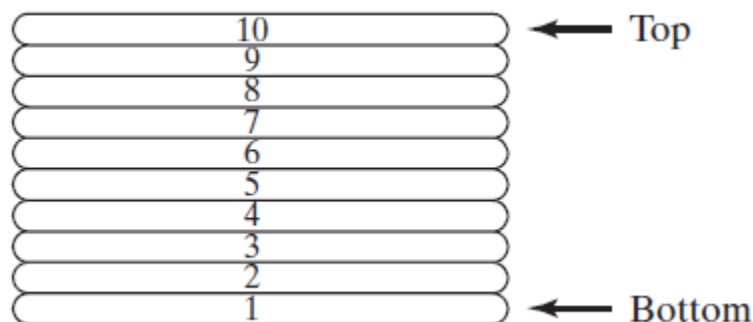


Lab 6 - Stack and Procedures

6.1 Stack

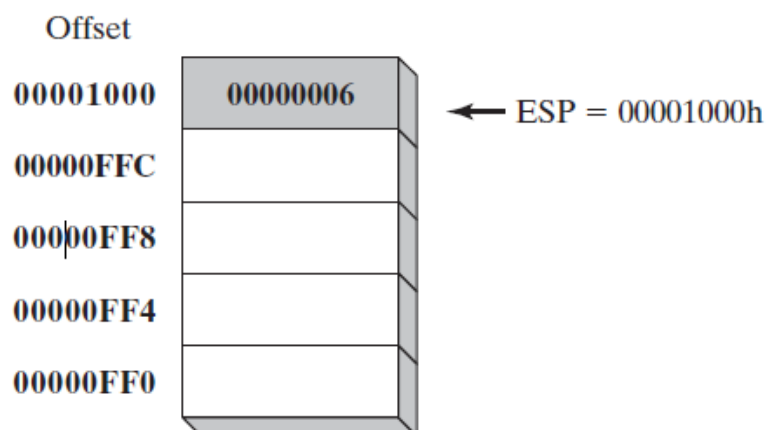
A *stack* is also called a LIFO structure (Last-In, First-Out) because the last value put into the stack is always the first value taken out. New values are added to the top of the stack, and existing values are removed from the top. In this Lab, we concentrate specifically on the *runtime stack*. It is supported directly by hardware in the CPU, and it is an essential part of the mechanism for calling and returning from procedures. Most of the time, we just call it the *stack*.



6.2 Runtime Stack (32-Bit Mode)

The *runtime stack* is a memory array managed directly by the CPU, using the ESP (extended stack pointer) register, known as the *stack pointer register*. In 32-bit mode, ESP register holds a 32-bit offset into some location on the stack. We rarely manipulate ESP directly; instead, it is indirectly modified by instructions such as CALL, RET, PUSH, and POP.

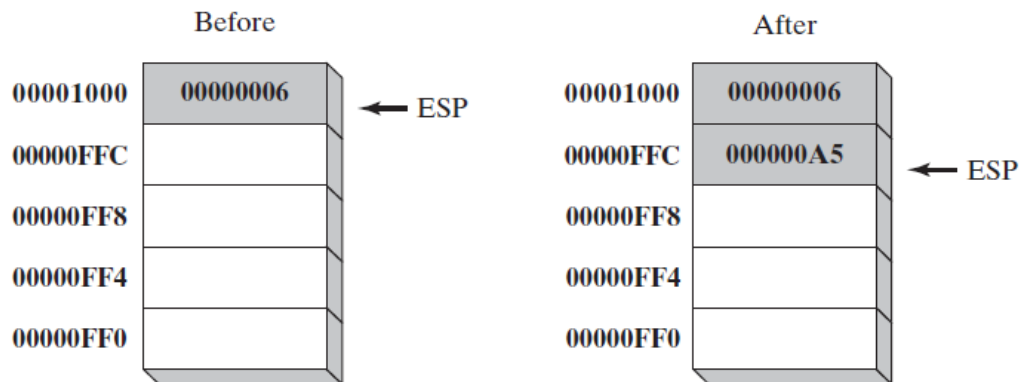
ESP register always points to the last value to be added to, or pushed on, the top of stack. To demonstrate, let's begin with a stack containing one value. In below figure, the ESP contains hexadecimal 00001000, the offset of the most recently pushed value (00000006). In diagrams, the top of the stack moves downward when the stack pointer decreases in value:



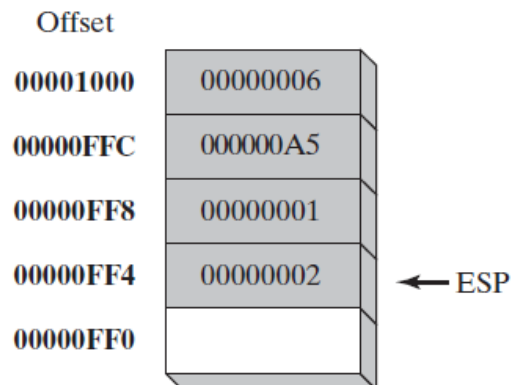
Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode.

6.2.1 Push Operation

A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location in the stack pointed to by the stack pointer. The below figure shows the effect of pushing 000000A5 on a stack that already contains one value (00000006). Notice that the ESP register always points to the last item pushed on the stack. The figure shows the stack ordering opposite to that of the diagram of stack we saw earlier in start of Lab, because the runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, ESP = 00001000h; after the push, ESP = 0000FFCh.

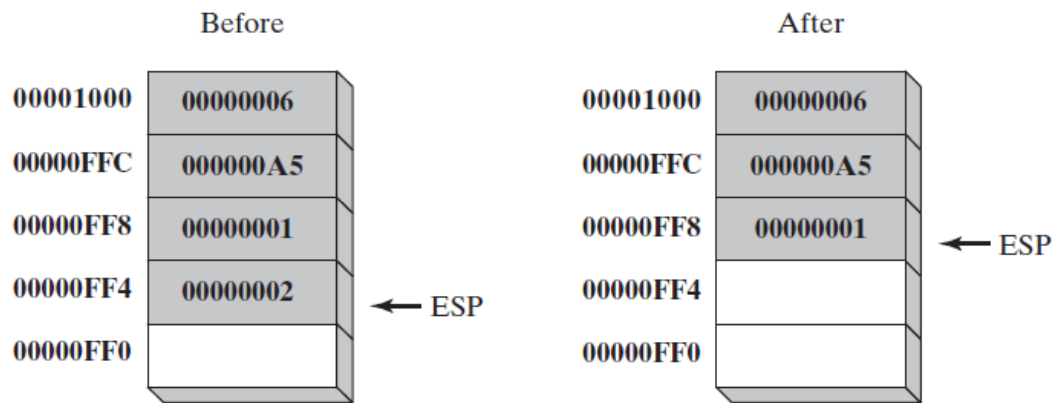


The below figure shows the same stack after pushing a total of four integers.



6.2.2 Pop Operation

A pop operation removes a value from the stack. After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next-highest location in the stack. The below figure shows the stack before and after the value 00000002 is popped.



6.2.3 Stack Applications

- A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
- When the CALL instruction executes, the CPU saves the current subroutine's return address on the stack.
- When calling a subroutine, you pass input values called *arguments* by pushing them on the stack.
- The stack provides temporary storage for local variables inside subroutines.

6.3 PUSH and POP Instructions

6.3.1 PUSH Instruction

The PUSH instruction first decrements ESP and then copies a source operand into the stack. A 16-bit operand causes ESP to be decremented by 2. A 32-bit operand causes ESP to be decremented by 4. There are three instruction formats:

```
PUSH reg/mem16
PUSH reg/mem32
PUSH imm32
```

6.3.2 POP Instruction

The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

```
POP reg/mem16
POP reg/mem32
```

6.3.3 PUSHFD and POPFD Instructions

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

```
pushfd
popfd
```

The MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags:

```
pushfd      ;save the flags
popfd       ;restore the flags
```

Note: When using pushes and pops of this type, be sure the program's execution path does not skip over the POPFD instruction.

A less error-prone way to save and restore the flags is to push them on the stack and immediately pop them into a variable:

```
.data
    saveFlags DWORD ?
.code
    pushfd                ;push flags on stack
    pop saveFlags         ;copy into a variable
```

The following statements restore the flags from the same variable:

```
push saveFlags            ;push saved flag values
popfd                    ;copy into the flags
```

6.3.4 PUSH and POP Instructions

PUSH instruction pushes the 16-bit general-purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed. The POP instruction pops the same registers in reverse. You should only use PUSH and POP when programming in 16-bit mode.

6.3.5 PUSHAD and POPAD Instructions

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order:

- EAX
- ECX
- EDX
- EBX
- ESP
- EBP
- ESI
- EDI

The POPAD instruction pops the same registers off the stack in reverse order.

If you write a procedure that modifies a number of 32-bit registers, use PUSHAD at the beginning of the procedure and POPAD at the end to save and restore the registers. The following code fragment is an example:

```
MySub PROC
    pushad                ;save general-purpose registers
    .
    .
    mov eax,...
    mov edx,...
    mov ecx,...
    .
    .
    popad                 ;restore general-purpose registers
    ret                   ;return from procedure
MySub ENDP
```

Note: Procedures returning results in one or more registers should not use PUSHA and PUSHAD. Suppose the following **ReadValue** procedure returns an integer in EAX; the call to POPAD overwrites the return value from EAX:

```
ReadValue PROC
pushad                ;save general-purpose registers
.
.
mov eax,return_value
.
.
popad                ;overwrites EAX!
ret
ReadValue ENDP
```

6.3.6 Show Run-Time Stack in Visual Studio

Debug the following code in Visual Studio:

```
Include Irvine32.inc

.data
var1 DWORD 59h
var2 DWORD 61h
var3 WORD 5Fh,8Ah,6Bh,2Bh

.code
main PROC
    mov     EAX, var1
    PUSH EAX
    mov     EBX, var2
    PUSH EBX

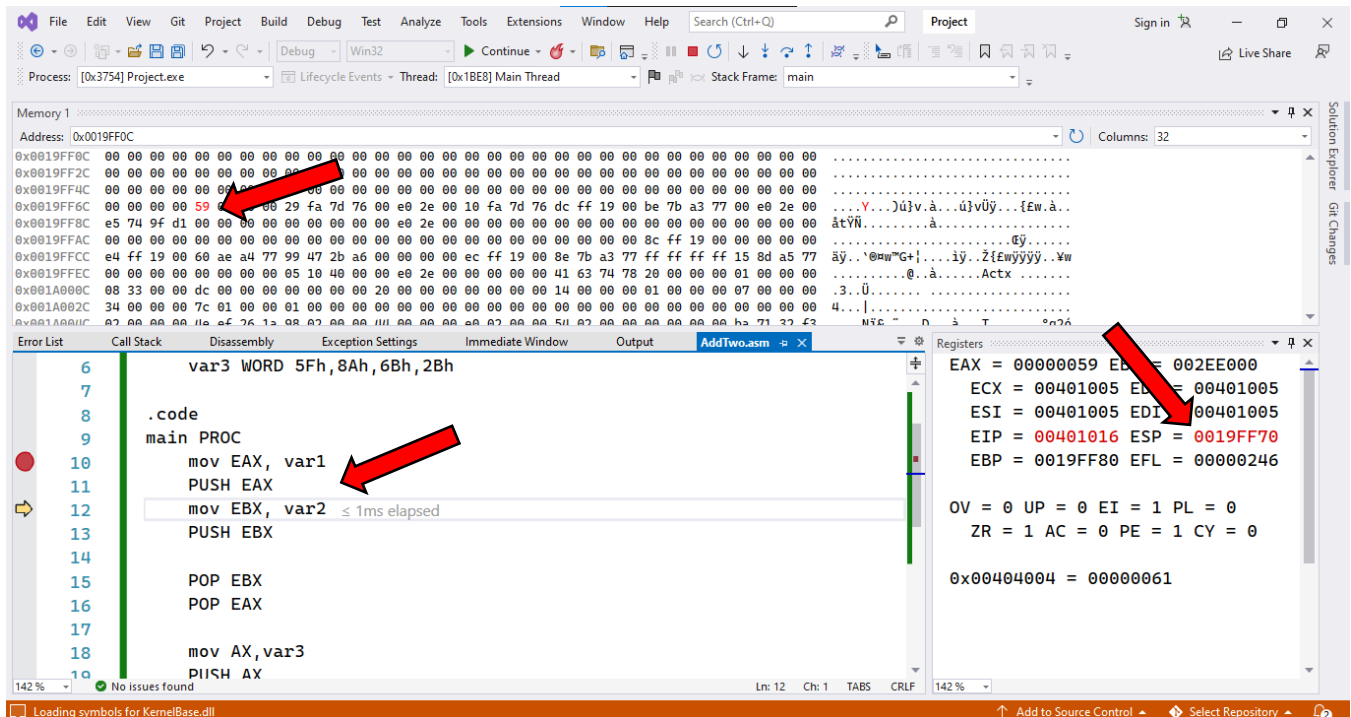
    POP EBX
    POP EAX

    mov     AX,var3
    PUSH AX

    exit
main ENDP
END main
```

Start program in debugging and go to Memory window.

After pushing the value to stack. Copy the ESP value to Memory Window Address box.



The Memory window shows the stack data in Memory window.

6.3.7 Example: Reversing a String

Include Irvine32.inc

.data

```
aName BYTE "Hello World",0
nameSize = ($ - aName) - 1
```

.code

main PROC

```
;Push the name on the stack.
```

```
mov ecx,nameSize
```

```
mov esi,0
```

```
L1: movzx eax,aName[esi]
```

```
; get character
```

```
push eax
```

```
; push on stack
```

```
inc esi
```

```
loop L1
```

```
; Pop the name from the stack, in reverse order
```

```
mov ecx,nameSize
```

```
mov esi,0
```

```
L2: pop eax
```

```
; get character
```

```
mov aName[esi],al
```

```
; store in string
```

```
inc esi
```

```
loop L2
```

```
mov EDX,OFFSET aName
```

```
call WriteString
```

```
INVOKE ExitProcess,0
```

main ENDP

END main

6.4 Procedures

We can divide programs into *subroutines*. A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively.

6.4.1 PROC Directive

A procedure is declared using the PROC and ENDP directives. It must be assigned a name (a valid identifier). Each program we've written so far contains a procedure named main.

```
main PROC
.
.
main ENDP
```

When you create a procedure other than your program's startup procedure, end it with a RET instruction. RET forces the CPU to return to the location from where the procedure was called:

```
sample PROC
.
.
ret
sample ENDP
```

6.4.2 Labels in Procedures

By default, labels are visible only within the procedure in which they are declared. This rule often affects jump and loop instructions. In the following example, the label named Destination must be located in the same procedure as the JMP instruction:

```
jmp Destination
```

It is possible to work around this limitation by declaring a global label, identified by a double colon (::) after its name:

```
Destination::
```

Note: It is not a good idea to jump or loop outside of the current procedure. Procedures have an automated way of returning and adjusting the runtime stack. If you directly transfer out of a procedure, the runtime stack can easily become corrupted.

6.4.3 Documenting Procedures

- A description of all tasks accomplished by the procedure.
- A list of input parameters and their usage, labeled by a word such as **Receives**. If any input parameters have specific requirements for their input values, list them here.
- A description of any values returned by the procedure, labeled by a word such as **Returns**.
- A list of any special requirements, called preconditions, that must be satisfied before the procedure is called. These can be labeled by the word **Requires**. For example, for a procedure that draws a graphics line, a useful precondition would be that the video display adapter must already be in graphics mode.

For example:

```

;-----
; sumof
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
; signed or unsigned.
; Returns: EAX = sum
;-----

SumOf PROC
    add eax,ebx
    add eax,ecx
    ret
SumOf ENDP

```

6.4.4 CALL and RET Instructions

The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location. The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called.

The CALL instruction pushes its return address on the stack and copies the called procedure's address into the instruction pointer. When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer. In 32-bit mode, the CPU executes the instruction in memory pointed to by EIP (instruction pointer register). In 16-bit mode, IP points to the instruction.

Suppose that in **main**, a CALL statement is located at offset 00000020. Typically, this instruction requires 5 bytes of machine code, so the next statement (a MOV in this case) is located at offset 00000025:

```

main PROC
00000020      call MySub
00000025      mov  eax,ebx

```

Next, suppose that the first executable instruction in **MySub** is located at offset 00000040:

```

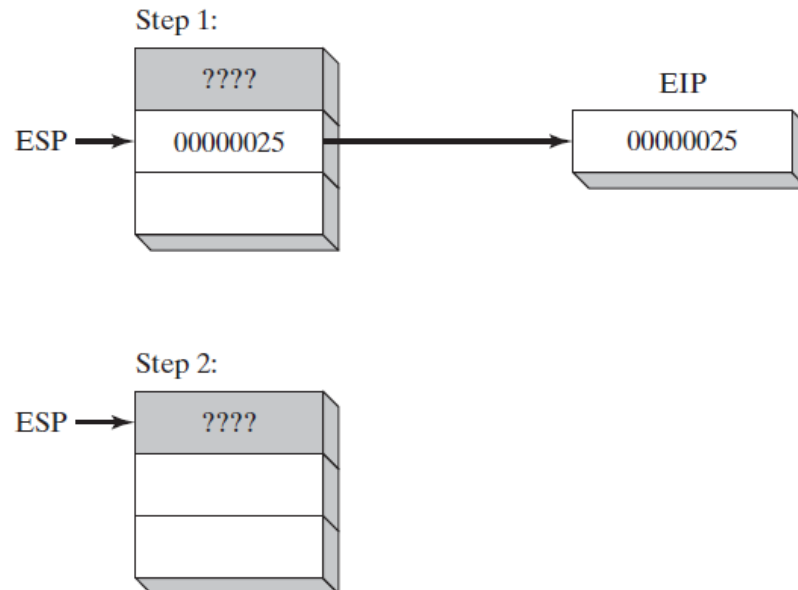
MySub PROC
00000040      mov  eax,edx
.
.
      ret
MySub ENDP

```

When the CALL instruction executes, the address following the call (00000025) is pushed on the stack and the address of **MySub** is loaded into EIP. All instructions in **MySub** execute up to its RET instruction.



When the RET instruction executes, in Step 1, the value in the stack pointed to by ESP is popped into EIP. In step 2, ESP is incremented so it points to the previous value on the stack.



6.4.5 Passing Register Arguments to Procedures

While using procedures, it is not a good idea to include references to specific variable names inside the procedure. If you did, the procedure could only be used with one array. A better approach is to pass the offset of an array to the procedure and pass an integer specifying the number of array elements. We call these *arguments* (or *input parameters*). In assembly language, it is common to pass arguments inside general-purpose registers.

In the below example we created a simple procedure named **SumOf** that added the integers in the EAX, EBX, and ECX registers. In main, before calling **SumOf**, we assign values to EAX, EBX, and ECX:

```
.data
    theSum DWORD ?

.code
main PROC
    mov eax,10000h    ; argument
    mov ebx,20000h    ; argument
    mov ecx,30000h    ; argument
    call SumOf        ; EAX = (EAX + EBX + ECX)
    mov theSum,eax    ; save the sum
main ENDP

SumOf PROC
    add eax,ebx
    add eax,ecx
    ret
SumOf ENDP
```

After the CALL statement, we have the option of copying the sum in EAX to a variable.

6.4.6 Example: Summing an Integer Array

```

Include Irvine32.inc
.data
    array DWORD 10000h,20000h,30000h,40000h,50000h
    theSum DWORD ?
.code
main PROC
    mov esi,OFFSET array          ;ESI points to array
    mov ecx,LENGTHOF array       ;ECX = array count
    call ArraySum                 ;calculate the sum
    mov theSum,eax                ;returned in EAX
    INVOKE ExitProcess,0
main ENDP

;-----
; ArraySum
;
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
; ECX = number of elements in the array
; Returns: EAX = sum of the array elements
;-----
ArraySum PROC
    push esi                      ;save ESI, ECX
    push ecx
    mov eax,0                     ;set the sum to zero
L1:
    add eax,[esi]                 ;add each integer to sum
    add esi,TYPE DWORD            ;point to next integer
    loop L1                       ;repeat for array size
    pop ecx                       ;restore ECX, ESI
    pop esi
    ret                           ;sum is in EAX
ArraySum ENDP

```

6.4.7 USES Operator

The USES operator is used with the PROC directive. It lets you list the names of all registers modified within a procedure. USES tells the assembler to do two things:

- First, generate PUSH instructions that save the registers on the stack at the beginning of the procedure.
- Second, generate POP instructions that restore the register values at the end of the procedure.

The USES operator immediately follows PROC and is itself followed by a list of registers on the same line separated by spaces or tabs (not commas). The **ArraySum** procedure from above example used PUSH and POP instructions to save and restore ESI and ECX. The USES operator can more easily do the same:

```

ArraySum PROC USES esi ecx
    mov eax,0                     ;set the sum to zero
L1:
    add eax,[esi]                 ;add each integer to sum
    add esi,TYPE DWORD            ;point to next integer
    loop L1                       ;repeat for array size
    ret                           ;sum is in EAX
ArraySum ENDP

```

6.5 Linking to an External Library

This section 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.

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
```

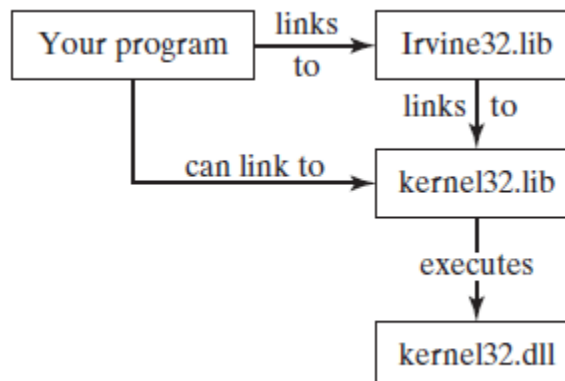
When the program is assembled, the assembler leaves the target address of the **CALL** instruction blank, knowing that it will be filled in by the linker. The linker looks for **WriteString** in the link library and copies the appropriate machine instructions from the library into the program's executable 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.

6.5.1 Linker Command

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
```

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. The figure below shows how *kernel32.lib* is a bridge to *kernel32.dll*.



6.6 The Irvine32 Library

- There is no Microsoft-approved standard library for assembly language programming.
- To display an integer on the console, you had to write a complicated procedure that converts the internal binary representation of integers to a sequence of ASCII characters and display the integer on the screen.
- Professional programmers often prefer to build their own libraries.
- The Irvine32 library is designed to provide a simple interface for input-output for beginners.

6.6.1 Procedures in the Irvine32 Library

The table below contains a complete list of 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 FFFFFFFh.
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.
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.

6.7 Irvine32 Library Procedures Examples

6.7.1 Testing Colors

```
; Testing SetTextColor and GetTextColor.
INCLUDE Irvine32.inc
```

```
.data
str1 BYTE "Sample string, in color",0dh,0ah,0
```

```
.code
main PROC
```

```
    mov     ax,yellow + (blue * 16)
    call    SetTextColor
```

```
    mov     edx,OFFSET str1
    call    WriteString
```

```
    call    GetTextColor
    call    DumpRegs
```

```
    exit
main ENDP
END main
```

For more details about "SetTextColor" procedure read Book's page number 167.

6.7.2 Testing the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor, WaitMsg, WriteBin, WriteHex, and WriteString procedures.

```
; 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
```

```
    ; Set text color to 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
```

```
    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
```

```
    ; Return console window to default colors.
    call    WaitMsg                ; "Press any key..."
    mov     eax,DefaultColor
    call    SetTextColor
    call    Clrscr
```

```
    exit
```

```
main ENDP
```

```
END main
```

6.7.3 Message Boxes

```
INCLUDE Irvine32.inc

.data
caption db "Dialog Title", 0

HelloMsg BYTE "This is a pop-up message box.", 0dh,0ah
          BYTE "Click OK to continue...", 0

.code
main PROC

    mov     ebx,0                ; no caption
    mov     edx,OFFSET HelloMsg  ; contents
    call    MsgBox

    mov     ebx,OFFSET caption   ; caption
    mov     edx,OFFSET HelloMsg  ; contents
    call    MsgBox

    exit
main ENDP
END main
```

Another example:

```
Include Irvine32.inc

.data
caption BYTE "LAB-6 Completed",0
question BYTE "Did you complete your LAB Tasks?."
          BYTE 0dh,0ah
          BYTE "Would you like to receive the Graded?",0
results BYTE "The results will be uploaded on LMS.",0dh,0ah,0

.code
main PROC

    mov ebx,OFFSET caption
    mov edx,OFFSET question
    call MsgBoxAsk
    ;(check return value in EAX)

    exit
main ENDP
END main
```

6.7.4 Random Number Generation

```

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 between -50 and +49
    mov     ecx,10                      ; loop 10 times

L1:  mov     eax,100                    ; values 0-99
     call    RandomRange                 ; generate random int
     sub     eax,50                      ; vaues -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

```


6.7.5 Calculation of 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                            ; eat 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

```