Lecture No.12

Lecture Outlines

5.3 Linking to an External Library
    5.3.1 Background Information

5.4 The Irvine32 Library
    5.4.1 Motivation for Creating the Library
    5.4.2 Overview
    5.4.3 Individual Procedure Descriptions

5.3 Linking to an External Library

If you spend the time, you can write detailed code for input–output in assembly language. It's a lot like building your own automobile from scratch so that you can drive somewhere. The work is both interesting and time consuming. In Chapter 11 you will get a chance to see how input–output is handled in MS-Windows protected mode. It is great fun, and a new world opens up when you see the available tools. For now, however, input–output should be easy while you are learning assembly language basics. Section 5.3 shows how to call procedures from the book's link libraries, named *Irvine32.lib* and *Irvine64.obj*. The complete library source code is available at the author's web site (asmirvine.com). It should be installed on your computer in the *Examples\Lib32* subfolder of the book's install file (usually named *C:\Irvine*).

The Irvine32 library can only be used by programs running in 32-bit mode. It contains procedures that link to the MS-Windows API when they generate input–output. The Irvine64 library is a more limited library for 64-bit applications that is limited to essential display and string operations.

5.3.1 Background Information

A *link library* is a file containing procedures (subroutines) that have been assembled into machine code. A link library begins as one or more source files, which are assembled into object files. The object files are inserted into a specially formatted file recognized by the linker utility. Suppose a program displays a string in the console window by calling a procedure named **WriteString**. The program source must contain a PROTO directive identifying the WriteString procedure:

```
WriteString proto
```

Next, a CALL instruction executes **WriteString**:

```
call WriteString
```

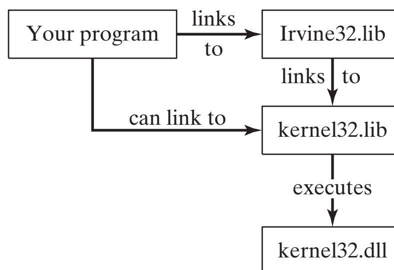
When the program is assembled, the assembler leaves the target address of the CALL instruction blank, knowing that it will be filled in by the linker. The linker looks for **WriteString** in the link library and copies the appropriate machine instructions from the library into the program's executable file. In addition, it inserts **WriteString's** address into the CALL instruction. If a procedure you're calling is not in the link library, the linker issues an error message and does not generate an executable file.

Linker Command Options The linker utility combines a program's object file with one or more object files and link libraries. The following command, for example, links hello.obj to the irvine32.lib and kernel32.lib libraries:

```
link hello.obj irvine32.lib kernel32.lib
```

Linking 32-Bit Programs The kernel32.lib file, part of the Microsoft Windows Platform *Software Development Kit*, contains linking information for system functions located in a file named kernel32.dll. The latter is a fundamental part of MS-Windows, and is called a *dynamic link library*. It contains executable functions that perform character-based input–output. Figure 5-9 shows how kernel32.lib is a bridge to kernel32.dll.

Figure 5–9 Linking 32-bit programs.



In Chapters 1 through 10, our programs link either Irvine32.lib or Irvine64.obj. Chapter 11 shows how to link programs directly to kernel32.lib.

5.4 The Irvine32 Library

5.4.1 Motivation for Creating the Library

There is no Microsoft-sanctioned standard library for assembly language programming. When programmers first started writing assembly language for x86 processors in the early 1980s, MS-DOS was the commonly used operating system. These 16-bit programs were able to call MS-DOS functions (known as INT 21h services) to do simple input/output. Even at that time, if you wanted to display an integer on the console, you had to write a fairly complicated procedure that converted from the internal binary representation of integers to a sequence of ASCII characters that would display the integer on the screen. We called it **WriteInt**, and this is the logic, abstracted into pseudocode:

Initialization:

```
let n equal the binary value
let buffer be an array of char[size]
```

Algorithm:

```
i = size  -1                    ; last position of buffer
repeat
    r = n mod 10                ; remainder
    n = n / 10                  ; integer division
    digit = r OR 30h            ; conver r to ASCII digit
    buffer[i--] = digit         ; store in buffer
until n = 0

if n is negative
    buffer[i] = "-"             ; insert a negative sign

while i > 0
    print buffer[i]
    i++
```

Notice that the digits are generated in reverse order and inserted into a buffer, moving from the back to the front. Then the digits are written to the console in forward order. While this code is easy enough to implement in C/C++, it requires some advanced skills in assembly language.

Professional programmers often prefer to build their own libraries, and doing so is an excellent educational experience. In 32-bit mode running under Windows, an input–output library must make calls directly into the operating system. The learning curve is rather steep, and it presents some challenges for beginning programmers. Therefore, the *Irvine32*

library is designed to provide a simple interface for input–output for beginners. As you continue through the chapters in this book, you will acquire the knowledge and skills to create your own library. You are free to modify and reuse the library, as long as you give credit to its original author. Another alternative, which we will discuss in Chapter 13, is to call Standard C library functions from your assembly language programs. Again, that requires some additional background.

Table 5-1 contains a complete list of procedures in the Irvine32 library.

Table 5-1 Procedures in the Irvine32 Library.

Procedure	Description
CloseFile	Closes a disk file that was previously opened.
Clrscr	Clears the console window and locates the cursor at the upper left corner.
CreateOutputFile	Creates a new disk file for writing in output mode.
Crlf	Writes an end-of-line sequence to the console window.
Delay	Pauses the program execution for a specified *n*-millisecond interval.
DumpMem	Writes a block of memory to the console window in hexadecimal.
DumpRegs	Displays the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, and EIP registers in hexadecimal. Also displays the most common CPU status flags.
GetCommandTail	Copies the program's command-line arguments (called the *command tail*) into an array of bytes.
GetDateTime	Gets the current date and time from the system.
GetMaxXY	Gets the number of columns and rows in the console window's buffer.
GetMseconds	Returns the number of milliseconds elapsed since midnight.
GetTextColor	Returns the active foreground and background text colors in the console window.
Gotoxy	Locates the cursor at a specific row and column in the console window.
IsDigit	Sets the Zero flag if the AL register contains the ASCII code for a decimal digit (0–9).
MsgBox	Displays a popup message box.
MsgBoxAsk	Display a yes/no question in a popup message box.
OpenInputFile	Opens an existing disk file for input.
ParseDecimal32	Converts an unsigned decimal integer string to 32-bit binary.
ParseInteger32	Converts a signed decimal integer string to 32-bit binary.
Random32	Generates a 32-bit pseudorandom integer in the range 0 to FFFFFFFFh.
Randomize	Seeds the random number generator with a unique value.
RandomRange	Generates a pseudorandom integer within a specified range.
ReadChar	Waits for a single character to be typed at the keyboard and returns the character.
ReadDec	Reads an unsigned 32-bit decimal integer from the keyboard, terminated by the Enter key.
ReadFromFile	Reads an input disk file into a buffer.
ReadHex	Reads a 32-bit hexadecimal integer from the keyboard, terminated by the Enter key.

Table 5-1 *(Continued)*

Procedure	Description
ReadInt	Reads a 32-bit signed decimal integer from the keyboard, terminated by the Enter key.
ReadKey	Reads a character from the keyboard's input buffer without waiting for input.
ReadString	Reads a string from the keyboard, terminated by the Enter key.
SetTextColor	Sets the foreground and background colors of all subsequent text output to the console.
Str_compare	Compares two strings.
Str_copy	Copies a source string to a destination string.
Str_length	Returns the length of a string in EAX.
Str_trim	Removes unwanted characters from a string.
Str_ucase	Converts a string to uppercase letters.
WaitMsg	Displays a message and waits for a key to be pressed.
WriteBin	Writes an unsigned 32-bit integer to the console window in ASCII binary format.
WriteBinB	Writes a binary integer to the console window in byte, word, or doubleword format.
WriteChar	Writes a single character to the console window.
WriteDec	Writes an unsigned 32-bit integer to the console window in decimal format.
WriteHex	Writes a 32-bit integer to the console window in hexadecimal format.
WriteHexB	Writes a byte, word, or doubleword integer to the console window in hexadecimal format.
WriteInt	Writes a signed 32-bit integer to the console window in decimal format.
WriteStackFrame	Writes the current procedure's stack frame to the console.
WriteStackFrameName	Writes the current procedure's name and stack frame to the console.
WriteString	Writes a null-terminated string to the console window.
WriteToFile	Writes a buffer to an output file.
WriteWindowsMsg	Displays a string containing the most recent error generated by MS-Windows.

5.4.2 Overview

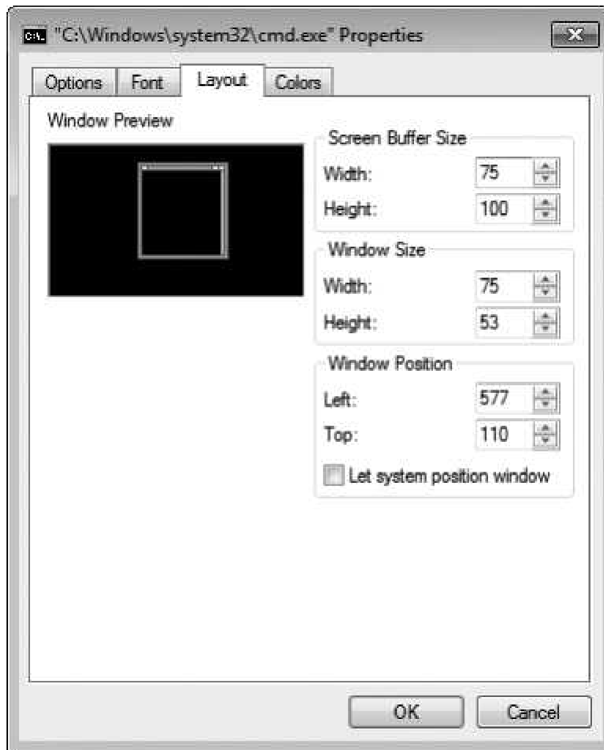
Console Window The *console window* (or *command window*) is a text-only window created by MS-Windows when a command prompt is displayed.

To display a console window in Microsoft Windows, click the Start button on the desktop, type *cmd* into the *Start Search* field, and press Enter. Once a console window is open, you can resize the console window buffer by right-clicking on the system menu in the window's upper-left corner, selecting *Properties* from the popup menu, and then modifying the values, as shown in Fig. 5-10.

You can also select various font sizes and colors. The console window defaults to 25 rows by 80 columns. You can use the *mode* command to change the number of columns and lines. The following, typed at the command prompt, sets the console window to 40 columns by 30 lines:

```
mode con cols=40 lines=30
```

Figure 5–10 Modifying the console window properties.



A *file handle* is a 32-bit integer used by the Windows operating system to identify a file that is currently open. When your program calls a Windows service to open or create a file, the operating system creates a new file handle and makes it available to your program. Each time you call an OS service method to read from or write to the file, you must pass the same file handle as a parameter to the service method.

Note: If your program calls procedures in the Irvine32 library, you must always push 32-bit values onto the runtime stack; if you do not, the Win32 Console functions called by the library will not work correctly.

5.4.3 Individual Procedure Descriptions

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.

CloseFile The CloseFile procedure closes a file that was previously created or opened (see CreateOutputFile and OpenInputFile). The file is identified by a 32-bit integer *handle*, which is passed in EAX. If the file is closed successfully, the value returned in EAX will be nonzero. Sample call:

```
mov   eax,fileHandle
call  CloseFile
```

Clrscr The Clrscr procedure clears the console window. This procedure is typically called at the beginning and end of a program. If you call it at other times, you may need to pause the program by first calling WaitMsg. Doing this allows the user to view information already on the screen before it is erased. Sample call:

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

CreateOutputFile The CreateOutputFile procedure creates a new disk file and opens it for writing. When you call the procedure, place the offset of a filename in EDX. When the procedure returns, EAX will contain a valid file handle (32-bit integer) if the file was created successfully. Otherwise, EAX equals INVALID_HANDLE_VALUE (a predefined constant). Sample call:

```
.data
filename BYTE "newfile.txt",0
.code
mov  edx,OFFSET filename
call CreateOutputFile
```

The following pseudocode describes the possible outcomes after calling CreateOutputFile:

```
if EAX = INVALID_HANDLE_VALUE
    the file was not created successfully
else
    EAX = handle for the open file
endif
```

Crlf The Crlf procedure advances the cursor to the beginning of the next line in the console window. It writes a string containing the ASCII character codes 0Dh and 0Ah. Sample call:

```
call  Crlf
```

Delay The Delay procedure pauses the program for a specified number of milliseconds. Before calling Delay, set EAX to the desired interval. Sample call:

```
mov   eax,1000                   ; 1 second
call  Delay
```

DumpMem The DumpMem procedure writes a range of memory to the console window in hexadecimal. Pass it the starting address in ESI, the number of units in ECX, and the unit size in EBX (1 = byte, 2 = word, 4 = doubleword). The following sample call displays an array of 11 doublewords in hexadecimal:

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,0Ah,0Bh
.code
main PROC
    mov   esi,OFFSET array           ; starting OFFSET
    mov   ecx,LENGTHOF array         ; number of units
    mov   ebx,TYPE array             ; doubleword format
    call  DumpMem
```

The following output is produced:

```
00000001  00000002  00000003  00000004  00000005  00000006
00000007  00000008  00000009  0000000A  0000000B
```

DumpRegs The DumpRegs procedure displays the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, and EFL (EFLAGS) registers in hexadecimal. It also displays the values of the Carry, Sign, Zero, Overflow, Auxiliary Carry, and Parity flags. Sample call:

```
call DumpRegs
```

Sample output:

```
EAX=00000613  EBX=00000000  ECX=000000FF  EDX=00000000
ESI=00000000  EDI=00000100  EBP=0000091E  ESP=000000F6
EIP=00401026  EFL=00000286  CF=0  SF=1  ZF=0  OF=0  AF=0  PF=1
```

The displayed value of EIP is the offset of the instruction following the call to DumpRegs. DumpRegs can be useful when debugging programs because it displays a snapshot of the CPU. It has no input parameters and no return value.

GetCommandTail The GetCommandTail procedure copies the program's command line into a null-terminated string. If the command line was found to be empty, the Carry flag is set; otherwise, the Carry flag is cleared. This procedure is useful because it permits the user of a program to pass parameters on the command line. Suppose a program named **Encrypt.exe** reads an input file named **file1.txt** and produces an output file named **file2.txt**. The user can pass both filenames on the command line when running the program:

```
Encrypt file1.txt file2.txt
```

When it starts up, the Encrypt program can call GetCommandTail and retrieve the two filenames. When calling GetCommandTail, EDX must contain the offset of an array of at least 129 bytes. Sample call:

```
.data
cmdTail BYTE 129 DUP(0)          ; empty buffer
.code
mov   edx,OFFSET cmdTail
call  GetCommandTail             ; fills the buffer
```

There is a way to pass command-line arguments when running an application in Visual Studio. From the Project menu, select *<projectname> Properties*. In the Property Pages window, expand the entry under *Configuration Properties*, and select *Debugging*. Then enter your command arguments into the edit line on the right panel named *Command Arguments*.

GetMaxXY The GetMaxXY procedure gets the size of the console window's buffer. If the console window buffer is larger than the visible window size, scroll bars appear automatically. GetMaxXY has no input parameters. When it returns, the DX register contains the number of buffer columns and AX contains the number of buffer rows. The possible range of each value can be no greater than 255, which may be smaller than the actual window buffer size. Sample call:

```
.data
rows  BYTE ?
cols  BYTE ?
.code
call  GetMaxXY
mov   rows,al
mov   cols,dl
```

GetMseconds The GetMseconds procedure gets the number of milliseconds elapsed since midnight on the host computer, and returns the value in the EAX register. The procedure is a great tool for measuring the time between events. No input parameters are required. The following example calls GetMseconds, storing its return value. After the loop executes, the code call GetMseconds a second time and subtract the two time values. The difference is the approximate execution time of the loop:

```
.data
startTime DWORD ?
.code
call GetMseconds
mov  startTime,eax
L1:
  ; (loop body)
  loop L1
call GetMseconds
sub  eax,startTime                ; EAX = loop time, in milliseconds
```

GetTextColor The GetTextColor procedure gets the current foreground and background colors of the console window. It has no input parameters. It returns the background color in the upper four bits of AL and the foreground color in the lower four bits. Sample call:

```
.data
color byte ?
.code
call  GetTextColor
mov   color,AL
```

Gotoxy The Gotoxy procedure locates the cursor at a given row and column in the console window. By default, the console window's X-coordinate range is 0 to 79 and the Y-coordinate range is 0 to 24. When you call Gotoxy, pass the Y-coordinate (row) in DH and the X-coordinate (column) in DL. Sample call:

```
mov   dh,10                       ; row 10
mov   dl,20                       ; column 20
call  Gotoxy                      ; locate cursor
```

The user may have resized the console window, so you can call GetMaxXY to find out the current number of rows and columns.

IsDigit The IsDigit procedure determines whether the value in AL is the ASCII code for a valid decimal digit. When calling it, pass an ASCII character in AL. The procedure sets the Zero flag if AL contains a valid decimal digit; otherwise, it clears Zero flag. Sample call:

```
mov   AL,somechar
call  IsDigit
```

MsgBox The MsgBox procedure displays a graphical popup message box with an optional caption. (This works when the program is running in a console window.) Pass it the offset of a string in EDX, which will appear inside the box. Optionally, pass the offset of a string for the box's title in EBX. To leave the title blank, set EBX to zero. Sample call:

```
.data
caption BYTE "Dialog Title", 0
HelloMsg BYTE "This is a pop-up message box.", 0dh,0ah
       BYTE "Click OK to continue...", 0
.code
mov   ebx,OFFSET caption
mov   edx,OFFSET HelloMsg
call  MsgBox
```

Sample output:

MsgBoxAsk The MsgBoxAsk procedure displays a graphical popup message box with Yes and No buttons. (This works when the program is running in a console window.) Pass it the offset of a question string in EDX, which will appear inside the box. Optionally, pass the offset of a string for the box's title in EBX. To leave the title blank, set EBX to zero. MsgBoxAsk returns an integer in EAX that tells you which button was selected by the user. The value will be one of two predefined Windows constants: IDYES (equal to 6) or IDNO (equal to 7). Sample call:

```
.data
caption BYTE "Survey Completed",0
question BYTE "Thank you for completing the survey."
  BYTE 0dh,0ah
  BYTE "Would you like to receive the results?",0
.code
mov   ebx,OFFSET caption
mov   edx,OFFSET question
call  MsgBoxAsk
;(check return value in EAX)
```

Sample output:

OpenInputFile The OpenInputFile procedure opens an existing file for input. Pass it the offset of a filename in EDX. When it returns, if the file was opened successfully, EAX will contain a valid file handle. Otherwise, EAX will equal INVALID_HANDLE_VALUE (a predefined constant).
Sample call:

```
.data
filename BYTE "myfile.txt",0
.code
mov  edx,OFFSET filename
call OpenInputFile
```

The following pseudocode describes the possible outcomes after calling OpenInputFile:

```
if EAX = INVALID_HANDLE_VALUE
     the file was not opened successfully
else
     EAX = handle for the open file
endif
```

ParseDecimal32 The ParseDecimal32 procedure converts an unsigned decimal integer string to 32-bit binary. All valid digits occurring before a nonnumeric character are converted. Leading spaces are ignored. Pass it the offset of a string in EDX and the string's length in ECX. The binary value is returned in EAX. Sample call:

```
.data
buffer BYTE "8193"
bufSize = ($ - buffer)
.code
mov   edx,OFFSET buffer
mov   ecx,bufSize
call  ParseDecimal32            ; returns EAX
```

- 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 contains the converted integer and CF = 0

See the description of the **ReadDec** procedure for details about how the Carry flag is affected.

ParseInteger32 The ParseInteger32 procedure converts a signed decimal integer string to 32-bit binary. All valid digits from the beginning of the string to the first nonnumeric character are converted. Leading spaces are ignored. Pass it the offset of a string in EDX and the string's length in ECX. The binary value is returned in EAX. Sample call:

```
.data
buffer byte "-8193"
bufSize = ($ - buffer)
.code
mov   edx,OFFSET buffer
mov   ecx,bufSize
call  ParseInteger32           ; returns EAX
```

The string may contain an optional leading plus or minus sign, followed only by decimal digits. The Overflow flag is set and an error message is displayed on the console if the value cannot be represented as a 32-bit signed integer (range: −2,147,483,648 to +2,147,483,647).

Random32 The Random32 procedure generates and returns a 32-bit random integer in EAX. When called repeatedly, Random32 generates a simulated random sequence. The numbers are created using a simple function having an input called a *seed*. The function uses the seed in a formula that generates the random value. Subsequent random values are generated using each previously generated random value as their seeds. The following code snippet shows a sample call to Random32:

```
.data
randVal DWORD ?
.code
call  Random32
mov   randVal,eax
```

Randomize The Randomize procedure initializes the starting seed value of the Random32 and RandomRange procedures. The seed equals the time of day, accurate to 1/100 of a second. Each time you run a program that calls Random32 and RandomRange, the generated sequence of random numbers will be unique. You need only to call Randomize once at the beginning of a program. The following example produces 10 random integers:

```
    call  Randomize
    mov   ecx,10
L1: call  Random32

    ; use or display random value in EAX here...

    loop  L1
```

RandomRange The RandomRange procedure produces a random integer within the range of 0 to $n - 1$, where n is an input parameter passed in the EAX register. The random integer is returned in EAX. The following example generates a single random integer between 0 and 4999 and places it in a variable named *randVal*.

```
.data
randVal DWORD ?

.code
mov  eax,5000
call RandomRange
mov  randVal,eax
```