Answers to Odd-Numbered Section Review Questions

for: Irvine, Kip R. *Assembly Language for Intel-Based Computers*, 4th Edition

Prepared by the author. Recent revision history:

- 11/24/2002: 11.2 (*notes*)
- 12/3/2002: 7.5 #3
- 1/1/2003: 5.4 #1

1 Basic Concepts

1.1 Welcome to Assembly Language

1. An assembler is a program that converts source-code programs from assembly language into machine language. A linker combines individual files created by an assembler into a single executable program.
2.
3. In a *one-to-many* relationship, a single statement expands into multiple assembly language or machine instructions.
4.
5. No. Each assembly language is based on either a processor family or specific computer.
6.
7. *Device drivers* are programs that translate general operating system commands into specific references to hardware details that only the manufacturer would know.
8.
9. Applications suited to assembly language: Hardware device driver, and embedded systems and computer games requiring direct hardware access.
10.

11. Assembly language has minimal formal structure, so structure must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.

12.

1.2 Virtual Machine Concept

1. Computers are constructed in layers, so that each layer represents a translation layer from a higher-level instruction set to a lower level instruction set.

2.

3. True

4.

5. The IA-32's virtual-86 operating mode emulates the architecture of the Intel 8086/8088 processor, used in the original IBM Personal Computer.

6.

7. Digital logic, microarchitecture, instruction set architecture, operating system, assembly language, high-level language.

8.

9. Instruction-set architecture

10.

1.3 Data Representation

1. Least significant bit (bit 0).

2.

3. (a) 248 (b) 202 (c) 240

4.

5. (a) 00010001 (b) 101000000 (c) 00011110

6.

7. (a) 2 (b) 4 (c) 8

8.

9. (a) 7 (b) 9 (c) 16

10.

11. (a) CF57 (b) 5CAD (c) 93EB

12.

13. (a) 1110 0101 1011 0110 1010 1110 1101 0111

    (b) 1011 0110 1001 0111 1100 0111 1010 0001

    (c) 0010 0011 0100 1011 0110 1101 1001 0010

14.

15. (a) 58 (b) 447 (c) 16534

16.

17. (a) FFE6 (b) FE3C

18.

19. (a) +31915 (b) –16093

20.

21. (a) –75 (b) +42 (c) –16

22.

23. (a) 11111011 (b) 11011100 (c) 11110000

24.

25. 58h and 88d

26.

27. To handle international character sets that require more than 256 codes.

28.

29. $+2^{255} - 1$

1.4 Boolean Operations

1. (NOT X) OR Y

2.

3. T

4.

5. T

6.

7. Truth table:

A	B	¬A	¬B	¬A ∧ ¬B
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	F	F	F

8.

9. 2 bits, producing the following values: 00, 01, 10, 11

2 IA-32 Processor Architecture

2.1 General Concepts

1. Control Unit, Arithmetic Logic Unit, and the clock.

2.

3. Conventional memory is outside the CPU, and it responds more slowly to access requests. Registers are hard-wired inside the CPU.

4.

5. Fetch memory operands, store memory operands

6.

7. Executing processor stages in parallel, making possible the overlapped execution of machine instructions.

8.

9. 12 cycles (5 + (8 – 1)

10.

11. 15 clock cycles (5 + 10)

12.

13. The OS executes a branch (like a GOTO) to the first machine instruction in the program.
14.
15. The OS scheduler determines how much time to allot to each task, and it switches between tasks.
16.

2.2 IA-32 Processor Architecture

1. Real-address mode, Protected mode, and System Managment mode
2.
3. CS, DS, SS, ES, FS, GS
4.
5. EBP
6.
7. Carry
8.
9. Sign
10.
11. 80 bits
12.
13. The Pentium
14.
15. CISC means *complex instruction set.* A large collection of instructions, some of which perform sophisticated operations that might be typical of a high-level language.
16.

2.3 IA-32 Memory Management

1. 4 GB (0 to FFFFFFFFh)
2.
3. linear (absolute)

4.

5. 0CFF0h

6.

7. SS register

8.

9. Global descriptor table

10.

11. This is an open-ended question, of course. It is a fact that MS-DOS first had to run on the 8086/8088 processors, which only supported Real-address mode. When later processors came out which supported Protected mode, my guess is that Microsoft wanted MS-DOS to continue to run on the older processors. Otherwise, customers with older computers would refuse to upgrade to new versions of MS-DOS.

12.

2.4 Components of an IA-32 Microcomputer

1. External cache memory is high-speed (static RAM) that is connected to the CPU via a special bus that permits data transfer at higher speeds than conventional memory.

2.

3. The 8259 is the interrupt controller chip, sometimes called PIC, that schedules hardware interrupts and interrupts the CPU.

4.

5. A beam of electrons illuminates phosphorus dots on the screen called pixels. Starting at the top of the screen, the gun fires electrons from the left side to the right in a horizontal row, briefly turns off, and returns to the left side of the screen to begin a new row. Horizontal retrace refers to the time period when the gun is off between rows. When the last row is drawn, the gun turns off (called the vertical retrace) and moves to the upper left corner of the screen to start all over.

6.

7. Static RAM

8.

9. Upstream and downstream

10.

2.5 Input-Output System

1. The application program level
2.
3. New devices are invented all the time, with capabilities that were often not anticipated when the BIOS was written.
4.
5. The operating system, BIOS, and hardware levels.
6.
7. No. The same BIOS would work for both operating systems. Many computer owners install two or three operating systems on the same computer. They would certainly not want to change the system BIOS every time they rebooted the computer!

3 Assembly Language Fundamentals

3.1 Basic Elements of Assembly Language

1. h,q,o,d,b,r,t,y
2.
3. No (they have the same precedence)
4.
5. Real number constant: +3.5E-02
6.
7. directives
8.
9. True
10.
11. False
12.
13. label, mnemonic, operand(s), comment
14.

15. True
16.
17. Because the addresses coded in the instructions would have to be updated whenever new variables were inserted before existing ones.

3.2 Example: Adding Three Integers

1. The INCLUDE directive copies necessary definitions and setup information from the *Irvine32.inc* text file. The data from this file is inserted into the data stream read by the assembler.
2.
3. code, data, and stack.
4.
5. The **exit** statement
6.
7. The ENDP directive
8.
9. PROTO declares the name of a procedure that is called by the current program.

3.3 Assembling, Linking, and Running Programs

1. Object (.OBJ) and listing (.LST) files.
2.
3. True
4.
5. Executable (.EXE) and map (.MAP).
6.
7. The /Zi option
8.
9. There are two many to mention here, but you can view their names by opening Kernel32.lib using the TextPad editor supplied on the book's CD-ROM. The file will display in hexadecimal. Scroll down to offset 1840h, and look at the various function names listed from that point on.

10.

3.4 Defining Data

1. var1 SWORD ?
2.
3. var3 SBYTE ?
4.
5. SDWORD
6.
7. wArray WORD 10,20,30
8.
9. dArray DWORD 50 DUP(?)
10.
11. bArray BYTE 20 DUP(0)
12.

3.5 Symbolic Constants

1. BACKSPACE = 08h
2.
3. ArraySize = ($ – myArray)
4.
5. PROCEDURE TEXTEQU <PROC>
6.
7. SetupESI TEXTEQU <mov esi,OFFSET myArray>

4 Data Transfers, Addressing, and Arithmetic

4.1 Data Transfer Instructions

1. Register, immediate, and memory
2.
3. False
4.
5. A 32-bit register or memory operand
6.
7. (a) not valid (b) valid (c) not valid (d) not valid (e) not valid (f) not valid (g) valid (h) not valid
8.
9. (a) 1000h (b) 3000h (c) FFF0h (d) 4000h
10.

4.2 Addition and Subtraction

1. inc val2
2.
3. Code:

```
mov ax,val4
sub val2,ax
```

4.
5. Both flags will be set.
6.
7. Code example:

```
mov ax,val2
neg ax
add ax,bx
sub ax,val4
```

8.
9. Yes

10.

11. No

12.

4.3 Data-Related Operators and Directives

1. False

2.

3. True

4.

5. True

6.

7. (a) 1 (b) 4 (c) 4 (d) 2 (e) 4 (f) 8 (g) 5

8.

9. mov al, BYTE PTR myWords+1

10.

11. Data directive:

```
myWordsD LABEL DWORD
myWords WORD 3 DUP(?),2000h
.data
mov eax,myWordsD
```

12.

4.4 Indirect Addressing

1. False

2.

3. False

4.

5. True - (the PTR operator is required)

6.

7. (a) 10h (b) 40h (c) 003Bh (d) 3 (e) 3 (f) 2

8.

4.5 JMP and LOOP Instructions

1. True
2.
3. 4,294,967,296 times
4.
5. True
6.
7. ECX
8.
9. This is a trick! The program does not stop, because the first LOOP instruction decrements ECX to zero. The second LOOP instruction decrements ECX to FFFFFFFFh, causing the outer loop to repeat.
10.

5 Procedures

5.1 Introduction

(no review questions)

5.2 Linking to an External Library

1. False - (it contains object code)
2.
3. Code example:

```
call MyProc
```

4.
5. Kernel32.lib
6.
7. %1

5.3 The Book's Link Library

1. RandomRange procedure
2.
3. Code example:

```
mov  eax,700
call Delay
```

4.
5. Gotoxy procedure
6.
7. PROTO statements (procedure prototypes) and constant definitions. (There are also text macros, but they are not mentioned in this chapter.)
8.
9. EDX contains the offset of an array of bytes, and ECX contains the maximum number of characters to read.
10.
11. Code example:

```
.data
str1 BYTE "Enter identification number: ",0
idStr BYTE 15 DUP(?)
.code
   mov  edx,OFFSET str1
   call WriteString
   mov  edx,OFFSET idStr
   mov  ecx,(SIZEOF idStr) - 1
   call ReadString
```

5.4 Stack Operations

1. SS and ESP
2.
3. LIFO stands for "last in, first out". The last value pushed into the stack is the first value popped out from the stack.
4.
5. True

6.

7. True

8.

9. PUSHAD

10.

11. POPFD

12.

5.5 Defining and Using Procedures

1. True

2.

3. Execution would contine beyond the end of the procedure, possibly into the beginning of another procedure. This type of programming bug is often difficult to detect!

4.

5. False - (it pushes the offset of the instruction *following* the call)

6.

7. True

8.

9. True

10.

11. True - (it also receives a count of the number of array elements)

12.

13. False

14.

15. The following statements would have to be modified:

```
add eax,[esi]   becomes -->  add ax,[esi]
add esi,4       becomes -->  add esi,2
```

5.6 Program Design Using procedures

1. functional decomposition, or top-down design
2.
3. A stub program contains all of its important procedures, but the procedures are either empty or nearly empty.
4.
5. The following statements would have to be modified:

```
mov [esi],eax   becomes -->   mov [esi],ax
add esi,4       becomes -->   add esi,2
```

6.

6 Conditional Processing

6.1 Introduction

(no review questions)

6.2 Boolean and Comparison Instructions

1. (a) 00001011 (b) 01001000 (c) 01101111 (d) 10100011
2.
3. (a) CF=0, ZF=0, SF=0

   (b) CF=0, ZF=0, SF=0

   (c) CF=1, ZF=0, SF=1
4.
5. `or ax,0FF00h`
6.
7. `test eax,1        ; (low bit set if eax is odd)`
8.

6.3 Conditional Loops

1. JA, JNBE, JAE, JNB, JB, JNAE, JBE, JNA
2.
3. JECXZ
4.
5. No (JB uses unsigned operands, whereas JL uses signed operands.)
6.
7. JL
8.
9. Yes
10.
11. Code:

```
cmp dx,cx
jbe L1
```

12.
13. Code:

```
and al,11111100b
jz  L3
jmp L4
```

6.4 Conditional Loop Instructions

1. False
2.
3. True
4.
5. If a matching value were not found, ESI would end up pointing beyond the end of the array. This could cause data to be corrupted if ESI were dereferenced and used to modify memory.

6.5 Conditional Structures

(We will assume that all values are unsigned in this section).

1. Code example:

```
     cmp bx,cx
     jna next
     mov X,1
  next:
```

2.

3. Code example:

```
     cmp val1,cx
     jna L1
     cmp cx,dx
     jna L1
     mov X,1
     jmp next
  L1: mov X,2
  next:
```

4.

5. Code example:

```
     cmp bx,cx              ; bx > cx?
     jna L1                 ; no: try condition after OR
     cmp bx,dx              ; yes: is bx > dx?
     jna L1                 ; no: try condition after OR
     jmp L2                 ; yes: set X to 1
  ;-----------------OR(dx > ax) -----------------------
  L1: cmp dx,ax             ; dx > ax?
     jna L3                 ; no: set X to 2
  L2: mov X,1               ; yes:set X to 1
     jmp next               ; and quit
  L3: mov X,2               ; set X to 2
  next:
```

6.6 Application: Finite-State Machines

1. A directed graph (also known as a *diagraph*).

2.

3. Each edge is a transition from one state to another, caused by some input.

4.

5. An infinite number of digits.

6.

7. No. The proposed FSM would permit a signed integer to consist of only a plus (+) or minus (–) sign. The FSM in Section 6.6.2 would not permit that.

8.

6.7 Using the .IF Directive (Optional)

(no review questions)

7 Integer Arithmetic

7.1 Introduction

(no review questions)

7.2 Shift and Rotate Instructions

1. ROL

2.

3. SAR

4.

5. Code example:

```
    shr al,1                ; shift AL into Carry flag
    jnc next                ; Carry flag set?
    or  al,80h              ; yes: set highest bit
next:                       ; no: do nothing
```

6.

7. `shl eax,4`

8.

9. `ror dl,4` (or: `rol dl,4`)

10.

11. (a) 6Ah (b) EAh (c) FDh (d) A9h

12.

13. Code example:

```
    shr ax,1                ; shift AX into Carry flag
    rcr bx,1                ; shift Carry flag into BX
```

```
; Using SHRD:
   shrd bx,ax,1
```

14.

7.3 Shift and Rotate Applications

1. This problem requires us to start with the high-order byte and work our way down to the lowest byte:

```
byteArray BYTE 81h,20h,33h
.code
shr byteArray+2,1
rcr byteArray+1,1
rcr byteArray,1
```

2.

3. The multiplier (24) can be factored into 16 * 8:

```
mov ebx,eax                ; save a copy of eax
shl eax,4                  ; multiply by 16
shl ebx,3                  ; multiply by 8
add eax,ebx                ; add the products
```

4.

5. Change the instruction at label L1 to: `shr eax,1`

6.

7.4 Multiplication and Division Instructions

1. The product is stored in registers that are twice the size of the multiplier and multiplicand. If you multiply 0FFh by 0FFh, for example, the product (FE01h) easily fits within 16 bits.

2.

3. With IMUL, the Carry and Overflow flags are set when the upper half of the product is not a sign extension of the lower half of the product.

4.

5. AX

6.

7. Code example:

```
mov ax,dividendLow
cwd                        ; sign-extend dividend
```

```
mov bx,divisor
idiv bx
```

8.

9. AX = 0306h

10.

11. The DIV will cause a divide overflow, so the values of AX and DX cannot be determined.

12.

13. Code example:

```
mov ax,-276
cwd                         ; sign-extend AX into DX
mov bx,10
idiv bx
mov val1,ax                 ; quotient
```

14.

15. Implement the signed expression: val1 = (val2 / val3) * (val1 + val2).

```
mov eax,val2
cdq                         ; extend EAX into EDX
idiv val3                   ; EAX = quotient
mov ebx,val1
add ebx,val2
imul ebx
mov val1,eax                ; lower 32 bits of product
```

(You can substitute any 32-bit general-purpose register for EBX in this example.)

7.5 Extended Addition and Subtraction

1. The ADC instruction adds both a source operand and the Carry flag to a destination operand.

2.

3. EAX = C0000000h, EDX = 00000010h

4.

5. DX = 0016h

6.

8 Advanced Procedures

8.1 Introduction

(no review questions)

8.2 Local Variables

1. (a) automatically restricts access to statements within a single procedure; (b) local variables make efficient use of memory; (c) you can use the same variable name in multiple procedures.
2.
3. False; you can define many more than three.
4.
5. Declaration: `LOCAL pArray:PTR DWORD`
6.
7. Declaration: `LOCAL pwArray:PTR WORD`
8.
9. Declaration: `LOCAL myArray[20]:DWORD`

8.3 Stack Parameters

1. True
2.
3. False
4.
5. False
6.
7. True
8.
9. True - when the immediate value is dereferenced, it will probably point to an invalid memory location.
10.

11. Declaration:

```
MultArray PROC ptr1:PTR DWORD,
  ptr2:PTR DWORD,
  count:DWORD          ; (may be byte, word, or dword)
```

12.

13. It uses input-output parameters.

14.

15. The following code is shown in the listing file, when the assembler's /Sg option is used. It shows that count, the second argument, was pushed on the stack first before the offset of myArray:

```
INVOKE SumArray, ADDR myArray, count
  push   +00000000Ah
  push   OFFSET myArray
  call   SumArray
```

(For more information about the assembler's command-line options, see Appendix D.)

8.4 Stack Frames

1. True

2.

3. True - (each stack position in Protected mode uses 4 bytes)

4.

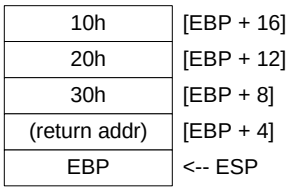
5. One code segment, and one data segment. All code and data are near, which means they can be reached using only 16-bit offsets.

6.

7. The C option preserves the case of identifiers and prepends a leading underscore to external names. The PASCAL option converts all identifiers to upper case.

8.

9. Stack frame diagram:

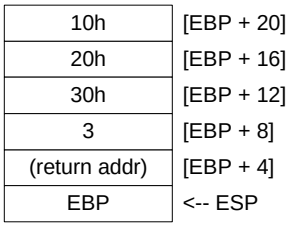


10.

11. LEA can return the offset of an indirect operand; it is particularly useful for obtaining the offset of a stack parameter.

12.

13. The C calling convention, because it specifies that arguments must be pushed on the stack in reverse order, makes it possible to create a procedure/function with a variable number of parameters. The last parameter pushed on the stack can be a count specifying the number of parameters already pushed on the stack. In the following diagram, for example, the count value is located at [EBP + 8]:



8.5 Recursion

1. False

2.

3. The following code executes after the recursive call:

```
ReturnFact:
    mov ebx,[ebp+8]
    mul ebx
L2: pop ebp
    ret 4
```

4.

5. 12! uses 156 bytes of stack space. *Rationale:* From Figure 8-1, we see that when $n = 0$, 12 stack bytes are used (3 entries). When $n = 1$, 24 bytes are used. When $n = 2$, 36 bytes are used. Therefore, the amount of stack space required for n! is $(n+1)*12$.
6.

8.6 Creating Multimodule Programs

1. True
2.
3. True
4.

9 Strings and Arrays

9.1 Introduction

(no review questions)

9.2 String Primitive Instructions

1. EAX
2.
3. (E)DI
4.
5. Repeat while ZF = 1
6.
7. 2
8.
9. one byte beyond the matching character
10.

9.3 Selected String Procedures

1. False (it stops when the null terminator of the shorter string is reached)
2.

3. False

4.

5. 1 (set)

6.

7. The digit is unchanged.

8.

9. The length would be: $(EDI_{final} - EDI_{initial}) - 1$

9.4 Two-Dimensional Arrays

1. Any general-purpose 32-bit registers.

2.

3. array[ebx + esi]

4.

5. Code example:

```
mov esi,2                    ; row
mov edi,3                    ; column
mov eax,[esi*16 + edi*4]
```

6.

7. No (the flat memory model has only one segment)

9.5 Searching and Sorting Integer Arrays

1. n – 1 times

2.

3. No: it decreases by 1 each

4.

5. $(\log_2 128) + 1 = 8$

6.

7. EDX and EDI were already compared

8.

10 Structures and Macros

10.1 Structures

1. Structures are essential whenever you need to pass a large amount of data between procedures. One variable can be used to hold all the data.
2.
3. temp1 MyStruct <>
4.
5. temp3 MyStruct <, 20 DUP(0)>
6.
7. mov ax,array.field1
8.
9. 82
10.
11. TYPE MyStruct.field2 (or: SIZEOF Mystruct.field2)
12.
13. Code example:

```
.data
time SYSTEMTIME <>
.code
mov ax,time.wHour
```

14.
15. Code example (initializes an array of Triangle structures):

```
.data
ARRAY_SIZE = 5
triangles Triangle ARRAY_SIZE DUP(<>)
.code
   mov  ecx,ARRAY_SIZE
   mov  esi,0
L1:mov  eax,11
   call RandomRange
   mov triangles[esi].Vertex1.X, ax
   mov  eax,11
   call RandomRange
```

```
    mov triangles[esi].Vertex1.Y, ax
    add esi,TYPE Triangle
    loop L1
```

10.2 Macros

1. False
2.
3. Macros can have parameters
4.
5. True
6.
7. To permit the use of labels in a macro that is invoked more than once by the same program.
8.
9. Code example:

```
OutChar MACRO aChar
   push eax
   mov  al,aChar
   call WriteChar
   pop  eax
ENDM
```

10.
11. Nested macro definition:

```
;-----------------------------------------------------
mAskInteger MACRO promptString
;
; Displays a prompt string, inputs an integer
; from the user, and returns its value in EAX.
;-----------------------------------------------------
   mWrite promptString
   call ReadInt
ENDM
; Sample call:
mAskInteger "Enter an integer between 1 and 50: "
```

12.
13. Code example:

```
mWriteStr namePrompt
1   push edx
1   mov edx,OFFSET namePrompt
1   call WriteString
1   pop edx
```

14.

15. Code example:

```
;-----------------------------------------------
mDumpMemx MACRO varName
;
; Displays a variable in hexadecimal, using the
; variable's attributes to determine the number
; of units and unit size.
;-----------------------------------------------
   push ebx
   push ecx
   push esi
   mov  esi,OFFSET varName
   mov  ecx,LENGTHOF varName
   mov  ebx,TYPE varName
   call DumpMem
   pop  esi
   pop  ecx
   pop  ebx
ENDM
; Sample calls:

.data
array1 BYTE  10h,20h,30h,40h,50h
array2 WORD  10h,20h,30h,40h,50h
array3 DWORD 10h,20h,30h,40h,50h
.code
mDumpMemx array1
mDumpMemx array2
mDumpMemx array3
```

10.3 Conditional-Assembly Directives

1. The IFB directive is used to check for blank macro parameters.

2.

3. EXITM

4.

5. The IFDEF returns true if a symbol has already been defined.
6.
7. Code example:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call Crlf
ENDM
```

8.
9. Code example:

```
mCopyWord MACRO intVal
    IF (TYPE intVal) EQ 2
      mov ax,intVal
    ELSE
      ECHO Invalid operand size
    ENDIF
ENDM
```

10.
11. The substitution (&) operator resolves ambiguous references to parameter names within a macro.
12.
13. The expansion operator (%) expands text macros or converts constant expressions into their text representations.
14.
15. Code example:

```
mLocate -2,20
;(no code generated because xval < 0)

mLocate 10,20
1  mov bx,0
1  mov ah,2
1  mov dh,20
1  mov dl,10
1  int 10h

mLocate col,row
1  mov bx,0
1  mov ah,2
1  mov dh,row
```

```
1  mov dl,col
1  int 10h
```

10.4 Defining Repeat Blocks

1. The WHILE directive repeats a statement block based on a boolean expression.
2.
3. The FOR directive repeats a statement block by iterating over a list of symbols.
4.
5. FORC
6.
7. Code example:

```
mRepeat MACRO 'X',50
   mov cx,50
??0000: mov ah,2
   mov dl,'X'
   int 21h
   loop ??0000

mRepeat MACRO AL,20
   mov cx,20
??0001: mov ah,2
   mov dl,AL
   int 21h
   loop ??0001

mRepeat MACRO byteVal,countVal
   mov cx,countVal
??0002: mov ah,2
   mov dl,byteVal
   int 21h
   loop ??0002
```

8.

11 32-Bit Windows Programming

11.1 Win32 Console Programming

1. /SUBSYSTEM:CONSOLE
2.
3. False
4.
5. True
6.
7. GetStdHandle
8.
9. Example from the *ReadConsole.asm* program in Section 11.1.3.1:

```
INVOKE ReadConsole, stdInHandle, ADDR buffer,
   BufSize - 2, ADDR bytesRead, 0
```

10.
11. Example from the *Console1.asm* program in Section 11.1.4.3:

```
INVOKE WriteConsole,
   consoleHandle,          ; console output handle
   ADDR message,           ; string pointer
   messageSize,            ; string length
   ADDR bytesWritten,      ; returns num bytes written
   0                       ; not used
```

12.
13. Calling **CreateFile** to create a new file:

```
INVOKE CreateFile,
   ADDR filename,
   GENERIC_WRITE,
   DO_NOT_SHARE,
   NULL,
   CREATE_ALWAYS,
   FILE_ATTRIBUTE_NORMAL,
   0
```

14.
15. Calling **WriteFile**:

```
INVOKE WriteFile,          ; write text to file
   fileHandle,             ; file handle
   ADDR buffer,            ; buffer pointer
   bufSize,                ; number of bytes to write
   ADDR bytesWritten,      ; number of bytes written
   0                       ; overlapped execution flag
```

16.

17. SetConsoleTitle

18.

19. SetConsoleCursorInfo

20.

21. WriteConsoleOutputAttribute

22.

11.2 Writing a Graphical Windows Application

Note: most of these questions can be answered by looking in *GraphWin.inc*, the include file supplied with MASM in the INCLUDE subdirectory.

1. A POINT structure contains two fields, ptX and ptY, that describe the X and Y coordinates (in pixels) of a point on the screen.

2.

3. *lpfnWndProc* is a pointer to a function in an application program that receives and processes event messages triggered by the user.

4.

5. *hInstance* holds a handle to the current program instance. Each programming running under MS-Windows is automatically assigned a handle by the operating system when the program is loaded into memory.

6.

7. Calling MessageBox:

```
INVOKE MessageBox, hMainWnd, ADDR GreetText,
   ADDR GreetTitle, MB_OK
```

8.

9. Icon constants (choose any two):

```
MB_ICONHAND, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONASTERISK
```

10.

11. The **WinProc** procedure receives and processes all event messages relating to a window. It decodes each message, and if the message is recognized, carries out application-oriented (or application-specific) tasks relating to the message.

12.

13. The ErrorHandler procedure, which is optional, is called if the system reports an error during the registration and creation of the program's main window.

14.

15. The message box appears before the main window closes.

11.3 IA-32 Memory Management

1. (a) Multitasking permits multiple programs (or tasks) to run at the same time. The processor divides up its time between all of the running programs.

   (b) Segmentation provides a way to isolate memory segments from each other. This permits multiple programs to run simultaneously without interfering with each other.

2.

3. True

4.

5. False

6.

7. A linear address is a 32-bit integer ranging between 0 and FFFFFFFFh, which refers to a memory location. The linear address may also be the physical address of the target data, if a feature called paging is disabled.

8.

9. The linear address is automatically a 32-bit physical memory address.

10.

11. The LDTR register

12.

13. One

14.

15. Choose any four from the following list: Base address, privilege level, segment type, segment present flag, granularity flag, segment limit.

16.

17. The Table field of a linear address (see Figure 11-4).

18.

12 High-Level Language Interface

12.1 Introduction

1. The naming convention used by a language refers to the rules or characteristics regarding the naming of variables and procedures.

2.

3. No, because the procedure name will not be found by the linker.

4.

5. C and C++ are case-sensitive, so they will only execute calls to procedures that are named in the same fashion.

6.

12.2 Inline Assembly Code

1. Inline assembly code is assembly language source code that is inserted directly into high-level language programs. The inline qualifier in C++, on the other hand, asks the C++ compiler to insert the body of a function directly into the program's compiled code, to avoid the extra execution time it would take to call and return from the function. (Note: answsering this question requires some knowledge of the C++ language, that is not found in the current book.)

2.

3. Examples of comments (select any two):

```
mov esi,buf            ; initialize index register
mov esi,buf            // initialize index register
mov esi,buf            /* initialize index register */
```

4.

5. Yes

6.

7. No

8.

9. Use the LEA instruction.

10.

11. The SIZE operator returns the product of TYPE (4) * LENGTH.

12.3 Linking to C++ Programs

1. X will be pushed last.

2.

3. If name decoration is in effect, an external function name generated by the C++ compiler will not be the same as the name of the called procedure written in assembly language. Understandably, the assembler does not have any knowledge of the name decoration rules used by C++ compilers.

4.

5. INT = 2, enum = 1, float = 4, double = 8.

6.

7. What SHLD instruction? Actually, there was a line in the LongRandom code originally, that read:

   ```
   shld edx,eax,16
   ```

   So using that instruction as the basis for the question, we can say that the equivalent statements would be:

   ```
   mov ecx,16
   L1: shl eax,1
       rcl edx,1
       Loop L1
   ```

   The current version of this procedure uses the follwoign statement to rotate out the lowest digit of EAX, which prevents a recurring pattern when generating sequences of small random numbers:

   ```
       ror eax,8
   ```

8.

13 16-Bit MS-DOS Programming

13.1 MS-DOS and the IBM-PC

1. 9FFFFh
2.
3. 00400h
4.
5. Suppose a program was named myProg.exe. The following would redirect its output to the default printer:

   ```
   myProg > prn
   ```

6.
7. An interrupt service routine (also called an *interrupt handler*) is an operating system procedure that (1) provides basic services to application programs, and (2) handles hardware events. For more details, see Section 16.4.
8.
9. See the 4 steps in Section 13.1.4.1.
10.
11. 10h
12.

13.2 MS-DOS Function Calls (INT 21h)

1. AH
2.
3. Functions 2 and 6 both write a single character.
4.
5. Function 40h
6.
7. Function 3Fh
8.

9. Functions 2Bh (set system date) and 2Dh (set system time).
10.

13.3 Standard MS-DOS File I/O Services

1. Device Handles: 0 = Keyboard (standard input), 1 = Console (standard output), 2 = Error output, 3 = Auxiliary device (asynchronous), 4 = Printer
2.
3. Parameters for function 716Ch

```
AX = 716Ch
BX = access mode (0 = read, 1 = write, 2 = read/write)
CX = attributes (0 = normal, 1 = read only, 2 = hidden,
 3 = system, 8 = volume ID, 20h = archive)
DX = action (1 = open, 2 = truncate, 10h = create)
DS:SI = segment/offset of filename
DI = alias hint (optional)
```

4.
5. Reading a binary array from a file is best done with INT 21h Function 3Fh. The following parameters are required

```
AH = 3Fh
BX = open file handle
CX = maximum bytes to read
DS:DX = address of input buffer
```

6.
7. The only difference is the value in BX. When reading from the keyboard, BX is set to the keyboard handle (0). When reading from a file, BX is set to the handle of the open file.
8.
9. Code example (BX already contains the file handle):

```
mov ah,42h                ; move file pointer
mov al,0                  ; method: offset from beginning
mov cx,0                  ; offsetHi
mov dx,50                 ; offsetLo
int 21h
```

14 Disk Fundamentals

14.1 Disk Storage Systems

1. True
2.
3. Cylinder
4.
5. 512
6.
7. The read/write heads must jump over other cylinders, wasting time and increasing the probablility that errors will occur.
8.
9. The average amount of time required to move the read/write heads between tracks.
10.
11. Up to three primary partitions if only one extended partition exists, or up to four primary partitions if there are no extended partitions.
12.
13. The disk partition table, and a program that locates a single partition’s boot sector and runs another program that loads the operating system.
14.
15. System

14.2 File Systems

1. True
2.
3. False - all systems, including NTFS, require at least one cluster to store a file
4.
5. False
6.

7. FAT32 and NTFS
8.
9. NTFS
10.
11. NTFS
12.
13. Boot record, file allocation table, root directory, and the data area.
14.
15. Two 8 KB clusters would be required, for a total of 16,384 bytes. The number of wasted bytes would be (16,384 – 8,200), or 8,184 bytes.
16.

14.3 Disk Directory

1. True
2.
3. False - (it contains the starting *cluster* number)
4.
5. 32
6.
7. The status bytes and their descriptions are listed in Table 14-5.
8.
9. The first byte of the entry is 4*x*h, where x indicates the number of long filename entries to be used for the file.
10.
11. Actually, there are three new fields: Last access date, create date, and create time
12.

14.4 Reading and Writing Disk Sectors (7305h)

1. True
2.
3. Parameters:

```
AX: 7305h
DS:BX: Segment/offset of a DISKIO structure variable
CX: 0FFFFh
DL: Drive number (0 = default, 1 = A, 2 = B, 3 = C, etc.)
SI: Read/write flag
```

4.
5. The Carry flag is set if function 7305h cannot read the requested sector, and the program displays an error message. (Remember that you cannot test this program under Windows NT, 2000, or XP.)

14.5 System-Level File Functions

1. Function 7303h
2.
3. Function 39h (create subdirectory) and Function 3Bh (set current directory).
4.

15 BIOS-Level Programming

15.1 Introduction

(no review questions)

15.2 Keyboard Input with INT 16h

1. INT 16h is best
2.
3. INT 9h reads the keyboard input port, retrieves the keyboard scan code and produces the corresponding ASCII code. It inserts both in the keyboard typeahead buffer.
4.
5. Function 10h

6.

7. No

8.

9. Bit 4 (see Table 15-2)

10.

11. To check for other keyboard keys, add more CMP and JE instructions after the existing ones currently in the loop. Suppose we wanted to check for the ESC, F1, and Home keys:

```
L1: .
    .
    cmp ah,1                ; ESC key's scan code?
    je  quit                ; yes: quit
    cmp ah,3Bh              ; F1 function key?
    je  quit                ; yes: quit
    cmp ah,47h              ; Home key?
    je  quit                ; yes: quit
    jmp L1                  ; no: check buffer again
```

15.3 Video Programming with INT 10h

1. MS-DOS level, BIOS level, and Direct video level.

2.

3. In MS-Windows, there are two ways to switch into full-screen mode:

- Create a shortcut to the program's EXE file. Then open the Properties dialog for the shortcut, select the Screen properties, and select Full-screen mode.
- Open a Command window from the Start menu, then press Alt-Enter to switch to full screen mode. Using the CD (change directory) command, navigate to your EXE file's directory, and run the program by typing its name. Alt-Enter is a toggle, so if you press it again, it will return the program to Window mode.

4.

5. ASCII code and attribute (2 bytes)

6.

7. Background: bits 4-7. Foreground: bits 0-3.

8.

9. Function 06h

10.

11. Function 01h
12.
13. AH = 2, DH = row, DL = column, and BH = video page
14.
15. AH = 6, AL = number of lines to scroll, BH = attribute of scrolled lines, CH & CL = upper-left window corner, and DH & DL = lower right window corner.
16.
17. Function 10h, Subfunction 03h (set AH to 10h, and AL to 03h)
18.
19. Every pixel on the screen is made of three colors: red, green, and blue. Dogs are color blind, so they cannot see pixels made from colors. I've tried displaying a picture of a cat on the screen, but my own dog seems not to notice.

15.4 Drawing Graphics Using INT 10h

1. Function 0Ch
2.
3. It's very slow.
4.
5. Mode 6Ah
6.
7. a. (350,150) b. (375,225) c. (150,400)

15.5 Memory-Mapped Graphics

1. False - (each byte corresponds to 1 pixel)
2.
3. Mode 13h maps each pixel's integer value into a table of colors called a palette.
4.
5. Each entry in the palette consists of three separate integer values (0–63) known as RGB (red, green, blue). Entry 0 in the color palette controls the screen's background color.
6.

7. (63,63,63)
8.
9. Code example:

```
; Set screen background color to bright green.
mov dx,3c8h               ; video paletter port
mov al,0                  ; index 0 (background color)
out dx,al
mov dx,3c9h               ; colors go to port 3C9h
mov al,0                  ; red
out dx,al
mov al,63                 ; green (intensity = 63)
out dx,al
mov al,0                  ; blue
out dx,al
```

10.

15.6 Mouse Programming

1. Function 0
2.
3. Functions 1 and 2
4.
5. Function 3
6.
7. Function 4
8.
9. Function 5
10.
11. Function 6
12.
13. Code example:

```
    mov ax,8              ; set vertical limits
    mov cx,200            ; lower limit
    mov dx,400            ; upper limit
    int 33h
```

Note: in the first printing of the book, the box that describes Function 8 had a few errors. AX must be set to 8, and INT 33h should be called *after* CX and DX have been set.

14.

15. Assuming that character cells are 8 pixels by 8 pixels, the X, Y coordinates values would be (8 * 20), (8 * 10). The cell will be at position 160, 80.

16.

16 Expert MS-DOS Programming

16.1 Introduction

(no review questions)

16.2 Defining Segments

1. Declares the beginning of a segment.

2.

3. The ASSUME directive makes it possible for the assembler to calculate the offsets of labels and variables at assembly time. A directive such as:

```
assume DS:myData
```

   says to the assembler, "assume that from this point on, all references to data labels (via DS) will be located in the segment named **myData**."

4.

5. PRIVATE, PUBLIC, MEMORY, STACK, COMMON, and AT

6.

7. The combine type tells the linker how to combine segments having the same name.

8.

9. A segment's class type provides another way of combining segments, in particular, those with different names. Segments having the same class type are loaded together, although they may be listed in a different order in the program source code.

10.

11. The third segment will also begin at address 1A060h.

16.3 Runtime Program Structure

1. The command processor checks to see if there is filename with extension COM in the current directory. If a file is found, it is executed. If a matching file is not found, see Section 16.3 for a description of the subsequent steps.

2.

3. Application programs loaded into the lowest 640K of memory. They are transient because when the finish executing, they are automatically unloaded from memory.

4.

5. At offset 2Ch inside the program segment prefix area.

6.

7. Tiny

8.

9. 64 Kilobytes

10.

11. One

12.

13. The ORG directive assigns a specific offset to the very next label or instruction following the directive. The addresses of all subsequent labels are calculated from that point onward. COM programs, for example, always have ORG 100h at the beginning of the program code, so the first executable instruction will be located at offset 100h.

14.

15. DS and ES point to the program segment prefix area of the program.

16.

17. The EXEMOD program displays statistics about a program's memory usage, and also permits many settings in the EXE header to be modified.

18.

16.4 Interrupt Handling

1. It displays a message on the screen "Abort, retry, or ignore?", and terminates the current program.
2.
3. At address 0000:0040h, because 0040h equals 10h * 4
4.
5. The CLI (clear interrupt flag) instruction
6.
7. IRQ 0 has highest priority
8.
9. INT 9h
10.
11. Functions 25h and 35h
12.
13. A terminate and stay resident (TSR) program leaves part of itself in memory when it exits. This is accomplished by calling INT 21h function 31h.
14.
15. Rather than executing an IRET instruction when it finishes, it can instead execute a JMP to the address that was previously stored in the interrupt vector.
16.
17. Ctrl + Alt + Right shift + Del