Procedures

* 1. Stack Operations
     1. Runtime Stack (32-Bit Mode)
     2. PUSH and POP Instructions
     3. Section Review
  2. Defining and Using Procedures
     1. PROC Directive
     2. CALL and RET Instructions
     3. Nested Procedure Calls
     4. Passing Register Arguments to Procedures
     5. Example: Summing an Integer Array
     6. Saving and Restoring Registers
     7. Section Review
  3. Linking to an External Library
     1. Background Information
     2. Section Review
  4. The Irvine32 Library
     1. Motivation for Creating the Library

**5**

* + 1. Overview
    2. Individual Procedure Descriptions
    3. Library Test Programs
    4. Section Review
  1. 64-Bit Assembly Programming
     1. The *Irvine64* Library
     2. Calling 64-Bit Subroutines
     3. The x64 Calling Convention
     4. Sample Program that Calls a Procedure
  2. Chapter Summary
  3. Key Terms
     1. Terms
     2. Instructions, Operators, and Directives
  4. Review Questions and Exercises
     1. Short Answer
     2. Algorithm Workbench
  5. Programming Exercises

This chapter introduces you to procedures, also known subroutines and functions. Any program of reasonable size needs to be divided into parts, and certain parts need to be used more than once. You will see that parameters can be passed in registers, and you will learn about the runtime stack that the CPU uses to track the calling location of procedures. Finally, we will introduce you to two code libraries supplied with this book, named Irvine32 and Irvine64, containing useful utilities that simplify input–output.

**139**

# Stack Operations

If we place ten plates on each other as in the following diagram, the result can be called a stack. While it might be possible to remove a dish from the middle of the stack, it is much more com- mon to remove from the top. New plates can be added to the top of the stack, but never to the bottom or middle (Fig. 5–1):

**Figure 5–1** Stack of plates

 Top

10

9

8

7

6

5

4

3

2

1

 Bottom

A *stack data structure* follows the same principle as a stack of plates: New values are added to the top of the stack, and existing values are removed from the top. Stacks in general are useful structures for a variety of programming applications, and they can easily be implemented using object-oriented programming methods. If you have taken a programming course that used data structures, you have worked with the *stack abstract data type*. A stack is also called a LIFO struc- ture (*Last-In, First-Out*) because the last value put into the stack is always the first value taken out.

In this chapter, 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.

## 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 indi- rectly modified by instructions such as CALL, RET, PUSH, and POP.

ESP always points to the last value to be added to, or *pushed* on, the top of stack. To demon- strate, let’s begin with a stack containing one value. In Fig. 5-2, the ESP contains hexadecimal 00001000, the offset of the most recently pushed value (00000006). In our diagrams, the top of the stack moves downward when the stack pointer decreases in value:

**Figure 5–2** A stack containing a single value

Offset

**00000006**

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

 ESP = 00001000h

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

The runtime stack discussed here is not the same as the *stack abstract data type* (ADT) discussed in data structures courses. The runtime stack works at the system level to handle subroutine calls. The stack ADT is a programming construct typically written in a high-level programming language such as C++ or Java. It is used when implementing algorithms that depend on last-in, first-out operations.

### 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. Figure 5-3 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 stack of plates we saw earlier, because the runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, ESP = 00001000h; after the push, ESP = 00000FFCh. Figure 5-4 shows the same stack after pushing a total of four integers.

**Figure 5–3** Pushing integers on the stack.

Before

**00000006**

After

**00000006**

**000000A5**

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

 ESP

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

 ESP

**Figure 5–4** Stack, after pushing 00000001 and 00000002.

Offset

00000006

000000A5

00000001

00000002

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

ESP

### 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. Figure 5-5 shows the stack before and after the value 00000002 is popped.

**Figure 5–5** Popping a value from the runtime stack.

Before

**00000002**

**00000001**

**000000A5**

**00000006**

After

**00000001**

**000000A5**

**00000006**

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

 ESP

**00001000**

**00000FFC**

**00000FF8**

**00000FF4**

**00000FF0**

 ESP

The area of the stack below ESP is logically empty, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

### Stack Applications

There are several important uses of runtime stacks in programs:

* 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.

## PUSH and POP Instructions

### 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 decre- mented by 4. There are three instruction formats:

PUSH *reg/mem16* PUSH *reg/mem32* PUSH *imm32*

### 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 incre- mented by 2; if the operand is 32 bits, ESP is incremented by 4:

POP *reg/mem16*

POP *reg/mem32*

### 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. There are times when it is useful to make a backup copy of the flags so you can restore them to their former values later. Often, we enclose a block of code within PUSHFD and POPFD:

pushfd ; save the flags

;

; any sequence of statements here...

;

popfd ; restore the flags

When using pushes and pops of this type, be sure the program’s execution path does not skip over the POPFD instruction. When a program is modified over time, it can be tricky to remem- ber where all the pushes and pops are located. The need for precise documentation is critical!

A less error-prone way to save and restore the flags is to push them on the stack and immedi- ately 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

### PUSHAD, PUSHA, POPAD, and POPA

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI. The POPAD instruction pops the same registers off the stack in reverse order. Similarly, the PUSHA instruc- tion, pushes the 16-bit general-purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed. The POPA instruction pops the same registers in reverse. You should only use PUSHA and POPA when programming in 16-bit mode. We cover 16-bit programming in Chapters 14–17.

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 follow- ing code fragment is an example:

MySub PROC

pushad ; save general-purpose registers

.

.

mov eax,...

mov edx,...

mov ecx,...

.

.

popad ; restore general-purpose registers ret

MySub ENDP

An important exception to the foregoing example must be pointed out; 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

### Example: Reversing a String

Let’s look at a program named *RevStr* that loops through a string and pushes each character on the stack. It then pops the letters from the stack (in reverse order) and stores them back into the same string variable. Because the stack is a LIFO (*last-in, first-out*) structure, the letters in the string are reversed:

; Reversing a String (RevStr.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess PROTO,dwExitCode:DWORD

.data

aName BYTE "Abraham Lincoln",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,

; and store in the aName array. mov ecx,nameSize

mov esi,0

L2: pop eax ; get character mov aName[esi],al ; store in string inc esi

loop L2

INVOKE ExitProcess,0 main ENDP

END main

## Section Review

* + - 1. Which register (in 32-bit mode) manages the stack?
      2. How is the runtime stack different from the stack abstract data type?
      3. Why is the stack called a LIFO structure?
      4. When a 32-bit value is pushed on the stack, what happens to ESP?
      5. *(True/False):* Local variables in procedures are created on the stack.
      6. *(True/False):* The PUSH instruction cannot have an immediate operand.

# Defining and Using Procedures

If you’ve already studied a high-level programming language, you know how useful it can be to divide programs into *subroutines.* A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively. In assembly language, we typi- cally use the term *procedure* to mean a subroutine. In other languages, subroutines are called methods or functions.

In terms of object-oriented programming, the functions or methods in a single class are roughly equivalent to the collection of procedures and data encapsulated in an assembly lan- guage module. Assembly language was created long before object-oriented programming, so it doesn’t have the formal structure found in object-oriented languages. Assembly programmers must impose their own formal structure on programs.

## PROC Directive

### Defining a Procedure

Informally, we can define a *procedure* as a named block of statements that ends in a return state- ment. 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**, for example,

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

### 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::

In terms of program design, it’s 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. For more informa- tion about the runtime stack, see Section 8.2.

### Example: SumOf Three Integers

Let’s create a procedure named **SumOf** that calculates the sum of three 32-bit integers. We will assume that relevant integers are assigned to EAX, EBX, and ECX before the procedure is called. The procedure returns the sum in EAX:

SumOf PROC

add eax,ebx add eax,ecx ret

SumOf ENDP

### Documenting Procedures

A good habit to cultivate is that of adding clear and readable documentation to your programs. The following are a few suggestions for information that you can put at the beginning of each procedure:

* 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 pro- cedure 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.

The descriptive labels we’ve chosen, such as Receives, Returns, and Requires, are not absolutes; other useful names are often used.

With these ideas in mind, let’s add appropriate documentation to the **SumOf** procedure:

;---------------------------------------------------------

; 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

Functions written in high-level languages like C and C++ typically return 8-bit values in AL, 16-bit values in AX, and 32-bit values in EAX.

## CALL and RET Instructions

The CALL instruction calls a procedure by directing the processor to begin execution at a new mem- ory 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. Mechanically speaking, 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 mem- ory pointed to by EIP (instruction pointer register). In 16-bit mode, IP points to the instruction.

### Call and Return Example

Suppose that in **main**, a CALL statement is located at offset 00000020. Typically, this instruc- tion 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 (Fig. 5-6), 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, the value in the stack

pointed to by ESP is popped into EIP (step 1 in Fig. 5-7). In step 2, ESP is incremented so it points to the previous value on the stack (step 2).

**Figure 5–6** Executing a CALL instruction.

 ESP

00000025

????

00000040

EIP

**Figure 5–7** Executing the RET instruction.

Step 1:

ESP

EIP

00000025

????

00000025

Step 2:

????

ESP

## Nested Procedure Calls

A *nested procedure call* occurs when a called procedure calls another procedure before the first procedure returns. Suppose that **main** calls a procedure named **Sub1**. While **Sub1** is executing, it calls the **Sub2** procedure. While **Sub2** is executing, it calls the **Sub3** procedure. The process is shown in Fig. 5-8.

When the RET instruction at the end of **Sub3** executes, it pops the value at stack[ESP] into the instruction pointer. This causes execution to resume at the instruction following the call **Sub3** instruction. The following diagram shows the stack just before the return from **Sub3** is executed:

 ESP

(ret to Sub2)

(ret to Sub1)

(ret to main)

**Figure 5–8** Nested procedure calls.

**main proc**

**.**

**.**

**call Sub1 exit**

**main endp**

**Sub1 proc**

**.**

**.**

**call Sub2 ret**

**Sub1 endp**

**Sub2 proc**

**.**

**.**

**call Sub3 ret**

**Sub2 endp**

**Sub3 proc**

**.**

**.**

**ret**

**Sub3 endp**

After the return, ESP points to the next-highest stack entry. When the RET instruction at the end of **Sub2** is about to execute, the stack appears as follows:

 ESP

(ret to Sub1)

(ret to main)

Finally, when **Sub1** returns, stack[ESP] is popped into the instruction pointer, and execution resumes in **main**:

 ESP

(ret to main)

Clearly, the stack proves itself a useful device for remembering information, including nested procedure calls. Stack structures, in general, are used in situations where programs must retrace their steps in a specific order.

## Passing Register Arguments to Procedures

If you write a procedure that performs some standard operation such as calculating the sum of an integer array, it’s not a good idea to include references to specific variable names inside the pro- cedure. 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 ele- ments. We call these *arguments* (or *input parameters)*. In assembly language, it is common to pass arguments inside general-purpose registers.

In the preceding section 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

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

## Example: Summing an Integer Array

A very common type of loop that you may have already coded in C++ or Java is one that calcu- lates the sum of an integer array. This is very easy to implement in assembly language, and it can be coded in such a way that it will run as fast as possible. For example, one can use registers rather than variables inside a loop.

Let’s create a procedure named **ArraySum** that receives two parameters from a calling pro- gram: a pointer to an array of 32-bit integers, and a count of the number of array values. It calcu- lates and returns the sum of the array in EAX:

;-----------------------------------------------------

; 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

Nothing in this procedure is specific to a certain array name or array size. It could be used in any program that needs to sum an array of 32-bit integers. Whenever possible, you should also create procedures that are flexible and adaptable.

### Testing the ArraySum Procedure

The following program tests the ArraySum procedure by calling it and passing the offset and length of an array of 32-bit integers. After calling ArraySum, it saves the procedure’s return value in a variable named theSum.

; Testing the ArraySum procedure (TestArraySum.asm)

.386

.model flat, stdcall

.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.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 END main

## Saving and Restoring Registers

In the **ArraySum** example, ECX and ESI were pushed on the stack at the beginning of the pro- cedure and popped at the end. This action is typical of most procedures that modify registers. Always save and restore registers that are modified by a procedure so the calling program can be sure that none of its own register values will be overwritten. The exception to this rule pertains to registers used as return values, usually EAX. Do not push and pop them.

### USES Operator

The USES operator, coupled with the PROC directive, 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 Section 5.2.5 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

The corresponding code generated by the assembler shows the effect of USES:

ArraySum PROC

**push esi**

**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**

**pop esi**

ret

ArraySum ENDP

**5.3 Linking to an External Library 153**

*Debugging Tip:* When using the Microsoft Visual Studio debugger, you can view the hidden machine instructions generated by MASM’s advanced operators and directives. Right-click in the Debugging window and select *Go to Disassembly*. This window displays your program’s source code along with hidden machine instructions generated by the assembler.

*Exception* There is an important exception to our standing rule about saving registers that applies when a procedure returns a value in a register (usually EAX). In this case, the return reg- ister should not be pushed and popped. For example, in the SumOf procedure in the following example, it pushes and pops EAX, causing the procedure’s return value to be lost:

SumOf PROC ; sum of three integers push eax ; save EAX

add eax,ebx ; calculate the sum

add eax,ecx ; of EAX, EBX, ECX

pop eax ; lost the sum! ret

SumOf ENDP

## Section Review

* + - 1. *(True/False):* The PROC directive begins a procedure and the ENDP directive ends a procedure.
      2. *(True/False):* It is possible to define a procedure inside an existing procedure.
      3. What would happen if the RET instruction was omitted from a procedure?
      4. How are the words *Receives* and *Returns* used in the suggested procedure documentation?
      5. *(True/False):* The CALL instruction pushes the offset of the CALL instruction on the stack.
      6. *(True/False):* The CALL instruction pushes the offset of the instruction following the CALL on the stack.

# 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 pro- cedures 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.

## 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 instruc- tion 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 pro- cedure 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.

links to

Irvine32.lib

can link to

Your program

kernel32.lib

links to

executes

kernel32.dll

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.

## Section Review

* + - 1. *(True/False)*: A link library consists of assembly language source code.
      2. Use the PROTO directive to declare a procedure named **MyProc** in an external link library.
      3. Write a CALL statement that calls a procedure named **MyProc** in an external link library.
      4. What is the name of the 32-bit link library supplied with this book?
      5. What type of file is **kernel32.dll**?

# The Irvine32 Library

## 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 con- tinue 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 Stan- dard 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.

## Overview

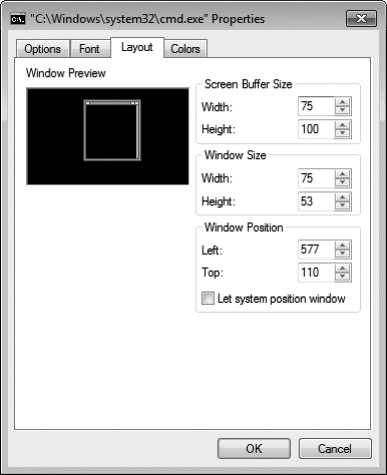
*Console Window* The *console window* (or *command window*) is a text-only window created by MS-Windows when a command prompt is displayed.

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 operat- ing 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.

## 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 pro- gram 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 writ- ing. 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 hexa- decimal. 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; other- wise, 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 file- names. 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 com- mand 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 con- sole 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 follow- ing 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 win- dow. 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 cur- rent 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 off- set 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

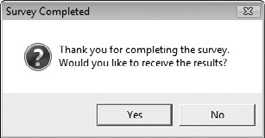
BYTE "Would you like to receive the results?",0

.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 off- set 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 232—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 dig- its. 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 pro- gram. 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

*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 proce- dure 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 = A/CII 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 232—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 charac- ters. 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 A/CII 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 num- ber of characters the user can enter, plus 1 (to save space for the terminating null byte). The pro- cedure 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 fol- lowing 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, multi- ply 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 back- ground 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 dis- play 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 sys- tem function. Sample call:

call WriteWindowsMsg

The following is an example of a message string:

Error 2: The system cannot find the file specified.

## 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 pro- gram’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 expres- sion 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 pro- cedure. 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 inte- gers 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 capa- bilities 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 per- formance. The *GetMseconds* procedure from the book’s library returns the number of millisec- onds 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 sub  ; Display | GetMSeconds eax,startTime  the elapsed time |  | |
| mov call call call | edx,OFFSET msg2 WriteString WriteDec  Crlf | ;  ; | "Elapsed milliseconds: " write the milliseconds |
| exit main ENDP |  |  |  |
| innerLoop  push | PROC  ecx | ; | save current ECX value |
| mov | ecx,0FFFFFFFh | ; | set the loop counter |
| L1: mul  mul mul | eax eax  eax | ; | use up some cycles |

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 mid- night 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 con- stant. 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 set- ting 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 repre- sents 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

## Section Review

* + - 1. Which procedure in the link library generates a random integer within a selected range?
      2. Which procedure in the link library displays “Press [Enter] to continue. . .” and waits for the user to press the Enter key?
      3. Write statements that cause a program to pause for 700 milliseconds.
      4. Which procedure from the link library writes an unsigned integer to the console window in decimal format?
      5. Which procedure from the link library places the cursor at a specific console window location?
      6. Write the INCLUDE directive that is required when using the Irvine32 library.
      7. What types of statements are inside the *Irvine32.inc* file?
      8. What are the required input parameters for the DumpMem procedure?
      9. What are the required input parameters for the ReadString procedure?
      10. Which processor status flags are displayed by the DumpRegs procedure?
      11. *Challenge:* Write statements that prompt the user for an identification number and input a string of digits into an array of bytes.

# 64-Bit Assembly Programming

* + 1. **The *Irvine64* Library**

Our book provides a minimal library to assist you with 64-bit programming, containing the fol- lowing procedures:

* **Crlf**: Writes an end-of-line sequence to the console.
* **Random64**: Generates a 64-bit pseudorandom integer in the range 0 to 264—1. The random value is returned in the RAX register.
* **Randomize**: Seeds the random number generator with a unique value.
* **ReadInt64**: Reads a 64-bit signed integer from the keyboard, terminated by the Enter key. It returns the integer value in the RAX register.
* **ReadString**: Reads a string from the keyboard, terminated by the Enter key. Pass it the offset of the input buffer in RDX, and set RCX to the maximum number of characters the user can enter, plus 1 (for the null terminator byte). It returns a count (in RAX) of the number of char- acters typed by the user.
* **Str\_compare:** Compares two strings. Pass it a pointer to the source string in RSI, and a pointer to the target string in RDI. Sets the Zero and Carry flags in the same way as the CMP (Compare) instruction.
* **Str\_copy:** Copies a source string to the location indicated by a target pointer. Pass the source offset in RSI, and the target offset in RDI.
* **Str\_length:** Returns the length of a null-terminated string in the RAX register. Pass it the string’s offset in RCX.
* **WriteInt64**: Displays the contents of the RAX register as a 64-bit signed decimal integer, with a leading plus or minus sign. It has no return value.
* **WriteHex64**: Displays the contents of the RAX register as a 64-bit hexadecimal integer. It has no return value.
* **WriteHexB**: Displays the contents of the RAX register as a hexadecimal integer in either a 1-byte, 2-byte, 4-byte, or 8-byte format. Pass it the display size (1, 2, 4, or 8) in the RBX register. It has no return value.
* **WriteString**: Displays a null-terminated ASCII string. Pass it the string’s 64-bit offset in RDX. It has no return value.

Although this library is much smaller than our 32-bit library, it contains many of the essential tools you need for making programs more interactive. You are also encouraged to expand this library with your own code as you progress through the book. The *Irvine64* library preserves the values of the RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers. On the other hand, the RAX, RCX, RDX, R8, R9, R10, and R11 register values are usually not preserved.

## Calling 64-Bit Subroutines

If you want to call a subroutine you have created, or a subroutine in the *Irvine64* library, all you have to do is place input parameters in registers and execute the CALL instruction. For example:

mov rax,12345678h call WriteHex64

There’s one other small thing you have to do, which is to use the PROTO directive at the top of your program to identify each procedure you plan to call that’s outside your own program:

|  |  |  |
| --- | --- | --- |
| ExitProcess | PROTO | ; located in the Windows API |
| WriteHex64 | PROTO | ; located in the Irvine64 library |

## The x64 Calling Convention

Microsoft follows a consistent scheme for passing parameters and calling procedures in 64-bit programs known as the *Microsoft x64 Calling Convention*. This convention is used by C/C++ compilers, as well as by the Windows Application Programming Interface (API). The only times you need to use this calling convention is when you either call a function in the Windows API, or you call a function written in C or C++. Here are some of the basic characteristics of this calling convention:

1. The CALL instruction subtracts 8 from the RSP (stack pointer) register, since addresses are 64-bits long.
2. The first four parameters passed to a procedure are placed in the RCX, RDX, R8, and R9, registers, in that order. If only one parameter is passed, it will be placed in RCX. If there is a second parameter, it will be placed in RDX, and so on. Additional parameters are pushed on the stack, in left-to-right order.
3. It is the caller’s responsibility to allocate at least 32 bytes of *shadow space* on the runtime stack, so the called procedures can optionally save the register parameters in this area.
4. When calling a subroutine, the stack pointer (RSP) must be aligned on a 16-byte boundary (a multiple of 16). The CALL instruction pushes an 8-byte return address on the stack, so the calling program must subtract 8 from the stack pointer, in addition to the 32 it already sub- tracts for the shadow space. We will soon show how this is done in a sample program.

The remaining details about the x64 calling convention will be introduced in Chapter 8, when we discuss the runtime stack in greater detail. Here’s the good news: you do not have to use the Microsoft x64 calling convention when calling subroutines in the *Irvine64* library. You only need to use it when calling Windows API functions.

## Sample Program that Calls a Procedure

Let’s create a short program that uses the Microsoft x64 calling convention to call a subroutine named **AddFour**. This subroutine adds the values in the four parameter registers (RCX, RDX, R8, and R9) and saves the sum in RAX. Because procedures normally return integer values in RAX, the calling program expects that value to be in this register when the subroutine returns. In this way, we can say that the subroutine is a function, because it receives four inputs and (deter- ministically) produces a single output.

1: ; Calling a subroutine in 64-bit mode (CallProc\_64.asm) 2: ; Chapter 5 example

3:

4: ExitProcess PROTO

5: WriteInt64 PROTO ; Irvine64 library 6: Crlf PROTO ; Irvine64 library 7:

8: .code

9: main PROC

10: sub rsp,8 ; align the stack pointer

11: sub rsp,20h ; reserve 32 bytes for shadow params 12:

13: mov rcx,1 ; pass four parameters, in order 14: mov rdx,2

15: mov r8,3

16: mov r9,4

17: call AddFour ; look for return value in RAX 18: call WriteInt64 ; display the number

19: call Crlf ; output a CR/LF 20:

21: mov ecx,0

22: call ExitProcess

23: main ENDP

24:

25: AddFour PROC

26: mov rax,rcx

27: add rax,rdx

28: add rax,r8

29: add rax,r9 ; sum is in RAX 30: ret

31: AddFour ENDP

32:

33: END

Let’s examine a few other details in the example: Line 10 aligns the stack pointer to an even 16-byte boundary. Why does this work? Before the OS called main, we assume the stack pointer

was aligned on a 16-byte boundary. Then, when the OS called main, the CALL instruction pushed an 8-byte return address on stack. Subtracting another 8 from the stack pointer drops it down to a multiple of 16.

You can run this program in the Visual Studio debugger and watch the RSP register (stack pointer) change values. When we did this, we saw the hexadecimal values shown graphically in Fig. 5-11. The figure shows only the lower 32 bits of each address, since the upper 32 bits con- tained all zeros:

1. Before line 10 executed, RSP = 01AFE48. This tells us that RSP was equal to 01AFE50 before the OS called our program. (The CALL instruction subtracts 8 from the stack pointer.)
2. After line 10 executed, RSP = 01AFE40, showing that the stack was properly aligned on a 16-byte boundary.
3. After line 11 executed, RSP = 01AFE20, showing that 32 bytes of shadow space were reserved at addresses 01AFE20 through 01AFE3F.
4. Inside the AddFour procedure, RSP = 01AFE18, showing that the caller’s return address had been pushed on the stack.
5. After AddFour returned, RSP again was equal to 01AFE20, the same value it had before call- ing AddFour.

Rather than calling ExitProcess to end the program, we might have chosen to execute a RET instruction, which would return to the process that launched our program. It would require, however, that we restore the stack pointer to the way it was when the **main** procedure began to execute. The following lines would be the replacement for lines 21–22 of the CallProc\_64 program:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 21: | add | rsp,28 | ; | restore the stack pointer |
| 22: | mov | ecx,0 | ; | process return code |
| 23: | ret |  | ; | return to the OS |

***Tip:*** When using the Irvine64 library, add the file named *Irvine64.obj* to your Visual Studio project. To do this in Visual Studio, right-click the project name in the Solution Explorer window, select *Add*, select *Existing Item*, and select the *Irvine64.obj* filename.

**Figure 5–11** Runtime stack for the CallProc\_64 program.

0 1 A F E 4 8

|  |
| --- |
| (return to OS) |
|  |
| shadow p1 |
| shadow p2 |
| shadow p3 |
| shadow p4 |
| (return to main) |

0 1 A F E 4 0

0 1 A F E 3 8

0 1 A F E 3 0

0 1 A F E 2 8

0 1 A F E 2 0

0 1 A F E 1 8