Lecture No.13

Lecture Outlines

5.4 The Irvine32 Library (Continued)

- 5.4.3 Individual Procedure Descriptions (Continued)
- 5.4.4 Library Test Programs

5.4.3 Individual Procedure Descriptions (Continued)

In this section, we describe how each of the procedures in the Irvine32 library is used. We will omit a few of the more advanced procedures, which will be explained in later chapters.

ReadChar The ReadChar procedure reads a single character from the keyboard and returns the character in the AL register. The character is not echoed in the console window. Sample call:

```
.data
char BYTE ?
.code
call ReadChar
mov  char,al
```

If the user presses an extended key such as a function key, arrow key, Ins, or Del, the procedure sets AL to zero, and AH contains a keyboard scan code. A list of scan codes is shown on the page facing the book's inside front cover. The upper half of EAX is not preserved. The following pseudocode describes the possible outcomes after calling ReadChar:

```
if an extended key was pressed
     AL = 0
     AH = keyboard scan code
else
     AL = ASCII key value
endif
```

ReadDec The ReadDec procedure reads a 32-bit unsigned decimal integer from the keyboard and returns the value in EAX. Leading spaces are ignored. The return value is calculated from all valid digits found until a nondigit character is encountered. For example, if the user enters 123ABC, the value returned in EAX is 123. Following is a sample call:

```
.data
intVal DWORD ?
.code
call ReadDec
mov  intVal,eax
```

ReadDec affects the Carry flag in the following ways:

- If the integer is blank, EAX = 0 and CF = 1
- If the integer contains only spaces, EAX = 0 and CF = 1
- If the integer is larger than $2^{32}-1$, EAX = 0 and CF = 1
- Otherwise, EAX holds the converted integer and CF = 0

ReadFromFile The ReadFromFile procedure reads an input disk file into a memory buffer. When you call ReadFromFile, pass it an open file handle in EAX, the offset of a buffer in EDX, and the maximum number of bytes to read in ECX. When ReadFromFile returns, check the value of the Carry flag: If CF is clear, EAX contains a count of the number of bytes read from the file. But if CF is set, EAX contains a numeric system error code. You can call the WriteWindowsMsg procedure to get a text representation of the error.

In the following example, as many as 5000 bytes are copied from the file into the buffer variable:

```
.data
BUFFER_SIZE = 5000
buffer BYTE BUFFER_SIZE DUP(?)
bytesRead DWORD ?

.code
mov   edx,OFFSET buffer         ; points to buffer
mov   ecx,BUFFER_SIZE           ; max bytes to read
call  ReadFromFile              ; read the file
```

If the Carry flag were clear at this point, you could execute the following instruction:

```
mov  bytesRead,eax              ; count of bytes actually read
```

But if the Carry flag were set, you would call WriteWindowsMsg procedure, which displays a string that contains the error code and description of the most recent error generated by the application:

```
call  WriteWindowsMsg
```

ReadHex The ReadHex procedure reads a 32-bit hexadecimal integer from the keyboard and returns the corresponding binary value in EAX. No error checking is performed for invalid characters. You can use both uppercase and lowercase letters for the digits A through F. A maximum of eight digits may be entered (additional characters are ignored). Leading spaces are ignored. Sample call:

```
.data
hexVal DWORD ?
.code
call ReadHex
mov  hexVal,eax
```

ReadInt The ReadInt procedure reads a 32-bit signed integer from the keyboard and returns the value in EAX. The user can type an optional leading plus or minus sign, and the rest of the number may only consist of digits. ReadInt sets the Overflow flag and display an error message if the value entered cannot be represented as a 32-bit signed integer (range: −2,147,483,648 to +2,147,483,647). The return value is calculated from all valid digits found until a nondigit character is encountered. For example, if the user enters +123ABC, the value returned is +123. Sample call:

```
.data
intVal SDWORD ?
.code
call  ReadInt
mov   intVal,eax
```

ReadKey The ReadKey procedure performs a no-wait keyboard check. In other words, it inspects the keyboard input buffer to see if a key has been pressed by the user. If no keyboard data is found, the Zero flag is set. If a keypress is found by ReadKey, the Zero flag is cleared and AL is assigned either zero or an ASCII code. If AL contains zero, the user may have pressed a special key (function key, arrow key, etc.) The AH register contains a virtual scan code, DX contains a virtual key code, and EBX contains the keyboard flag bits. The following pseudocode describes the various outcomes when calling ReadKey:

```
if no_keyboard_data then
   ZF = 1
else
   ZF = 0
   if AL = 0 then
     extended key was pressed, and AH = scan code, DX = virtual
        key code, and EBX =  keyboard flag bits
   else
     AL = the key's ASCII code
   endif
endif
```

The upper halves of EAX and EDX are overwritten when ReadKey is called.

ReadString The ReadString procedure reads a string from the keyboard, stopping when the user presses the Enter key. Pass the offset of a buffer in EDX and set ECX to the maximum number of characters the user can enter, plus 1 (to save space for the terminating null byte). The procedure returns the count of the number of characters typed by the user in EAX. Sample call:

```
.data
buffer BYTE 21 DUP(0)           ; input buffer
byteCount DWORD ?               ; holds counter
.code
mov   edx,OFFSET buffer         ; point to the buffer
mov   ecx,SIZEOF buffer         ; specify max characters
call  ReadString                ; input the string
mov   byteCount,eax             ; number of characters
```

ReadString automatically inserts a null terminator in memory at the end of the string. The following is a hexadecimal and ASCII dump of the first 8 bytes of **buffer** after the user has entered the string "ABCDEFG":

```
41 42 43 44 45 46 47 00        ABCDEFG
```

The variable **byteCount** equals 7.

SetTextColor The SetTextColor procedure *(Irvine32 library only)* sets the foreground and background colors for text output. When calling SetTextColor, assign a color attribute to EAX. The following predefined color constants can be used for both foreground and background:

black = 0	red = 4	gray = 8	lightRed = 12
blue = 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

Color constants are defined in the *Irvine32.inc* file. To get a complete color byte value, multiply the background color by 16 and add it to the foreground color. The following constant, for example, indicates yellow characters on a blue background:

```
yellow + (blue * 16)
```

The following statements set the color to white on a blue background:

```
mov   eax,white + (blue * 16)   ; white on blue
call  SetTextColor
```

An alternative way to express color constants is to use the SHL operator. You shift the background color leftward by 4 bits before adding it to the foreground color.

```
yellow + (blue SHL 4)
```

The bit shifting is performed at assembly time, so it can only have constant operands. In Chapter 7, you will learn how to shift integers at runtime. You can find a detailed explanation of video attributes in Section 16.3.2.

Str_length The Str_length procedure returns the length of a null-terminated string. Pass the string's offset in EDX. The procedure returns the string's length in EAX. Sample call:

```
.data
buffer BYTE "abcde",0
bufLength DWORD ?
.code
mov   edx,OFFSET buffer         ; point to string
call  Str_length                ; EAX = 5
mov   bufLength,eax             ; save length
```

WaitMsg The WaitMsg procedure displays the message "Press any key to continue. . ." and waits for the user to press a key. This procedure is useful when you want to pause the screen display before data scrolls off and disappears. It has no input parameters. Sample call:

```
call  WaitMsg
```

WriteBin The WriteBin procedure writes an integer to the console window in ASCII binary format. Pass the integer in EAX. The binary bits are displayed in groups of four for easy reading. Sample call:

```
mov   eax,12346AF9h
call  WriteBin
```

The following output would be displayed by our sample code:

```
0001 0010 0011 0100 0110 1010 1111 1001
```

WriteBinB The WriteBinB procedure writes a 32-bit integer to the console window in ASCII binary format. Pass the value in the EAX register and let EBX indicate the display size in bytes (1, 2, or 4). The bits are displayed in groups of four for easy reading. Sample call:

```
mov   eax,00001234h
mov   ebx,TYPE WORD             ; 2 bytes
call  WriteBinB                 ; displays 0001 0010 0011 0100
```

WriteChar The WriteChar procedure writes a single character to the console window. Pass the character (or its ASCII code) in AL. Sample call:

```
mov   al,'A'
call  WriteChar                    ; displays: "A"
```

WriteDec The WriteDec procedure writes a 32-bit unsigned integer to the console window in decimal format with no leading zeros. Pass the integer in EAX. Sample call:

```
mov   eax,295
call  WriteDec                     ; displays: "295"
```

WriteHex The WriteHex procedure writes a 32-bit unsigned integer to the console window in 8-digit hexadecimal format. Leading zeros are inserted if necessary. Pass the integer in EAX. Sample call:

```
mov   eax,7FFFh
call  WriteHex                     ; displays: "00007FFF"
```

WriteHexB The WriteHexB procedure writes a 32-bit unsigned integer to the console window in hexadecimal format. Leading zeros are inserted if necessary. Pass the integer in EAX and let EBX indicate the display format in bytes (1, 2, or 4). Sample call:

```
mov   eax,7FFFh
mov   ebx,TYPE WORD                ; 2 bytes
call  WriteHexB                    ; displays: "7FFF"
```

WriteInt The WriteInt procedure writes a 32-bit signed integer to the console window in decimal format with a leading sign and no leading zeros. Pass the integer in EAX. Sample call:

```
mov   eax,216543
call  WriteInt                     ; displays: "+216543"
```

WriteString The WriteString procedure writes a null-terminated string to the console window. Pass the string's offset in EDX. Sample call:

```
.data
prompt BYTE "Enter your name: ",0
.code
mov   edx,OFFSET prompt
call  WriteString
```

WriteToFile The WriteToFile procedure writes the contents of a buffer to an output file. Pass it a valid file handle in EAX, the offset of the buffer in EDX, and the number of bytes to write in ECX. When the procedure returns, if EAX is greater than zero, it contains a count of the number of bytes written; otherwise, an error occurred. The following code calls WriteToFile:

```
BUFFER_SIZE = 5000
.data
fileHandle   DWORD ?
buffer       BYTE BUFFER_SIZE DUP(?)
```

```
.code
mov  eax,fileHandle
mov  edx,OFFSET buffer
mov  ecx,BUFFER_SIZE
call WriteToFile
```

The following pseudocode describes how to handle the value returned in EAX after calling WriteToFile:

```
if EAX = 0 then
   error occurred when writing to file
   call WriteWindowsMessage to see the error
else
   EAX = number of bytes written to the file
endif
```

WriteWindowsMsg The WriteWindowsMsg procedure writes a string containing the most recent error generated by your application to the Console window when executing a call to a system function. Sample call:

```
call WriteWindowsMsg
```

The following is an example of a message string:

```
Error 2: The system cannot find the file specified.
```

5.4.4 Library Test Programs

Tutorial: Library Test #1

In this hands-on tutorial, you will write a program that demonstrates integer input–output with screen colors.

Step 1: Begin the program with a standard heading:

```
; Library Test #1: Integer I/O (InputLoop.asm)

; Tests the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor,
; WaitMsg, WriteBin, WriteHex, and WriteString procedures.
INCLUDE Irvine32.inc
```

Step 2: Declare a **COUNT** constant that will determine the number of times the program's loop repeats later on. Then two constants, **BlueTextOnGray** and **DefaultColor**, are defined here so they can be used later on when we change the console window colors. The color byte stores the background color in the upper 4 bits, and the foreground (text) color in the lower 4 bits. We have not yet discussed bit shifting instructions, but you can multiply the background color by 16 to shift it into the high 4 bits of the color attribute byte:

```
.data
COUNT = 4
BlueTextOnGray = blue + (lightGray * 16)
DefaultColor = lightGray + (black * 16)
```

Step 3: Declare an array of signed doubleword integers, using hexadecimal constants. Also, add a string that will be used as prompt when the program asks the user to input an integer:

```
arrayD SDWORD 12345678h,1A4B2000h,3434h,7AB9h
prompt BYTE "Enter a 32-bit signed integer: ",0
```

Step 4: In the code area, declare the main procedure and write code that initializes ECX to blue text on a light gray background. The **SetTextColor** method changes the foreground and background color attributes of all text written to the window from this point onward in the program's execution:

```
.code
main PROC
    mov   eax,BlueTextOnGray
    call  SetTextColor
```

In order to set the background of the console window to the new color, you must use the Clrscr procedure to clear the screen:

```
    call  Clrscr                    ; clear the screen
```

Next, the program will display a range of doubleword values in memory, identified by the variable named **arrayD**. The DumpMem procedure requires parameters to be passed in the ESI, EBX, and ECX registers.

Step 5: Assign to ESI the offset of **arrayD**, which marks the beginning of the range we wish to display:

```
    mov   esi,OFFSET arrayD
```

Step 6: EBX is assigned an integer value that specifies the size of each array element. Since we are displaying an array of doublewords, EBX equals 4. This is the value returned by the expression TYPE arrayD:

```
    mov   ebx,TYPE arrayD           ; doubleword = 4 bytes
```

Step 7: ECX must be set to the number of units that will be displayed, using the LENGTHOF operator. Then, when DumpMem is called, it has all the information it needs:

```
    mov   ecx,LENGTHOF arrayD   ; number of units in arrayD
    call  DumpMem               ; display memory
```

The following figure shows the type of output that would be generated by DumpMem:

```
Dump of offset 00405000
-------------------------------
12345678  1A4B2000  00003434  00007AB9
```

Next, the user will be asked to input a sequence of four signed integers. After each integer is entered, it is redisplayed in signed decimal, hexadecimal, and binary.

Step 8: Output a blank line by calling the Crlf procedure. Then, initialize ECX to the constant value COUNT so ECX can be the counter for the loop that follows:

```
call  Crlf
mov   ecx,COUNT
```

Step 9: We need to display a string that asks the user to enter an integer. Assign the offset of the string to EDX, and call the WriteString procedure. Then, call the ReadInt procedure to receive input from the user. The value the user enters will be automatically stored in EAX:

```
L1: mov   edx,OFFSET prompt
    call  WriteString
    call  ReadInt                     ; input integer into EAX
    call  Crlf                        ; display a newline
```

Step 10: Display the integer stored in EAX in signed decimal format by calling the WriteInt procedure. Then call Crlf to move the cursor to the next output line:

```
    call  WriteInt                    ; display in signed decimal
    call  Crlf
```

Step 11: Display the same integer (still in EAX) in hexadecimal and binary formats, by calling the WriteHex and WriteBin procedures:

```
    call  WriteHex                    ; display in hexadecimal
    call  Crlf
    call  WriteBin                    ; display in binary
    call  Crlf
    call  Crlf
```

Step 12: You will insert a Loop instruction that allows the loop to repeat at Label L1. This instruction first decrements ECX, and then jumps to label L1 only if ECX is not equal to zero:

```
    Loop  L1                          ; repeat the loop
```

Step 13: After the loop ends, we want to display a "Press any key…" message and then pause the output and wait for a key to be pressed by the user. To do this, we call the WaitMsg procedure:

```
    call  WaitMsg                     ; "Press any key..."
```

Step 14: Just before the program ends, the console window attributes are returned to the default colors (light gray characters on a black background).

```
    mov   eax, DefaultColor
    call  SetTextColor
    call  Clrscr
```

Here are the closing lines of the program:

```
    exit
main ENDP
END main
```

The remainder of the program's output is shown in the following figure, using four sample integers entered by the user:

```
Enter a 32-bit signed integer: -42

-42
FFFFFFD6
1111 1111 1111 1111 1111 1111 1101 0110

Enter a 32-bit signed integer: 36

+36
00000024
0000 0000 0000 0000 0000 0000 0010 0100

Enter a 32-bit signed integer: 244324

+244324
0003BA64
0000 0000 0000 0011 1011 1010 0110 0100

Enter a 32-bit signed integer: -7979779

-7979779
FF863CFD
1111 1111 1000 0110 0011 1100 1111 1101
```

A complete listing of the program appears below, with a few added comment lines:

```
; Library Test #1: Integer I/O   (InputLoop.asm)

; Tests the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor,
; WaitMsg, WriteBin, WriteHex, and WriteString procedures.

include Irvine32.inc

.data
COUNT = 4
BlueTextOnGray = blue + (lightGray * 16)
DefaultColor = lightGray + (black * 16)
arrayD SDWORD 12345678h,1A4B2000h,3434h,7AB9h
prompt BYTE "Enter a 32-bit signed integer: ",0

.code
main PROC

; Select blue text on a light gray background

    mov   eax,BlueTextOnGray
    call  SetTextColor
    call  Clrscr                     ; clear the screen

    ; Display an array using DumpMem.

    mov   esi,OFFSET arrayD          ; starting OFFSET
    mov   ebx,TYPE arrayD            ; doubleword = 4 bytes
    mov   ecx,LENGTHOF arrayD        ; number of units in arrayD
    call  DumpMem                    ; display memory
```

```
    ; Ask the user to input a sequence of signed integers
    call  Crlf                          ; new line
    mov   ecx,COUNT
L1: mov   edx,OFFSET prompt
    call  WriteString
    call  ReadInt                       ; input integer into EAX
    call  Crlf                          ; new line
; Display the integer in decimal, hexadecimal, and binary
    call  WriteInt                      ; display in signed decimal
    call  Crlf
    call  WriteHex                      ; display in hexadecimal
    call  Crlf
    call  WriteBin                      ; display in binary
    call  Crlf
    call  Crlf
    Loop  L1                            ; repeat the loop
; Return the console window to default colors
    call  WaitMsg                       ; "Press any key..."
    mov   eax,DefaultColor
    call  SetTextColor
    call  Clrscr
    exit
main ENDP
END main
```

Library Test #2: Random Integers

Let's look at a second library test program that demonstrates random-number-generation capabilities of the link library, and introduces the CALL instruction (to be covered fully in Section 5.5). First, it randomly generates 10 unsigned integers in the range 0 to 4,294,967,294. Next, it generates 10 signed integers in the range −50 to +49:

```
; Link Library Test #2 (TestLib2.asm)
; Testing the Irvine32 Library procedures.
include Irvine32.inc
TAB = 9                         ; ASCII code for Tab
.code
main PROC
    call  Randomize             ; init random generator
    call  Rand1
    call  Rand2
    exit
main ENDP
Rand1 PROC
; Generate ten pseudo-random integers.
    mov   ecx,10                ; loop 10 times
L1: call  Random32              ; generate random int
```

```
    call  WriteDec              ; write in unsigned decimal
    mov   al,TAB                ; horizontal tab
    call  WriteChar             ; write the tab
    loop  L1

    call  Crlf
    ret
Rand1 ENDP

Rand2 PROC
; Generate ten pseudo-random integers from -50 to +49
    mov   ecx,10                ; loop 10 times

L1: mov   eax,100               ; values 0-99
    call  RandomRange           ; generate random int
    sub   eax,50                ; values -50 to +49
    call  WriteInt              ; write signed decimal
    mov   al,TAB                ; horizontal tab
    call  WriteChar             ; write the tab
    loop  L1

    call  Crlf
    ret
Rand2 ENDP
END main
```

Here is sample output from the program:

```
3221236194   2210931702   974700167   367494257   2227888607
926772240    506254858    1769123448   2288603673  736071794
-34   +27   +38   -34   +31   -13   -29   +44   -48   -43
```

Library Test #3: Performance Timing

Assembly language is often used to optimize sections of code seen as critical to a program's performance. The *GetMseconds* procedure from the book's library returns the number of milliseconds elapsed since midnight. In our third library test program, we call *GetMseconds*, execute a nested loop, and call *GetMSeconds* a second time. The difference between the two values returned by these procedure calls gives us the elapsed time of the nested loop:

```
; Link Library Test #3          (TestLib3.asm)

; Calculate the elapsed execution time of a nested loop

include Irvine32.inc

.data
OUTER_LOOP_COUNT = 3
startTime DWORD ?
msg1 byte "Please wait...",0dh,0ah,0
msg2 byte "Elapsed milliseconds: ",0

.code
```

```
main PROC
     mov   edx,OFFSET msg1      ; "Please wait..."
     call  WriteString

; Save the starting time

     call  GetMSeconds
     mov   startTime,eax

; Start the outer loop

     mov   ecx,OUTER_LOOP_COUNT

L1:  call  innerLoop
     loop  L1

; Calculate the elapsed time

     call  GetMSeconds
     sub   eax,startTime

; Display the elapsed time

     mov   edx,OFFSET msg2       ; "Elapsed milliseconds: "
     call  WriteString
     call  WriteDec              ; write the milliseconds
     call  Crlf

     exit
main ENDP

innerLoop PROC
     push  ecx                   ; save current ECX value

     mov   ecx,0FFFFFFFh         ; set the loop counter
L1:  mul   eax                   ; use up some cycles
     mul   eax
     mul   eax
     loop  L1                    ; repeat the inner loop

     pop   ecx                   ; restore ECX's saved value
     ret
innerLoop ENDP

END main
```

Here is sample output from the program running on an Intel Core Duo processor:

```
Please wait....

Elapsed milliseconds: 4974
```

Detailed Analysis of the Program

Let us study Library Test #3 in greater detail. The *main* procedure displays the string “Please wait…” in the console window:

```
main PROC
     mov   edx,OFFSET msg1       ; "Please wait..."
     call  WriteString
```

When *GetMSeconds* is called, it returns the number of milliseconds that have elapsed since midnight into the EAX register. This value is saved in a variable for later use:

```
call  GetMSeconds
mov   startTime,eax
```

Next, we create a loop that executes based on the value of the OUTER_LOOP_COUNT constant. That value is moved to ECX for use later in the LOOP instruction:

```
mov   ecx,OUTER_LOOP_COUNT
```

The loop begins with label L1, where the *innerLoop* procedure is called. This CALL instruction repeats until ECX is decremented down to zero:

```
L1: call  innerLoop
    loop  L1
```

The **innerLoop** procedure uses an instruction named PUSH to save ECX on the stack before setting it to a new value. (We will discuss PUSH and POP in the upcoming Section 5.4.) Then, the loop itself has a few instructions designed to use up clock cycles:

```
innerLoop PROC
    push ecx                     ; save current ECX value

    mov   ecx,0FFFFFFFh          ; set the loop counter
L1: mul   eax                    ; use up some cycles
    mul   eax
    mul   eax
    loop  L1                     ; repeat the inner loop
```

The LOOP instruction will have decremented ECX down to zero at this point, so we pop the saved value of ECX off the stack. It will now have the same value on leaving this procedure that it had when entering. The PUSH and POP sequence is necessary because the *main* procedure was using ECX as a loop counter when it called the *innerLoop* procedure. Here are the last few lines of *innerLoop:*

```
    pop   ecx                    ; restore ECX's saved value
    ret
innerLoop ENDP
```

Back in the *main* procedure, after the loop finishes, we call GetMSeconds, which returns its result in EAX. All we have to do is subtract the starting time from this value to get the number of milliseconds that elapsed between the two calls to GetMSeconds:

```
call  GetMSeconds
sub   eax,startTime
```

The program displays a new string message, and then displays the integer in EAX that represents the number of elapsed milliseconds:

```
    mov   edx,OFFSET msg2        ; "Elapsed milliseconds: "
    call  WriteString
    call  WriteDec               ; display the value in EAX
    call  Crlf
    exit
main ENDP
```