23. When 16-bit numbers are divided, in which register is the quotient found?
24. What errors are detected during a division?
25. Explain the difference between the IDIV and DIV instructions.
26. Where is the remainder found after an 8-bit division?
27. Write a short sequence of instructions that divides the number in BL by the number in CL, and then multiplies the result by 2. The final answer must be a 16-bit number stored in the DX register.
28. Which instructions are used with BCD arithmetic operations?
29. Which instructions are used with ASCII arithmetic operations?
30. Explain how the AAM instruction converts from binary to BCD.
31. Develop a sequence of instructions that converts the unsigned number in AX (values of 0–65535) into a 5-digit BCD number stored in memory, beginning at the location addressed by the BX register in the data segment. Note that the most-significant character is stored first and no attempt is made to blank leading zeros.
32. Develop a sequence of instructions that adds the 8-digit BCD number in AX and BX to the 8-digit BCD number in CX and DX. (AX and CX are the most-significant registers. The result must be found in CX and DX after the addition.)
33. Select an AND instruction that will:
    (a) AND BX with DX and save the result in BX
    (b) AND 0EAH with DH
    (c) AND DI with BP and save the result in DI
    (d) AND the data addressed by BP with CX and save the result in memory
    (e) AND the data stored in four words before the location addressed by SI with DX and save the result in DX
    (f) AND AL with memory location WHAT and save the result at location WHAT
34. Develop a short sequence of instructions that clears (0) the three leftmost bits of DH without changing the remainder DH and stores the result in BH.
35. Select an OR instruction that will:
    (a) OR BL with AH and save the result in AH
    (b) OR 88H with ECX
    (c) OR DX with SI and save the result in SI
    (d) OR 1122H with BP
    (e) OR the data addressed by BX with CX and save the result in memory
    (f) OR the data stored 40 bytes after the location addressed by BP with AL and save the result in AL
    (g) OR AH with memory location WHEN and save the result in WHEN
36. Develop a short sequence of instructions that sets (1) the rightmost five bits of DI without changing the remaining bits of DI. Save the results in SI.
37. Select the XOR instruction that will:
    (a) XOR BH with AH and save the result in AH
    (b) XOR 99H with CL
    (c) XOR DX with DI and save the result in DX
    (d) XOR the data stored 30 words after the location addressed by BP with DI and save the result in DI
    (e) XOR DI with memory location WELL and save the result in DI
38. Develop a sequence of instructions that sets (1) the rightmost four bits of AX; clears (0) the leftmost three bits of AX; and inverts bits 7, 8, and 9 of AX.
39. Describe the difference between the AND and TEST instructions.
40. Select an instruction that tests bit position 2 of register CH.
41. What is the difference between the NOT and the NEG instruction?
42. Select the correct instruction to perform each of the following tasks:
    (a) shift DI right three places, with zeros moved into the leftmost bit

(b) move all bits in AL left one place, making sure that a 0 moves into the rightmost bit position
(c) rotate all the bits of AL left three places
(d) move the DH register right one place, making sure that the sign of the result is the same as the sign of the original number

43. What does the SCASB instruction accomplish?
44. For string instructions, DI always addresses data in the _____ segment.
45. What is the purpose of the D flag bit?
46. Explain what the REPE prefix does when coupled with the SCASB instruction.
47. What condition or conditions will terminate the repeated string instruction REPNE SCASB?
48. Describe what the CMPSB instruction accomplishes.
49. Develop a sequence of instructions that scans through a 300H-byte section of memory called LIST, located in the data segment searching for a 66H.
50. What happens if AH = 02H and DL = 43H when the INT 21H instruction is executed?

CHAPTER 6

Program Control Instructions

INTRODUCTION

The program control instructions direct the flow of a program and allow the flow to change. A change in flow often occurs after a decision, made with the CMP or TEST instruction, is followed by a conditional jump instruction. This chapter explains the program control instructions, including the jumps, calls, returns, interrupts, and machine control instructions.

Also presented in this chapter are the relational assembly language statements (.IF, .ELSE, .ELSEIF, .ENDIF, .WHILE, .ENDW, .REPEAT, and .UNTIL) that are available in version 6.X and above of MASM or TASM, with version 5.X set for MASM compatibility. These relational assembly language commands allow the programmer to develop control flow portions of the program with C/C++ language efficiency.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use both conditional and unconditional jump instructions to control the flow of a program.
2. Use the relational assembly language statements .IF, .REPEAT, .WHILE, and so forth in programs.
3. Use the call and return instructions to include procedures in the program structure.
4. Explain the operation of the interrupts and interrupt control instructions.
5. Use machine control instructions to modify the flag bits.

6–1 THE JUMP GROUP

The main program control instruction, **jump** (JMP), allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction. A conditional jump instruction allows the programmer to make decisions based upon numerical tests. The results of numerical tests are held in the flag bits, which are then tested by conditional jump instructions. Another instruction similar to the conditional jump, the conditional set, is explained with the conditional jump instructions in this section.

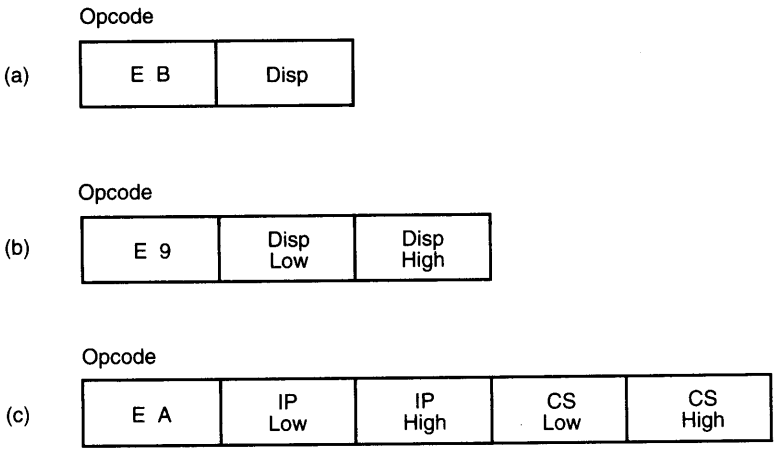

FIGURE 6–1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.

In this section of the text, all jump instructions are illustrated with their uses in sample programs. Also revisited are the LOOP and conditional LOOP instructions, first presented in Chapter 3, because they are also forms of the jump instruction.

Unconditional Jump (JMP)

Three types of unconditional jump instructions (see Figure 6–1) are available to the microprocessor: short jump, near jump, and far jump. The **short jump** is a two-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes from the address following the jump. The three-byte **near jump** allows a branch or jump within ±32K bytes (or anywhere in the current code segment) from the instruction in the current code segment. Remember that segments are cyclic in nature, which means that one location above offset address FFFFH is offset address 0000H. For this reason, if you jump two bytes ahead in memory and the instruction pointer addresses offset address FFFFH, the flow continues at offset address 0001H. Thus, a displacement of ±32K bytes allows a jump to any location within the current code segment. Finally, the five-byte **far jump** allows a jump to any memory location within the real memory system. The short and near jumps are often called **intrasegment** jumps, and the far jumps are often called **intersegment** jumps.

In the 80386 through the Pentium 4 processors, the near jump is within ±2G if the machine is operated in the protected mode, with a code segment that is 4G bytes long. If operated in the real mode, the near jump is within ±32K bytes. In the protected mode, the 80386 and above use a 32-bit displacement that is not shown in Figure 6–1.

Short Jump. Short jumps are called **relative jumps** because they can be moved, along with their related software, to any location in the current code segment without a change. This is because the jump address is not stored with the opcode. Instead of a jump address, a **distance,** or displacement, follows the opcode. The short jump displacement is a distance represented by a one-byte signed number whose value ranges between +127 and –128. The short jump instruction appears in Figure 6–2. When the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment. The short jump instruction branches to this new address for the next instruction in the program.

Example 6–1 shows how short jump instructions pass control from one part of the program to another. It also illustrates the use of a **label** (a symbolic name for a memory address) with the jump instruction. Notice how one jump (JMP SHORT NEXT) uses the SHORT directive to force a short jump, while the other does not. Most assembler programs choose the best form of the jump instruction so the second jump instruction (JMP START) also assembles as a short jump. If the address of the next instruction (0009H) is added to the sign-extended dis-

placement (0017H) of the first jump, the address of NEXT is at location 0017H + 0009H or 0020H.

EXAMPLE 6–1

```
0000  33 DB                    XOR   BX,BX

0002  B8 0001      START:      MOV   AX,1
0005  03 C3                    ADD   AX,BX
0007  EB 17                    JMP   SHORT NEXT

0020  8B D8        NEXT:       MOV   BX,AX
0022  EB DE                    JMP   START
```

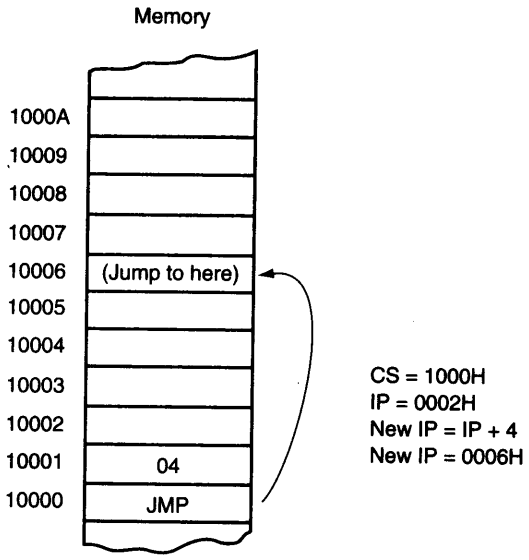

FIGURE 6–2 A short jump to four memory locations beyond the address of the next instruction.

Whenever a jump instruction references an address, a label normally identifies the address. The JMP NEXT instruction is an example; it jumps to label NEXT for the next instruction. It is very rare to ever use an actual hexadecimal address with any jump instruction, but the assembler supports addressing in relation to the instruction pointer by using the $ + a displacement. For example, a JMP $+2 jumps over the next two memory locations following the JMP instruction. The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it for a jump. If a colon does not follow a label, you cannot jump to it. Note that the only time a colon is used after a label is when the label is used with a jump or call instruction.

Near Jump. The near jump is similar to the short jump, except that the distance is farther. A **near jump** passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction. The near jump is a three-byte instruction that contains an opcode followed by a signed 16-bit displacement. The signed displacement adds to the instruction pointer (IP) to generate the jump address. Because the signed displacement is in the range of ±32K, a near jump can jump to any memory location within the current real mode code segment. Figure 6-3 illustrates the operation of the real mode near jump instruction.

The near jump is also relocatable (as was the short jump) because it is also a relative jump. If the code segment moves to a new location in the memory, the distance between the jump instruction and the operand address remains the same. This allows a code segment to be relocated by simply moving it. This feature, along with the relocatable data segments, makes the Intel family of microprocessors ideal for use in a general purpose computer system. Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

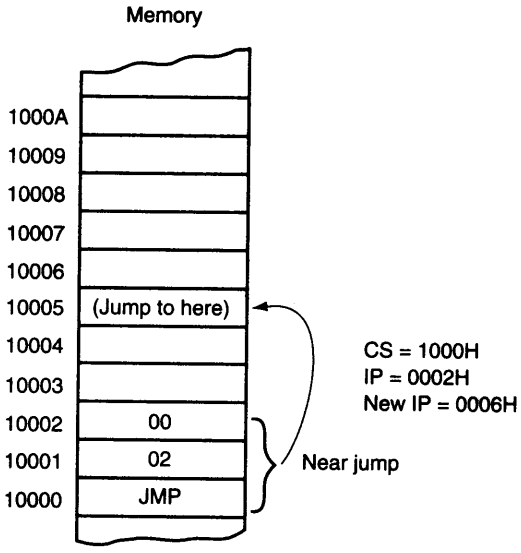

FIGURE 6–3 A near jump that adds the displacement (0002H) to the contents of IP.

Example 6–2 shows the same basic program that appeared in Example 6–1, except that the jump distance is greater. The first jump (JMP NEXT) passes control to the instruction at offset memory location 0200H within the code segment. Notice that the instruction assembles as an E9 0200 R. The letter R denotes a **relocatable jump** address of 0200H. The relocatable address of 0200H is for the assembler program's internal use only. The **actual machine language instruction** assembles as an E9 F6 01, which does not appear in the assembler listing. The actual displacement is a 01F6H for this jump instruction. The assembler lists the jump address as 0200 R, so the address is easier to interpret as software is developed. If the linked execution file (.EXE) or command file (.COM) is displayed in hexadecimal code, the jump instruction appears as an E9 F6 01.

EXAMPLE 6–2

```
0000  33 DB                    XOR   BX,BX

0002  B8 0001          START:  MOV   AX,1
0005  03 C3                    ADD   AX,BX
0007  E9 0200 R                JMP   NEXT

0200  8B D8            NEXT:   MOV   BX,AX
0202  E9 0002 R                JMP   START
```

Far Jump. A far jump instruction (see Figure 6–4) obtains a new segment and offset address to accomplish the jump. Bytes 2 and 3 of this five-byte instruction contain the new offset address; bytes 4 and 5 contain the new segment address. The offset address, which is 16-bits, contains the offset location within the new code segment.

Example 6–3 lists a short program that uses a far jump instruction. The far jump instruction sometimes appears with the FAR PTR directive, as illustrated. Another way to obtain a far jump is to define a label as a **far label.** A label is far only if it is external to the current code segment or procedure. The JMP UP instruction in the example references a far label. The label UP is defined as a far label by the EXTRN UP:FAR directive. External labels appear in programs that contain more than one program file. Another way of defining a label as global is to use a double colon (LABEL::), following the label in place of the single colon. This is required inside procedure blocks that are defined as near if the label is accessed from outside the procedure block.

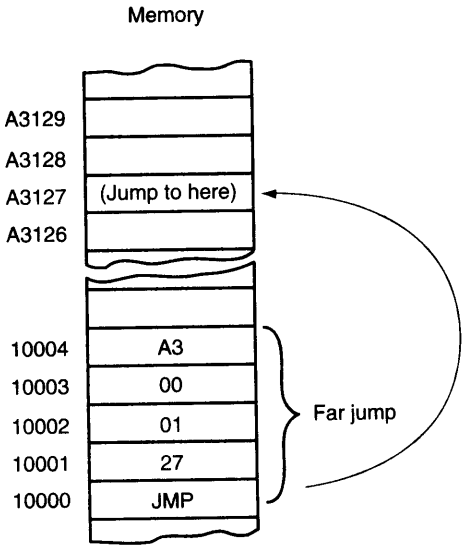

FIGURE 6–4 A far jump instruction replaces the contents of both CS and IP with four bytes following the opcode.

EXAMPLE 6–3

```
                               EXTRN   UP:FAR

0000  33 DB                    XOR   BX,BX
0002  B8 0001          START:  MOV   AX,1
0005  03 C3                    ADD   AX,BX
0007  E9 0200 R                JMP   NEXT

0200  8B D8            NEXT:   MOV   BX,AX
0202  EA 0002 ---- R           JMP   FAR PTR START

0207  EA 0000 ---- E           JMP   UP
```

When the program files are joined, the linker inserts the address for the UP label into the JMP UP instruction. It also inserts the segment address in the JMP START instruction. The segment address in JMP FAR PTR START is listed as – – – – R for relocatable; the segment address in JMP UP is listed as – – – – E for external. In both cases, the – – – – is filled in by the linker when it links or joins the program files.

Jumps with Register Operands. The jump instruction can also use a 16- or 32-bit register as an operand. This automatically sets up the instruction as an **indirect jump.** The address of the jump is in the register specified by the jump instruction. Unlike the displacement associated with the near jump, the contents of the register are transferred directly into the instruction pointer. An indirect jump does not add to the instruction pointer, as with short and near jumps. The JMP AX instruction, for example, copies the contents of the AX register into the IP when the jump occurs. This allows a jump to any location within the current code segment. In the 80386 and above, a JMP EAX instruction also jumps to any location within the current code segment; the difference is that in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed.

Example 6–4 shows how the JMP AX instruction accesses a jump table in the code segment. This program reads a key from the keyboard and then modifies the ASCII code to a 00H in AL for a '1', a 01H for a '2', and a 02H for a '3'. If a '1', '2', or '3' is typed, AH is cleared to 00H. Because the jump table contains 16-bit offset addresses, the contents of AX are doubled to 0, 2, or 4, so a 16-bit entry in the table can be accessed. Next, the offset address of the start of the jump table is loaded to SI, and AX is added to form the reference to the jump address. The MOV AX,[SI] instruction then fetches an address from the jump table, so the JMP AX instruction jumps to the addresses (ONE, TWO, or THREE) stored in the jump table.

EXAMPLE 6–4

```
                        ;A program that reads 1, 2, or 3 from the keyboard
                        ;if a 1, 2, or 3 is typed, a 1, 2, or 3 is displayed.
                        ;
                        .MODEL SMALL                        ;select SMALL model
0000                    .DATA                               ;start of DATA segment
0000  0030 R            TABLE   DW      ONE                 ;define lookup table
0002  0034 R                    DW      TWO
0004  0038 R                    DW      THREE
0000                    .CODE                               ;start of CODE segment
                        .STARTUP                            ;start of program
0017                    TOP:
0017  B4 01                     MOV     AH,1                ;read key into AL
0019  CD 21                     INT     21H

001B  2C 31                     SUB     AL,31H              ;convert to binary
001D  72 F8                     JB      TOP                 ;if below '1' typed
001F  3C 02                     CMP     AL,2
0021  77 F4                     JA      TOP                 ;if above '3' typed

0023  B4 00                     MOV     AH,0                ;double to 0, 2, or 4
0025  03 C0                     ADD     AX,AX
0027  BE 0000 R                 MOV     SI,OFFSET TABLE     ;address lookup table
002A  03 F0                     ADD     SI,AX               ;form lookup address
002C  8B 04                     MOV     AX,[SI]             ;get ONE, TWO, or THREE
002E  FF E0                     JMP     AX                  ;jump address
0030                    ONE:
0030  B2 31                     MOV     DL,'1'              ;load '1' for display
0032  EB 06                     JMP     BOT                 ;go display '1'
0034                    TWO:
0034  B2 32                     MOV     DL,'2'              ;load '2' for display
0036  EB 02                     JMP     BOT                 ;go display '2'
0038                    THREE:
0038  B2 33                     MOV     DL,'3'              ;load '3' for display
003A                    BOT:
```

```
003A  B4 02              MOV     AH,2            ;display number
003C  CD 21              INT     21H
                  .EXIT                          ;exit to DOS
                  END                            ;end of file
```

Indirect Jumps Using an Index. The jump instruction may also use the [ ] form of addressing to directly access the jump table. The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps. (This type of jump is also known as a **double-indirect** jump if the register jump is called an **indirect jump.**) The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction. Here Example 6–5 repeats Example 6–4 by using the JMP TABLE [SI] instead of JMP AX. This reduces the length of the program.

EXAMPLE 6–5

```
                  .MODEL SMALL                   ;select SMALL model
0000              .DATA                          ;start of DATA segment
0000  002D R      TABLE   DW      ONE            ;lookup table
0002  0031 R              DW      TWO
0004  0035 R              DW      THREE
0000              .CODE                          ;start of CODE segment
                  .STARTUP                       ;start of program
0017              TOP:
0017  B4 01               MOV     AH,1           ;read key to AL
0019  CD 21               INT     21H

001B  2C 31               SUB     AL,31H         ;test for below '1'
001D  72 F8               JB      TOP            ;if below '1'
001F  3C 02               CMP     AL,2
0021  77 F4               JA      TOP            ;if above '3'
0023  B4 00               MOV     AH,0           ;calculate table address
0025  03 C0               ADD     AX,AX
0027  03 F0               ADD     SI,AX
0029  FF A4 0000 R        JMP     TABLE [SI]     ;jump to ONE, TWO, or THREE
002D              ONE:
002D  B2 31               MOV     DL,'1'         ;load DL with '1'
002F  EB 06               JMP     BOT
0031              TWO:
0031  B2 32               MOV     DL,'2'         ;load DL with '2'
0033  EB 02               JMP     BOT
0035              THREE:
0035  B2 33               MOV     DL,'3'         ;load DL with '3'
0037              BOT:
0037  B4 02               MOV     AH,2           ;display ONE, TWO, or THREE
0039  CD 21               INT     21H
                  .EXIT                          ;exit to DOS
                  END                            ;end of file
```

The mechanism used to access the jump table is identical with a normal memory reference. The JMP TABLE [SI] instruction points to a jump address stored at the code segment offset location addressed by SI. It jumps to the address stored in the memory at this location. Both the register and indirect indexed jump instructions usually address a 16-bit offset. This means that both types of jumps are near jumps. If a JMP FAR PTR [SI] or JMP TABLE [SI], with TABLE data defined with the DD directive, appears in a program, the microprocessor assumes that the jump table contains doubleword, 32-bit addresses (IP and CS).

Conditional Jumps and Conditional Sets

Conditional jump instructions are always short jumps in the 8086 through the 80286 microprocessors. This limits the range of the jump to within +127 bytes and –128 bytes from the location following the conditional jump. In the

TABLE 6–1 Conditional jump instructions.

Assembly Language	*Condition Tested*	*Operation*
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry set
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = O	Jump if greater than
JGE	S = O	Jump if greater than or equal
JL	S <> O	Jump if less than
JLE	Z = 1 or S <> O	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow set
JP or JPE	P = 1	Jump if parity set or jump if parity even
JS	S = 1	Jump if sign is set
JCXZ	CX = 0	Jump if CX is zero

80386 and above, conditional jumps are either short or near jumps. This allows these microprocessors to use a conditional jump to any location within the current code segment. Table 6–1 lists all the conditional jump instructions with their test conditions. Note that the Microsoft MASM version 6.X/TASM 5.0 assembler automatically adjusts conditional jumps if the distance is too great.

The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (O). If the condition under test is true, a branch to the label associated with the jump instruction occurs. If the condition is false, the next sequential step in the program executes. For example, a JC will jump if the carry bit is set.

The operation of most conditional jump instructions is straightforward because they often test just one flag bit, although some test more than one. Relative magnitude comparisons require more complicated conditional jump instructions that test more than one flag bit.

Because both signed and unsigned numbers are used in programming, and because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons. Figure 6–5 shows the order of both signed and unsigned 8-bit numbers. The 16- and 32-bit numbers follow the same order as the 8-bit numbers, except that they are larger. Notice that an FFH (255) is above the 00H in the set of unsigned numbers, but an FFH (–1) is less than 00H for signed numbers. Therefore, an unsigned FFH is above 00H, but a signed FFH is less than 00H.

When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions. The terms *greater than* and *less than* refer to signed numbers. When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions. The terms *above* and *below* refer to unsigned numbers.

The remaining conditional jumps test individual flag bits, such as overflow and parity. Notice that JE has an alternative opcode JZ. All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test. (The alternates appear in Appendix B with the instruction set listing.) For example, the JA instruction (jump if above) has the alternative JNBE (jump if not below or equal). A JA functions exactly as a JNBE, but a JNBE is awkward in many cases when compared to a JA.

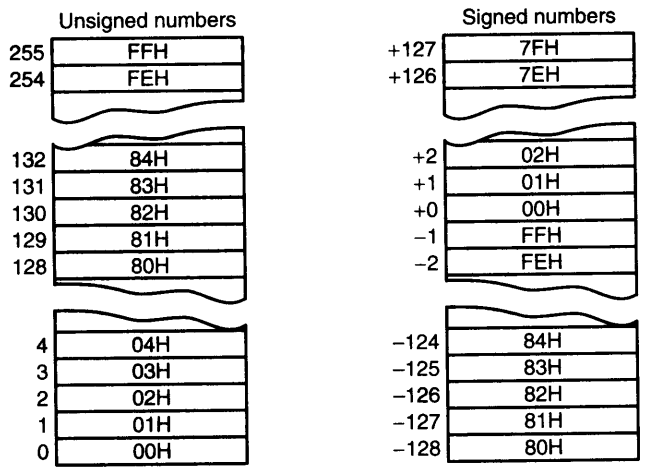

FIGURE 6–5 Signed and unsigned numbers follow different orders.

The conditional jump instructions all test flag bits except for JCXZ (jump if CX = 0). Instead of testing flag bits, JCXZ directly tests the contents of the CX register without affecting the flag bits. For the JCXZ instruction, if CX = 0, a jump occurs, and if CX <> 0, no jump occurs. Likewise for the CX <> 0, no jump occurs.

A program that uses JCXZ appears in Example 6–6. Here, the SCASB instruction searches a table for a 0AH. Following the search, a JCXZ instruction tests CX to see if the count has reached zero. If the count is zero, the 0AH is not found in the table. The carry flag is used in this example to pass the not found condition back to the calling program. Another method used to test to see if the data are found is the JNE instruction. If JNE replaces JCXZ, it performs the same function. After the SCASB instruction executes, the flags indicate a not-equal condition if the data were not found in the table.

EXAMPLE 6–6

```
                ;A procedure that searches a table of 100 bytes for 0AH.
                ;The address, TABLE, is transferred to the procedure
                ;through the SI register.
                ;
0017            SCAN    PROC    NEAR

0017  B9 0064           MOV     CX,100          ;load count of 100
001A  B0 0A             MOV     AL,0AH          ;load AL with 0AH
001C  FC                CLD                     ;select increment
001D  F2/AE             REPNE   SCASB           ;test 100 bytes for 0AH
001F  F9                STC                     ;set carry for not found
0020  E3 01             JCXZ    NOT_FOUND       ;if not found
0022  F8                CLC                     ;clear carry if found

0023            NOT_FOUND:

0023  C3                RET                     ;return from procedure

0024            SCAN    ENDP
```

LOOP

The LOOP instruction is a combination of a decrement CX and the JNZ conditional jump. In the 8086 through the 80286 processors, LOOP decrements CX; if CX <> 0, it jumps to the address indicated by the label. If CX becomes a 0, the next sequential instruction executes.

Example 6–7 shows how data in one block of memory (BLOCK1) adds to data in a second block of memory (BLOCK2), using LOOP to control how many numbers add. The LODSW and STOSW instructions access the data in BLOCK1 and BLOCK2. The ADD AX,ES:[DI] instruction accesses the data in BLOCK2 located in the extra segment. The only reason that BLOCK2 is in the extra segment is that DI addresses extra segment data for the STOSW instruction. The .STARTUP directive only loads DS with the address of the data segment. In this example, the extra segment also addresses data in the data segment, so the contents of DS are copied to ES through the accumulator. Unfortunately, there is no direct move from segment register-to-segment register instruction.

EXAMPLE 6–7

```
                        ;A program that sums the contents of BLOCK1 and BLOCK2
                        ;and stores the results over top of the data in BLOCK2.
                        ;
                        .MODEL SMALL                        ;select SMALL model
0000                    .DATA                               ;start of DATA segment
0000  0064 [            BLOCK1 DW     100 DUP (?)           ;100 bytes for BLOCK1
           0000
               ]
00C8  0064 [            BLOCK2 DW     100 DUP (?)           ;100 bytes for BLOCK2
           0000
               ]
0000                    .CODE                               ;start of CODE segment
                        .STARTUP                            ;start of program
0017  8C D8                   MOV     AX,DS                 ;overlap DS and ES
0019  8E C0                   MOV     ES,AX

001B  FC                      CLD                           ;select increment
001C  B9 0064                 MOV     CX,100                ;load count of 100
001F  BE 0000 R               MOV     SI,OFFSET BLOCK1      ;address BLOCK1
0022  BF 00C8 R               MOV     DI,OFFSET BLOCK2      ;address BLOCK2

0025                    L1:
0025  AD                      LODSW                         ;load AX with BLOCK1
0026  26:03 05                ADD     AX,ES:[DI]            ;add BLOCK2 data to AX
0029  AB                      STOSW                         ;store sum in BLOCK2
002A  E2 F9                   LOOP    L1                    ;repeat 100 times
                        .EXIT                               ;exit to DOS
                        END                                 ;end of file
```

Conditional LOOPs. As with REP, the LOOP instruction also has conditional forms: LOOPE and LOOPNE. The LOOPE **(loop while equal)** instruction jumps if CX <> 0 while an equal condition exists. It will exit the loop if the condition is not equal or if the CX register decrements to 0. The LOOPNE **(loop while not equal)** instruction jumps if CX <> 0 while a not-equal condition exists. It will exit the loop if the condition is equal or if the CX register decrements to 0.

As with the conditional repeat instructions, alternates exist for LOOPE and LOOPNE. The LOOPE instruction is the same as LOOPZ, and the LOOPNE instruction is the same as LOOPNZ. In most programs, only the LOOPE and LOOPNE apply.

6–2 PROCEDURES

The procedure or subroutine is an important part of any computer system's architecture. A **procedure** is a group of instructions that usually performs one task. A procedure is a reusable section of the software that is stored in memory once, but used as often as necessary. This saves memory space and makes it easier to develop software. The only disadvantage of a procedure is that it takes the computer a small amount of time to link to the procedure

and return from it. The CALL instruction links to the procedure, and the RET (**return**) instruction returns from the procedure.

The stack stores the return address whenever a procedure is called during the execution of a program. The CALL instruction pushes the address of the instruction following the CALL (**return address**) on the stack. The RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

With the assembler, there are specific rules for storing procedures. A procedure begins with the PROC directive and ends with the ENDP directive. Each directive appears with the name of the procedure. This programming structure makes it easy to locate the procedure in a program listing. The PROC directive is followed by the type of procedure: NEAR or FAR. Example 6–8 shows how the assembler uses the definition of both a near (intrasegment) and far (intersegment) procedure. In MASM version 6.X, the NEAR or FAR type can be followed by the USES statement. The USES statement allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure. The USES statement is also illustrated in Example 6–8.

EXAMPLE 6–8

```
0000                SUMS    PROC NEAR

0000  03 C3                 ADD  AX,BX
0002  03 C1                 ADD  AX,CX
0004  03 C2                 ADD  AX,DX
0006  C3                    RET

0007                SUMS    ENDP

0007                SUMS1   PROC FAR

0007  03 C3                 ADD  AX,BX
0009  03 C1                 ADD  AX,CX
000B  03 C2                 ADD  AX,DX
000D  CB                    RET

000E                SUMS1   ENDP

000E                SUMS2   PROC NEAR  USES BX CX DX

0011  03 C3                 ADD  AX,BX
0013  03 C1                 ADD  AX,CX
0015  03 C2                 MOV  AX,DX
                            RET

001B                SUMS2   ENDP
```

When these two procedures are compared, the only difference is the opcode of the return instruction. The near return instruction uses opcode C3H and the far return uses opcode CBH. A near return removes a 16-bit number from the stack and places it into the instruction pointer to return from the procedure in the current code segment. A far return removes a 32-bit number from the stack and places it into both IP and CS to return from the procedure to any memory location.

Procedures that are to be used by all software (**global**) should be written as far procedures. Procedures that are used by a given task (**local**) are normally defined as near procedures.

CALL

The CALL instruction transfers the flow of the program to the procedure. The CALL instruction differs from the jump instruction because a CALL saves a return address on the stack. The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

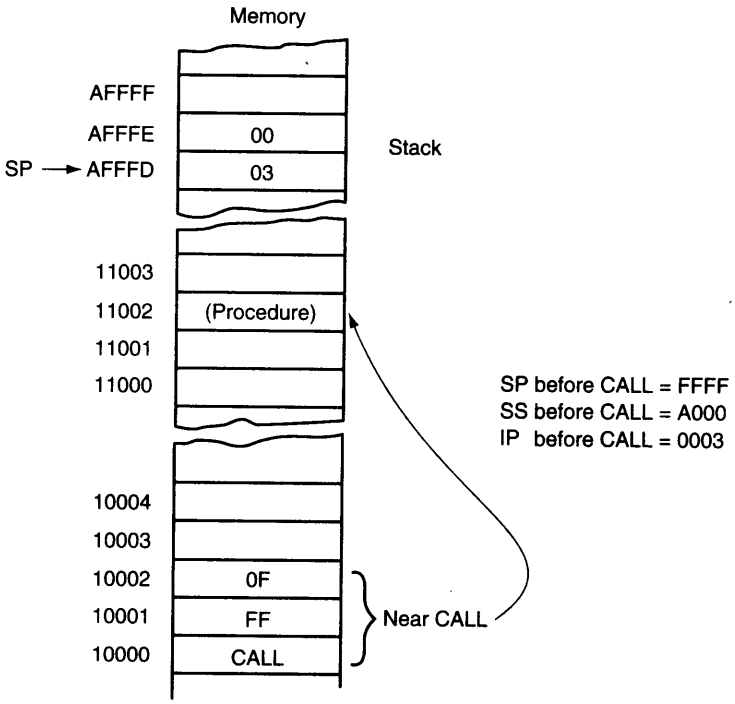

FIGURE 6–6 The effect of a near CALL on the stack and the instruction pointer.

Near CALL. The near CALL instruction is three bytes long; the first byte contains the opcode, and the second and third bytes contain the displacement, or distance of ±32K in the 8086. This is identical to the form of the near jump instruction. When the near CALL executes, it first pushes the offset address of the next instruction on the stack. The offset address of the next instruction appears in the instruction pointer (IP). After saving this return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure. There is no short CALL instruction.

Why save the IP on the stack? The instruction pointer always points to the next instruction in the program. For the CALL instruction, the contents of IP are pushed onto the stack, so program control passes to the instruction following the CALL after a procedure ends. Figure 6–6 shows the return address (IP) stored on the stack and the call to the procedure.

Far CALL. The far CALL instruction is like a far jump because it can call a procedure stored in any memory location in the system. The far CALL is a five-byte instruction that contains an opcode, followed by the next value for the IP and CS registers. Bytes 2 and 3 contain the new contents of the IP, and bytes 4 and 5 contain the new contents for CS.

The far CALL instruction places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2–5 of the instruction. This allows the far CALL to call a procedure located anywhere in the memory and return from that procedure.

Figure 6–7 shows how the far CALL instruction calls a far procedure. Here, the contents of IP and CS are pushed onto the stack. Next, the program branches to the procedure. A variant of the far call exists as CALLF, but this should be avoided in favor of defining the type of call instruction with the PROC statement.

CALLs with Register Operands. Like jump instructions, call instructions also may contain a register operand. An example is the CALL BX instruction, which pushes the contents of IP onto the stack. It then jumps to the offset address, lo-

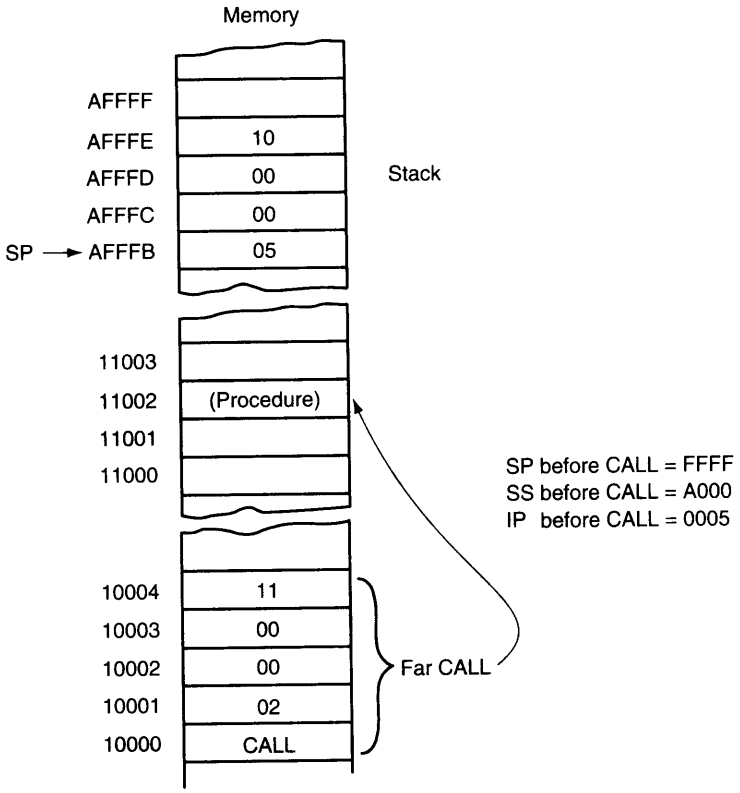

FIGURE 6–7 The effect of a far CALL instruction.

cated in register BX, in the current code segment. This type of CALL always uses a 16-bit offset address, stored in any 16-bit register except the segment registers.

Example 6–9 illustrates the use of the CALL register instruction to call a procedure that begins at offset address DISP. (This call could also directly call the procedure by using the CALL DISP instruction.) The OFFSET address DISP is placed into the BX register, and then the CALL BX instruction calls the procedure beginning at address DISP. This program displays an "OK" on the monitor screen.

EXAMPLE 6–9

```
                    ;A program that displays OK on the monitor screen
                    ;using procedure DISP.
                    ;
                    .MODEL TINY                     ;select TINY model
0000                .CODE                           ;start of CODE segment
                    .STARTUP                        ;start of program

0100  BB 0110 R             MOV  BX,OFFSET DISP     ;address DISP with BX
0103  B2 4F                 MOV  DL,'O'             ;display 'O'
0105  FF D3                 CALL BX
0107  B2 4B                 MOV  DL,'K'             ;display 'K'
0109  FF D3                 CALL BX

                    .EXIT                           ;exit to DOS
                    ;A procedure that displays the ASCII contents of DL on
                    ;the monitor screen.
                    ;
```

```
0110                DISP    PROC NEAR

0110  B4 02                  MOV  AH,2              ;select function 02H
0112  CD 21                  INT  21H               ;execute DOS function
0114  C3                    RET                    ;return from procedure

0115                DISP    ENDP

                    END                            ;end of file
```

CALLs with Indirect Memory Addresses. A CALL with an indirect memory address is particularly useful whenever different subroutines need to be chosen in a program. This selection process is often keyed with a number that addresses a CALL address in a lookup table.

Example 6–10 shows three separate subroutines referenced by the number 1, 2, and 3 as read from the keyboard on the personal computer. The calling sequence adjusts the value of AL and extends it to a 16-bit number before adding it to the location of the lookup table. This references one of the three subroutines using the CALL TABLE [BX] instruction. When this program executes, the letter A is displayed when a 1 is typed, the letter B is displayed when a 2 is typed, and the letter C is displayed when a 3 is typed.

EXAMPLE 6–10

```
                    ;A program that uses a CALL lookup table to access one of
                    ;three different procedures: ONE, TWO, or THREE.
                    ;
                    .MODEL SMALL                   ;select SMALL model
0000                .DATA                          ;start of DATA segment
0000  0000 R        TABLE   DW   ONE               ;define lookup table
0002  0007 R                DW   TWO
0004  000E R                DW   THREE
0000                .CODE                          ;start of CODE segment

0000                ONE     PROC NEAR

0000  B4 02                 MOV  AH,2              ;display a letter A
0002  B2 41                 MOV  DL,'A'
0004  CD 21                 INT  21H
0006  C3                   RET

0007                ONE     ENDP

0007                TWO     PROC NEAR

0007  B4 02                MOV  AH,2              ;display letter B
0009  B2 42                MOV  DL,'B'
000B  CD 21                INT  21H
000D  C3                   RET

000E                TWO     ENDP

000E                THREE   PROC NEAR

000E  B4 02                MOV  AH,2              ;display letter C
0010  B2 43                MOV  DL,'C'
0012  CD 21                INT  21H
0014  C3                   RET

0015                THREE   ENDP

                    .STARTUP                       ;indicate start of program
002C                TOP:
002C  B4 01                MOV  AH,1              ;read key into AL
002E  CD 21                INT  21H
```

```
0030  2C 31                  SUB  AL,31H        ;convert to binary
0032  72 F8                  JB   TOP           ;if below 0
0034  3C 02                  CMP  AL,2
0036  77 F4                  JA   TOP           ;if above 2

0038  B4 00                  MOV  AH,0          ;form lookup address
003A  8B D8                  MOV  BX,AX
003C  03 DB                  ADD  BX,BX
003E  FF 97 0000 R           CALL TABLE [BX]    ;call procedure

                    .EXIT                       ;exit to DOS
                    END                         ;end of file
```

The CALL instruction also can reference far pointers if the instruction appears as a CALL FAR PTR [SI] or as a CALL TABLE [SI], if the data in the table are defined as doubleword data with the DD directive. These instructions retrieve a 32-bit address from the data segment memory location addressed by SI and use it as the address of a far procedure.

RET

The return instruction (RET) removes a 16-bit number **(near return)** from the stack and places it into IP, or removes a 32-bit number **(far return)** and places it into IP and CS. The near and far return instructions are both defined in the procedure's PROC directive, which automatically selects the proper return instruction.

When IP or IP and CS are changed, the address of the next instruction is at a new memory location. This new location is the address of the instruction that immediately follows the most recent CALL to a procedure. Figure 6–8 shows how the CALL instruction links to a procedure and how the RET instruction returns in the 8086.

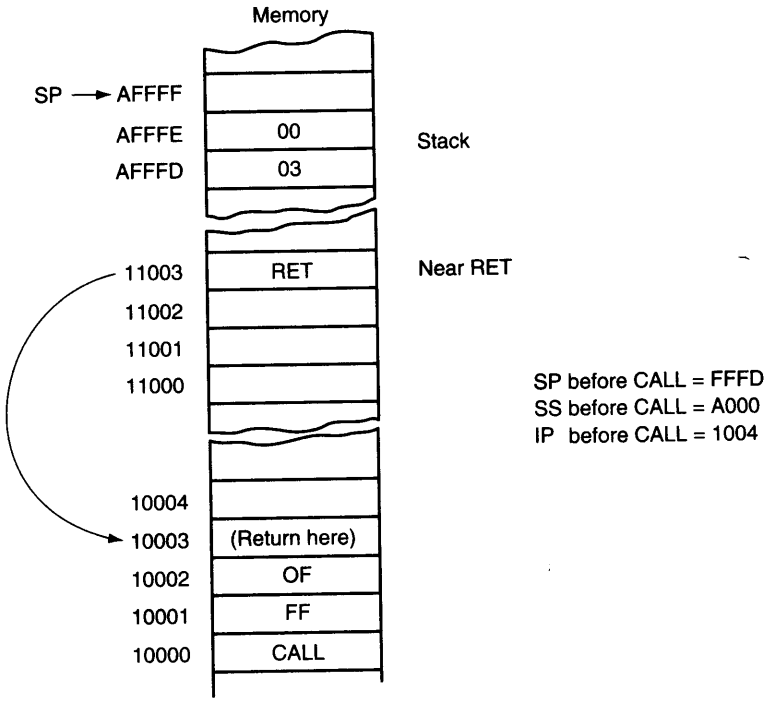

FIGURE 6–8 The effect of a near return instruction on the stack and instruction pointer.

There is one other form of the return instruction, which adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack. A return that uses an immediate operand is ideal for use in a system that uses the C or Pascal calling conventions. (This is true, even though the C and PASCAL calling conventions require the caller to remove stack data for many functions.) These conventions push parameters on the stack before calling a procedure. If the parameters are to be discarded upon return, the return instruction contains a number that represents the number of bytes pushed to the stack as parameters.

Example 6–11 shows how this type of return erases the data placed on the stack by a few pushes. The RET four adds a 4 to SP after removing the return address from the stack. Because the PUSH AX and PUSH BX together place four bytes of data on the stack, this return effectively deletes AX and BX from the stack. This type of return rarely appears in assembly language programs, but it is used in high-level programs to clear stack data after a procedure. Notice how parameters are addressed on the stack by using the BP register, which by default addresses the stack segment. Parameter stacking is common in procedures written for C or PASCAL by using the C or PASCAL calling conventions.

EXAMPLE 6–11

```
0000  B8 001E             MOV  AX,30
0003  BB 0028             MOV  BX,40
0006  50                  PUSH AX              ;stack parameter 1
0007  53                  PUSH BX              ;stack parameter 2

0008  E8 0066             CALL ADDM            ;add parameters from stack
                           .    .
                           .    .              ;program continues here
0071              ADDM    PROC NEAR


0071  55                  PUSH BP              ;save BP
0072  8B EC               MOV  BP,SP           ;address stack with BP
0074  8B 46 04            MOV  AX,[BP+4]       ;get parameter 1
0077  03 46 06            ADD  AX,[BP+6]       ;add parameter 2
007A  5D                  POP  BP              ;restore BP
007B  C2 0004             RET  4               ;return, dump parameters


007E              ADDM    ENDP
```

As with the CALLN and CALLF instructions, there are also variants of the return instruction: RETN and RETF. As with the CALLN and CALLF instructions, these variants should also be avoided in favor of using the PROC statement to define the type of call and return.

6–3 INTRODUCTION TO INTERRUPTS

An interrupt is either a hardware-generated CALL (externally derived from a hardware signal) or a software-generated CALL (internally derived from the execution of an instruction or by some other internal event). At times, an internal interrupt is called an *exception.* Either type interrupts the program by calling an interrupt service procedure or interrupt handler.

This section explains software interrupts, which are special types of CALL instructions. This section descibes the three types of software interrupt instructions (INT, INTO, and INT 3), provides a map of the interrupt vectors, and explains the purpose of the special interrupt return instruction (IRET).

TABLE 6–2 Interrupt vectors.

Number	*Address*	*Microprocessor*	*Function*
0	0H–3H	All	Divide error
1	4H–7H	All	Aingle-step
2	8H–BH	All	NMI pin
3	CH–FH	All	Breakpoint
4	10H–13H	All	Interrupt on overflow
5	14H–17H	80186–Pentium 4	Bound instruction
6	18H–1BH	80186–Pentium 4	Invalid opcode
7	1CH–1FH	80186–Pentium 4	Coprocessor emulation
8	20H–23H	80386–Pentium 4	Double fault
9	24H–27H	80386	Coprocessor segment overrun
A	28H–2BH	80386–Pentium 4	Invalid task state segment
B	2CH–2FH	80386–Pentium 4	Segment not present
C	30H–33H	80386–Pentium 4	Stack fault
D	34H–37H	80386–Pentium 4	General protection fault (GPF)
E	38H–3BH	80386–Pentium 4	Page fault
F	3CH–3FH	—	Reserved
10	40H–43H	80286–Pentium 4	Floating-point error
11	44H–47H	80486SX	Alignment check interrupt
12	48H–4FH	Pentium/Pentium 4	Machine check exception
13–1F	50H–7FH	—	Reserved
20–FF	80H–3FFH	—	User interrupts

Interrupt Vectors

An **interrupt vector** is a four-byte number stored in the first 1024 bytes of the memory (000000H–0003FFH) when the microprocessor operates in the real mode. In the protected mode, the vector table is replaced by an interrupt descriptor table that uses eight-byte descriptors to describe each of the interrupts. There are 256 different interrupt vectors, and each vector contains the address of an interrupt service procedure. Table 6–2 lists the interrupt vectors, with a brief description and the memory location of each vector for the real mode. Each vector contains a value for IP and CS that forms the address of the interrupt service procedure. The first two bytes contain the IP, and the last two bytes contain the CS.

Intel reserves the first 32 interrupt vectors for the present and future microprocessor products. The remaining interrupt vectors (32–255) are available for the user. Some of the reserved vectors are for errors that occur during the execution of software, such as the divide error interrupt. Some vectors are reserved for the coprocessor. Still others occur for normal events in the system. In a personal computer, the reserved vectors are used for system functions, as detailed later in this section. Vectors 1–6, 7, 9, 16, and 17 function in the real mode and protected mode; the remaining vectors function only in the protected mode.

Interrupt Instructions

The microprocessor has three different interrupt instructions that are available to the programmer: INT, INTO, and INT 3. In the real mode, each of these instructions fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector. In the protected mode, each of these instructions fetches an interrupt descriptor from the interrupt descriptor table. The descriptor specifies the address of the interrupt service procedure. The interrupt call is similar to a far CALL instruction because it places the return address (IP/EIP and CS) on the stack.

INTs. There are 256 different software interrupt instructions (INTs) available to the programmer. Each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH). For example, the INT 100 uses interrupt vector 100, which appears at memory address 190H–193H. The address of the interrupt vector is determined by multiplying the interrupt type number times 4. For example, the INT 10H instruction calls the interrupt service procedure whose address is stored beginning at memory location 40H (10H × 4) in the real mode. In the protected mode, the interrupt descriptor is located by multiplying the type number by 8 instead of 4 because each descriptor is eight bytes long.

Each INT instruction is two bytes long. The first byte contains the opcode, and the second byte contains the vector type number. The only exception to this is INT 3, a one-byte special software interrupt used for breakpoints.

Whenever a software interrupt instruction executes, it (1) pushes the flags onto the stack, (2) clears the T and I flag bits, (3) pushes CS onto the stack, (4) fetches the new value for CS from the interrupt vector, (5) pushes IP onto the stack, (6) fetches the new value for IP from the vector, and (7) jumps to the new location addressed by CS and IP

The INT instruction performs as a far CALL except that it not only pushes CS and IP onto the stack, but it also pushes the flags onto the stack. The INT instruction performs the operation of a PUSHF, followed by a far CALL instruction.

Notice that when the INT instruction executes, it clears the interrupt flag (I), which controls the external hardware interrupt input pin INTR (interrupt request). When I = 0, the microprocessor disables the INTR pin; when I = 1, the microprocessor enables the INTR pin.

Software interrupts are most commonly used to call system procedures because the address of the system function need not be known. The system procedures are common to all system and application software. The interrupts often control printers, video displays, and disk drives. Besides relieving the program from remembering the address of the system call, the INT instruction replaces a far CALL that would otherwise be used to call a system function. The INT instruction is two bytes long whereas the far CALL is five bytes long. Each time that the INT instruction replaces a far CALL, it saves three bytes of memory in a program. This can amount to a sizable saving if the INT instruction often appears in a program, as it does for system calls.

IRET. The interrupt return instruction (IRET) is used only with software or hardware interrupt service procedures. Unlike a simple return instruction (RET), the IRET instruction will (1) pop stack data back into the IP, (2) pop stack data back into CS, and (3) pop stack data back into the flag register. The IRET instruction accomplishes the same tasks as the POPF, followed by a far RET instruction.

Whenever an IRET instruction executes, it restores the contents of I and T from the stack. This is important because it preserves the state of these flag bits. If interrupts were enabled before an interrupt service procedure, they are automatically re-enabled by the IRET instruction because it restores the flag register.

INT 3. An INT 3 instruction is a special software interrupt designed to function as a breakpoint. The difference between it and the other software interrupts is that INT 3 is a one-byte instruction, while the others are two-byte instructions.

It is common to insert an INT 3 instruction in software to interrupt or break the flow of the software. This function is called a **breakpoint.** A breakpoint occurs for any software interrupt, but because INT 3 is one byte long, it is easier to use for this function. Breakpoints help to debug faulty software.

INTO. Interrupt on overflow (INTO) is a conditional software interrupt that tests the overflow flag (O). If O = 0, the INTO instruction performs no operation; if O = 1 and an INTO instruction executes, an interrupt occurs via vector type number 4.

The INTO instruction appears in software that adds or subtracts signed binary numbers. With these operations, it is possible to have an overflow. Either the JO instruction or INTO instruction detects the overflow condition.

An Interrupt Service Procedure. Suppose that, in a particular system, a procedure is required to add the contents of DI, SI, BP, and BX and then save the sum in AX. Because this is a common task in this system, it is may occasionally be worthwhile to develop the task as a software interrupt. Realize that interrupts are usually reserved for system events and this is merely an example showing how an interrupt service procedure appears. Example 6–12 shows this software interrupt. The main difference between this procedure and a normal far procedure is that it ends with the IRET instruction instead of the RET instruction, and the contents of the flag register are saved on the stack during its execution.

EXAMPLE 6–12

```
0000                    INTS    PROC FAR

0000  03 C3                     ADD  AX,BX
0002  03 C5                     ADD  AX,BP
0004  03 C7                     ADD  AX,DI
0006  03 C6                     ADD  AX,SI
0008  CF                        IRET

0009                    INTS    ENDP
```

Interrupt Control

Although this section does not explain hardware interrupts, two instructions are introduced that control the INTR pin. The set interrupt flag instruction (STI) places a 1 into the I flag bit, which enables the INTR pin. The clear interrupt flag instruction (CLI) places a 0 into the I flag bit, which disables the INTR pin. The STI instruction enables INTR and the CLI instruction disables INTR. In a software interrupt service procedure, hardware interrupts are enabled as one of the first steps. This is accomplished by the STI instruction. The reason interrupts are enabled early in an interrupt service procedure is that just about all of the I/O devices in the personal computer are interrupt-processed. If the interrupts are disabled too long, severe system problems result.

Interrupts in the Personal Computer

The interrupts found in the personal computer differ somewhat from the ones presented in Table 6–2. The reason that they differ is that the original personal computers are 8086/8088-based systems. This meant that they only contained Intel-specified interrupts 0–4. This design is carried forward so that newer systems are compatible with the early personal computers.

Because the personal computer is operated in the real mode, the interrupt vector table is located at addresses 00000H–003FFH. The assignments used by computer system are listed in Table 6–3. Notice that these differ somewhat from the assignments in Table 6–2. Some of the interrupts shown in this table are used in example programs in later chapters. An example is the clock tick, which is extremely useful for timing events because it occurs 18.2 times per second in all personal computers.

Interrupts 00H–1FH and 70H–77H are present in the computer, no matter what operating system is installed. If DOS is installed, interrupts 20H–2FH are also present. The BIOS uses interrupts 11H through 1FH, the video BIOS uses INT 10H, and the hardware in the system uses interrupts 00H through 0FH and 70H through 77H.

6–4 MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

The last category of real mode instructions found in the microprocessor are the machine control and miscellaneous group. These instructions provide control of the carry bit, sample the TEST pin, and perform various other functions. Because many of these instructions are used in hardware control, they need only be explained briefly at this point.

Controlling the Carry Flag Bit

The carry flag (C) propagates the carry or borrow in multiple-word/double word addition and subtraction. It also indicates errors in procedures. There are three instructions that control the contents of the carry flag: STC (set carry), CLC (clear carry), and CMC (complement carry).

Because the carry flag is seldom used, except with multiple word addition and subtraction, it is available for other uses. The most common task for the carry flag is to indicate an error upon return from a procedure. Suppose that

TABLE 6–3 The hexadecimal interrupt assignments for the personal computer.

Number	*Function*
0	Divide error
1	Single-step
2	NMI pin (often parity error checks)
3	Breakpoint
4	Overflows
5	Print screen key and BOUND instruction
6	Illegal instruction
7	Coprocessor emulation
8	Clock tick (18.2 Hz)
9	Keyboard
A	IRQ2 (cascade in AT system)
B–F	IRQ3–IRQ7
10	Video BIOS
11	Equipment environment
12	Conventional memory size
13	Direct disk services
14	Serial COM port service
15	Miscellaneous
16	Keyboard service
17	Parallel port (LPT) service
18	ROM BASIC
19	Reboot
1A	Clock service
1B	Control-break handler
1C	User timer service
1D	Pointer for video parameter table
1E	Pointer for disk parameter table
1F	Pointer for graphic character pattern table
20	Terminate program (DOS 1.0)
21	DOS services
22	Program termination handler
23	Control-C handler
24	Critical error handler
25	Read disk
26	Write disk
27	Terminate and stay resident (TSR)
28	DOS idle
2F	Multiplex handler
31	DPMI (DOS protected mode interface) provided by Windows
33	Mouse driver
67	VCPI (virtual control program interface) provided by HIMEM.SYS
70–77	IRQ8–IRQ15

a procedure reads data from a disk memory file. This operation can be successful, or an error such as file-not-found can occur. Upon return from this procedure, if C = 1, an error has occurred; if C = 0, no error occurred. Most of the DOS and BIOS procedures use the carry flag to indicate error conditions.

21. The LOCK prefix causes the LOCK pin to become a logic 0 for the duration of the locked instruction. The ESC instruction passes instruction to the numeric coprocessor.

6–6 QUESTIONS AND PROBLEMS

1. What is a short JMP?
2. Which type of JMP is used when jumping to any location within the current code segment?
3. Which JMP instruction allows the program to continue execution at any memory location in the system?
4. Which JMP instruction is five bytes long?
5. What is the range of a near jump in the 80386–Pentium 4 microprocessors?
6. Which type of JMP instruction (short, near, or far) assembles for the following:
   (a) if the distance is 0210H bytes
   (b) if the distance is 0020H bytes
   (c) if the distance is 10000H bytes
7. What can be said about a label that is followed by a colon?
8. The near jump modifies the program address by changing which register or registers?
9. The far jump modifies the program address by changing which register or registers?
10. Explain what the JMP AX instruction accomplishes. Also identify it as a near or a far jump instruction.
11. Contrast the operation of a JMP DI with a JMP [DI].
12. Contrast the operation of a JMP [DI] with a JMP FAR PTR [DI].
13. List the five flag bits tested by the conditional jump instructions.
14. Describe how the JA instruction operates.
15. When will the JO instruction jump?
16. Which conditional jump instructions follow the comparison of signed numbers?
17. Which conditional jump instructions follow the comparison of unsigned numbers?
18. Which conditional jump instructions test both the Z and C flag bits?
19. When does the JCXZ instruction jump?
20. The 8086 LOOP instruction decrements register _____ and tests it for a 0 to decide if a jump occurs.
21. Explain how the LOOPE instruction operates.
22. Develop a short sequence of instructions that stores a 00H into 150H bytes of memory, beginning at extra segment memory location DATA. You must use the LOOP instruction to help perform this task.
23. Develop a sequence of instructions that searches through a block of 100H bytes of memory. This program must count all the unsigned numbers that are above 42H and all that are below 42H. Byte-sized data segment memory location UP must contain the count of numbers above 42H, and data segment location DOWN must contain the count of numbers below 42H.
24. What is a procedure?
25. Explain how the near and far CALL instructions function.
26. How does the near RET instruction function?
27. The last executable instruction in a procedure must be a(n) _____.
28. Which directive identifies the start of a procedure?
29. How is a procedure identified as near or far?
30. Explain what the RET 6 instruction accomplishes.
31. Write a near procedure that cubes the contents of the CX register. This procedure may not affect any register except CX.

32. Write a procedure that multiplies DI by SI and then divides the result by 100H. Make sure that the result is left in AX upon returning from the procedure. This procedure may not change any register except AX.
33. What is an interrupt?
34. Which software instructions call an interrupt service procedure?
35. How many different interrupt types are available in the microprocessor?
36. What is the purpose of interrupt vector type number 0?
37. Illustrate the contents of an interrupt vector and explain the purpose of each part.
38. How does the IRET instruction differ from the RET instruction?
39. What is the IRETD instruction?
40. The INTO instruction only interrupts the program for what condition?
41. The interrupt vector for an INT 40H instruction is stored at which memory locations?
42. What instructions control the function of the INTR pin?
43. Which personal computer interrupt services the parallel LPT port?
44. Which personal computer interrupt services the keyboard?
45. What instruction tests the TEST pin?

CHAPTER 7

Programming the Microprocessor

INTRODUCTION

This chapter develops programs and programming techniques using the MASM macro assembler program, the DOS function calls, and the BIOS function calls. Many of the DOS function calls and BIOS function calls are used in this chapter, but all are explained in complete detail in Appendix A. Please scan the function calls listed in Appendix A as you read this chapter. The MASM assembler has already been explained and demonstrated in prior chapters, but there are still more features to learn at this point.

Some programming techniques explained in this chapter include macro sequences, keyboard and display manipulation, program modules, library files, using the mouse, interrupt hooks, and other important programming techniques. This chapter is meant as an introduction to programming, yet it provides valuable programming techniques that provide a wealth of background so that programs can be easily developed for the personal computer by using MSDOS as a springboard.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use the MASM assembler and linker program to create programs that contain more than one module.
2. Explain the use of EXTRN and PUBLIC as they apply to modular programming.
3. Set up a library file that contains commonly used subroutines.
4. Write and use MACRO and ENDM to develop macro sequences used with linear programming.
5. Develop programs using DOS function calls.
6. Differentiate a DOS function call from a BIOS function call.
7. Show how to hook into interrupts using DOS function calls.
8. Use conditional assembly language statements in programs.

7-1 MODULAR PROGRAMMING

Many programs are too large to be developed by one person. This means that programs are routinely developed by teams of programmers. The linker program is provided with MSDOS so that programming modules can be linked together into a complete program. Linking is also an internal function of the Programmer's WorkBench program that is bundled with MASM version 6.X. This section of the text describes the linker, the linking task, library files, EXTRN, and PUBLIC as they apply to program modules and modular programming. It also introduces the use of Programmer's WorkBench, which is also used to manage programs generated by teams.

The Assembler and Linker

The **assembler program** converts a symbolic **source module** (file) into a hexadecimal **object file.** We have seen many examples of symbolic source files, written in assembly language, in prior chapters. Example 7–1 shows how the assembler dialog that appears as a source module named NEW.ASM is assembled. Note that this dialog is used with version 6.11 at the DOS command line. This assembler also uses the Programmer's WorkBench program for development, without resorting to the DOS command line. Whenever you create a source file, it should have an extension of ASM. Source files are created by using WorkBench, an editor that comes with the assembler, or by almost any other word processor or editor capable of generating an ASCII file.

EXAMPLE 7–1

```
C:\MASM611\FILES>ml /Flnew.lst new.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: new.asm

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

Object Modules [.obj]: new.obj
Run File [new.exe]: "new.exe"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:

C:\MASM611\FILES>
```

The assembler program (ML) requires the source file name following ML. In the example, the /Fl switch is used to create a listing file named NEW.LST. Although this is optional, it is recommended so the output of the assembler can be viewed for troubleshooting problems. The source listing file (.LST) contains the assembled version of the source file and its hexadecimal machine language equivalent. The cross-reference file (.CRF), which is not generated in this example, lists all labels and pertinent information required for cross-referencing.

The **linker program,** which executes as the second part of ML, reads the object files that are created by the assembler program and links them together into a single execution file. An **execution file** is created with the file name extension EXE. Execution files are selected by typing the file name at the DOS prompt (A:\). An example execution file is FROG.EXE, which is executed by typing FROG at the DOS command prompt.

If a file is short enough (less than 64K bytes long) it can be converted from an execution file to a **command file** (.COM). The command file is slightly different from an execution file in that the program must be originated at location 100H before it can execute. This means that the program must be no larger than 64K–100H in length. The ML program generates a command file if the tiny model is used with a starting address of 100H. Note that Programmer's WorkBench can also be configured to generate a command file. The main advantage of a command file is that it loads off the disk into the computer much more quickly than an execution file. It also requires less disk storage space than the equivalent execution file.

Example 7–2 shows the linker program protocol when it is used to link the files NEW, WHAT, and DONUT. The linker also links library files (LIBS) so procedures, located within LIBS, can be used with the linked execution file. To invoke the linker, type LINK at the DOS command prompt, as illustrated in Example 7–2. Note that before files are linked, they must first be assembled and they must be **error-free.** ML not only links the files, but it also assembles them prior to linking.

EXAMPLE 7–2

```
C:\MASM611\FILES>ml new.asm what.asm donut.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: new.asm
 Assembling: what.asm
 Assembling: donut.asm

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

Object Modules [.obj]: new.obj+
Object Modules [.obj]: "what.obj"+
Object Modules [.obj]: "donut.obj"
Run File [new.exe]: "new.exe"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:

C:\MASM611\FILES>
```

In this example, after typing ML, the linker program asks for the "Object Modules," which are created by the assembler. In this example, we have three object modules: NEW, WHAT, and DONUT. If more than one object file exists, the main program file (NEW, in this example) is typed first, followed by any other supporting modules.

Library files are entered after the file name and after the switch /LINK. In this example, we did not enter a library file name. To use a library called NUMB.LIB while assembling a program called NEW.ASM, type ML NEW.ASM /LINK NUMB.LIB.

PUBLIC and EXTRN

The PUBLIC and EXTRN directives are very important to modular programming. We use PUBLIC to declare that labels of code, data, or entire segments are available to other program modules. EXTRN (external) declares that labels are external to a module. Without these statements, modules could not be linked together to create a program by using modular programming techniques. They might link, but one module would not be able to communicate to another.

The PUBLIC directive is placed in the opcode field of an assembly language statement to define a label as public, so that the label can be used by other modules. The label declared as public can be a jump address, a data address, or an entire segment. Example 7–3 shows the PUBLIC statement used to define some labels and make them public to other modules. When segments are made public, they are combined with other public segments that contain data with the same segment name.

EXAMPLE 7–3

```
.MODEL SMALL
.DATA
        PUBLIC   DATA1        ;declare DATA1 and DATA2 public
        PUBLIC   DATA2
```

```
0000  0064[      DATA1    DB        100 DUP (?)
            00
              ]
0064  0064[      DATA2    DB        100 DUP (?)
            00
              ]
                .CODE
                .STARTUP
                         PUBLIC    READ          ;declare READ public
                READ     PROC      FAR

0006  B4 06              MOV       AH,6          ;read keyboard
0008  B2 FF              MOV       DL,0FFH
000A  CD 21              INT       21H
000C  74 F8              JE        READ          ;if no key typed
000E  CB                 RET

                READ     ENDP
                         END
```

The EXTRN statement appears in both data and code segments to define labels as external to the segment. If data are defined as external, their sizes must be defined as BYTE, WORD, or DWORD. If a jump or call address is external, it must be defined as NEAR or FAR. Example 7–4 shows how the external statement is used to indicate that several labels are external to the program listed. Notice in this example that any external address or data is defined with the letter E in the hexadecimal assembled listing.

EXAMPLE 7–4

```
                        .MODEL SMALL
                        .DATA

                              EXTRN DATA1:BYTE
                              EXTRN DATA2:BYTE
                              EXTRN DATA3:WORD
                              EXTRN DATA4:DWORD
                        .CODE
                              EXTRN READ:FAR
                        .STARTUP

0005 BF 0000 E                MOV   DX,OFFSET DATA1
0008 B9 000A                  MOV   CX,10
000B                    START:
000B 9A 0000 ---- E           CALL  READ
0010 AA                       STOSB
0011 E2 F8                    LOOP  START
                        .EXIT
                        END
```

Libraries

Library files are collections of procedures that are used by many different programs. These procedures are assembled and compiled into a library file by the LIB program that accompanies the MASM assembler program. Libraries allow common procedures to be collected into one place so they can be used by many different applications. The library file (FILENAME.LIB) is invoked when a program is linked with the linker program.

Why bother with library files? A library file is a good place to store a collection of related procedures. When the library file is linked with a program, only the procedures required by the program are removed from the library file and added to the program. If any amount of assembly language programming is to be accomplished efficiently, a good set of library files is essential and saves many hours in recoding common functions.

Creating a Library File. A library file is created with the LIB command, typed at the DOS prompt. A library file is a collection of assembled .OBJ files that each perform one procedure or task. Example 7–5 shows two separate files (READ_KEY and ECHO) that will be used to structure a library file. Please notice that the name of the procedure must be declared PUBLIC in a library file and does not necessarily need to match the file name, although it does in this example. Each procedure in this example is defined as a FAR procedure, so that the linker can place the procedures in a code segment separate from the main program. When FAR is used to define a procedure, we usually call it a global procedure.

EXAMPLE 7–5

```
                        ;The first library module is called READ_KEY.  This
                        ;procedure reads a key from the keyboard and returns with
                        ;its ASCII code in AL.
                        .MODEL TINY
                              PUBLIC READ_KEY

                        READ_KEY  PROC  FAR

0000  52                      PUSH   DX
                        READ_KEY1:
0001  B4 06                   MOV    AH,6
0003  B2 FF                   MOV    DH,0FFH
0005  CD 21                   INT    21H
0007  74 F8                   JE     READ_KEY1
0009  5A                      POP    DX
000A  CB                      RET

                        READ_KEY      ENDP
                              END
                        ;
                        ;The second library module is called ECHO.  This
                        ;procedure displays the ASCII character in AL on the
                        ;video screen.
                        .MODEL TINY
                              PUBLIC  ECHO

                        ECHO  PROC    FAR

0000  52                      PUSH   DX
0001  B4 06                   MOV    AH,6
0003  8A D0                   MOV    DL,AL
0005  CD 21                   INT    21H
0007  5A                      POP    DX
0008  CB                      RET

                        ECHO  ENDP
                              END
```

After each file is assembled (note that there are two complete example procedures in Example 7–5), the LIB program is used to combine them into a single library file. The LIB program prompts for information, as illustrated in Example 7–6, in which these files are combined to form the library IO.

EXAMPLE 7–6

```
C:\MASM611\FILES\LIB

Microsoft (R) Library Manager  Version 3.20.010
Copyright (C) Microsoft Corp. 1983-1992.  All rights reserved.
```

```
Library name: IO
Library file does not exist.  Create? Y
Operations: READ_KEY+ECHO
List file: IO
```

The LIB program begins with the copyright message from Microsoft, followed by the prompt *Library name*. The library name chosen is IO for the IO.LIB file. Because this is a new file, the library program asks if we wish to create the library file. The *Operations:* prompt is where the library module names are typed. In this case, we create a library by using two procedure files (READ_KEY and ECHO). Note that these files were created and assembled as READ_KEY.ASM and ECHO.ASM from Example 7–5. The list file shows the contents of the library and is illustrated in Example 7–7. The list file shows the size and names of the files used to create the library, and the public label (procedure name) that is used in the library file.

To add additional library modules, type the name of the library file after invoking LIB. At the *Operations*: prompt, type the new module name, preceded by a *plus sign* to add a new procedure. If you must delete a library module, use a *minus sign* before the operation file name.

EXAMPLE 7–7

```
ECHO..... . ECHO          READ_KEY..... . READ_KEY

READ_KEY     Offset: 00000010H Code and data size: BH
  READ_KEY

ECHO         Offset: 00000070H Code and data size: 9H
```

Once the library file is linked to your program file, only the library procedures actually used by your program are placed in the execution file. Don't forget to use the label EXTRN when specifying library calls from your program module. For example, to use the ECHO procedure in a program, type EXTRN ECHO:FAR.

Macros

A **macro** is a group of instructions that perform one task, just as a procedure performs one task. The difference is that a procedure is accessed via a CALL instruction, while a macro, and all the instructions defined in the macro, is inserted in the program at the point of usage. Creating a macro is very similar to creating a new opcode that can be used in the program. The name of the macro and any parameters associated with it are typed, and the assembler then inserts them into the program. Macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The instructions of the macro are placed in your program by the assembler at the point they are invoked.

The MACRO and ENDM directives delineate a macro sequence. The first statement of a macro is the MACRO instruction, which contains the name of the macro and any parameters associated with it. An example is MOVE MACRO A,B, which defines the macro name as MOVE. This new pseudo opcode uses two parameters: A and B. The last statement of a macro is the ENDM instruction, which is placed on a line by itself. Never place a label in front of the ENDM statement, or the macro will not assemble.

Example 7–8 shows how a macro is created and used in a program. The first six lines of code define the macro. This macro moves the word-sized contents of memory location B into word-sized memory location A. After the macro is defined in the example, it is used twice. The macro is **expanded** by the assembler in this example, so that you can see how it assembles to generate the moves. Any hexadecimal machine language statement followed by a number (1, in this example) is a macro expansion statement. The expansion statements are not typed in the source program; they are generated by the assembler to show that the assembler has inserted them into the program. Notice that the comment in the macro is preceded with ;; instead of ; as is customary. Macro sequences must always be defined before they are used in a program, so they generally appear at the top of the code segment.

EXAMPLE 7–8

```
                    MOVE  MACRO A,B

                          PUSH  AX
                          MOV   AX,B
                          MOV   A,AX
                          POP   AX

                          ENDM

                          MOVE  VAR1,VAR2   ;use the MOVE macro

0000  50          1       PUSH  AX
0001  A1 0002 R   1       MOV   AX,VAR2
0004  A3 0000 R   1       MOV   VAR1,AX
0007  58          1       POP   AX

                          MOVE  VAR3,VAR4   ;use the MOVE macro

0008  50          1       PUSH  AX
0009  A1 0006 R   1       MOV   AX,VAR4
000C  A3 0004 R   1       MOV   VAR3,AX
000F  58          1       POP   AX
```

Local Variables in a Macro. Sometimes, macros contain local variables. A **local variable** is one that appears in the macro, but is not available outside the macro. To define a local variable, we use the LOCAL directive. Example 7–9 shows how a local variable, used as a jump address, appears in a macro definition. If this jump address is not defined as local, the assembler will flag it with errors on the second and subsequent attempts to use the macro.

EXAMPLE 7–9

```
                    READ  MACRO A           ;;reads keyboard
                          LOCAL READ1       ;;define READ1 as local

                          PUSH  DX
                    READ1:
                          MOV   AH,6
                          MOV   DL,0FFH
                          INT   21H
                          JE    READ1
                          MOV   A,AL
                          POP   DX
                          ENDM


                          READ  VAR5        ;read key into VAR5

0000  52          1       PUSH  DX
0001              1   ??0000:
0001  B4 06       1       MOV   AH,6
0003  B2 FF       1       MOV   DL,0FFH
0005  CD 21       1       INT   21H
0007  74 F8       1       JE    ??0000
0009  A2 0008 R   1       MOV   VAR5,AL
000C  5A          1       POP   DX

                          READ  VAR6        ;read key into VAR6

000D  52          1       PUSH  DX
```

```
000E                1   ??0001:
000E  B4 06         1           MOV     AH,6
0010  B2 FF         1           MOV     DL,0FFH
0012  CD 21         1           INT     21H
0014  74 F8         1           JE      ??0001
0016  A2 0009 R     1           MOV     VAR6,AL
0019  5A            1           POP     DX
```

This example reads a character from the keyboard and stores it into the byte-sized memory location indicated as a parameter with the macro. Notice how the local label READ1 is treated in the expanded macros. The assembler uses labels that start with ?? to designate them as assembler- generated labels.

The LOCAL directive must always immediately follow the MACRO directive, without any intervening spaces or comments. If a comment or space appears between MACRO and LOCAL, the assembler indicates an error and will not accept the variable as local.

Placing MACRO Definitions in Their Own Module. Macro definitions can be placed in the program file as shown, or they can be placed in their own macro module. A file can be created that contains only macros to be included with other program files. We use the INCLUDE directive to indicate that a program file will include a module that contains external macro definitions. Although this is not a library file, for all practical purposes it functions as a library of macro sequences.

When macro sequences are placed in a file (often with the extension INC or MAC), they do not contain PUBLIC statements. If a file called MACRO.MAC contains macro sequences, the INCLUDE statement is placed in the program file as INCLUDE C:\ASSM\MACRO.MAC. Notice that the macro file is on drive C, subdirectory ASSM in this example. The INCLUDE statement includes these macros, just as if you had typed them into the file. No EXTRN statement is needed to access the macro statements that have been included. Programs may contain both macro include files and library files.

Conditional Statements in Macro Sequences

Conditional assembly language statements are available to the assembler for use in the assembly process and in macro sequences. The conditional statements create instructions that control the flow of the program and are variations of the IF-THEN, IF-THEN-ELSE, DO-WHILE, and REPEAT-UNTIL constructs used in high-level language programming languages, which were presented in the last chapter. The conditional statements for macro sequence control—presented here—are also available, but they function to create instructions only at assembly time within macro sequences. The assembler distinguishes conditional statements for macro control and condition statements for program flow with a period. For example, the .IF statement is used for program flow control, while the IF statement is used for macro assembly control. Both types of conditional statements may be used in a macro, but the macro conditionals may only be used in a macro.

Conditional Assembly Statements

As mentioned, conditional assembly is implemented with the IF-THEN or IF-THEN-ELSE construct found in high-level languages. Table 7–1 shows the forms used for the IF statement in the conditional assembly process.

The IF and ENDIF statements allow portions of the program to assemble if some condition is met. Otherwise, the statements between IF and ENDIF do not assemble and generate code.

Example 7–10 shows how the IF, ELSE, and ENDIF statements are used to conditionally assemble values for the width and length of paper in a program. Note that TRUE and FALSE are defined as 1 and 0. This is important because these values are not predefined by the assembler. Next, the width and length of the paper are adjusted by using TRUE and FALSE statements. This can be expanded to ask an entire series of questions about a program so that custom versions can be created. Example 7–10(a) is the original source-code, and Example 7–10(b) shows how the program assembles for TRUE answers for both the width and length. Example 7–10(c) shows the assembled output for a false width and a true length.

When Example 7–10(a) is assembled, TRUE and FALSE are equated to WIDT and LENGT to modify the way that the assembler forms the program. In Example 7–10(b), both WIDT and LENGT are defined as TRUE, which causes the assembler to modify the way the program is assembled, so that a page is 72 columns wide and the length is continuous. Example 7–10(c) is another example in which the WIDT is FALSE and LENGT is TRUE, causing the assembler to form the instructions that make the page width 80 columns and the length continuous. The only form not shown is where the page length is 66 lines.

Examples of some of the other forms listed in Table 7–1 appear later in the text. When one of these new conditional statements appears it is explained and shown with an example.

TABLE 7–1 Conditional assembly language IF statements.

Statement	*Function*
IF	If the expression is true
IFB	If argument is blank
IFE	If the expression is not true
IFDEF	If the label has been defined
IFNB	If argument is not blank
IFNDEF	If the label has not been defined
IFIDN	If argument 1 equals argument 2
IFDIF	If argument 1 does not equal argument 2

EXAMPLE 7–10(a)

```
;source program
;
TRUE    EQU  1                ;define true
FALSE   EQU  0                ;define false
WIDT    EQU  FALSE            ;set to true if 72 columns
                              ;and false if 80 columns
LENGT   EQU  TRUE             ;set to true if continuous
                              ;and false if 66 lines

                IF      WIDT  ;72 columns
WIDE            DB      72
                ELSE
WIDE            DB      80     ;80 columns
                ENDIF

                IF      LENGT  ;if continuous
LONG            DB      -1
                ELSE
LONG            DB      66      ;if 66 lines
                ENDIF
```

EXAMPLE 7–10(b)

```
                    ;assembled portion with WIDT = TRUE and LENGT = TRUE
                    ;
                            IF   WIDT             ;72 columns
0000  48            WIDE    DB   72
                            ELSE
                            ENDIF

                            IF   LENGT            ;if continuous
0001  FF            LONG    DB   -1
                            ELSE
                            ENDIF
```

EXAMPLE 7–10(c)

```
                    ;assembled portion with WIDT = FALSE and LENGT = TRUE
                    ;
                            IF   WIDT             ;72 columns
```

```
                        ELSE
0000  50          WIDE  DB   80              ;80 columns
                        ENDIF
                        IF   LENGT          ;if continuous
0001  FF          LONG  DB -1
                        ELSE
                        ENDIF
```

Using Conditional Statements in Macros

Macro sequences contain their own set of conditional instructions that differ somewhat from the ones used with the assembler, as presented in Chapter 6. For example, macros can use REPEAT and WHILE, but they do so without the period in front of the keywords REPEAT and WHILE. The REPEAT has no corresponding UNTIL, and the WHILE statement has no corresponding ENDW when used in a macro. These statements are available to all versions of the assembler.

Table 7–2 lists the relational operators used with WHILE and REPEAT. These operators are also used with any of the statements listed in Table 7–1. Note that these are different from the operators specified in Table 6–3 for the .WHILE and .REPEAT statements.

TABLE 7–2 Relational operators used with WHILE and REPEAT in macro sequences.

Operator	*Function*
EQ	Equal
NE	Not equal
LE	Less than or equal
LT	Less than
GE	Greater than or equal
GT	Greater than
NOT	Logical inversion
AND	Logical AND
OR	Logical OR
XOR	Logical exclusive-OR

REPEAT Statement in a Macro. The REPEAT statement has a parameter associated with it to repeat the macro sequence a fixed number of times. As with any macro sequence, the repeat sequence must end with the ENDM statement. The repeat sequence inserts the instructions that appear between the REPEAT statement and the ENDM statement into the program the number of times indicated with the REPEAT statement.

Example 7–11 shows a macro called TESTS and its calling program, which sends the 10 ASCII characters from 0 through 9 to the video screen. Notice how this macro is formed by using the MACRO statement to name the macro TESTS, and how the REPEAT statement appears within macro TESTS with its own ENDM statement. Notice that the macro starts by placing a 6 into AH and the ASCII code for a 0 in DL. This sets up the DOS INT 21H function call, so a 0 is displayed on the video screen. Next, the REPEAT statement appears (note that it does not contain a period, as in .REPEAT). This is a different REPEAT statement, used only in macro sequences and available to all versions of MASM.

The repeated statements in this example are INT 21H, which display the ASCII contents of DL and INC DL, which modifies the ASCII code displayed. In this case, the REPEAT 10 causes the statements between REPEAT 10 and the first ENDM to be repeated 10 times, as illustrated. Note that the 1 and 2 to the left of the instructions are listed to show that these statements are assembler-generated and not entered as part of the source program.

EXAMPLE 7–11

```
                TESTS   MACRO

                        MOV  AH,6
                        MOV  DL,'0'

                        REPEAT 10
                            INT  21H
                            INC  DL             ;;increment to next number
                        ENDM
                        ENDM

0000            MAIN    PROC FAR
```

```
                        TESTS                   ;display 0 through 9
0000 B4 06      1       MOV  AH,6
0002 B2 30      1       MOV  DL,'0'
0004 CD 21      2       INT  21H
0006 FE C2      2       INC  DL
0008 CD 21      2       INT  21H
000A FE C2      2       INC  DL
000C CD 21      2       INT  21H
000E FE C2      2       INC  DL
0010 CD 21      2       INT  21H
0012 FE C2      2       INC  DL
0014 CD 21      2       INT  21H
0016 FE C2      2       INC  DL
0018 CD 21      2       INT  21H
001A FE C2      2       INC  DL
001C CD 21      2       INT  21H
001E FE C2      2       INC  DL
0020 CD 21      2       INT  21H
0022 FE C2      2       INC  DL
0024 CD 21      2       INT  21H
0026 FE C2      2       INC  DL
0028 CD 21      2       INT  21H
002A FE C2      2       INC  DL

                        .EXIT

0031            MAIN    ENDP
```

WHILE Statement in a Macro. The WHILE statement appears in macro sequences in much the same way as REPEAT appears. That is, the while loop is terminated with the ENDM statement. The expression associated with WHILE determines how many times the loop is repeated. The WHILE statement is available to all versions of MASM.

Example 7–12 shows how the WHILE statement is used to generate a table of squares from 2 squared to whatever value fits into an array of byte-sized memory called SQUARE. The first statement of the sequence defines the label SQUARE for the first byte of data generated. The WHILE RES LT 255 repeats the calculation (SEED*SEED), while the result is less than or equal to 255. Notice that the table generated contains the square of the numbers from 2 to 15, or 225 (E1H). If you look closely at Example 7–12, the value of the SEED + 1 and SEED*SEED shows the number and its square.

EXAMPLE 7–12

```
                ;table of byte-sized squares
                ;
0000            SQUARE  LABEL BYTE              ;;define label
 = 0001         SEED    =   1
 = 0001         RES     =   SEED*SEED           ;;compute square
                WHILE   RES LT 255
                        DB   RES
                SEED =  SEED+1
                RES  =  SEED*SEED
                ENDM
0000 01         1       DB   RES
 = 0002         1 SEED =  SEED+1
 = 0004         1 RES  =  SEED*SEED
0001 04         1       DB   RES
 = 0003         1 SEED =  SEED+1
 = 0009         1 RES  =  SEED*SEED
0002 09         1       DB   RES
 = 0004         1 SEED =  SEED+1
```

```
 = 0010          1 RES   =   SEED*SEED
0003 10          1           DB    RES
 = 0005          1 SEED  =   SEED+1
 = 0019          1 RES   =   SEED*SEED
0004 19          1           DB    RES
 = 0006          1 SEED  =   SEED+1
 = 0024          1 RES   =   SEED*SEED
0005 24          1           DB    RES
 = 0007          1 SEED  =   SEED+1
 = 0031          1 RES   =   SEED*SEED
0006 31          1           DB    RES
 = 0008          1 SEED  =   SEED+1
 = 0040          1 RES   =   SEED*SEED
0007 40          1           DB    RES
 = 0009          1 SEED  =   SEED+1
 = 0051          1 RES   =   SEED*SEED
0008 51          1           DB    RES
 = 000A          1 SEED  =   SEED+1
 = 0064          1 RES   =   SEED*SEED
0009 64          1           DB    RES
 = 000B          1 SEED  =   SEED+1
 = 0079          1 RES   =   SEED*SEED
000A 79          1           DB    RES
 = 000C          1 SEED  =   SEED+1
 = 0090          1 RES   =   SEED*SEED
000B 90          1           DB    RES
 = 000D          1 SEED  =   SEED+1
 = 00A9          1 RES   =   SEED*SEED
000C A9          1           DB    RES
 = 000E          1 SEED  =   SEED+1
 = 00C4          1 RES   =   SEED*SEED
000D C4          1           DB    RES
 = 000F          1 SEED  =   SEED+1
 = 00E1          1 RES   =   SEED*SEED
```

FOR Statement in a Macro. The FOR statement iterates a list of data. If you are familiar with BASIC, the FOR statement functions like the READ statement, and the list of data associated with it functions like the DATA statement. Example 7–13 shows how the FOR statement is used to display a series of characters on the video display. Notice that the CHR:VARARG indicates the variable name CHR that is of variable size (VARARG). The first use of DISP generates the code required to display BARRY. The second use of the DISP macro generates the code required to display BREY. The FOR statement counts the variable used after display and repeats the commands between FOR and ENDM for each variable; in this case, each ASCII character.

EXAMPLE 7–13

```
                    DISP    MACRO   CHR:VARARG
                            MOV     AH,2
                            FOR     ARG,<CHR>
                                    MOV     DL,ARG
                                    INT     21H
                            ENDM
                            ENDM

                            DISP    'B','A','R','R','Y',' '

0000  B4 02       1         MOV     AH,2
0002  B2 42       2         MOV     DL,'B'
0004  CD 21       2         INT     21H
0006  B2 41       2         MOV     DL,'A'
0008  CD 21       2         INT     21H
```

```
000A  B2 52       2          MOV  DL,'R'
000C  CD 21       2          INT  21H
000E  B2 52       2          MOV  DL,'R'
0010  CD 21       2          INT  21H
0012  B2 59       2          MOV  DL,'Y'
0014  CD 21       2          INT  21H
0016  B2 20       2          MOV  DL,' '
0018  CD 21       2          INT  21H

                             DISP  'B','R','E','Y'

001A  B4 02       1          MOV  AH,2
001C  B2 42       2          MOV  DL,'B'
001E  CD 21       2          INT  21H
0020  B2 52       2          MOV  DL,'R'
0022  CD 21       2          INT  21H
0024  B2 45       2          MOV  DL,'E'
0026  CD 21       2          INT  21H
0028  B2 59       2          MOV  DL,'Y'
002A  CD 21       2          INT  21H
```

IF, ELSE, and ENDIF Statements in a Macro. The IF statement is used in a macro to make decisions, based on the parameters sent to the macro. As before, note that IF is used in a macro and .IF is used in a program. The IF statement is available to all versions of the assembler, whereas .IF is available only to version 6.X.

In Example 7–14, a macro is developed that uses a number of conditional assembly statements to read a key, display a character, or display a carriage return and line feed combination. This example illustrates the use of IF, IFB, INB, ENDIF, and ELSE. The macro is called IO. If IO is used on a line by itself, the assembler generates the code to read a key. If IO –1 appears as a statement, the assembler generates the code required to display a carriage return and line feed. If IO 'B' appears as a statement, the assembler generates the code required to display the letter B. This example is listed in expanded form, so that the code generated by the assembler can be viewed and studied. As before, the lines that contain a number between the hexadecimal code and the statement in the program are assembler-generated, and are not included in the original source program.

EXAMPLE 7–14

```
                             .MODEL TINY
0000                         .CODE
                    ;the IO macro functions in 3 ways
                    ;
                    ;(1) IO          read a key with echo
                    ;(2) IO -1       display a carriage return & line feed
                    ;(3) IO 'B'      display the letter 'B'
                    ;or  IO AL       display contents of AL
                    ;
                    IO      MACRO  CHAR
                            IFB   <CHAR>                  ;;if CHAR is blank
                                MOV  AH,1                 ;;read key function
                            ENDIF


                            IFNB  <CHAR>                  ;;if CHAR not blank
                                MOV  AH,2                 ;;display character

                                    IF  CHAR EQ -1        ;;if CHAR equals -1
                                        MOV  DL,13        ;;display return
                                        INT  21H
                                        MOV  DL,10        ;;display line feed
```

```
                                    ELSE                    ;;if CHAR not -1

                                        MOV  DL,CHAR        ;;load CHAR to DL
                                    ENDIF

                                ENDIF

                                INT  21H
                                ENDM
                                .STARTUP
                            ;
                            ;This program does a carriage return, line feed then
                            ;displays the letters BE on the video screen.  Next it
                            ;waits for a key to be typed.  Following the key, a
                            ;carriage return/line feed is displayed.
                            ;
                                IO   -1                     ;return & line feed

0100  B4 02       1         MOV  AH,2
0102  B2 0D       1         MOV  DL,13
0104  CD 21       1         INT  21H
0106  B2 0A       1         MOV  DL,10
0108  CD 21       1         INT  21H

                                IO   'B'                    ;display 'B'

010A  B4 02       1         MOV  AH,2
010C  B2 42       1         MOV  DL,'B'
010E  CD 21       1         INT  21H

                                IO   'E'                    ;display 'E'

0110  B4 02       1         MOV  AH,2
0112  B2 45       1         MOV  DL,'E'
0114  CD 21       1         INT  21H

                                IO                          ;read key

0116  B4 01       1         MOV  AH,1
0118  CD 21       1         INT  21H

                                IO   -1                     ;return & line feed

011A  B4 02       1         MOV  AH,2
011C  B2 0D       1         MOV  DL,13
011E  CD 21       1         INT  21H
0120  B2 0A       1         MOV  DL,10
0122  CD 21       1         INT  21H

                                .EXIT
                                END
```

The first part of the macro uses the IFB <CHAR> statement to test CHAR for a blank condition. If CHAR is blank, the assembler generates the MOV AH,1 instruction followed by the very last instruction in the macro, INT 21H, to read a key with echo. This is used in the program with the IO statement.

The second part of the macro contains the IFNB <CHAR> statement to test if CHAR is not blank. If CHAR is not blank, another IF-ELSE-ENDIF sequence appears to test the contents of CHAR. If CHAR is a –1, the assembler generates the code required to display a carriage return and line feed combination. If CHAR is not a –1, the ELSE statement places CHAR into DL for display. This very powerful macro can handle most keyboard and single-character display functions. It also illustrates the power of the conditional assembly statements, when used within a macro.

The Modular Programming Approach

The modular programming approach often involves a team of people with different programming tasks. This allows the team manager to assign portions of the program to different team members. Often, the team manager develops the system flowchart or shell, and then divides it into modules for team members.

A team member might be assigned the task of developing a macro definition file. This file might contain macro definitions that handle the I/O operations for the system. Another team member might be assigned the task of developing the procedures used for the system. In most cases, the procedures are organized as a library file that is linked to the program modules. Finally, several program files or modules might be used for the final system, each developed by different team members.

This approach requires considerable communications between team members and good documentation. Documentation is the key so that modules interface correctly. Communication among team members plays an essential role in this approach.

7–2 USING THE KEYBOARD AND VIDEO DISPLAY

Today, there are few programs that don't use the keyboard and video display. This section of the text explains how to use the keyboard and video display connected to the IBM PC or compatible computer running under MSDOS.

Reading the Keyboard with DOS Functions

The keyboard of the personal computer is read via a DOS function call. A complete listing of the DOS function calls appears in Appendix A. This section uses INT 21H with various DOS function calls to read the keyboard. Data read from the keyboard are either in ASCII-coded form or in extended ASCII-coded form.

The ASCII-coded data appear as outlined in Table 1–7 in Section 1–4. The extended character set of Table 1–8 applies to printed or displayed data only, and not to keyboard data. Notice that the ASCII codes in Table 1–7 correspond to most of the keys on the keyboard. Also available through the keyboard are extended ASCII-coded keyboard data. Table 7–3 lists most of the extended ASCII codes obtained with various keys and key combinations. Notice that most keys on the keyboard have alternative key codes. Each function key has four sets of codes selected by the function key alone, the shift-function key combination, the alternate-function key combination, and the control-function key combination.

There are three ways to read the keyboard. The first method reads a key and echoes (or displays) the key on the video screen. A second way simply tests to see if a key is pressed. If it is, it reads the key; otherwise it returns without any key. The third way allows an entire character string or line to be read from the keyboard.

Reading a Key with an Echo. Example 7–15 shows how a key is read from the keyboard and **echoed** (sent) back out to the video display by using a procedure called KEY. Although this method is the easiest way to read a key, it is also the most limited because it always echoes the character to the screen, even if it is an unwanted character. The DOS function number 01H also responds to the control-C key combination, and exits to DOS if it is typed.

EXAMPLE 7–15

```
0000                    KEY     PROC    FAR

0000  B4 01                     MOV     AH,1            ;function 01H
0002  CD 21                     INT     21H             ;read key
0004  0A C0                     OR      AL,AL           ;test for 00H, clear carry
0006  75 03                     JNZ     KEY1
0008  CD 21                     INT     21H             ;get extended
000A  F9                        STC                     ;indicate extended
000B                    KEY1:
000B  CB                        RET

000C                    KEY     ENDP
```

TABLE 7-3 The keyboard scanning and extended ASCII codes as returned from the keyboard.

		Extended ASCII code with....			
Key	*Scan Code*	*Nothing*	*Shift*	*Control*	*Alternate*
Esc	01				01
1	02				78
2	03			03	79
3	04				7A
4	05				7B
5	06				7C
6	07				7D
7	08				7E
8	09				7F
9	0A				80
0	0B				81
-	0C				82
+	0D				83
Bksp	0E				0E
Tab	0F		0F	94	A5
Q	10				10
W	11				11
E	12				12
R	13				13
T	14				14
Y	15				15
U	16				16
I	17				17
O	18				18
P	19				19
[	1A				1A
]	1B				1B
Enter	1C				1C
Enter	1C				A6
Lctrl	1D				
Rctrl	1D				
A	1E				1E
S	1F				1F
D	20				20
F	21				21
G	22				22

(continued on next page)

TABLE 7–3 *(continued)*

		Extended ASCII code with....			
Key	*Scan Code*	*Nothing*	*Shift*	*Control*	*Alternate*
H	23				23
J	24				24
K	25				25
L	26				26
;	27				27
'	28				28
`	29				29
Lshft	2A				
\	2B				
Z	2C				2C
X	2D				2D
C	2E				2E
V	2F				2F
B	30				30
N	31				31
M	32				32
,	33				33
.	34				34
/	35				35
Gray /	35			95	A4
Rshft	36				
PrtSc	E0 2A E0 37				
L alt	38				
R alt	38				
Space	39				
Caps	3A				
F1	3B	3B	54	5E	68
F2	3C	3C	55	5F	69
F3	3D	3D	56	60	6A
F4	3E	3E	57	61	6B
F5	3F	3F	58	62	6C
F6	40	40	59	63	6D
F7	41	41	5A	64	6E
F8	42	42	5B	65	6F
F9	43	43	5C	66	70
F10	44	44	5D	67	71
F11	57	85	87	89	8B
F12	58	86	88	8A	8C
Num	45				
Scroll	46				
Home	E0 47	47	47	77	97

(continued on next page)

TABLE 7–3 *(continued)*

		Extended ASCII code with....			
Key	*Scan Code*	*Nothing*	*Shift*	*Control*	*Alternate*
Up	48	48	48	8D	98
Pgup	E0 49	49	49	84	99
Gray -	4A				
Left	4B	4B	4B	73	9B
Center	4C				
Right	4D	4D	4D	74	9D
Gray +	4E				
End	E0 4F	4F	4F	75	9F
Down	E0 50	50	50	91	A0
Pgdn	E0 51	51	51	76	A1
Ins	E0 52	52	52	92	A2
Del	E0 53	53	53	93	A3
Pause	E0 10 45				

To read and echo a character, the AH register is loaded with DOS function number 01H. This is followed by the INT 21H instruction, which calls a procedure that processes DOS function calls. Upon return from the INT 21H, the AL register contains the ASCII character typed; the video display also shows the typed character. If AL = 0 after the return, the INT 21H instruction must again be executed to obtain the extended ASCII-coded character (refer to Table 7–3). The procedure of Example 7–15 returns with carry set (1) to indicate an extended ASCII character and carry cleared (0) to indicate a normal ASCII character. When this procedure is called, the CALL instruction might be followed by a JC EXTENDED to process the extended ASCII character.

Reading a Key without an Echo. The best single character key-reading function is function number 06H. This function reads a key without an echo to the screen. It also allows extended ASCII characters and *does not* respond to the control-C key combination. This function uses AH for the function number (06H) and DL = 0FFH to indicate that the function call (INT 21H) will read the keyboard without an echo. I usually use DL = –1 instead of DL = 0FFH because it is easier to type and has the same value (because 0FFH = –1).

Example 7–16 shows a procedure that uses function number 06H to read the keyboard. This performs as shown in Example 7–15, except that no character is echoed to the video display.

EXAMPLE 7–16

```
000                KEYS    PROC    FAR

0000  B4 06                MOV     AH,6            ;function 06H
0002  B2 FF                MOV     DL,0FFH
0004  CD 21                INT     21H             ;read key
0006  74 F8                JE      KEYS            ;if no key
0008  0A C0                OR      AL,AL           ;test for 00H, clear carry
000A  75 03                JNE     KEYS1
000C  CD 21                INT     21H             ;get extended
000E  F9                   STC                     ;indicate extended
000F                KEYS1:
000F  CB                   RET

0010                KEYS    ENDP
```

If you examine the procedure, there is one other difference. Function call number 06H returns from the INT 21H, even if no key is typed; function call 01H waits for a key to be typed. This is an important difference that should be noted. This feature allows software to perform other tasks between checking the keyboard for a character.

Read an Entire Line with an Echo. Sometimes, it is advantageous to read an entire line of data with one function call. Function call number 0AH reads an entire line of information—up to 255 characters—from the keyboard. It continues to acquire keyboard data until either the enter key (0DH) is typed or the character count expires. This function requires that AH = 0AH, and DS:DX addresses the keyboard buffer (a memory area where the ASCII data are stored). The first byte of the buffer area must contain the maximum number of keyboard characters read by this function. If the number typed exceeds this maximum number, the function returns, just as if the enter key were typed. The second byte of the buffer contains the count of the actual number of characters typed, and the remaining locations in the buffer contain the ASCII keyboard data.

Example 7–17 shows how this function reads two lines of information into two memory buffers (BUF1 and BUF2). Before the call to the DOS function through the LINE procedure, the first byte of the buffer is loaded with a 255, so up to 255 characters can be typed. If you assemble and execute this program, the first and second lines are accepted. The only problem is that the second line appears on top of the first line. The next section of the text explains how to output characters to the video display to solve this problem.

EXAMPLE 7–17

```
                    ;A program that reads two lines of data from the keyboard
                    ;using DOS INT 21H function number 0AH.
                    ;***uses***
                    ;LINE procedure to read a line.
                    ;
                            .MODEL SMALL            ;select SMALL model
0000                        .DATA                   ;start DATA segment
0000  0101 [        BUF1    DB   257 DUP (?)        ;define BUF1
           00
              ]
0101  0101 [        BUF2    DB   257 DUP (?)        ;define BUF2
           00
              ]
0000                        .CODE                   ;start CODE segment
                            .STARTUP                ;start program
0017  C6 06 0000 R FF       MOV  BUF1,255           ;character count of 255
001C  BA 0000 R             MOV  DX,OFFSET BUF1     ;address BUF1
001F  E8 000F               CALL LINE               ;read a line

0022  C6 06 0101 R FF       MOV  BUF2,255           ;character count of 255
0027  BA 0101 R             MOV  DX,OFFSET BUF2     ;address BUF2
002A  E8 0004               CALL LINE               ;read a line
                            .EXIT                   ;exit to DOS
                    ;
                    ;The LINE procedure uses DOS INT 21H function 0AH to
                    ;read and echo an entire line from the keyboard.
                    ;***parameters***
                    ;DX must contain the data segment offset address of the
                    ;buffer. The first location in the buffer contains the
                    ;number of characters to be read for the line.
                    ;Upon return the second location in the buffer contains
                    ;the line length.
                    ;
0031                LINE    PROC NEAR

0031  B4 0A                 MOV  AH,0AH             ;select function 0AH
0033  CD 21                 INT  21H                ;access DOS
0035  C3                    RET                     ;return from procedure

0036                LINE    ENDP
                            END                     ;end of file
```

Writing to the Video Display with DOS Functions

With most programs, data must be displayed on the video display. Video data are displayed in a number of different ways with DOS function calls. We use function 02H or 06H for displaying one character at a time, or function 09H for displaying an entire string of characters. Because functions 02H and 06H are identical, we tend to use function 06H because it is also used to read a key and, as mentioned, does not respond to a control-C key combination.

Displaying One ASCII Character. Both DOS functions 02H and 06H are explained together because they are identical for displaying ASCII data. Example 7–18 shows how this function displays a carriage return (0DH) and a line feed (0AH). Here a macro sequence, called DISP (display), displays the carriage return and line feed. The combination of a carriage return and a line feed moves the cursor to the next line at the left margin of the video screen. This two-step process is used to correct the problem that occurred between the lines typed through the keyboard in Example 7–17.

EXAMPLE 7–18

```
                        ;A program that displays a carriage return and a line
                        ;feed using the DISP macro.
                        ;
                         .MODEL TINY                    ;select TINY model
                         .CODE                          ;start CODE segment
                         DISP   MACRO A                 ;;display A macro

                                MOV  AH,06H             ;;DOS function 06H
                                MOV  DL,A               ;;place parameter A in DL
                                INT  21H                ;;display parameter A

                                ENDM

                         .STARTUP                       ;start program

                                DISP 0DH                ;display carriage return

0100  B4 06       1             MOV  AH,06H
0102  B2 0D       1             MOV  DL,0DH
0104  CD 21       1             INT  21H

                                DISP 0AH                ;display line feed

0106  B4 06       1             MOV  AH,06H
0108  B2 0A       1             MOV  DL,0AH
010A  CD 21       1             INT  21H
                          .EXIT                         ;exit to DOS
                                END                     ;end of file
```

Displaying a Character String. A character string is a series of ASCII-coded characters that end with a $ (24H) when used with DOS function call number 09H. Example 7–19 shows how a message is displayed at the current cursor position on the video display. Function call number 09H requires that DS:DX address the character string before executing the INT 21H instruction.

EXAMPLE 7–19

```
                          .MODEL SMALL          ;select SMALL model
0000                      .DATA                 ;start DATA segment

0000  0D 0A 0A 54  MES    DB   13,10,10,'This is a test line.$'
      68 69 73 20
      69 73 20 61
      20 74 65 73
      74 20 6C 69
```

```
      6E 65 2E 24
0000                          .CODE                 ;start CODE segment
                              .STARTUP              ;start program

0017  B4 09                   MOV  AH,9             ;select function 09H
0019  BA 0000 R               MOV  DX,OFFSET MES    ;address character string
001C  CD 21                   INT  21H              ;access DOS

                              .EXIT                 ;exit to DOS
                              END                   ;end of file
```

This example program can be entered into the assembler, linked, and executed to produce *"This is a test line"* on the video display.

The .EXIT directive embodies the DOS function 4CH. As shown in Appendix A, DOS function 4CH terminates a program. The .EXIT directive inserts a series of two instructions in the program MOV AH,4CH, followed by an INT 21H instruction.

Using BIOS Video Function Calls

In addition to the DOS function call INT 21H, we also have video BIOS (basic I/O system) function calls at INT 10H. The DOS function calls allow a key to be read and a character to be displayed with ease, but the cursor is difficult to position at the desired screen location. The video BIOS function calls allow more control over the video display than the DOS function calls do.The video BIOS function calls also require less time to execute than the DOS function calls do. The DOS function calls do not allow cursor placement, while the video BIOS function calls do.

Cursor Position. Before any information is placed on the video screen, the position of the cursor should be known. This allows the screen to be cleared and started at any desired location. Video BIOS function number 03H allows the cursor position to be read from the video interface. Video BIOS function number 02H allows the cursor to be placed at any screen position. Table 7–4 shows the contents of various registers for both functions 02H and 03H.

The page number in register BH should be 0 before setting the cursor position. Most software does not normally access the other pages (1–7) of the video display. The page number is often ignored after a cursor read. The 0 page is available in the CGA (color graphics adapter), EGA (enhanced graphics adapter), and VGA (variable graphics array) text modes of operation.

The cursor position assumes that the left-hand page column is column 0, progressing across a line to column 79. The row number corresponds to the character line number on the screen. Row 0 is the uppermost line, while row 24 is the last line on the screen. This assumes that the text mode selected for the video adapter is 80 characters per line by 25 lines. Other text modes are also available, such as 40 × 25 and 96 × 43.

Example 7–20 shows how the video BIOS function call INT 10H is used to clear the video screen. This is just one method of clearing the screen. Notice that the first function call positions the cursor to row 0 and column 0, which is called the **home position.** Next, we use the DOS function call to write 2000 (80 characters per line × 25 character lines) blank spaces (20H) on the video display. Finally, the cursor is again moved to the home position.

TABLE 7–4 Video BIOS function INT 10H.

AH	*Description*	*Parameters*
02H	Sets cursor position	DH = row, DL = column, and BH = page number
03H	Reads cursor position	DH = row, DL = column, and BH = page number

EXAMPLE 7–20

```
                        ;A program that clears the screen and homes the
                        ;cursor to the upper left-hand corner of the screen.
                        ;
                                .MODEL TINY             ;select TINY model
0000                            .CODE                   ;start CODE segment
                        HOME    MACRO                   ;;home cursor macro
                                MOV  AH,2               ;;function 02H
                                MOV  BH,0               ;;page 0
                                MOV  DX,0               ;;row 0, line 0
                                INT  10H                ;;home cursor
                                ENDM

                                .STARTUP                ;start program
                                HOME                    ;home cursor
0100  B4 02       1             MOV  AH,2
0102  B7 00       1             MOV  BH,0
0104  BA 0000     1             MOV  DX,0
0107  CD 10       1             INT  10H
0109  B9 07D0                   MOV  CX,25*80           ;load character count
010C  B4 06                     MOV  AH,6               ;select function 06H
010E  B2 20                     MOV  DL,' '             ;select a space
0110                    MAIN1:
0110  CD 21                     INT  21H                ;display a space
0112  E2 FC                     LOOP MAIN1              ;repeat 2000 times
                                HOME                    ;home cursor
0114  B4 02       1             MOV  AH,2
0116  B7 00       1             MOV  BH,0
0118  BA 0000     1             MOV  DX,0
011B  CD 10       1             INT  10H
                                .EXIT                   ;exit to DOS
                                END                     ;end of file
```

If this example is assembled, linked, and executed, a problem surfaces. This program is too slow to be useful in most cases. To correct this situation, another video BIOS function call is used. We can use the scroll function (06H) to clear the screen at a much higher speed.

Function 06H is used with a 00H in AL to blank the entire screen. This allows Example 7–20 to be rewritten so that the screen clears at a much higher speed. See Example 7–21 for a faster clear and home cursor program. Here, function call number 08H reads the character attributes for blanking the screen. Next, they are positioned in the correct registers and DX is loaded with the screen size, 4FH (79) and 19H (25). If this program is assembled, linked, executed, and compared with Example 7–20, there is a big difference in the speed at which the screen is cleared. (Make sure that the lines in the program that are macro expansion ending in a 1 are not typed into the program.) Please refer to Appendix A for other video BIOS INT 10H function calls that may prove useful in your applications. Also listed in Appendix A is a complete listing of all the INT functions available in most computers.

EXAMPLE 7–21

```
                        ;A program that clears the screen and homes the cursor.
                        ;
                                .MODEL TINY             ;select TINY model
0000                            .CODE                   ;start code segment
                                HOME MACRO              ;;home cursor
                                MOV  AH,2
                                MOV  BH,0
                                MOV  DX,0
                                INT  10H
                                ENDM

                                .STARTUP                ;start program
```

```
0100  B7 00              MOV  BH,0
0102  B4 08              MOV  AH,8
0104  CD 10              INT  10H              ;read video attribute

0106  8A DF              MOV  BL,BH            ;load page number
0108  8A FC              MOV  BH,AH
010A  B9 0000            MOV  CX,0             ;load attributes
010D  BA 194F            MOV  DX,194FH         ;line 25, column 79
0110  B8 0600            MOV  AX,600H          ;select scroll function
0113  CD 10              INT  10H              ;scroll screen
                         HOME                  ;home cursor
0115  B4 02       1      MOV  AH,2
0117  B7 00       1      MOV  BH,0
0119  BA 0000     1      MOV  DX,0
011C  CD 10       1      INT  10H

                         .EXIT                 ;exit to DOS
                         END                   ;end program
```

Display Macro

One of the more usable macro sequences is the one illustrated in Example 7–22. Although it is simple and has been presented before, it saves much typing when creating programs that must display many individual characters. What makes this macro so useful is that a register can be specified as the argument, an ASCII character in quotes, or the numeric value for an ASCII character.

EXAMPLE 7–22

```
                         ;A program that displays AB followed by a carriage
                         ;return and line feed combination using the DISP macro.
                         ;
                                .MODEL TINY           ;select TINY model
                                .CODE                 ;start CODE segment
                         DISP   MACRO VAR             ;;display VAR macro
                                MOV  DL,VAR
                                MOV  AH,6
                                INT  21H
                                ENDM
                                .STARTUP              ;start program
                                DISP  'A'             ;display 'A'
0100 B2  41       1      MOV  DL,'A'
0102 B4  06       1      MOV  AH,6
0104 CD  21       1      INT  21H

0106 B0  42              MOV  AL,'B'           ;load AL with 'B'
                         DISP  AL              ;display 'B'
0008 8A  D0       1      MOV  DL,AL
000A B4  06       1      MOV  AH,6
000C CD  21       1      INT  21H

                         DISP  13              ;display carriage return
000E B2  0D       1      MOV  DL,13
0010 B4  06       1      MOV  AH,6
0012 CD  21       1      INT  21H

                         DISP  10              ;display line feed
0014 B2  0A       1      MOV  DL,10
0016 B4  06       1      MOV  AH,6
0018 CD  21       1      INT  21H
                         .EXIT                 ;exit to DOS
                         END                   ;end of file
```

7-3 DATA CONVERSIONS

In computer systems, data are seldom in the correct form. One main task of the system is to convert data from one form to another. This section of the chapter describes conversions between binary and ASCII. Binary data are removed from a register or memory and converted to ASCII for the video display. In many cases, ASCII data are converted to binary as they are typed on the keyboard. We also explain converting between ASCII and hexadecimal data.

Converting from Binary to ASCII

Conversion from binary to ASCII is accomplished in two ways: (1) by the AAM instruction if the number is less than 100, or (2) by a series of decimal divisions (divide by 10). Both techniques are presented in this section.

The AAM instruction converts the value in AX into a two-digit unpacked BCD number in AX. If the number in AX is 0062H (98 decimal) before AAM executes, AX contains a 0908H after AAM executes. This is not ASCII code, but it is converted to ASCII code by adding a 3030H to AX. Example 7–23 illustrates a program that uses the procedure DISP, which processes the binary value in AL (0–99) and displays it on the video screen as decimal. The DISP procedure blanks a leading zero, which occurs for the numbers 0–9, with an ASCII space code. This example program displays the number 74 (test data) on the video screen.

EXAMPLE 7–23

```
                    ;A program that uses the DISP procedure to display 74
                    ;decimal on the video display.
                    ;
                            .MODEL TINY             ;select TINY mode
0000                        .CODE                   ;start code segment
                            .STARTUP                ;start program
0100  B0 4A                 MOV  AL,4AH             ;load test data to AL
0102  E8 0004               CALL DISP               ;display AL in decimal
                            .EXIT                   ;exit to DOS
                    ;
                    ;The DISP procedure displays AL (0 to 99) as a decimal
                    ;number. AX is destroyed by this procedure.
                    ;
0109                DISP    PROC NEAR

0109  52                   PUSH DX                  ;save DX
010A  B4 00                MOV  AH,0                ;clear AH
010C  D4 0A                AAM                      ;convert to BCD
010E  80 C4 20             ADD  AH,20H
0111  80 FC 20             CMP  AH,20H              ;test for leading zero
0114  74 03                JE   DISP1               ;if leading zero
0116  80 C4 10             ADD  AH,10H              ;convert to ASCII
0119                DISP1:
0119  8A D4                MOV  DL,AH               ;display first digit
011B  B4 06                MOV  AH,6
011D  50                   PUSH AX
011E  CD 21                INT  21H
0120  58                   POP  AX
0121  8A D0                MOV  DL,AL
0123  80 C2 30             ADD  DL,30H              ;convert second digit to ASCII
0126  CD 21                INT  21H                 ;display second digit
0128  5A                   POP  DX                  ;restore DX
0129  C3                   RET

012A                DISP    ENDP
                            END                     ;end of file
```

The reason that AAM converts any number between 0 and 99 to a two-digit unpacked BCD number is because it divides AX by 10. The result is left in AX so AH contains the quotient and AL the remainder. This same scheme of dividing by 10 can be expanded to convert any whole number of any number system from binary to an ASCII-coded character string that can be displayed on the video screen. For example, if AX is divided by 8 instead of 10, the number is displayed in octal.

The algorithm for converting from binary to ASCII code is:

1. Divide by the 10, then save the remainder on the stack as a significant BCD digit.
2. Repeat step 1 until the quotient is a 0.
3. Retrieve each remainder and add a 30H to convert to ASCII before displaying or printing.

Example 7–24 shows how the unsigned 16-bit content of AX is converted to ASCII and displayed on the video screen. Here, we divide AX by 10 and save the remainder on the stack after each division for later conversion to ASCII. After all the digits have been converted, the result is displayed on the video screen by removing the remainders from the stack and converting them to ASCII code. This procedure (DISPX) also blanks any leading zeros that occur.

EXAMPLE 7–24

```
                        ;A program that uses DISPX to display AX in decimal.
                        ;
                                .MODEL TINY             ;select TINY model
0000                            .CODE                   ;start CODE segment
                                .STARTUP                ;start program
0100  B8 04A3                   MOV  AX,4A3H            ;load AX with test data
0103  E8 0004                   CALL DISPX              ;display AX in decimal
                                .EXIT                   ;exit to DOS
                        ;
                        ;The DISPX procedure displays AX in decimal.
                        ;AX is destroyed.
                        ;
010A                    DISPX   PROC NEAR

010A  52                        PUSH DX                 ;save DX, CX, and BX
010B  51                        PUSH CX
010C  53                        PUSH BX
010D  B9 0000                   MOV  CX,0               ;clear digit counter
0110  BB 000A                   MOV  BX,10              ;set for decimal
0113                    DISPX1:
0113  BA 0000                   MOV  DX,0               ;clear DX
0116  F7 F3                     DIV  BX                 ;divide DX:AX by 10
0118  52                       PUSH DX                 ;save remainder
0119  41                       INC  CX                 ;count remainder
011A  0B C0                     OR   AX,AX              ;test for quotient of zero
011C  75 F5                     JNZ  DISPX1             ;if quotient is not zero
011E                    DISPX2:
011E  5A                        POP  DX                 ;get remainder
011F  B4 06                     MOV  AH,6               ;select function 06H
0121  80 C2 30                  ADD  DL,30H             ;convert to ASCII
0124  CD 21                     INT  21H                ;display digit
0126  E2 F6                     LOOP DISPX2             ;repeat for all digits

0128  5B                        POP  BX                 ;restore BX, CX, and DX
0129  59                       POP  CX
012A  5A                        POP  DX
012B  C3                        RET

012C                    DISPX   ENDP
                                END                     ;end of file
```