **JECXZ** instructions do not have logical negations, expansion of the jump instruction to handle targets with unspecified distances cannot be performed for those instructions. Therefore, the distance must always be short.

The size and distance of the target operand determines the encoding for conditional or unconditional jumps to externals or targets in different segments. The jump-extending and optimization features do not apply in this case.

Note Conditional jumps on the 80386 and 80486 processors can be to targets up to 32K away, so jump extension occurs only for targets greater than that distance.

Anonymous Labels

When you code jumps in assembly language, you must invent many label names. One alternative to continually thinking up new label names is to use anonymous labels, which you can use anywhere in your program. But because anonymous labels do not provide meaningful names, they are best used for jumping over only a few lines of code. You should mark major divisions of a program with actual named labels.

Use two at signs (@@) followed by a colon (:) as an anonymous label. To jump to the nearest preceding anonymous label, use **@B** (back) in the jump instruction's operand field; to jump to the nearest following anonymous label, use **@F** (forward) in the operand field.

The jump in the following example targets an anonymous label:

```
        jge     @F
        .
        .
        .
@@:
```

The items **@B** and **@F** always refer to the nearest occurrences of @@:, so there is never any conflict between different anonymous labels.

Decision Directives

The high-level structures you can use for decision-making are the **.IF**, **.ELSEIF**, and **.ELSE** statements. These directives generate conditional jumps. The expression following the **.IF** directive is evaluated, and if true, the following instructions are executed until the next **.ENDIF**, **.ELSE**, or **.ELSEIF** directive is reached. The **.ELSE** statements execute if the expression is false. Using the **.ELSEIF** directive puts a new expression inside the alternative part of the original **.IF** statement to be evaluated. The syntax is:

.IF *condition1*
statements
[[**.ELSEIF** *condition2*
statements]]
[[**.ELSE**
statements]]
.ENDIF

The decision structure

```
        .IF     cx == 20
        mov     dx, 20
        .ELSE
        mov     dx, 30
        .ENDIF
```

generates this code:

```
                            .IF  cx == 20
0017  83 F9 14          *         cmp    cx, 014h
001A  75 05             *         jne    @C0001
001C  BA 0014                     mov    dx, 20
                            .ELSE
001F  EB 03             *         jmp    @C0003
0021                    *@C0001:
0021  BA 001E                     mov   dx, 30
                            .ENDIF
0024                    *@C0003:
```

Loops

Loops repeat an action until a termination condition is reached. This condition can be a counter or the result of an expression's evaluation. MASM 6.1 offers many ways to set up loops in your programs. The following list compares MASM loop structures:

Instructions	Action
LOOP	Automatically decrements CX. When CX = 0, the loop ends. The top of the loop cannot be greater than 128 bytes from the **LOOP** instruction. (This is true for all **LOOP** instructions.)
LOOPE/**LOOPZ**, **LOOPNE**/**LOOPNZ**	Loops while equal or not equal. Checks both CX and the state of the zero flag. **LOOPZ** ends when either CX=0 or the zero flag is clear, whichever occurs first. **LOOPNZ** ends when either CX=0 or the zero flag is set, whichever occurs first. **LOOPE** and **LOOPZ** assemble to the same machine instruction, as do **LOOPNE** and **LOOPNZ**. Use whichever mnemonic best fits the context of your loop. Set CX to a number out of range if you don't want a count to control the loop.
JCXZ, **JECXZ**	Branches to a label only if CX = 0 or ECX = 0. Unlike other conditional-jump instructions, which can jump to either a near or a short label under the 80386 or 80486, **JCXZ** and **JECXZ** always jump to a short label.
Conditional jumps	Acts only if certain conditions met. Necessary if several conditions must be tested. See "Conditional Jumps," page 164.

The following examples illustrate these loop constructions.

```
; The LOOP instruction: For 200 to 0 do task
        mov     cx, 200         ; Set counter
next:   .                       ; Do the task here
        .
        .
        loop    next            ; Do again
                                ; Continue after loop

; The LOOPNE instruction: While AX is not 'Y', do task
        mov     cx, 256         ; Set count too high to interfere
wend:   .                       ; But don't do more than 256 times
        .                       ; Some statements that change AX
        .
        cmp     al, 'Y'         ; Is it Y or too many times?
        loopne  wend            ; No? Repeat
                                ; Yes? Continue
```

The **JCXZ** and **JECXZ** instructions provide an efficient way to avoid executing loops when the loop counter CX is empty. For example, consider the following loops:

```
mov     cx, LoopCount           ; Load loop counter
next:   .                       ; Iterate loop CX times
        .
        .
        loop    next            ; Do again
```

If `LoopCount` is zero, CX decrements to -1 on the first pass. It then must decrement 65,535 more times before reaching 0. Use a **JCXZ** to avoid this problem:

```
mov     cx, LoopCount           ; Load loop counter
        jcxz    done            ; Skip loop if count is 0
next:   .                       ; Else iterate loop CX times
        .
        .
        loop    next            ; Do again
done:                           ; Continue after loop
```

Loop-Generating Directives

The high-level control structures generate loop structures for you. These directives are similar to the **while** and **repeat** loops of C or Pascal, and can make your assembly programs easier to code and to read. The assembler generates the appropriate assembly code. These directives are summarized as follows:

Directives	Action
.WHILE ... **.ENDW**	The statements between **.WHILE** *condition* and **.ENDW** execute while the condition is true.
.REPEAT ... **.UNTIL**	The loop executes at least once and continues until the condition given after **.UNTIL** is true. Generates conditional jumps.
.REPEAT ... **.UNTILCXZ**	Compares label to an expression and generates appropriate loop instructions.
.BREAK	End a **.REPEAT** or a **.WHILE** loop unconditionally.
.CONTINUE	Jump unconditionally past any remaining code to bottom of loop.

These constructs work much as they do in a high-level language such as C or Pascal. Keep in mind the following points:

- These directives generate appropriate processor instructions. They are not new instructions.
- They require proper use of signed and unsigned data declarations.

These directives cause a set of instructions to execute based on the evaluation of some *condition*. This *condition* can be an expression that evaluates to a signed or unsigned value, an expression using the binary operators in C (**&&**, ||, or **!**), or the state of a flag. For more information about expression operators, see page 178.

The evaluation of the *condition* requires the assembler to know if the operands in the condition are signed or unsigned. To state explicitly that a named memory location contains a signed integer, use the signed data allocation directives **SBYTE**, **SWORD**, and **SDWORD**.

.WHILE Loops

As with **while** loops in C or Pascal, the test condition for **.WHILE** is checked before the statements inside the loop execute. If the test condition is false, the loop does not execute. While the condition is true, the statements inside the loop repeat.

Use the **.ENDW** directive to mark the end of the **.WHILE** loop. When the condition becomes false, program execution begins at the first statement following the **.ENDW** directive. The **.WHILE** directive generates appropriate compare and jump statements. The syntax is:

.WHILE *condition*
statements
.ENDW

For example, this loop copies the contents of one buffer to another until a '$' character (marking the end of the string) is found:

```
.DATA
buf1    BYTE "This is a string",'$'
buf2    BYTE 100 DUP (?)
.CODE
sub     bx, bx                  ; Zero out bx
.WHILE  (buf1[bx] != '$')
mov     al, buf1[bx]            ; Get a character
mov     buf2[bx], al            ; Move it to buffer 2
inc     bx                      ; Count forward
.ENDW
```

.REPEAT Loops

MASM's **.REPEAT** directive allows for loop constructions like the **do** loop of C and the **REPEAT** loop of Pascal. The loop executes until the condition following the **.UNTIL** (or **.UNTILCXZ**) directive becomes true. Since the condition is checked at the end of the loop, the loop always executes at least once. The **.REPEAT** directive generates conditional jumps. The syntax is:

.REPEAT
statements
.UNTIL *condition*

.REPEAT
statements
.UNTILCXZ [[*condition*]]

where *condition* can also be *expr1 == expr2* or *expr1 != expr2*. When two conditions are used, *expr2* can be an immediate expression, a register, or (if *expr1* is a register) a memory location.

For example, the following code fills a buffer with characters typed at the keyboard. The loop ends when the ENTER key (character 13) is pressed:

```
        .DATA
buffer  BYTE    100 DUP (0)
        .CODE
        sub     bx, bx              ; Zero out bx
        .REPEAT
        mov     ah, 01h
        int     21h                 ; Get a key
        mov     buffer[bx], al      ; Put it in the buffer
        inc     bx                  ; Increment the count
        .UNTIL (al == 13)           ; Continue until al is 13
```

The **.UNTIL** directive generates conditional jumps, but the **.UNTILCXZ** directive generates a **LOOP** instruction, as shown by the listing file code for these examples. In a listing file, assembler-generated code is preceded by an asterisk.

```
ASSUME  bx:PTR SomeStruct

            .REPEAT
       *@C0001:
                 inc    ax
            .UNTIL  ax==6
       *         cmp    ax, 006h
       *         jne    @C0001
.REPEAT
       *@C0003:
                 mov    ax, 1
            .UNTILCXZ
       *         loop   @C0003

            .REPEAT
       *@C0004:
            .UNTILCXZ   [bx].field != 6
       *         cmp    [bx].field, 006h
       *         loope  @C0004
```

.BREAK and .CONTINUE Directives

The **.BREAK** and **.CONTINUE** directives terminate a **.REPEAT** or **.WHILE** loop prematurely. These directives allow an optional **.IF** clause for conditional breaks. The syntax is:

.BREAK [[**.IF** *condition*]]
.CONTINUE [[**.IF** *condition*]]

Note that **.ENDIF** is not used with the **.IF** forms of **.BREAK** and **.CONTINUE** in this context. The **.BREAK** and **.CONTINUE** directives work the same way as the **break** and **continue** instructions in C. Execution continues at the instruction following the **.UNTIL**, **.UNTILCXZ**, or **.ENDW** of the nearest enclosing loop.

Instead of ending the loop execution as **.BREAK** does, **.CONTINUE** causes loop execution to jump directly to the code that evaluates the loop condition of the nearest enclosing loop.

The following loop accepts only the keys in the range '0' to '9' and terminates when you press ENTER.

```
.WHILE 1                ; Loop forever
mov     ah, 08h         ; Get key without echo
int     21h
.BREAK .IF al == 13     ; If ENTER, break out of the loop
.CONTINUE .IF (al < '0') || (al > '9')
                        ; If not a digit, continue looping
mov     dl, al          ; Save the character for processing
mov     ah, 02h         ; Output the character
int     21h
.ENDW
```

If you assemble the preceding source code with the /Fl and /Sg command-line options and then view the results in the listing file, you will see this code:

```
                        .WHILE 1
0017                   *@C0001:
0017  B4 08                     mov    ah, 08h
0019  CD 21                     int    21h
                        .BREAK .IF al == 13
001B  3C 0D            *        cmp    al, 00Dh
001D  74 10            *        je     @C0002
                        .CONTINUE .IF (al  '0') || (al  '9')
001F  3C 30            *        cmp    al, '0'
0021  72 F4            *        jb     @C0001
0023  3C 39            *        cmp    al, '9'
0025  77 F0            *        ja     @C0001
0027  8A D0                     mov    dl, al
0029  B4 02                     mov    ah, 02h
002B  CD 21                     int    21h
                        .ENDW
002D  EB E8            *        jmp    @C0001
002F                   *@C0002:
```

The high-level control structures can be nested. That is, **.REPEAT** or **.WHILE** loops can contain **.REPEAT** or **.WHILE** loops as well as **.IF** statements.

If the code generated by a **.WHILE** loop, **.REPEAT** loop, or **.IF** statement generates a conditional or unconditional jump, MASM encodes the jump using the jump extension and jump optimization techniques described in “Unconditional Jumps,” page 162, and “Conditional Jumps,” page 164.

Writing Loop Conditions

You can express the conditions of the **.IF**, **.REPEAT**, and **.WHILE** directives using relational operators, and you can express the attributes of the operand with the **PTR** operator. To write loop conditions, you also need to know how the assembler evaluates the operators and operands in the condition. This section explains the operators, attributes, precedence level, and expression evaluation order for the conditions used with loop-generating directives.

Expression Operators

The binary relational operators in MASM 6.1 are the same binary operators used in C. These operators generate MASM compare, test, and conditional jump instructions. High-level control instructions include:

Operator	Meaning
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
&	Bit test
!	Logical NOT
&&	Logical AND
||	Logical OR

A condition without operators (other than **!**) tests for nonzero as it does in C. For example, `.WHILE (x)` is the same as `.WHILE (x != 0)`, and `.WHILE (!x)` is the same as `.WHILE (x == 0)`.

You can also use the flag names (**ZERO?**, **CARRY?**, **OVERFLOW?**, **SIGN?**, and **PARITY?**) as operands in conditions with the high-level control structures. For example, in `.WHILE (CARRY?)`, the value of the carry flag determines the outcome of the condition.

Signed and Unsigned Operands

Expression operators generate unsigned jumps by default. However, if either side of the operation is signed, the assembler considers the entire operation signed.

You can use the **PTR** operator to tell the assembler that a particular operand in a register or constant is a signed number, as in these examples:

```
        .WHILE  SWORD PTR [bx] <= 0
        .IF     SWORD PTR mem1 >  0
```

Without the **PTR** operator, the assembler would treat the contents of BX as an unsigned value.

You can also specify the size attributes of operands in memory locations with **SBYTE**, **SWORD**, and **SDWORD**, for use with **.IF**, **.WHILE**, and **.REPEAT**.

```
        .DATA
mem1    SBYTE   ?
mem2    WORD    ?
        .IF     mem1 > 0
        .WHILE  mem2 < bx
        .WHILE  SWORD PTR ax < count
```

Precedence Level

As with C, you can concatenate conditions with the **&&** operator for AND, the || operator for OR, and the **!** operator for negate. The precedence level is **!**, **&&**, and ||, with **!** having the highest priority. Like expressions in high-level languages, precedence is evaluated left to right.

Expression Evaluation

The assembler evaluates conditions created with high-level control structures according to short-circuit evaluation. If the evaluation of a particular condition automatically determines the final result (such as a condition that evaluates to false in a compound statement concatenated with **AND**), the evaluation does not continue.

For example, in this **.WHILE** statement,

```
        .WHILE (ax > 0) && (WORD PTR [bx] == 0)
```

the assembler evaluates the first condition. If this condition is false (that is, if AX is less than or equal to 0), the evaluation is finished. The second condition is not checked and the loop does not execute, because a compound condition containing **&&** requires both expressions to be true for the entire condition to be true.

Procedures

Organizing your code into procedures that execute specific tasks divides large programs into manageable units, allows for separate testing, and makes code more efficient for repetitive tasks.

Assembly-language procedures are similar to functions, subroutines, and procedures in high-level languages such as C, FORTRAN, and Pascal. Two instructions control the use of assembly-language procedures. **CALL** pushes the return address onto the stack and transfers control to a procedure, and **RET** pops the return address off the stack and returns control to that location.

The **PROC** and **ENDP** directives mark the beginning and end of a procedure. Additionally, **PROC** can automatically:

- Preserve register values that should not change but that the procedure might otherwise alter.
- Set up a local stack pointer, so that you can access parameters and local variables placed on the stack.
- Adjust the stack when the procedure ends.

Defining Procedures

Procedures require a label at the start of the procedure and a **RET** instruction at the end. Procedures are normally defined by using the **PROC** directive at the start of the procedure and the **ENDP** directive at the end. The **RET** instruction normally is placed immediately before the **ENDP** directive. The assembler makes sure the distance of the **RET** instruction matches the distance defined by the **PROC** directive. The basic syntax for **PROC** is:

label **PROC** [[**NEAR** | **FAR**]]
.
.
.
RET [[*constant*]]
label **ENDP**

The **CALL** instruction pushes the address of the next instruction in your code onto the stack and passes control to a specified address. The syntax is:

CALL {*label* | *register* | *memory*}

The operand contains a value calculated at run time. Since that operand can be a register, direct memory operand, or indirect memory operand, you can write call tables similar to the example code on page 164.

Calls can be near or far. Near calls push only the offset portion of the calling address and therefore must target a procedure within the same segment or group. You can specify the type for the target operand. If you do not, MASM uses the declared distance (**NEAR** or **FAR**) for operands that are labels and for the size of register or memory operands. The assembler then encodes the call appropriately, as it does with unconditional jumps. (See previous "Unconditional Jumps" and "Conditional Jumps.")

MASM optimizes a call to a far non-external label when the label is in the current segment by generating the code for a near call, saving one byte.

You can define procedures without **PROC** and **ENDP**, but if you do, you must make sure that the size of the **CALL** matches the size of the **RET**. You can specify the **RET** instruction as **RETN** (Return Near) or **RETF** (Return Far) to override the default size:

```
        call    NEAR PTR task   ; Call is declared near
        .                       ; Return comes to here
        .
        .
task:                           ; Procedure begins with near label
        .
        .                       ; Instructions go here
        .
        retn                    ; Return declared near
```

The syntax for **RETN** and **RETF** is:

label: | *label* **LABEL NEAR**
statements
RETN [[*constant*]]

label **LABEL FAR**
statements
RETF [[*constant*]]

The **RET** instruction (and its **RETF** and **RETN** variations) allows an optional constant operand that specifies a number of bytes to be added to the value of the SP register after the return. This operand adjusts for arguments passed to the procedure before the call, as shown in the example in "Using Local Variables," following.

When you define procedures without **PROC** and **ENDP**, you must make sure that calls have the same size as corresponding returns. For example, **RETF** pops two words off the stack. If a **NEAR** call is made to a procedure with a far return, the popped value is meaningless, and the stack status may cause the execution to return to a random memory location, resulting in program failure.

An extended **PROC** syntax automates many of the details of accessing arguments and saving registers. See "Declaring Parameters with the PROC Directive," later in this chapter.

Passing Arguments on the Stack

Each time you call a procedure, you may want it to operate on different data. This data, called "arguments," can be passed to the procedure in various ways. Although you can pass arguments to a procedure in registers or in variables, the most common method is the stack. Microsoft languages have specific conventions for passing arguments. These conventions for assembly-language modules shared with modules from high-level languages are explained in Chapter 12, "Mixed-Language Programming."

This section describes how a procedure accesses the arguments passed to it on the stack. Each argument is accessed as an offset from BP. However, if you use the **PROC** directive to declare parameters, the assembler calculates these offsets for you and lets you refer to parameters by name. The next section, "Declaring Parameters with the PROC Directive," explains how to use **PROC** this way. This example shows how to pass arguments to a procedure. The procedure expects to find those arguments on the stack. As this example shows, arguments must be accessed as offsets of BP.

```
; C-style procedure call and definition

        mov     ax, 10          ; Load and
        push    ax              ;   push constant as third argument
        push    arg2            ; Push memory as second argument
        push    cx              ; Push register as first argument
        call    addup           ; Call the procedure
        add     sp, 6           ; Destroy the pushed arguments
        .                       ;   (equivalent to three pops)
        .
        .
addup   PROC    NEAR            ; Return address for near call
                                ;   takes two bytes
        push    bp              ; Save base pointer - takes two bytes
                                ;   so arguments start at fourth byte
        mov     bp, sp          ; Load stack into base pointer
        mov     ax, [bp+4]      ; Get first argument from
                                ;   fourth byte above pointer
        add     ax, [bp+6]      ; Add second argument from
                                ;   sixth byte above pointer
        add     ax, [bp+8]      ; Add third argument from
                                ;   eighth byte above pointer
        pop     bp              ; Restore BP
        ret                     ; Return result in AX
addup   ENDP
```

Figure 7.1 shows the stack condition at key points in the process.

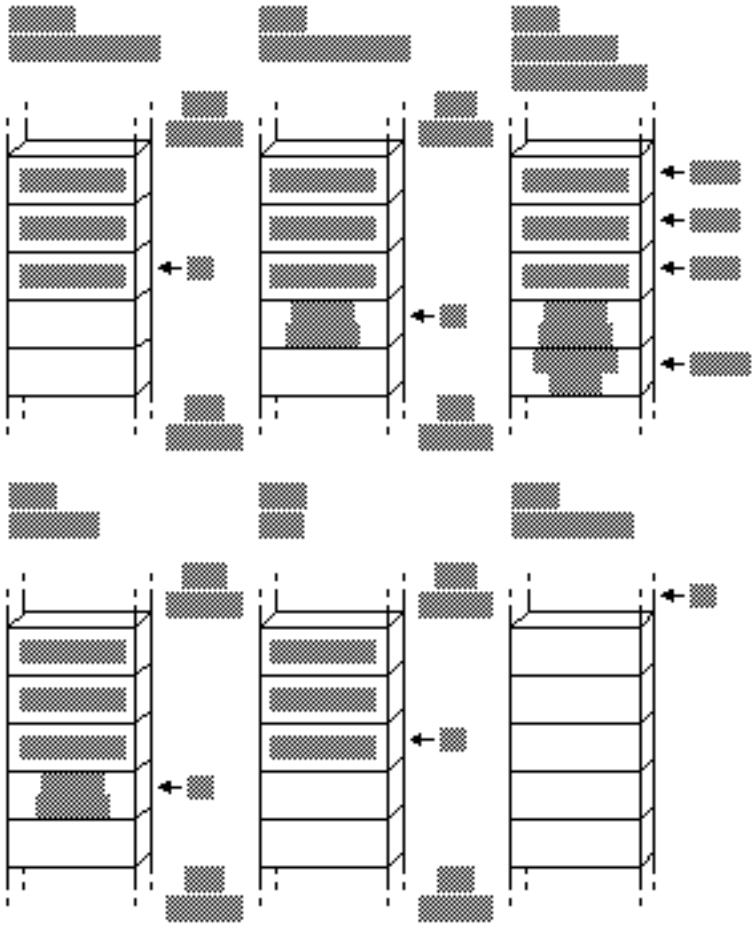


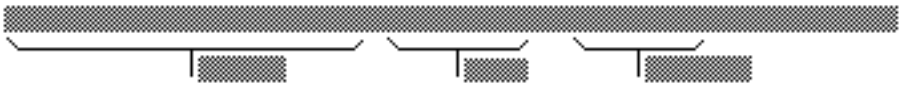
Figure 7.1 Procedure Arguments on the Stack

Starting with the 80186 processor, the **ENTER** and **LEAVE** instructions simplify the stack setup and restore instructions at the beginning and end of procedures. However, **ENTER** uses a lot of time. It is necessary only with nested, statically-scoped procedures. Thus, a Pascal compiler may sometimes generate **ENTER**. The **LEAVE** instruction, on the other hand, is an efficient way to do the stack cleanup. **LEAVE** reverses the effect of the last **ENTER** instruction by restoring BP and SP to their values before the procedure call.

Declaring Parameters with the PROC Directive

With the **PROC** directive, you can specify registers to be saved, define parameters to the procedure, and assign symbol names to parameters (rather than as offsets from BP). This section describes how to use the **PROC** directive to automate the parameter-accessing techniques described in the last section.

For example, the following diagram shows a valid **PROC** statement for a procedure called from C. It takes two parameters, `var1` and `arg1`, and uses (and must save) the DI and SI registers:



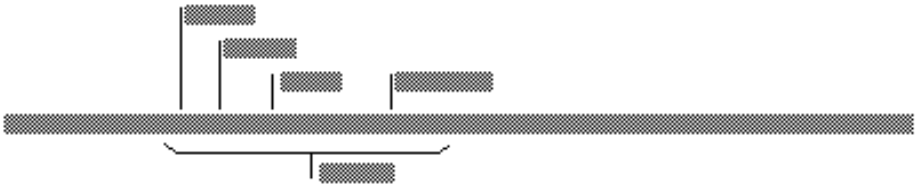
The syntax for **PROC** is:

label **PROC** [[*attributes*]] [[**USES** *reglist*]] [[,]] [[*parameter*[[:*tag*]]**...**]]

The parts of the **PROC** directive include:

Argument	Description
label	The name of the procedure.
attributes	Any of several attributes of the procedure, including the distance, *langtype*, and *visibility* of the procedure. The syntax for *attributes* is given on the following page.
reglist	A list of registers following the **USES** keyword that the procedure uses, and that should be saved on entry. Registers in the list must be separated by blanks or tabs, not by commas. The assembler generates prologue code to push these registers onto the stack. When you exit, the assembler generates epilogue code to pop the saved register values off the stack.
parameter	The list of parameters passed to the procedure on the stack. The list can have a variable number of parameters. See the discussion following for the syntax of *parameter*. This list can be longer than one line if the continued line ends with a comma.

This diagram shows a valid **PROC** definition that uses several attributes:



Attributes

The syntax for the attributes field is:

⟦*distance*⟧ ⟦*langtype*⟧ ⟦*visibility*⟧ ⟦<*prologuearg*>⟧

The explanations for these options include:

Argument	Description
distance	Controls the form of the **RET** instruction generated. Can be **NEAR** or **FAR**. If *distance* is not specified, it is determined from the model declared with the **.MODEL** directive. **NEAR** distance is assumed for **TINY, SMALL**, **COMPACT**, and **FLAT**. The assembler assumes **FAR** distance for **MEDIUM**, **LARGE**, and **HUGE**. For 80386/486 programming with 16- and 32-bit segments, you can specify **NEAR16**, **NEAR32**, **FAR16**, or **FAR32**.
langtype	Determines the calling convention used to access parameters and restore the stack. The **BASIC**, **FORTRAN**, and **PASCAL** *langtypes* convert procedure names to uppercase, place the last parameter in the parameter list lowest on the stack, and generate a **RET** *num* instruction to end the procedure. The **RET** adjusts the stack upward by *num*, which represents the number of bytes in the argument list. This step, called "cleaning the stack," returns the stack pointer SP to the value it had before the caller pushed any arguments. The **C** and **STDCALL** *langtype* prefixes an underscore to the procedure name when the procedure's scope is **PUBLIC** or **EXPORT** and places the first parameter lowest on the stack. **SYSCALL** is equivalent to the **C** calling convention with no underscore prefixed to the procedure's name. **STDCALL** uses caller stack cleanup when **:VARARG** is specified; otherwise the called routine must clean up the stack (see Chapter 12).
visibility	Indicates whether the procedure is available to other modules. The *visibility* can be **PRIVATE, PUBLIC**, or **EXPORT**. A procedure name is **PUBLIC** unless it is explicitly declared as **PRIVATE**. If the *visibility* is **EXPORT**, the linker places the procedure's name in the export table for segmented executables. **EXPORT** also enables **PUBLIC** visibility. You can explicitly set the default *visibility* with the **OPTION** directive. **OPTION PROC:PUBLIC** sets the default to public. For more information, see Chapter 1, "Using the Option Directive."
prologuearg	Specifies the arguments that affect the generation of prologue and epilogue code (the code MASM generates when it encounters a **PROC** directive or the end of a procedure). For an explanation of prologue and epilogue code, see "Generating Prologue and Epilogue Code," later in this chapter.

Parameters

The comma that separates *parameters* from *reglist* is optional, if both fields appear on the same line. If *parameters* appears on a separate line, you must end the *reglist* field with a comma. In the syntax:

parmname [[**:***tag*]]

parmname is the name of the parameter. The *tag* can be the *qualifiedtype* or the keyword **VARARG**. However, only the last parameter in a list of parameters can use the **VARARG** keyword. The *qualifiedtype* is discussed in "Data Types," Chapter 1. An example showing how to reference **VARARG** parameters appears later in this section. You can nest procedures if they do not have parameters or **USES** register lists. This diagram shows a procedure definition with one parameter definition.

The procedure presented in "Passing Arguments on the Stack," page 182, is here rewritten using the extended **PROC** functionality. Prior to the procedure call, you must push the arguments onto the stack unless you use **INVOKE.** (See "Calling Procedures with INVOKE," later in this chapter.)

```
addup   PROC NEAR C,
        arg1:WORD, arg2:WORD, count:WORD
        mov     ax, arg1
        add     ax, count
        add     ax, arg2
        ret
addup   ENDP
```

If the arguments for a procedure are pointers, the assembler does not generate any code to get the value or values that the pointers reference; your program must still explicitly treat the argument as a pointer. (For more information about using pointers, see Chapter 3, "Using Addresses and Pointers.")

In the following example, even though the procedure declares the parameters as near pointers, you must code two **MOV** instructions to get the values of the param-
eters. The first **MOV** gets the address of the parameters, and the second **MOV** gets the parameter.

```
; Call from C as a FUNCTION returning an integer

        .MODEL medium, c
        .CODE
myadd   PROC   arg1:NEAR PTR WORD, arg2:NEAR PTR WORD

        mov     bx, arg1     ; Load first argument
        mov     ax, [bx]
        mov     bx, arg2     ; Add second argument
        add     ax, [bx]

        ret

myadd   ENDP
```

You can use conditional-assembly directives to make sure your pointer parameters are loaded correctly for the memory model. For example, the following version of `myadd` treats the parameters as **FAR** parameters, if necessary.

```
        .MODEL  medium, c       ; Could be any model
        .CODE
myadd   PROC    arg1:PTR WORD,   arg2:PTR WORD

        IF      @DataSize
        les     bx, arg1        ; Far parameters
        mov     ax, es:[bx]
        les     bx, arg2
        add     ax, es:[bx]
        ELSE
        mov     bx, arg1          ; Near parameters
        mov     ax, [bx]
        mov     bx, arg2
        add     ax, [bx]
        ENDIF

        ret
myadd   ENDP
```

Using VARARG

In the **PROC** statement, you can append the **:VARARG** keyword to the last parameter to indicate that the procedure accepts a variable number of arguments. However, **:VARARG** applies only to the **C**, **SYSCALL**, or **STDCALL** calling conventions (see Chapter 12). A symbol must precede **:VARARG** so the procedure can access arguments as offsets from the given variable name, as this example illustrates:

```
addup3  PROTO NEAR C, argcount:WORD, arg1:VARARG

        invoke  addup3, 3, 5, 2, 4

addup3  PROC    NEAR C, argcount:WORD, arg1:VARARG
        sub     ax, ax          ; Clear work register
        sub     si, si

        .WHILE  argcount > 0    ; Argcount has number of arguments
        add     ax, arg1[si]    ; Arg1 has the first argument
        dec     argcount        ; Point to next argument
        inc     si
        inc     si
        .ENDW

        ret                     ; Total is in AX
addup3  ENDP
```

You can pass non-default-sized pointers in the **VARARG** portion of the parameter list by separately passing the segment portion and the offset portion of the address.

Note When you use the extended **PROC** features and the assembler encounters a **RET** instruction, it automatically generates instructions to pop saved registers, remove local variables from the stack, and, if necessary, remove parameters. It generates this code for each **RET** instruction it encounters. You can reduce code size by having only one return and jumping to it from various locations.

Using Local Variables

In high-level languages, local variables are visible only within a procedure. In Microsoft languages, these variables are usually stored on the stack. In assembly-language programs, you can also have local variables. These variables should not be confused with labels or variable names that are local to a module, as described in Chapter 8, "Sharing Data and Procedures Among Modules and Libraries."

This section outlines the standard methods for creating local variables. The next section shows how to use the **LOCAL** directive to make the assembler

automatically generate local variables. When you use this directive, the assembler generates the same instructions as those demonstrated in this section but handles some of the details for you.

If your procedure has relatively few variables, you can usually write the most efficient code by placing these values in registers. Use local (stack) data when you have a large amount of temporary data for the procedure.

To use a local variable, you must save stack space for it at the start of the procedure. A procedure can then reference the variable by its position in the stack. At the end of the procedure, you must clean the stack by restoring the stack pointer. This effectively throws away all local variables and regains the stack space they occupied.

This example subtracts 2 bytes from the SP register to make room for a local word variable, then accesses the variable as `[bp-2]`.

```
        push    ax                ; Push one argument
        call    task              ; Call
        .
        .
        .

task    PROC    NEAR
        push    bp                ; Save base pointer
        mov     bp, sp            ; Load stack into base pointer
        sub     sp, 2             ; Save two bytes for local variable
        .
        .
        .
        mov     WORD PTR [bp-2], 3 ; Initialize local variable
        add     ax, [bp-2]        ; Add local variable to AX
        sub     [bp+4], ax        ; Subtract local from argument
        .                         ; Use [bp-2] and [bp+4] in
        .                         ;   other operations
        .
        mov     sp, bp            ; Clear local variables
        pop     bp                ; Restore base
        ret     2                 ; Return result in AX and pop
task    ENDP                      ;   two bytes to clear parameter
```

Notice the instruction `mov sp,bp` at the end of the procedure restores the original value of SP. The statement is required only if the value of SP changes inside the procedure (usually by allocating local variables). The argument passed to the procedure is removed with the **RET** instruction. Contrast this to the example in "Passing Arguments on the Stack," page 182, in which the calling code adjusts the stack for the argument.

Figure 7.2 shows the stack at key points in the process.

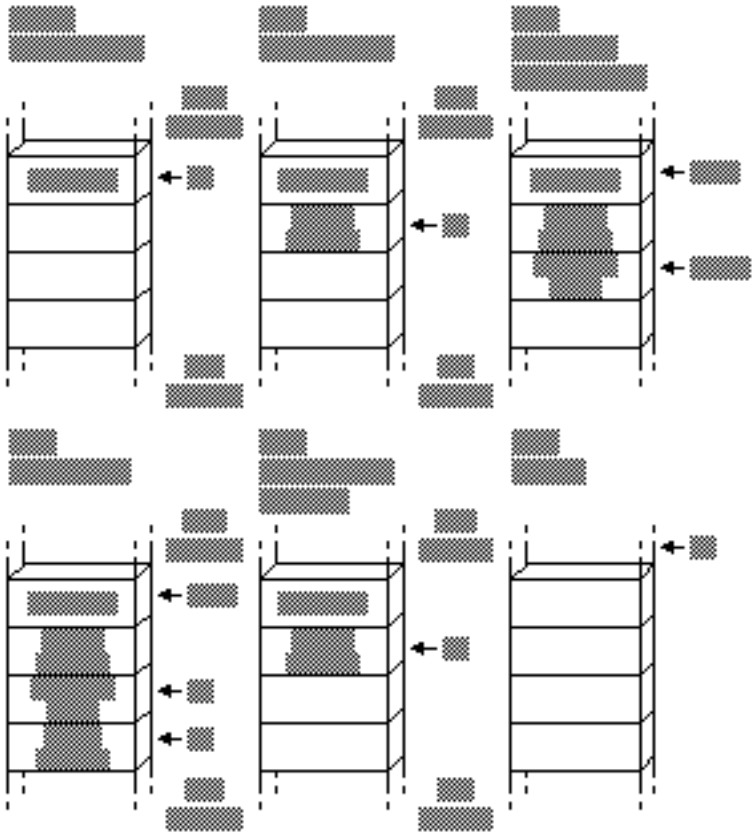


Figure 7.2 Local Variables on the Stack

Creating Local Variables Automatically

MASM's **LOCAL** directive automates the process for creating local variables on the stack. **LOCAL** frees you from having to count stack words, and it makes your code easier to write and maintain. This section illustrates the advantages of creating temporary data with the **LOCAL** directive.

To use the **LOCAL** directive, list the variables you want to create, giving a type for each one. The assembler calculates how much space is required on the stack. It also generates instructions to properly decrement SP (as described in the previous section) and to reset SP when you return from the procedure.

When you create local variables this way, your source code can refer to each local variable by name rather than as an offset of the stack pointer. Moreover,

the assembler generates debugging information for each local variable. If you have programmed before in a high-level language that allows scoping, local variables will seem familiar. For example, a C compiler sets up variables with automatic storage class in the same way as the **LOCAL** directive.

We can simplify the procedure in the previous section with the following code:

```
task    PROC    NEAR    arg:WORD
        LOCAL   loc:WORD
        .
        .
        .
        mov     loc, 3      ; Initialize local variable
        add     ax, loc     ; Add local variable to AX
        sub     arg, ax     ; Subtract local from argument
        .                   ; Use "loc" and "arg" in other operations
        .
        .
        ret
task    ENDP
```

The **LOCAL** directive must be on the line immediately following the **PROC** statement with the following syntax:

LOCAL *vardef* [[**,** *vardef*]]...

Each *vardef* defines a local variable. A local variable definition has this form:

label[[[*count***]**]][[**:***qualifiedtype*]]

These are the parameters in local variable definitions:

Argument	Description
label	The name given to the local variable. You can use this name to access the variable.
count	The number of elements of this name and type to allocate on the stack. You can allocate a simple array on the stack with *count*. The brackets around *count* are required. If this field is omitted, one data object is assumed.
qualifiedtype	A simple MASM type or a type defined with other types and attributes. For more information, see "Data Types" in Chapter 1.

If the number of local variables exceeds one line, you can place a comma at the end of the first line and continue the list on the next line. Alternatively, you can use several consecutive **LOCAL** directives.

The assembler does not initialize local variables. Your program must include code to perform any necessary initializations. For example, the following code fragment sets up a local array and initializes it to zero:

```
arraysz EQU     20

aproc   PROC    USES di
        LOCAL   var1[arraysz]:WORD, var2:WORD
        .
        .
        .
; Initialize local array to zero
        push    ss
        pop     es              ; Set ES=SS
        lea     di, var1        ; ES:DI now points to array
        mov     cx, arraysz     ; Load count
        sub     ax, ax
        rep     stosw           ; Store zeros
; Use the array...
        .
        .
        .
        ret
aproc   ENDP
```

Even though you can reference stack variables by name, the assembler treats them as offsets of BP, and they are not visible outside the procedure. In the following procedure, `array` is a local variable.

```
index   EQU   10
test    PROC  NEAR
LOCAL   array[index]:WORD
        .
        .
        .
        mov     bx, index
;       mov     array[bx], 5          ; Not legal!
```

The second **MOV** statement may appear to be legal, but since `array` is an offset of BP, this statement is the same as

```
;       mov [bp + bx + arrayoffset], 5   ; Not legal!
```

BP and BX can be added only to SI and DI. This example would be legal, however, if the index value were moved to SI or DI. This type of error in your program can be difficult to find unless you keep in mind that local variables in procedures are offsets of BP.

Declaring Procedure Prototypes

MASM provides the **INVOKE** directive to handle many of the details important to procedure calls, such as pushing parameters according to the correct calling conventions. To use **INVOKE**, the procedure called must have been declared previously with a **PROC** statement, an **EXTERNDEF** (or **EXTERN**) statement, or a **TYPEDEF**. You can also place a prototype defined with **PROTO** before the **INVOKE** if the procedure type does not appear before the **INVOKE**. Procedure prototypes defined with **PROTO** inform the assembler of types and numbers of arguments so the assembler can check for errors and provide automatic conversions when **INVOKE** calls the procedure.

Declaring procedure prototypes is good programming practice, but is optional. Prototypes in MASM perform the same function as prototypes in C and other high-level languages. A procedure prototype includes the procedure name, the types, and (optionally) the names of all parameters the procedure expects. Prototypes usually are placed at the beginning of an assembly program or in a separate include file so the assembler encounters the prototype before the actual procedure.

Prototypes enable the assembler to check for unmatched parameters and are especially useful for procedures called from other modules and other languages. If you write routines for a library, you may want to put prototypes into an include file for all the procedures used in that library. For more information about using include files, see Chapter 8, "Sharing Data and Procedures among Modules and Libraries."

The **PROTO** directive provides one way to define a procedure prototype. The syntax for a prototype definition is the same as for a procedure declaration (see "Declaring Parameters with the PROC Directive," earlier in this chapter), except that you do not include the list of registers, *prologuearg* list, or the scope of the procedure.

Also, the **PROTO** keyword precedes the *langtype* and *distance* attributes. The attributes (like **C** and **FAR**) are optional. However, if they are not specified, the defaults are based on any **.MODEL** or **OPTION LANGUAGE** statement. The names of the parameters are also optional, but you must list parameter types. A label preceding **:VARARG** is also optional in the prototype but not in the **PROC** statement.

If a **PROTO** and a **PROC** for the same function appear in the same module, they must match in attribute, number of parameters, and parameter types. The easiest way to create prototypes with **PROTO** is to write your procedure and then copy the first line (the line that contains the **PROC** keyword) to a location in your program that follows the data declarations. Change **PROC** to **PROTO** and remove the **USES** *reglist*, the *prologuearg* field, and the *visibility* field. It

is important that the prototype follow the declarations for any types used in it to avoid any forward references used by the parameters in the prototype.

The following example illustrates how to define and then declare two typical procedures. In both prototype and declaration, the comma before the argument list is optional only when the list does not appear on a separate line:

```
; Procedure prototypes.

addup     PROTO NEAR C argcount:WORD, arg2:WORD, arg3:WORD
myproc    PROTO FAR C, argcount:WORD, arg2:VARARG

; Procedure declarations

addup     PROC NEAR C, argcount:WORD, arg2:WORD, arg3:WORD
.
.
.
myproc    PROC FAR C PUBLIC <callcount> USES di si,
          argcount:WORD,
          arg2:VARARG
```

When you call a procedure with **INVOKE**, the assembler checks the arguments given by **INVOKE** against the parameters expected by the procedure. If the data types of the arguments do not match, MASM reports an error or converts the type to the expected type. These conversions are explained in the next section.

Calling Procedures with INVOKE

INVOKE generates a sequence of instructions that push arguments and call a procedure. This helps maintain code if arguments or *langtype* for a procedure are changed. **INVOKE** generates procedure calls and automatically:

- Converts arguments to the expected types.
- Pushes arguments on the stack in the correct order.
- Cleans the stack when the procedure returns.

If arguments do not match in number or if the type is not one the assembler can convert, an error results.

If the procedure uses **VARARG**, **INVOKE** can pass a number of arguments different from the number in the parameter list without generating an error or warning. Any additional arguments must be at the end of the **INVOKE** argument list. All other arguments must match those in the prototype parameter list.

The syntax for **INVOKE** is:

INVOKE *expression* [[**,** *arguments*]]

where *expression* can be the procedure's label or an indirect reference to a procedure, and *arguments* can be an expression, a register pair, or an expression preceded with **ADDR**. (The **ADDR** operator is discussed later in this chapter.)

Procedures with these prototypes

```
addup   PROTO NEAR C argcount:WORD, arg2:WORD, arg3:WORD
myproc  PROTO FAR C, argcount:WORD, arg2:VARARG
```

and these procedure declarations

```
addup   PROC NEAR C, argcount:WORD, arg2:WORD, arg3:WORD
.
.
.
myproc  PROC FAR C PUBLIC <callcount> USES di si,
        argcount:WORD,
        arg2:VARARG
```

can be called with **INVOKE** statements like this:

```
        INVOKE  addup,   ax, x,  y
        INVOKE  myproc,  bx, cx, 100, 10
```

The assembler can convert some arguments and parameter type combinations so that the correct type can be passed. The signed or unsigned qualities of the arguments in the **INVOKE** statements determine how the assembler converts them to the types expected by the procedure.

The `addup` procedure, for example, expects parameters of type **WORD**, but the arguments passed by **INVOKE** to the `addup` procedure can be any of these types:

- **BYTE**, **SBYTE**, **WORD**, or **SWORD**
- An expression whose type is specified with the **PTR** operator to be one of those types
- An 8-bit or 16-bit register
- An immediate expression in the range –32K to +64K
- A **NEAR PTR**

If the type is smaller than that expected by the procedure, MASM widens the argument to match.

Widening Arguments

For **INVOKE** to correctly handle type conversions, you must use the signed data types for any signed assignments. MASM widens an argument to match the type expected by a procedure's parameters in these cases:

Type Passed	Type Expected
BYTE, **SBYTE**	**WORD**, **SWORD**, **DWORD**, **SDWORD**
WORD, **SWORD**	**DWORD**, **SDWORD**

The assembler can extend a segment if far data is expected, and it can convert the type given in the list to the types expected. If the assembler cannot convert the type, however, it generates an error.

Detecting Errors

If the assembler needs to widen an argument, it first copies the value to AL or AX. It widens an unsigned value by placing a zero in the higher register area, and widens a signed value with a **CBW**, **CWD**, or **CWDE** instruction as required. Similarly, the assembler copies a constant argument value into AL or AX when the **.8086** directive is in effect. You can see these generated instructions in the listing file when you include the /Sg command-line option.

Using the accumulator register to widen or copy an argument may lead to an error if you attempt to pass AX as another argument. For example, consider the following **INVOKE** statement for a procedure with the C calling convention

```
INVOKE  myprocA, ax, cx, 100, arg
```

where `arg` is a **BYTE** variable and `myproc` expects four arguments of type **WORD**. The assembler widens and then pushes `arg` like this:

```
mov     al, DGROUP:arg
xor     ah, ah
push    ax
```

The generated code thus overwrites the last argument (AX) passed to the procedure. The assembler generates an error in this case, requiring you to rewrite the **INVOKE** statement.

To summarize, the **INVOKE** directive overwrites AX and perhaps DX when widening arguments. It also uses AX to push constants on the 8088 and 8086. If you use these registers (or EAX and EDX on an 80386/486) to pass arguments, they may be overwritten. The assembler's error detection prevents this from ever becoming a run-time bug, but AX and DX should remain your last choice for holding arguments.

Invoking Far Addresses

You can pass a **FAR** pointer in a *segment***::***offset* pair, as shown in the following. Note the use of double colons to separate the register pair. The registers could be any other register pair, including a pair that an MS-DOS call uses to return values.

```
FPWORD   TYPEDEF FAR PTR WORD
SomeProc PROTO var1:DWORD, var2:WORD, var3:WORD

        pfaritem    FPWORD     faritem
        .
        .
        .
        les         bx, pfaritem
        INVOKE      SomeProc, ES::BX, arg1, arg2
```

However, **INVOKE** cannot combine into a single address one argument for the segment and one for the offset.

Passing an Address

You can use the **ADDR** operator to pass the address of an expression to a procedure that expects a **NEAR** or **FAR** pointer. This example generates code to pass a far pointer (to `arg1`) to the procedure `proc1`.

```
PBYTE   TYPEDEF FAR PTR BYTE
arg1    BYTE    "This is a string"
proc1   PROTO   NEAR C fparg:PBYTE
        .
        .
        .
INVOKE  proc1, ADDR arg1
```

For information on defining pointers with **TYPEDEF,** see "Defining Pointer Types with TYPEDEF" in Chapter 3.

Invoking Procedures Indirectly

You can make an indirect procedure call such as `call [bx + si]` by using a pointer to a function prototype with **TYPEDEF**, as shown in this example:

```
FUNCPROTO       TYPEDEF PROTO NEAR ARG1:WORD
FUNCPTR         TYPEDEF PTR FUNCPROTO

        .DATA
pfunc   FUNCPTR OFFSET proc1, OFFSET proc2

        .CODE
        .
        .
        .
        mov     bx, OFFSET pfunc          ; BX points to table
        mov     si, Num                   ; Num contains 0 or 2
        INVOKE  FUNCPTR PTR [bx+si], arg1 ; Call proc1 if Num=0
                                          ;   or proc2 if Num=2
```

You can also use **ASSUME** to accomplish the same task. The following **ASSUME** statement associates the type `FUNCPTR` with the BX register.

```
        ASSUME  BX:FUNCPTR
        mov     bx, OFFSET pfunc
        mov     si, Num
        INVOKE  [bx+si], arg1
```

Checking the Code Generated

Code generated by the **INVOKE** directive may vary depending on the processor mode and calling conventions in effect. You can check your listing files to see the code generated by the **INVOKE** directive if you use the /Sg command-line option.

Generating Prologue and Epilogue Code

When you use the **PROC** directive with its extended syntax and argument list, the assembler automatically generates the prologue and epilogue code in your procedure. “Prologue code” is generated at the start of the procedure. It sets up a stack pointer so you can access parameters from within the procedure. It also saves space on the stack for local variables, initializes registers such as DS, and pushes registers that the procedure uses. Similarly, “epilogue code” is the code at the end of the procedure that pops registers and returns from the procedure.

The assembler automatically generates the prologue code when it encounters the first instruction or label after the **PROC** directive. This means you cannot label the prologue for the purpose of jumping to it. The assembler generates the epilogue code when it encounters a **RET** or **IRET** instruction. Using the assembler-generated prologue and epilogue code saves time and decreases the number of repetitive lines of code in your procedures.

The generated prologue or epilogue code depends on the:

- Local variables defined.
- Arguments passed to the procedure.
- Current processor selected (affects epilogue code only).
- Current calling convention.
- Options passed in the *prologuearg* of the **PROC** directive.
- Registers being saved.

The *prologuearg* list contains options specifying how to generate the prologue or epilogue code. The next section explains how to use these options, gives the standard prologue and epilogue code, and explains the techniques for defining your own prologue and epilogue code.

Using Automatic Prologue and Epilogue Code

The standard prologue and epilogue code handles parameters and local variables. If a procedure does not have any parameters or local variables, the prologue and epilogue code that sets up and restores a stack pointer is omitted, unless
FORCEFRAME is included in the *prologuearg* list. (**FORCEFRAME** is discussed later in this section.) Prologue and epilogue code also generates a push and pop for each register in the register list.

The prologue code consists of three steps:

1. Point BP to top of stack.
2. Make space on stack for local variables.
3. Save registers the procedure must preserve.

The epilogue cancels these three steps in reverse order, then cleans the stack, if necessary, with a **RET** *num* instruction. For example, the procedure declaration

```
myproc  PROC NEAR PASCAL USES di si,
        arg1:WORD, arg2:WORD, arg3:WORD
        LOCAL local1:WORD, local2:WORD
```

generates the following prologue code:

```
        push    bp              ; Step 1:
        mov     bp, sp          ;   point BP to stack top
        sub     sp, 4           ; Step 2: space for 2 local words
        push    di              ; Step 3:
        push    si              ;   save registers listed in USES
```

The corresponding epilogue code looks like this:

```
        pop     si              ; Undo Step 3
        pop     di
        mov     sp, bp          ; Undo Step 2
        pop     bp              ; Undo Step 1
        ret     6               ; Clean stack of pushed arguments
```

Notice the **RET 6** instruction cleans the stack of the three word-sized arguments. The instruction appears in the epilogue because the procedure does not use the C calling convention. If `myproc` used C conventions, the epilogue would end with a **RET** instruction without an operand.

The assembler generates standard epilogue code when it encounters a **RET** instruction without an operand. It does not generate an epilogue if **RET** has a nonzero operand. To suppress generation of a standard epilogue, use **RETN** or **RETF** with or without an operand, or use **RET 0**.

The standard prologue and epilogue code recognizes two operands passed in the *prologuearg* list, **LOADDS** and **FORCEFRAME**. These operands modify the prologue code. Specifying **LOADDS** saves and initializes DS. Specifying **FORCEFRAME** as an argument generates a stack frame even if no arguments are sent to the procedure and no local variables are declared. If your procedure has any parameters or locals, you do not need to specify **FORCEFRAME**.

For example, adding **LOADDS** to the argument list for `myproc` creates this prologue:

```
push    bp              ; Step 1:
mov     bp, sp          ;   point BP to stack top
sub     sp, 4           ; Step 2: space for 2 locals
push    ds              ; Save DS and point it
mov     ax, DGROUP      ;   to DGROUP, as
mov     ds, ax          ;   instructed by LOADDS
push    di              ; Step 3:
push    si              ;   save registers listed in USES
```

The epilogue code restores DS:

```
pop     si              ; Undo Step 3
pop     di
pop     ds              ; Restore DS
mov     sp, bp          ; Undo Step 2
pop     bp              ; Undo Step 1
ret     6               ; Clean stack of pushed arguments
```

User-Defined Prologue and Epilogue Code

If you want a different set of instructions for prologue and epilogue code in your procedures, you can write macros that run in place of the standard prologue and epilogue code. For example, while you are debugging your procedures, you may want to include a stack check or track the number of times a procedure is called. You can write your own prologue code to do these things whenever a procedure executes. Different prologue code may also be necessary if you are writing applications for Windows. User-defined prologue macros will respond correctly if you specify **FORCEFRAME** in the *prologuearg* of a procedure.

To write your own prologue or epilogue code, the **OPTION** directive must appear in your program. It disables automatic prologue and epilogue code generation. When you specify

OPTION PROLOGUE : *macroname*

OPTION EPILOGUE : *macroname*

the assembler calls the macro specified in the **OPTION** directive instead of generating the standard prologue and epilogue code. The prologue macro must be a macro function, and the epilogue macro must be a macro procedure.

The assembler expects your prologue or epilogue macro to have this form:

macroname **MACRO** *procname*, \
flag, \
parmbytes, \
localbytes, \
<*reglist*>, \
userparms

Your macro must have formal parameters to match all the actual arguments passed. The arguments passed to your macro include:

Argument	Description	
procname	The name of the procedure.	
flag	A 16-bit flag containing the following information:	
	Bit = Value	**Description**
	Bit 0, 1, 2	For calling conventions (000=unspecified language type, 001=**C**, 010=**SYSCALL**, 011=**STDCALL**, 100=**PASCAL**, 101=**FORTRAN**, 110=**BASIC**).
	Bit 3	Undefined (not necessarily zero).
	Bit 4	Set if the caller restores the stack (use **RET**, not **RET***n*).
	Bit 5	Set if procedure is **FAR**.
	Bit 6	Set if procedure is **PRIVATE**.
	Bit 7	Set if procedure is **EXPORT**.
	Bit 8	Set if the epilogue is generated as a result of an **IRET** instruction and cleared if the epilogue is generated as a result of a **RET** instruction.
	Bits 9–15	Undefined (not necessarily zero).
parmbytes	The accumulated count in bytes of all parameters given in the **PROC** statement.	
localbytes	The count in bytes of all locals defined with the **LOCAL** directive.	
reglist	A list of the registers following the **USES** operator in the procedure declaration. Enclose this list with angle brackets (< >) and separate each item with commas. Reverse the list for epilogues.	
userparms	Any argument you want to pass to the macro. The prologuearg (if there is one) specified in the **PROC** directive is passed to this argument.	

Your macro function must return the *parmbytes* parameter. However, if the prologue places other values on the stack after pushing BP and these values are not referenced by any of the local variables, the exit value must be the number of bytes for procedure locals plus any space between BP and the locals. Therefore, *parmbytes* is not always equal to the bytes occupied by the locals.

The following macro is an example of a user-defined prologue that counts the number of times a procedure is called.

```
ProfilePro      MACRO procname,       \
                      flag,           \
                      bytecount,      \
                      numlocals,      \
                      regs,           \
                      macroargs

                .DATA
procname&count  WORD 0
                .CODE
        inc     procname&count  ; Accumulates count of times the
                                ;   procedure is called
        push    bp
        mov     bp, sp
                                ; Other BP operations
        IFNB <regs>
           FOR r, regs
              push r
           ENDM
        ENDIF
        EXITM %bytecount
ENDM
```

Your program must also include this statement before calling any procedures that use the prologue:

```
OPTION PROLOGUE:ProfilePro
```

If you define either a prologue or an epilogue macro, the assembler uses the standard prologue or epilogue code for the one you do not define. The form of the code generated depends on the **.MODEL** and **PROC** options used.

If you want to revert to the standard prologue or epilogue code, use `PROLOGUEDEF` or `EPILOGUEDEF` as the *macroname* in the **OPTION** statement.

```
OPTION EPILOGUE:EPILOGUEDEF
```

You can completely suppress prologue or epilogue generation with

```
OPTION PROLOGUE:None
OPTION EPILOGUE:None
```

In this case, no user-defined macro is called, and the assembler does not generate a default code sequence. This state remains in effect until the next **OPTION**
PROLOGUE or **OPTION EPILOGUE** is encountered.

For additional information about writing macros, see Chapter 9, "Using Macros." The PROLOGUE.INC file provided in the MASM 6.1 distribution disks can create the prologue and epilogue sequences for the Microsoft C professional development system.

MS-DOS Interrupts

In addition to jumps, loops, and procedures that alter program execution, interrupt routines transfer execution to a different location. In this case, control goes to an interrupt routine.

You can write your own interrupt routines, either to replace an existing routine or to use an undefined interrupt number. For example, you may want to replace an MS-DOS interrupt handler, such as the Critical Error (Interrup 24h) and CONTROL+C (Interrupt 23h) handlers. The **BOUND** instruction checks array bounds and calls Interrupt 5 when an error occurs. If you use this instruction, you need to write an interrupt handler for it.

This section summarizes the following:

- How to call interrupts
- How the processor handles interrupts
- How to redefine an existing interrupt routine

The example routine in this section handles addition or multiplication overflow and illustrates the steps necessary for writing an interrupt routine. For additional information about MS-DOS and BIOS interrupts, see Chapter 11, "Writing Memory-Resident Software."

Calling MS-DOS and ROM-BIOS Interrupts

Interrupts provide a way to access MS-DOS and ROM-BIOS from assembly language. They are called with the **INT** instruction, which takes an immediate value between 0 and 255 as its only operand.

MS-DOS and ROM-BIOS interrupt routines accept data through registers. For instance, most MS-DOS routines (and many BIOS routines) require a function number in the AH register. Many handler routines also return values in registers. To use an interrupt, you must know what data the handler routine expects and what data, if any, it returns. For information, consult Help or one of the other references mentioned in the Introduction.

The following fragment illustrates a simple call to MS-DOS Function 9, which displays the string `msg` on the screen:

```
        .DATA
msg     BYTE    "This writes to the screen$"
        .CODE
        mov     ax, SEG msg     ; Necessary only if DS does not
        mov     ds, ax          ;   already point to data segment
        mov     dx, offset msg  ; DS:DX points to msg
        mov     ah, 09h         ; Request Function 9
        int     21h
```

When the **INT** instruction executes, the processor:

1. Looks up the address of the interrupt routine in the Interrupt Vector Table. This table starts at the lowest point in memory (segment 0, offset 0) and consists of a series of far pointers called vectors. Each vector comprises a 4-byte address (segment:offset) pointing to an interrupt handler routine. The table sequence implies the number of the interrupt the vector references: the first vector points to the Interrupt 0 handler, the second vector to the Interrupt 1 handler, and so forth. Thus, the vector at 0000:*i**4 holds the address of the handler routine for Interrupt *i*.
2. Clears the trap flag (TF) and interrupt enable flag (IF).
3. Pushes the flags register, the current code segment (CS), and the current instruction pointer (IP), in that order. (The current instruction is the one following the **INT** statement.) As with a **CALL**, this ensures control returns to the next logical position in the program.
4. Jumps to the address of the interrupt routine, as specified in the Interrupt Vector Table.
5. Executes the code of the interrupt routine until it encounters an **IRET** instruction.
6. Pops the instruction pointer, code segment, and flags.

Figure 7.3 illustrates how interrupts work.

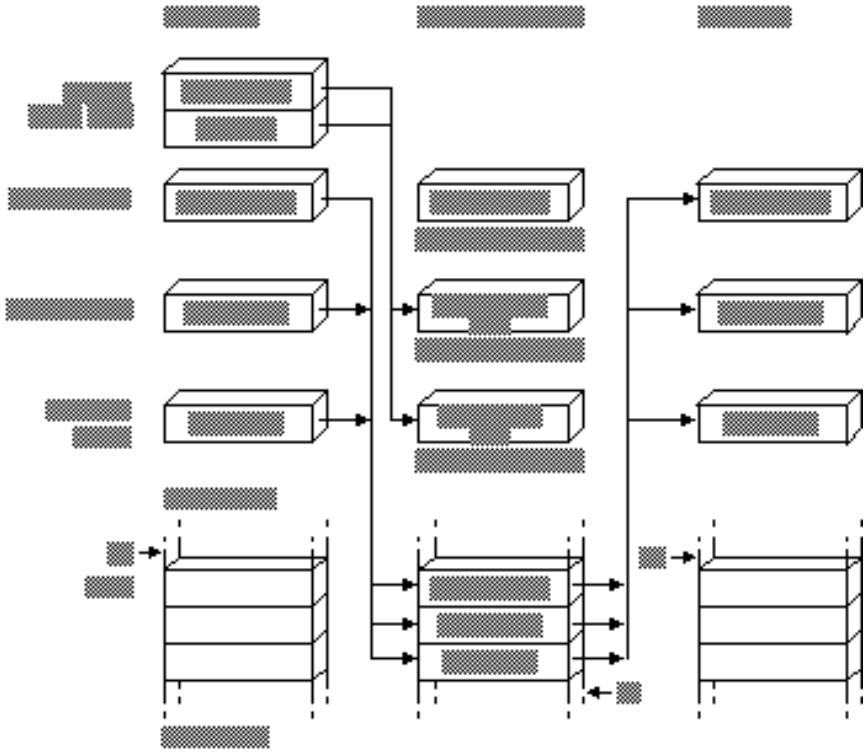


Figure 7.3 Operation of Interrupts

Replacing an Interrupt Routine

To replace an existing interrupt routine, your program must:

- Provide a new routine to handle the interrupt.
- Replace the old routine's address in the Interrupt Vector Table with the address of your new routine.
- Replace the old address back into the vector table before your program ends.

You can write an interrupt routine as a procedure by using the **PROC** and **ENDP** directives. The routine should always be defined as **FAR** and should end with an **IRET** instruction instead of a **RET** instruction.

Note You can use the full extended **PROC** syntax (described in "Declaring Parameters with the PROC Directive," earlier in this chapter) to write interrupt procedures. However, you should not make interrupt procedures **NEAR** or specify arguments for them. You can use the **USES** keyword, however, to correctly generate code to save and restore a register list in interrupt procedures.

The **IRET** instruction in MASM 6.1 has two forms that suppress epilogue code. This allows an interrupt to have local variables or use a user-defined prologue. **IRETF** pops a **FAR16** return address, and **IRETFD** pops a **FAR32** return address.

The following example shows how to replace the handler for Interrupt 4. Once registered in the Interrupt Vector Table, the new routine takes control when the processor encounters either an **INT 4** instruction or its special variation **INTO** (Interrupt on Overflow). **INTO** is a conditional instruction that acts only when the overflow flag is set. With **INTO** after a numerical calculation, your code can automatically route control to a handler routine if the calculation results in a numerical overflow. By default, the routine for Interrupt 4 simply consists of an **IRET**, so it returns without doing anything. Using **INTO** is an alternative to using **JO** (Jump on Overflow) to jump to another set of instructions.

The following example program first executes **INT 21h** to invoke MS-DOS Function 35h (Get Interrupt Vector). This function returns the existing vector for Interrupt 4. The program stores the vector, then invokes MS-DOS Function 25h (Set Interrupt Vector) to place the address of the `ovrflow` procedure in the Interrupt Vector Table. From this point on, `ovrflow` gains control whenever the processor executes **INTO** while the overflow flag is set. The new routine displays a message and returns with AX and DX set to 0.

```
        .MODEL LARGE, C
FPFUNC  TYPEDEF FAR PTR
        .DATA
msg     BYTE    "Overflow - result set to 0",13,10,'$'
vector  FPFUNC  ?
        .CODE
        .STARTUP

        mov     ax, 3504h               ; Load Interrupt 4 and call DOS
        int     21h                     ;   Get Interrupt Vector
        mov     WORD PTR vector[2],es   ; Save segment
        mov     WORD PTR vector[0],bx   ;   and offset
```

```
        push    ds                      ; Save DS
        mov     ax, cs                  ; Load segment of new routine
        mov     ds, ax
        mov     dx, OFFSET ovrflow      ; Load offset of new routine
        mov     ax, 2504h               ; Load Interrupt 4 and call DOS
        int     21h                     ;   Set Interrupt Vector
        pop     ds                      ; Restore
        .
        .
        .
        add     ax, bx                  ; Do arithmetic
        into                            ; Call Interrupt 4 if overflow
        .
        .
        .
        lds     dx, vector              ; Load original address
        mov     ax, 2504h               ; Restore it to vector table
        int     21h                     ;   with DOS set vector function
        mov     ax, 4C00h               ; Terminate function
        int     21h


ovrflow PROC    FAR
        sti                             ; Enable interrupts
                                        ;   (turned off by INT)
        mov     ah, 09h                 ; Display string function
        mov     dx, OFFSET msg          ; Load address
        int     21h                     ; Call DOS
        sub     ax, ax                  ; Set AX to 0
        cwd                             ; Set DX to 0
        iret                            ; Return
ovrflow ENDP
        END
```

Before the program ends, it again uses MS-DOS Function 25h to reset the original Interrupt 4 vector back into the Interrupt Vector Table. This reestablishes the original routine as the handler for Interrupt 4.

The first instruction of the `ovrflow` routine warrants further discussion. When the processor encounters an **INT** instruction, it clears the interrupt flag before branching to the specified interrupt handler routine. The interrupt flag serves a crucial role in smoothing the processor's tasks, but must not be abused. When clear, the flag inhibits hardware interrupts such as the keyboard or system timer. It should be left clear only briefly and only when absolutely necessary. Unless you have a

compelling reason to leave the flag clear, always include an **STI** (Set Interrupt Flag) instruction at the beginning of your interrupt handler routine to reenable hardware interrupts.

CLI (Clear Interrupt Flag) and its corollary **STI** are designed to protect small sections of time-dependent code from interruptions by the hardware. If you use **CLI** in your program, be sure to include a matching **STI** instruction as well. The sample interrupt handlers in Chapter 11, "Writing Memory-Resident Software," illustrate how to use these important instructions.