Contents

PROCEDURES IN ASSEMBLY LANGAUGE

Linking External Libraries in Assembly: A Friendly Guide

Want to use an external library in your assembly code? The linker is your best friend! Just use the -l and -L options on the command line.

- **-l** tells the linker *which* library to use.
- **-L** tells it *where* to find the library file.

Example: Let's say you want to link your hello.obj file to the Irvine32 library. Here's how you'd do it:

```
link hello.obj -l Irvine32.lib kernel32.lib
```

This command links your object file to both the Irvine32 library and the kernel32 library.

Why kernel32? It's a system library packed with essential functions, like reading and writing to the console—super handy!

Putting It All Together Here's a simple assembly program that calls a procedure from the Irvine32 library:

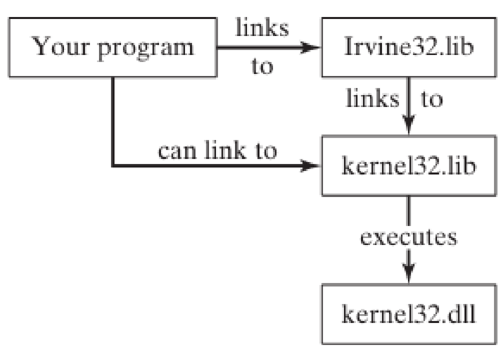


Figure 5–9 Linking 32-bit programs.

```
; This program displays the string "Hello, world!" on the console.

include Irvine32.inc

.data
msg db "Hello, world!", 0

.code
start:
    mov eax, OFFSET msg
    call WriteString
    mov eax, 1 ; exit program
    int 0x80
```

This little program prints "Hello, world!" to the console. Here's how it works:

How Linking Works: Your Flowchart Explained

Your flowchart does a great job showing how the linker turns your code into a working program! Here's a quick breakdown:

1. **Reads Your Code:** The linker starts by reading your program's object file and building a *symbol table*—a list of all the names (like functions or variables) and their addresses.
2. **Searches Libraries:** It then looks through Irvine32.lib and kernel32.lib for any functions or data your program needs. If it finds a match, it copies the code into your final program.
3. **Updates Addresses:** The linker makes sure everything points to the right place, updating the symbol table as it goes.
4. **Repeats Until Done:** It keeps going until every reference is resolved.
5. **Creates the Executable:** Finally, it writes your finished program to disk—ready to run!

Bonus: The flowchart skips some details (like fixing up addresses for different memory locations), but it's a perfect big-picture view!

The Runtime Stack: Your Program's Scratch Pad

The stack is like a stack of plates—you can only add or remove from the top. It's where your program stores function arguments, return values, and local variables.

- **Push:** Decrements the stack pointer (ESP) by 4 bytes and stores a 32-bit value at the new top.
- **Pop:** Grabs the top 32-bit value and increments ESP by 4 bytes.

Example: Here's how you'd push and pop a value in assembly:

```
; Push the value 10 onto the stack.
push 10

; Pop the value off of the stack and store it in the register EAX.
pop eax
```

How the Stack Grows in Memory

Yes, you've got it! The stack grows *downward*—meaning it starts at a high memory address and moves toward lower ones as you push more data.

Your Example:

- Suppose your stack starts at address 4096 (the top).
- You push 10: the stack pointer moves down to 4092.
- You push 11: the stack pointer moves down to 4088.
- You pop: the stack pointer moves back up to 4092.

In short:

- **Push:** Stack pointer moves to a lower address.
- **Pop:** Stack pointer moves back up.

This "last in, first out" behavior is what makes the stack so useful for managing function calls and local variables.

Figure 5–3 Pushing integers on the stack.

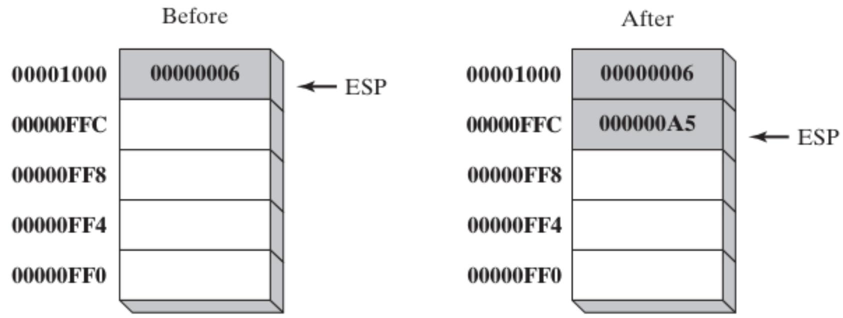
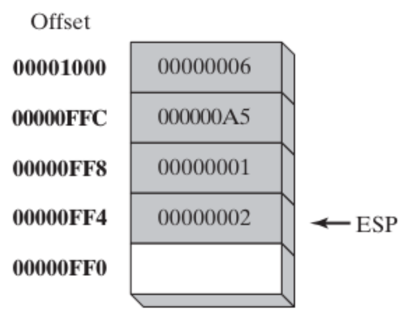


Figure 5–4 Stack, after pushing 00000001 and 00000002.



Push Operation: Adding to the Stack

In a 32-bit system, pushing a value onto the stack is a two-step process:

1. **Move the Stack Pointer:** The stack pointer (ESP) is decremented by 4 bytes, making room for the new value.
2. **Store the Value:** The value is copied into the memory location now pointed to by ESP.

Key Point: The stack pointer always points to the last item pushed, and the stack grows *downward*—from higher memory addresses to lower ones.

Pop Operation: Removing from the Stack

Popping a value off the stack is just as simple:

1. **Remove the Value:** The value at the memory location pointed to by ESP is "removed" (i.e., it's read and no longer considered part of the stack).
2. **Move the Stack Pointer:** ESP is incremented by 4 bytes, pointing to the next item on the stack.

Example:

- Before pop: ESP points to 00000002.
- After pop: 00000002 is removed, and ESP moves up by 4 bytes.

Saving and Restoring Flags

In assembly, you can save and restore the processor flags using PUSHFD and POPFD. This is handy for preserving the state of your program, especially before and after interrupts or critical sections.

Tip: If you want to be extra safe, you can push the flags onto the stack and immediately pop them into a variable. This makes your code clearer and less prone to errors—you can always see where the flags are stored and when they're restored.

Just remember: Always make sure your POPFD (or flag restore) happens at the right time, or your program might not behave as expected!

PUSHFD and POPFD Operation

```
.data
savedFlags DWORD ?

.code
main PROC
    ; Save the flags using PUSHFD
    pushfd

    ; Store the saved flags into the 'savedFlags' variable
    pop savedFlags

    ; Modify some flags (for demonstration purposes)
    cli  ; Clear Interrupt Flag
    sti  ; Set Interrupt Flag

    ; Restore the original flags using 'savedFlags'
    push savedFlags
    popfd

    ; Your code continues here

    ; Exit the program
    mov eax, 0x4C00  ; Exit code 0
    int 0x21        ; Call DOS function to exit
main ENDP

END main
```

PUSHFD in x86 Assembly: Your Friendly Flags Saver

The PUSHFD instruction is like a helpful assistant for your flags register: it knows exactly where to find all your flags and automatically pushes the entire 32-bit EFLAGS register onto the stack. No need to tell it which flags to grab—it just does it all for you!

This means it saves everything: status flags like the Zero Flag (ZF), Carry Flag (CF), and Overflow Flag (OF), plus control flags such as the Direction Flag (DF) and Interrupt Flag (IF). When you're ready to restore things, just use POPFD to pull those flags right back where they belong.

How It Works in Practice

Let's say you want to temporarily tweak some flags, but you don't want to lose the original settings. Here's how you can do it safely:

1. **Save the Flags:** Use PUSHFD to push the current flags onto the stack.
2. **Store Them:** Pop the flags off the stack and tuck them away in a variable (like savedFlags).
3. **Play Around:** Clear or set flags as needed—maybe disable interrupts with CLI or re-enable them with STI.
4. **Restore Order:** When you're done, push your saved flags back onto the stack and use POPFD to restore the original state.

Example Walkthrough

```
.data
    savedFlags DWORD 0  ; A cozy spot to store your flags

.code
main PROC
    PUSHFD           ; Save all flags to the stack
    POP savedFlags   ; Store them in savedFlags

    ; --- Do your thing ---
    CLI              ; Disable interrupts (just for fun)
    STI              ; Re-enable interrupts

    ; --- Restore the original flags ---
    PUSH savedFlags ; Put your saved flags back on the stack
    POPFD            ; Restore the flags register

    ; Exit gracefully
    MOV eax, 0x4C00
    INT 0x21
main ENDP
```

or

```
; 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
    ; Initialize registers and counters
    mov ecx,nameSize
    mov esi,0
```

```
L1:
    ; Get the next character from the 'aName' string
    movzx eax,aName[esi]

    ; Push the character onto the stack
    push eax

    ; Move to the next character
    inc esi

    ; Continue the loop until all characters are processed
    loop L1

    ; Pop the characters from the stack (in reverse order)
    ; and store them back into the 'aName' array

    ; Reset counters
    mov ecx,nameSize
    mov esi,0

L2:
    ; Get the character from the stack
    pop eax

    ; Store the character in the 'aName' string
    mov aName[esi],al

    ; Move to the next character
    inc esi

    ; Continue the loop until all characters are restored
    loop L2

    ; Exit the program
    INVOKE ExitProcess,0

main ENDP
END main
```

A Word of Caution While PUSHFD and POPFD make flag management a breeze, always double-check that you're saving and restoring them correctly. Messing up here can lead to unexpected program behavior—so handle with care!

How This Code Reverses a String Using the Stack

This assembly program reverses the string "Abraham Lincoln" using the stack's natural Last-In-First-Out (LIFO) behavior. Here's the breakdown:

1. Setup

- **.data Section:**
  - aName holds the string "Abraham Lincoln" (plus a null terminator).
  - nameSize calculates the string's length (excluding the null terminator).
- **.code Section:**
  - ecx is loaded with the string length (loop counter).
  - esi starts at 0 (string index).

2. Pushing Characters onto the Stack (Loop L1)

- **What happens:**
  - Load each character from aName[esi] into eax.
  - Push eax onto the stack.
  - Move to the next character (inc esi).
  - Repeat until all characters are pushed.
- **Why it works:** The stack's LIFO order means the first character pushed will be the last one popped—perfect for reversing!

3. Popping Characters Back (Loop L2)

- **What happens:**
  - Pop each character from the stack into eax.
  - Store it back into aName[esi].
  - Move to the next position (inc esi).
  - Repeat until all characters are restored.
- **Result:** The string is now reversed in place!

4. Wrap-Up

- The program exits cleanly with INVOKE ExitProcess, 0.

In short: The code uses the stack to flip the string—simple, clever, and efficient!

The Tricky Bits: What Makes This Assembly Code Challenging

```
; Example: Reverse "Abraham Lincoln" using stack operations - MASM syntax, Windows environment
.386
.model flat, stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib

.data
    aName db "Abraham Lincoln",0        ; original string with null terminator
    nameSize equ ($ - aName) - 1        ; length of string excluding null terminator

.code
main PROC
    ; Initialize registers
    mov esi, 0              ; index into string
    mov ecx, nameSize       ; loop counter = string length

L1: ; Push each character onto stack
    movzx eax, byte ptr aName[esi]  ; load character into eax (zero-extended)
    push eax                        ; push onto stack
    inc esi                         ; move to next character
    loop L1                         ; repeat until ecx = 0

    ; Reset index and counter for popping
    mov esi, 0
    mov ecx, nameSize

L2: ; Pop characters back into string (reversing order)
    pop eax                         ; get character from stack
    mov al, byte ptr eax            ; use only low 8 bits
    mov aName[esi], al              ; store back into string
    inc esi
    loop L2

    ; Exit program gracefully
    INVOKE ExitProcess, 0
main ENDP
END main
```

🎯 1. Playing with Strings

We're working with the string "Abraham Lincoln" as a bunch of bytes in memory. By using aName[esi], the code steps through each character one at a time. The star of the show here is movzx: it grabs a single byte (your character) and safely drops it into a 32-bit register, padding the rest with zeros so you don't end up with random junk data.

🔄 2. Loops Made Simple

ecx is our countdown timer. It starts at the length of the string and ticks down to zero. The loop instruction is like a built-in shortcut—it automatically subtracts one from ecx and jumps back to the loop until we're done. If you're new to assembly, this feels a little magical at first, but once you see it, it clicks.

📦 3. Stack Magic

Think of the stack as a "last in, first out" box. Push characters in order, then pop them back out, and voilà—you've reversed the string! It's clever, but it can be tricky to picture until you imagine the stack as a spring-loaded box that spits out the last thing you put in.

🤹 4. Register Juggling

Each register has its own job:

- **eax**: Holds the current character.
- **esi**: Points to where we are in the string.
- **ecx**: Counts how many times we loop.
- **al**: The tiny lower 8 bits of eax, perfect for writing characters back.

It's like juggling—you've got to keep track of which ball (register) is doing what.

✅ 5. Ending Gracefully

INVOKE ExitProcess, 0 is our polite way of saying "all done!" The 0 means everything went fine. No drama, just a clean exit.

📏 6. Measuring the String

nameSize is calculated as ($ - aName) - 1. Here, $ is the current address, and aName is the start of the string. Subtracting gives us the length, and we shave off one to ignore the null terminator. If pointer math feels weird, don't worry—it's just subtraction with addresses.

⛰️ 7. Control Flow Adventures

The code hops between loops (L1 and L2), pushing and popping characters. Understanding how the program jumps around is key—it's like following a treasure map. Once you see the flow, debugging and writing your own assembly gets way easier.

🌟 Why It's Worth the Effort

Assembly is all about precision and control. At first, these tricks feel like riddles, but once you get the hang of them, you'll see how powerful (and even fun!) low-level programming can be. Keep practicing, and soon these "tricky bits" will feel like second nature.

QUICK REVIEW: STACK QUESTIONS MADE SIMPLE

Which register manages the stack in 32-bit mode?

That's **ESP (Extended Stack Pointer)**. It always points to the top of the stack, and every time you push or pop something, ESP moves to keep track.

📚 Runtime Stack vs. Stack Data Type

- **Runtime stack**: A real chunk of memory used by programs to handle function calls, local variables, and flow control. It grows when you call a function and shrinks when you return.
- **Stack data type**: A higher-level idea in computer science—just a structure that supports push and pop operations, following the **Last-In-First-Out (LIFO)** rule. Think of it this way: the runtime stack is the *real-world implementation*, while the stack data type is the *concept*.

🔄 Why is the stack called LIFO?

Because the **last thing you put in is the first thing that comes out**. Just like stacking plates—you grab the top one first.

What happens to ESP when you push a 32-bit value?

ESP moves down by **4 bytes** (since 32 bits = 4 bytes). That way, ESP always points to the new top of the stack.

✅ True/False Quickies

- **Local variables live on the stack? → True.** That's how functions keep their own private workspace.
- **PUSH can't take an immediate operand? → False.** You *can* push a value directly, like PUSH 42.

⚙️ Assembly Instructions & Directives

PROC / ENDP → Mark the start and end of a procedure. Think of them as bookends for your function.

- **PUSH** → Put a value on the stack.
- **POP** → Take the top value off the stack and store it somewhere.
- **PUSHA / PUSHAD** → Push *all* general-purpose registers at once (16-bit vs. 32-bit versions).
- **POPA / POPAD** → Pop them all back in the same order.
- **PUSHFD / POPFD** → Save and restore the processor's flags (EFLAGS). Handy for preserving state.
- **RET** → Return from a procedure. It pops the return address off the stack and jumps back.
- **USES** → A directive that documents which registers a procedure touches. Example:

```
MyProcedure PROC USES RAX, RBX
```

- This makes it clear to other programmers which registers are affected.

🌟 Why This Matters

These instructions are the building blocks of how assembly manages memory, flow, and state. Once you see them as simple "stack tricks" and "procedure bookends," they stop feeling intimidating and start feeling like neat little tools.

```
.model flat, stdcall
.stack 4096

.data
    message db "Hello, World!", 0

.code
    MyProcedure PROC USES EAX, EBX
        ; Your code here that uses EAX and EBX registers
        mov eax, 42
        mov ebx, 24
        add eax, ebx
        ; Rest of your code
        ret
    MyProcedure ENDP

    main PROC
        ; Push all general-purpose registers onto the stack
        pushad
        ; Call MyProcedure
        call MyProcedure
        ; Pop all general-purpose registers from the stack
        popad
        ; Push processor flags onto the stack
        pushfd
        ; Modify some flags (for demonstration purposes)
        cld ; Clear direction flag
        ; Pop processor flags from the stack
        popfd
        ; Display a message
        mov eax, 4          ; syscall number for sys_write (Linux)
        mov ebx, 1          ; file descriptor 1 (stdout)
        mov ecx, offset message  ; pointer to the message
        mov edx, 13         ; message length
        int 0x80            ; invoke syscall
        ; Exit program
        mov eax, 1          ; syscall number for sys_exit (Linux)
        int 0x80            ; invoke syscall
    main ENDP
END main
```

🧩 PROCEDURES IN ASSEMBLY LANGUAGE

Procedures are like mini-programs inside your program.

They break big tasks into smaller, manageable pieces, making your code easier to understand, test, and reuse.

In assembly, a **procedure** is simply a named block of code that you can call from anywhere.

Unlike higher-level languages, assembly doesn't enforce structure, so programmers use procedures to bring order and modularity to their work.

✨ The PROC Directive

The PROC directive defines a procedure. You give it a name, write the code inside, and close it with ENDP. Think of PROC and ENDP as the "start" and "end" markers for your subroutine.

Example: A Simple Sum Procedure

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

This procedure adds three 32-bit integers together and leaves the result in the EAX register.

📞 Calling a Procedure

To use a procedure, you just call it by name. For example:

```
mov eax, 1
mov ebx, 2
mov ecx, 3

call SumOf

; eax now contains the sum of 1, 2, and 3, which is 6
```

Here, the program sets up values in registers, calls SumOf, and gets the result back in EAX.

🔙 Returning from a Procedure

Procedures return control with the RET instruction. RET pops the return address off the stack and jumps back to wherever the procedure was called. Simple, clean, and efficient.

```
.code
main PROC
    ; Call the MyProcedure
    call MyProcedure

    ; Continue executing after the call

    ; Exit the program
    exit

MyProcedure PROC
    ; Your procedure code here

    ; Return to the caller
    ret

MyProcedure ENDP

main ENDP
END main
```

🌟 Why Procedures Matter

- **Clarity**: Breaks complex programs into smaller tasks.
- **Reusability**: Write once, call anywhere.
- **Structure**: Helps assembly programmers organize code in a language that doesn't enforce it.

Understanding the Example Step by Step

📂 .code Section

This is where the actual program lives. It contains both the **main program** and any procedures you define, like MyProcedure.

🚀 main PROC

This marks the start of the main program. Think of it as the entry point where execution begins.

📞 call MyProcedure

The CALL instruction is how you jump into a procedure. When the CPU executes CALL, two things happen:

1. **Return address pushed** → The address of the next instruction (right after CALL) is saved on the stack.
2. **Instruction pointer updated** → The CPU loads the address of MyProcedure into EIP (the instruction pointer), so execution continues inside the procedure.

After the procedure finishes, the program will pick up right where it left off—thanks to that saved return address.

🛠️ MyProcedure PROC

This is the start of the procedure itself. Inside here, you write the code that defines what the procedure does.

🔙 ret

The RET instruction is how you exit a procedure. It pops the saved return address off the stack and loads it back into EIP. That means the CPU "jumps back" to the instruction immediately after the CALL in the main program.

🏁 MyProcedure ENDP

Marks the end of the procedure. It's the closing bracket for your subroutine.

🏁 main ENDP

Marks the end of the main program.

🏁 END main

Tells the assembler: "This is the end of the program, and execution starts at main."

✨ Big Picture Summary

- **CALL** → jumps into a procedure and saves the return address on the stack.
- **RET** → pops that return address off the stack and jumps back to the caller. Together, they make structured program flow possible in assembly—like function calls in higher-level languages.

Nested Procedure Calls

A **nested procedure call** happens when one procedure calls another before finishing. In other words, Procedure A calls Procedure B, and only after B returns does A continue.

```
.code
main PROC
    ; Call the OuterProcedure
    call OuterProcedure
    ; Continue executing after the call to OuterProcedure
    ; Exit the program
    exit

OuterProcedure PROC
    ; Some code in the OuterProcedure
    ; Call the InnerProcedure
    call InnerProcedure
    ; Continue executing in OuterProcedure after the call to InnerProcedure
    ; Return from OuterProcedure
    ret

OuterProcedure ENDP
InnerProcedure PROC
    ; Some code in the InnerProcedure
    ; Return from InnerProcedure
    ret

InnerProcedure ENDP
main ENDP
END main
```

🗂️ .code Section

This is where the program lives. It contains the **main program** and the two procedures: OuterProcedure and InnerProcedure.

🚀 main PROC

Marks the start of the main program. This is the entry point where execution begins.

📞 call OuterProcedure

The main program calls OuterProcedure. When the CPU executes CALL, it:

1. Pushes the return address (the spot right after the CALL) onto the stack.
2. Jumps into OuterProcedure by loading its address into the instruction pointer (EIP).

So, control moves from main into OuterProcedure.

🛠️ OuterProcedure PROC

This is the start of OuterProcedure. It runs its own code, then calls another procedure—InnerProcedure.

📞 call InnerProcedure

Inside OuterProcedure, the CALL instruction jumps into InnerProcedure. Again, the return address is saved on the stack, and execution continues inside InnerProcedure.

🛠️ InnerProcedure PROC

This is the start of InnerProcedure. It runs its code, then finishes with RET.

🔙 ret in InnerProcedure

The RET pops the saved return address off the stack and jumps back to the instruction immediately after the CALL InnerProcedure in OuterProcedure. Execution resumes in OuterProcedure.

🔙 ret in OuterProcedure

Once OuterProcedure finishes, its RET pops the return address saved earlier and jumps back to the instruction immediately after CALL OuterProcedure in main. Execution resumes in the main program.

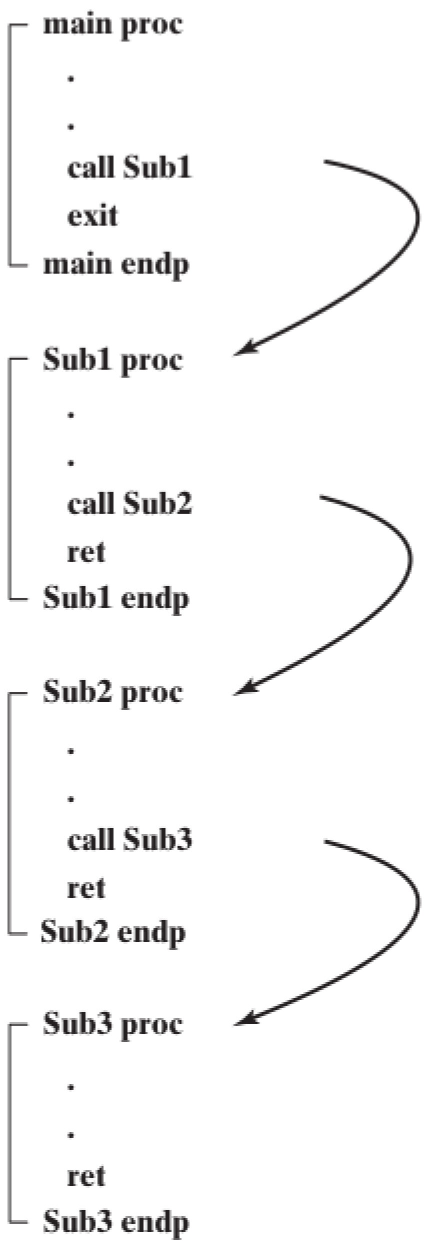
✨ Big Picture

- The **main program** calls OuterProcedure.
- **OuterProcedure** does its work, then calls InnerProcedure.
- **InnerProcedure** runs, returns to OuterProcedure.
- **OuterProcedure** finishes, returns to main.

The stack keeps track of all these return addresses, so the CPU always knows exactly where to go back.

🌟 Why It Matters

Nested procedure calls show how assembly supports structured program flow. With CALL and RET, you can build layered tasks—procedures that call other procedures—while the stack quietly manages all the “return-to-here” addresses behind the scenes.



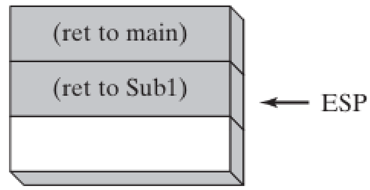
The main procedure calls the Sub1 procedure. Sub1 calls Sub2, and Sub2 calls Sub3. Sub3 then returns to Sub2, Sub2 returns to Sub1, and Sub1 returns to main.

Here is a more detailed explanation of the flow:

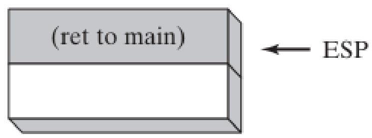
- The main procedure calls Sub1 by executing the CALL Sub1 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub1.
- The CALL instruction then loads the address of Sub1 into the instruction pointer.

- The Sub1 procedure begins executing. Sub1 calls Sub2 by executing the CALL Sub2 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub2.
- The CALL instruction then loads the address of Sub2 into the instruction pointer.
- The Sub2 procedure begins executing.
- Sub2 calls Sub3 by executing the CALL Sub3 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub3.
- The CALL instruction then loads the address of Sub3 into the instruction pointer.
- The Sub3 procedure begins executing. Sub3 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub2 from the stack into the instruction pointer.
- Sub2 begins executing again.
- Sub2 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub1 from the stack into the instruction pointer.
- Sub1 begins executing again.
- Sub1 returns by executing the RET instruction.
- The RET instruction pops the return address for main from the stack into the instruction pointer.
- The main procedure begins executing again.
- The stack is used to keep track of the return addresses for the nested procedure calls.
- When a procedure calls another procedure, it pushes its return address onto the stack.
- When a procedure returns, it pops its return address from the stack.
- This ensures that the procedures return to the correct location in the program.

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:



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

Here is a more detailed explanation of the image:

- The ret to main entry contains the address of the instruction following the CALL Sub1 instruction in the main procedure.
- This is the return address for the main procedure.
- The ret to Sub1 entry contains the address of the instruction following the CALL Sub2 instruction in the Sub1 procedure.
- This is the return address for the Sub1 procedure.
- The ret to Sub2 entry contains the address of the instruction following the CALL Sub3 instruction in the Sub2 procedure.
- This is the return address for the Sub2 procedure.
- The stack is used to keep track of the return addresses for the nested procedure calls.
- When a procedure calls another procedure, it pushes its return address onto the stack.
- When a procedure returns, it pops its return address from the stack.
- This ensures that the procedures return to the correct location in the program.
- In the image, Sub3 has just returned.

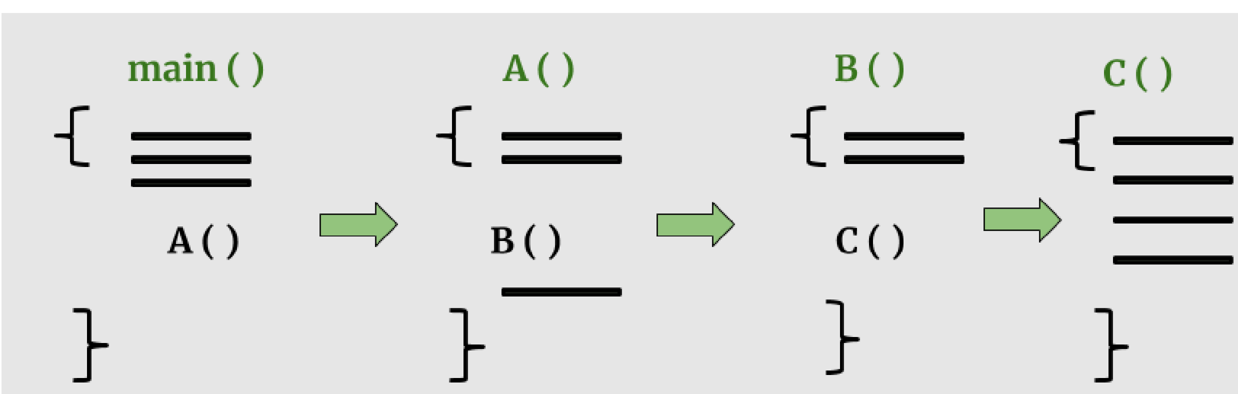
- The RET instruction in Sub3 popped the return address for Sub2 from the stack and loaded it into the instruction pointer.
- This caused Sub2 to begin executing again.
- The next instruction to be executed is the instruction after the CALL Sub3 instruction in the Sub2 procedure.
- After Sub2 finishes executing, it will return to Sub1 by executing the RET instruction.
- The RET instruction in Sub2 will pop the return address for Sub1 from the stack and load it into the instruction pointer.
- This will cause Sub1 to begin executing again.
- The same process will continue until all of the procedures have returned.
- The last procedure to return will be the main procedure.

So, this nesting is basically, putting the address of the procedure that called me onto the stack, executing my code, then returning causes a pop of the address of the procedure that called me? And I will only press return if I don't call another procedure within me?

Yes, your understanding of nested procedure calls is correct. When a procedure is called, the address of the instruction immediately following the CALL instruction is pushed onto the stack as the return address.

This allows the called procedure to execute its code and, when it's ready to return, it uses the RET instruction. The RET instruction pops the return address from the stack and transfers control back to the instruction with that address, effectively "returning" to the calling procedure.

Here's a step-by-step explanation of the process:



The calling procedure (let's call it Procedure A) encounters a CALL instruction to call another procedure (Procedure B).

Before transferring control to Procedure B, Procedure A pushes its own return address (the address of the instruction immediately following the CALL) onto the stack.

Procedure B is executed. When Procedure B is ready to return, it uses the RET instruction.

The RET instruction pops the return address (which is the address of the instruction following the CALL in Procedure A) from the stack.

Control is transferred back to Procedure A, continuing from where it left off.

Your statement, **"I will only press return if I don't call another procedure within me,"** is also accurate.

When a procedure calls another procedure within it (nested calls), it won't use RET until all nested calls are completed.

Each RET instruction corresponds to a single CALL instruction, and they must match.

If you call another procedure within the current one, you need to ensure that all inner procedure calls are resolved before the outer procedure returns.

This maintains the proper stack structure and allows for correct program execution.

```
ArraySum PROC
    ; Input:
    ;   ecx = array size
    ;   esi = pointer to the start of the array
    ; Output:
    ;   eax = sum of array elements

    xor eax, eax         ; Initialize eax to 0 (sum)
L1:
    add eax, [esi]      ; Add the current element to the sum
    add esi, 4          ; Move to the next element (assuming 32-bit integers)
    loop L1             ; Repeat for the remaining elements
    ret

ArraySum ENDP
```

The code snippet you provided appears to be part of a procedure named ArraySum, which is designed to calculate the sum of an array of 32-bit integers.

This code is not directly related to nested procedures but rather demonstrates the concept of using procedures for code modularity and reusability.

loop L1: This instruction is likely inside a loop that iterates through an array. It decrements the ecx register (loop counter) and checks if it's zero. If ecx is not zero, it jumps back to the L1 label, effectively looping through the array elements.

pop ecx: After the loop completes, it pops the original value of ecx from the stack. This is often done to restore the loop counter to its original value, allowing the caller to access it.

pop esi: Similarly, it pops the esi register from the stack. This might be used to restore the pointer or index used to access the array elements.

ret: This is the return instruction, indicating that the procedure is finished. The sum of the array is expected to be in the eax register when the procedure returns.

In this rewritten version, the ArraySum procedure takes ecx as the array size and esi as the pointer to the start of the array. It initializes eax to 0 for the sum and uses a loop to iterate through the array, adding each element to the sum. This makes the procedure more flexible and suitable for various array sizes and starting addresses.

```
; Testing the ArraySum procedure (TestArraySum.asm)
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h ; An array of 32-bit integers
    theSum DWORD ? ; Variable to store the sum

.code
    main PROC
        mov esi, OFFSET array ; ESI points to the array
        mov ecx, LENGTHOF array ; ECX = number of elements in the array
        call ArraySum ; Call the ArraySum procedure to calculate the sum
        mov theSum, eax ; Store the returned sum in theSum variable
        INVOKE ExitProcess, 0 ; Exit the program

    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 (pointer to the array)
        push ecx ; Save ECX (number of elements)
        mov eax, 0 ; Initialize the sum to zero
```

```
L1:
    add eax, [esi] ; Add each integer to the sum
    add esi, TYPE DWORD ; Move to the next integer (assuming 32-bit integers)
    loop L1 ; Repeat for the remaining elements

    pop ecx ; Restore ECX
    pop esi ; Restore ESI
    ret ; Return with the sum in EAX
ArraySum ENDP
```

This program tests the ArraySum procedure, which calculates the sum of an array of 32-bit integers. Let's break it down step by step:

- • • **.386 and .model:** These directives specify the target processor (80386) and the memory model (flat), respectively.
- • • **.stack 4096:** This directive reserves 4096 bytes of stack memory for the program.
- **ExitProcess PROTO, dwExitCode:DWORD:** This line declares the ExitProcess function prototype, which is used later to exit the program with an exit code.
- • • **array DWORD 10000h, 20000h, 30000h, 40000h, 50000h:** This defines an array of 32-bit integers named array. It contains five elements with the values 0x10000, 0x20000, 0x30000, 0x40000, and 0x50000. theSum DWORD ?: This declares a variable named theSum to store the sum of the array. The ? indicates that it's uninitialized.
- • • **main PROC:** This marks the beginning of the main procedure. mov esi, OFFSET array: This sets esi to the memory address (offset) of the array variable, effectively pointing to the beginning of the array.
- • • **mov ecx, LENGTHOF array:** This loads ecx with the number of elements in the array, which is 5 in this case.
- **call ArraySum:** This calls the ArraySum procedure to calculate the sum of the array elements.
- • • **mov theSum, eax:** After ArraySum returns, the sum is stored in eax, and this line moves it to the theSum variable.
- • • **INVOKE ExitProcess, 0:** This invokes the ExitProcess function with an exit code of 0, terminating the program.

Here's a detailed explanation of the ArraySum procedure:

- **push esi and push ecx:** These instructions push the values of esi (pointer to the array) and ecx (number of elements) onto the stack to preserve their values.

- • • **mov eax, 0:** This initializes eax to zero, which will be used to accumulate the sum of the array.
- • • **L1::** This label marks the beginning of a loop.
- • • **add eax, [esi]:** This instruction adds the value at the memory location pointed to by esi (the current element of the array) to eax, effectively accumulating the sum.
- **add esi, TYPE DWORD:** This increments esi by the size of a DWORD (32 bits) to move to the next element in the array.
- • • **loop L1:** This instruction decrements ecx (the loop counter) and checks if it's zero. If ecx is not zero, it jumps back to the L1 label, repeating the addition for the remaining elements of the array.
- • • **pop ecx and pop esi:** These instructions restore the original values of ecx and esi that were pushed onto the stack at the beginning of the procedure.
- • • **ret:** This instruction indicates the end of the procedure and returns with the sum of the array elements in eax.

In summary, the ArraySum procedure is called from the main program to calculate the sum of the array. It uses a loop to iterate through the elements, accumulating the sum in eax, and then returns the result.

Let's review the concepts and questions related to procedures and the RET instruction:

(True/False): The PROC directive begins a procedure, and the ENDP directive ends a procedure.

True. The PROC directive is used to begin the definition of a procedure, and the ENDP directive marks the end of that procedure.

(True/False): It is possible to define a procedure inside an existing procedure.

True. In many assembly languages, including x86 assembly, it is possible to define a procedure inside another procedure. These are called nested procedures. However, it's important to maintain proper stack and register management when doing so.

What would happen if the RET instruction was omitted from a procedure?

If the RET instruction is omitted from a procedure, the program would not properly return control to the calling code. This can lead to unpredictable behavior, memory leaks, or crashes, as the stack would not be properly adjusted, and the program counter would not be updated.

How are the words Receives and Returns used in the suggested procedure documentation?

In procedure documentation, the word "Receives" is often used to describe the input parameters or arguments that the procedure expects. These are the values or data that are passed to the procedure when it's called. The word "Returns" is used to describe what the procedure provides as output. This typically includes the result of the procedure's computation, often in a specific register like EAX.

(True/False): The CALL instruction pushes the offset of the CALL instruction on the stack.

False. The CALL instruction does not push the offset of the CALL instruction on the stack. It pushes the return address, which is the address of the instruction immediately following the CALL instruction.

(True/False): The CALL instruction pushes the offset of the instruction following the CALL on the stack.

True. The CALL instruction pushes the offset of the instruction immediately following the CALL on the stack. This is necessary so that the program knows where to continue execution when the called procedure returns. In summary, procedures are defined using the PROC and ENDP directives, and the RET instruction is used to return from a procedure. Proper stack management is essential to ensure correct program flow. The terms "Receives" and "Returns" are used in procedure documentation to describe input and output, respectively. Finally, the CALL instruction pushes the return address on the stack, allowing for structured program flow and function calls.

(True/False): A link library consists of assembly language source code.

False. A link library does not consist of assembly language source code. Instead, it contains precompiled machine code procedures (subroutines) that can be used by programs. Use the PROTO directive to declare a procedure named MyProc in an external link library.

To declare a procedure named MyProc in an external link library, you would typically use the PROTO directive like this:

```
MyProc PROTO
```

This declares the existence of a procedure named MyProc without specifying its parameters or return type. The linker will resolve the actual details when linking to the external library.

Write a CALL statement that calls a procedure named MyProc in an external link library. To call a procedure named MyProc in an external link library, you would use a CALL statement like this:

```
call MyProc
```

The linker will ensure that the correct machine code for MyProc is linked into your program when you build it. What is the name of the 32-bit link library supplied with this book?

The name of the **32-bit link library supplied with the book is Irvine32.lib.** This library contains procedures that link to the MS-Windows API for input and output operations and other functionality.

What type of file is kernel32.dll?

kernel32.dll is a dynamic link library (DLL) file. It contains executable functions that perform various system-level tasks, including character-based input and output. These DLLs are an essential part of the Windows operating system and can be linked to from assembly language programs to access their functionality. These answers provide a better understanding of how external libraries are used in assembly language programming, especially in the context of the book's examples.

ReadInt: Reads a 32-bit signed decimal integer from the keyboard, terminated by the Enter key.

READINT

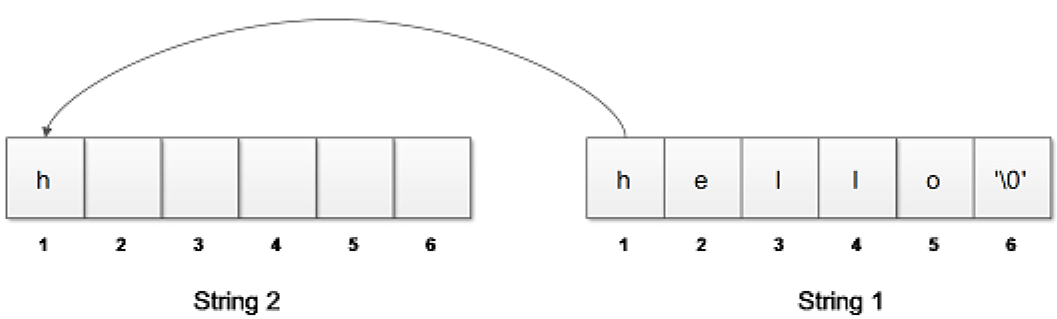
ReadKey: Reads a single character from the keyboard's input buffer without waiting for input. Useful for detecting key presses.

ReadString: Reads a string from the keyboard, terminated by the Enter key.

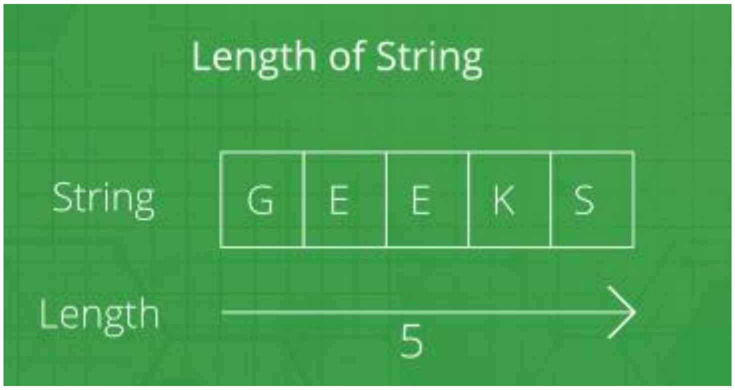
SetTextColor: Sets the foreground and background colors for all subsequent text output to the console.

Str_compare: Compares two strings and returns a result indicating their relationship.

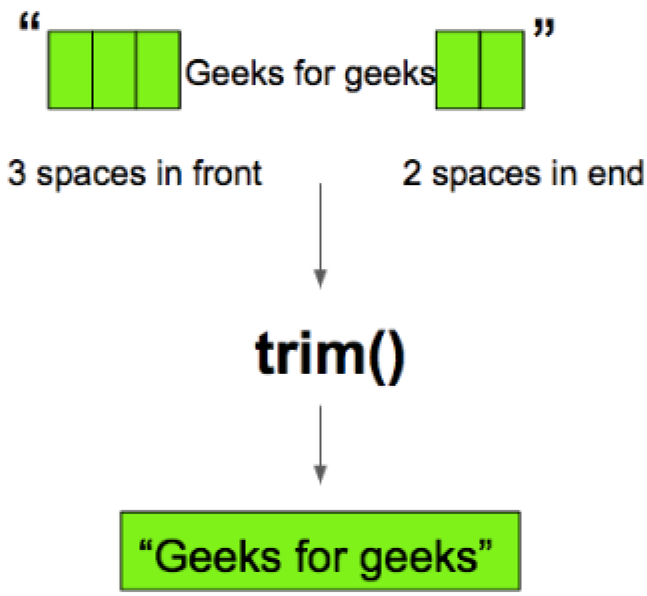
Str_copy: Copies a source string to a destination string.



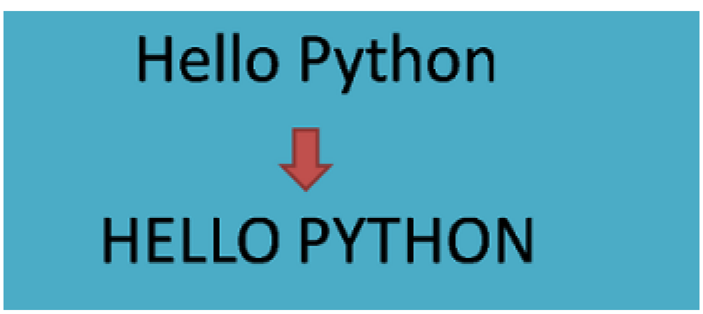
Str_length: Returns the length of a string in EAX (in characters).



Str_trim: Removes unwanted characters (specified by a given character set) from a string.

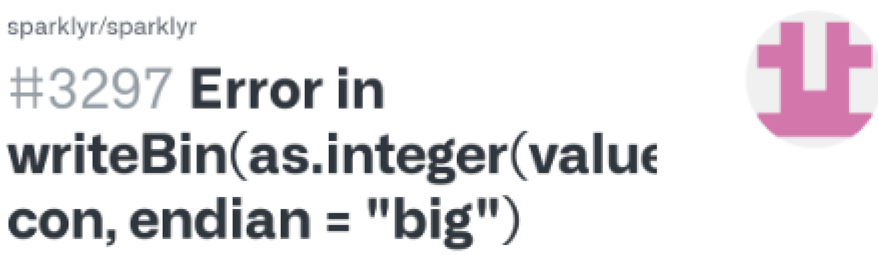


Str_ucase: Converts a string to uppercase letters.

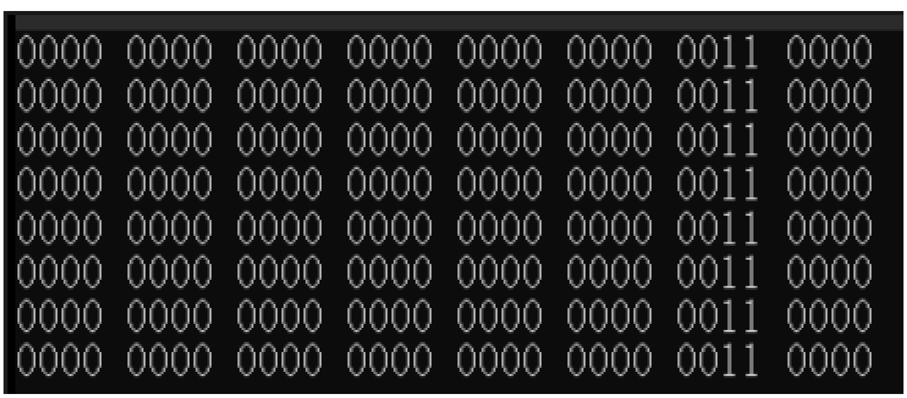


WaitMsg: Displays a message and waits for a key to be pressed before continuing.

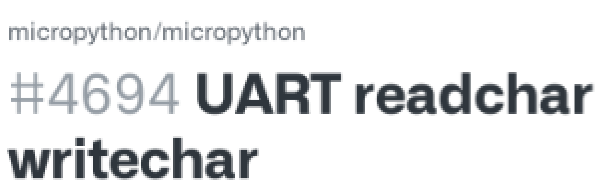
WriteBin: Writes an unsigned 32-bit integer to the console window in ASCII binary format.



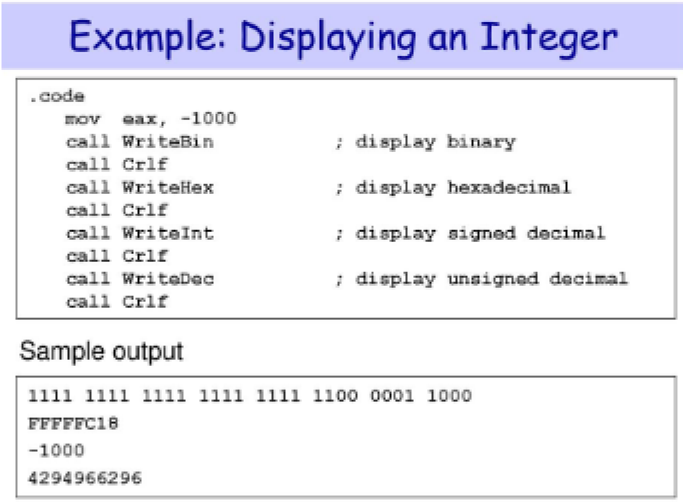
WriteBinB: Writes a binary integer to the console window in byte, word, or doubleword format.



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.

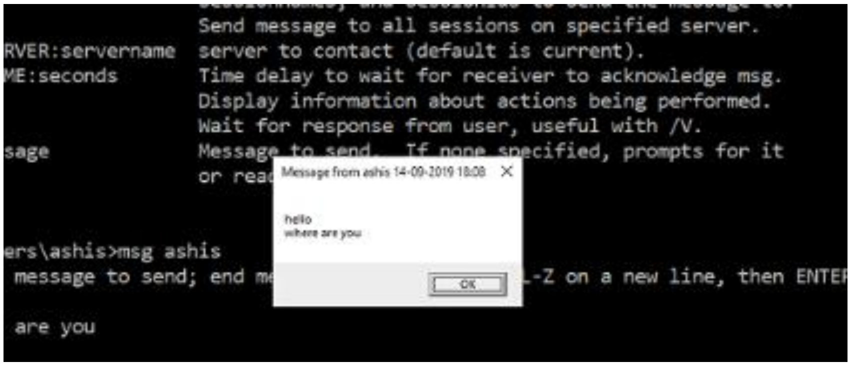
WriteStackFrame: Writes the current procedure's stack frame information to the console.

WriteStackFrameName: Writes the name of the current procedure and its stack frame information to the console.

WriteString: Writes a null-terminated string to the console window.

WriteToFile: Writes a buffer of data to an output file.

WriteWindowsMsg: Displays a string containing the most recent error message generated by MS-Windows.



Each of these procedures serves specific purposes, such as input/output operations, string manipulation, and text formatting, and can be extremely helpful for assembly language programmers to simplify common tasks.

Overview of Console Window The console window is a text-only window in MS-Windows created when a command prompt is displayed. You can customize its size, font size, and colors.

- Mode Command: You can use the "mode" command in the command prompt to change the number of columns and lines in the console window. For example:

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

==

Individual Procedure Descriptions

===

Here are descriptions of some of the procedures in the Irvine32 library:

1. 1.

1. CloseFile

- Closes a file that was previously created or opened using a file handle. Pass the file handle in EAX.

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

1. 2.

2. Clrscr

- Clears the console window. Typically called at the beginning and end of a program to clear the screen. If called at other times, you may need to pause using WaitMsg to allow the user to view existing screen contents.

```
call WaitMsg call Clrscr
```

1. 3.

3. CreateOutputFile

- Creates a new disk file and opens it for writing. Provide the filename offset in EDX. Returns a valid file handle (32-bit integer) in EAX if successful, otherwise INVALID_HANDLE_VALUE.

```
mov edx, offset filename call CreateOutputFile
```

1. 4.

4. Crlf

- The Crlf procedure advances the cursor to the beginning of the next line in the console window. It essentially moves to the next line. Sample Call:

```
call Crlf
```

This instruction calls the Crlf procedure. The Crlf procedure stands for "Carriage Return" and "Line Feed." It is used to advance the cursor to the beginning of the next line in the console window. Essentially, it simulates pressing the Enter key, which moves the cursor to the next line as in a typical text editor.

1. 5.

5. Delay

- The Delay procedure pauses the program for a specified number of milliseconds. You need to set EAX to the desired interval before calling it. Sample Call:

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

This instruction calls the Delay procedure. The Delay procedure is used to pause the program for a specified number of milliseconds. In this case, it's designed to create a 1-second delay since EAX is set to 1000 (milliseconds).

1. 6.

6. DumpMem

- The DumpMem procedure writes a range of memory to the console window in hexadecimal format. It requires you to pass the starting address in ESI, the number of units in ECX, and the unit size in EBX.

```
mov esi, OFFSET array ; starting OFFSET
mov ecx, LENGTHOF array ; number of units
mov ebx, TYPE array ; doubleword format
call DumpMem
```

This call displays the content of the array in hexadecimal.

Here's an explanation of the provided assembly code:

mov esi, OFFSET array: This instruction loads the memory address (offset) of the array into the ESI register. It's setting up ESI to point to the starting address of the array in memory.

mov ecx, LENGTHOF array: This instruction loads the length of the array in terms of the number of elements into the ECX register. It determines how many units of data will be displayed when calling DumpMem.

mov ebx, TYPE array: This instruction loads the data type of the array elements into the EBX register. It specifies the format of the data when calling DumpMem. In this case, it indicates that the elements are in doubleword format.

call DumpMem: This instruction calls the DumpMem procedure. The DumpMem procedure is responsible for displaying a range of memory in hexadecimal format. It takes the starting address in ESI, the number of units in ECX, and the unit size in EBX (in this case, doubleword format). It will display the contents of the array specified by ESI, ECX, and EBX in hexadecimal format.

In summary, these instructions set up the parameters for the DumpMem procedure to display the contents of the array in doubleword format from its starting address with the specified length.

1. 7.

7. DumpRegs

- The DumpRegs procedure displays the values of various registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFL), as well as the Carry, Sign, Zero, Overflow, Auxiliary Carry, and Parity flags in hexadecimal.

```
call DumpRegs
```

This instruction calls the DumpRegs procedure. The DumpRegs procedure displays the values of various CPU registers and flags in hexadecimal format. It provides a snapshot of the CPU's current state, which can be helpful for debugging and understanding the program's execution.

1. 8.

8. GetCommandTail

- GetCommandTail copies the program's command line into a null-terminated string and checks if the command line is empty.
- To use it, you must provide the offset of a buffer in EDX where the command line will be stored. Sample Call:

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

1. 9.

9. GetMaxXY

- The GetMaxXY procedure retrieves the size of the console window's buffer. The number of buffer columns is stored in DX, and the number of buffer rows is stored in AX. Sample Call:

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

1. 10.

10. GetMseconds

- GetMseconds returns the number of milliseconds elapsed since midnight on the host computer in the EAX register. It's useful for measuring time between events. Sample Call:

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

This example measures the execution time of a loop.

1. 11.

11. GetTextColor

- The GetTextColor procedure retrieves the current foreground and background colors of the console window. It has no input parameters. The background color is stored in the upper four bits of AL, and the foreground color is stored in the lower four bits. Sample Call:

```
.data
    color BYTE ?
.code
    call GetTextColor
    mov color, al
```

Here's a breakdown of what this code does:

1. **.data color BYTE ?:** This declares a byte-sized variable named color in the data section. This variable will be used to store the color information retrieved from the console.

1. 2. **2. .code call GetTextColor:** This line calls the GetTextColor procedure, which retrieves the current text color attributes of the console window. The result is returned in the AL register.

1. 3. **3. mov color, al:** This instruction moves the value in the AL register (which contains the retrieved color information) into the color variable declared earlier. It stores the current text color in the color variable.

So, after executing these instructions, the color variable will contain the current text color attribute of the console window, allowing you to use or manipulate it in your program as needed.

1. 12.

12. Gotoxy

- The Gotoxy procedure positions the cursor at a specific row and column in the console window. You should pass the row (Y-coordinate) in DH and the column (X-coordinate) in DL. Sample Call:

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

Here's a breakdown of what this code does:

mov dh, 10: This instruction moves the value 10 into the DH (Destination High) register, which represents the row number where you want to position the cursor. In this case, it sets the row to 10.

mov dl, 20: This instruction moves the value 20 into the DL (Destination Low) register, which represents the column number where you want to position the cursor. It sets the column to 20.

call Gotoxy: This is a procedure call to the Gotoxy procedure. When called, it takes the values in DH (row) and DL (column) and uses them to position the cursor in the console window.

So, after executing these instructions, the cursor will be moved to row 10 and column 20 in the console window. This can be useful for controlling the cursor's position when you want to display text or interact with the user at specific locations on the screen.

1. 13.

13. IsDigit

- IsDigit checks whether the value in AL is the ASCII code for a valid decimal digit. If AL contains a valid decimal digit, it sets the Zero flag (ZF); otherwise, it clears ZF. Sample Call:

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

You are trying to determine whether the value in the AL register represents a valid decimal digit by calling the IsDigit procedure.

You move the value of somechar into the AL register using mov AL, somechar. somechar presumably contains an ASCII character.

You call the IsDigit procedure with the current value in the AL register using call IsDigit.

Inside the IsDigit procedure, it checks if the value in AL is a valid ASCII code for a decimal digit. If it is, the procedure sets the Zero flag (ZF) to 1. If it's not a valid digit, the ZF is cleared (set to 0).

After calling IsDigit, you can check the state of the ZF to determine whether the character in AL is a valid decimal digit or not.

If ZF is set (ZF = 1), then somechar represents a valid decimal digit. If ZF is cleared (ZF = 0), then somechar does not represent a valid decimal digit.

This code is useful if you want to validate whether a character is a decimal digit in your program.

1. 14.

14. MsgBox

The MsgBox procedure displays a popup message box with an optional caption. Pass it the offset of a string in EDX to appear inside the box, and optionally, pass the offset of a string for the box's title in EBX. To leave the title blank, set EBX to zero.

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

You are setting up a message box with a title ("Dialog Title") and a message ("This is a pop-up message box. Click OK to continue..."). Then, you are calling the MsgBox procedure to display this message box.

You define two null-terminated strings in the .data section. caption contains the title of the message box, and HelloMsg contains the message along with a line break (0dh,0ah) and the instruction "Click OK to continue..."

You load the address of the caption string into the EBX register using mov ebx, OFFSET caption. This prepares the address of the title string to be passed as a parameter to the MsgBox procedure.

You load the address of the HelloMsg string into the EDX register using mov edx, OFFSET HelloMsg. This prepares the address of the message string to be passed as a parameter to the MsgBox procedure.

You call the MsgBox procedure using call MsgBox. The procedure displays a graphical popup message box with an OK button. It expects the address of the title in EBX and the address of the message in EDX.

After the user clicks the OK button in the message box, the MsgBox procedure returns, and your program continues executing any subsequent code.

This code demonstrates how to display a simple informational message box with a title and a message, allowing the user to acknowledge the message by clicking the OK button.

1. 15.

15. MsgBoxAsk

The MsgBoxAsk procedure displays a popup message box with Yes and No buttons. It returns an integer in EAX that indicates which button was selected by the user (IDYES or IDNO).

```
.data
    caption BYTE "Survey Completed", 0
    question BYTE "Thank you for completing the test.", 0dh,0ah, "Receive results?", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET question
    call MsgBoxAsk ; check return value in EAX
```

You are setting up a message box with a title ("Survey Completed") and a question ("Thank you for completing the test. Receive results?"). Then, you are calling the MsgBoxAsk procedure to display this message box.

Here's what's happening step by step:

1. You define two null-terminated strings in the .data section. caption contains the title of the message box, and question contains the message along with a line break (0dh,0ah) and the question "Receive results?"

1. You load the address of the caption string into the EBX register using mov ebx, OFFSET caption. This prepares the address of the title string to be passed as a parameter to the MsgBoxAsk procedure.

1. You load the address of the question string into the EDX register using mov edx, OFFSET question. This prepares the address of the message/question string to be passed as a parameter to the MsgBoxAsk procedure.

1. You call the MsgBoxAsk procedure using call MsgBoxAsk. The procedure displays a graphical popup message box with a Yes and No button. It expects the address of the title in EBX and the address of the message/question in EDX.

After the user interacts with the message box (by clicking either Yes or No), the MsgBoxAsk procedure returns an integer value in EAX, which tells you which button was selected. The value will be either IDYES (equal to 6) if the user clicked Yes or IDNO (equal to 7) if the user clicked No.

To check the return value and take further actions based on the user's choice, you can use conditional statements or other logic in your code, depending on whether EAX contains IDYES or IDNO.

1. ***16.***

16. OpenInputFile

The OpenInputFile procedure opens an existing file for input. Pass it the offset of the filename in EDX. If the file is opened successfully, EAX contains a valid file handle; otherwise, EAX equals INVALID_HANDLE_VALUE.

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

After the call, you can check the value of EAX to determine if the file was opened successfully. If EAX equals INVALID_HANDLE_VALUE, the file was not opened successfully.

You are defining a null-terminated string "myfile.txt" in the .data section and then using it as the filename to open an input file using the OpenInputFile procedure.

Here's what's happening step by step:

You define a null-terminated string filename with the content "myfile.txt" in the .data section.

You load the address of the filename string into the EDX register using mov edx, OFFSET filename. This prepares the address of the filename to be passed as a parameter to the OpenInputFile procedure.

You call the OpenInputFile procedure using call OpenInputFile. The procedure expects the filename (the address of the null-terminated string) in the EDX register. It will attempt to open the file with the given name for input.

After calling OpenInputFile, if the file was successfully opened, the EAX register will contain a valid file handle. If the file could not be opened, EAX will be set to INVALID_HANDLE_VALUE (a predefined constant).

In your code, you have not shown how you are handling the result of opening the file (the value in EAX). Depending on whether the file was opened successfully or not, you would typically check the value in EAX and take appropriate actions, such as reading from the file or displaying an error message.

1. *17.*

17. ReadString:

Use ReadString to read a string from the keyboard until the Enter key is pressed. Pass the offset of a buffer where the input will be stored in EDX and specify the maximum number of characters the user can enter in ECX. It returns the count of the number of characters typed by the user in EAX. Example:

```
.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
```

The provided code segment is using the ReadString procedure to read a string from the keyboard and save it in a buffer. Let's break down what each part of the code does:

.data section:

buffer BYTE 21 DUP(0): This defines a buffer named buffer that can hold up to 21 bytes (including the null terminator). It's initialized with zeros to ensure it's an empty string.

byteCount DWORD ?: This declares a DWORD variable named byteCount to hold the count of characters entered by the user.

.code section:

mov edx, OFFSET buffer: This line sets the EDX register to point to the buffer variable, effectively providing the address of the buffer to the ReadString procedure.

mov ecx, SIZEOF buffer: Here, the ECX register is loaded with the size of the buffer variable, which is the maximum number of characters that can be read. In this case, it's set to 21.

call ReadString: This line calls the ReadString procedure, which reads a string from the keyboard. It stops reading when the Enter key is pressed. The string entered by the user is stored in the buffer, and the number of characters read is returned in EAX.

mov byteCount, eax: Finally, this line stores the value of EAX (the number of characters read) into the byteCount variable, so you can later access and use this count in your program.

Where did eax come from?

The **ReadString procedure** reads a string from the keyboard and returns the number of characters read in the **EAX register.** After calling ReadString, the value in EAX represents the number of characters read, and it's stored in the byteCount variable for later use in your program.

In summary, this code segment initializes a buffer, reads a string from the keyboard into the buffer using ReadString, and stores the count of characters read in the byteCount variable.

1. 18.

18. SetTextColor:

Use SetTextColor to set the foreground and background colors for text output. Set the desired color attribute in EAX, combining foreground and background colors. Example (setting text to white on a blue background).

```
mov eax, white + (blue SHL 4) ; white on blue
call SetTextColor
```

The provided code segment is setting the text color for text output in a console window using the SetTextColor procedure. Let's break down what each part of the code does:

mov eax, white + (blue SHL 4): This line sets up the eax register to specify the desired text color attribute. It combines two color constants, white and blue, to achieve the desired color. Here's the breakdown:

white and blue are color constants defined in the Irvine32 library. white represents white text, and blue represents blue as the background color.

(blue SHL 4) shifts the value of blue four bits to the left. This bitwise shift operation is

used to specify the background color. In this case, it's indicating a blue background.

Adding white and the shifted blue value together combines the foreground and background colors. So, the result is a combination of white text on a blue background.

call SetTextColor: This line calls the SetTextColor procedure with the color attribute specified in eax. The procedure then sets the text color for subsequent text output to the console using the specified color combination.

In summary, this code segment sets the text color to white on a blue background using the SetTextColor procedure, so any text output that follows will be displayed with this color combination in the console window.

1. 19.

19. Str_length:

Use Str_length to find the length of a null-terminated string. Pass the offset of the string in EDX. It returns the string's length in EAX.

```
.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
```

The provided code snippet calculates the length of a null-terminated string stored in memory using the Str_length procedure and then saves the length in a DWORD variable named bufLength. Here's a breakdown of what each part of the code does:

.data: This section defines a data segment where you declare data variables.

buffer BYTE "abcde", 0: This line declares a null-terminated string named buffer with the value "abcde". The null terminator (0) signifies the end of the string.

bufLength DWORD ?: This line declares an uninitialized DWORD variable named bufLength to store the length of the string.

.code: This section is the code segment where you write the program's instructions.

mov edx, OFFSET buffer: This instruction loads the offset of the buffer string into the edx register. It sets edx to point to the beginning of the string.

call Str_length: This line calls the Str_length procedure, passing the address of the buffer string as a parameter in edx. The Str_length procedure calculates the length of the string and stores it in the eax register.

mov bufLength, eax: This instruction copies the value of eax (which now contains the length of the string) into the bufLength variable. This variable will now hold the length of the "abcde" string, which is 5 characters.

In summary, this code segment calculates and stores the length of the "abcde" string in the bufLength variable, resulting in bufLength being set to 5.

1. 20.

20. WaitMsg:

Use WaitMsg to display the message "Press any key to continue..." and wait for the user to press a key. Example:

```
call WaitMsg
```

The code snippet call WaitMsg is a function call to the WaitMsg procedure. Here's what it does:

call WaitMsg: This line of code calls the WaitMsg procedure, which is part of the Irvine32 library. When you call WaitMsg, it displays the message "Press any key to continue..." on the console window and waits for the user to press any key.

The **purpose of using WaitMsg** is to pause the program's execution temporarily and provide a message to the user, prompting them to press a key to continue. This can be useful when you want to give the user time to read the output on the console before it disappears.

So, when this line of code is executed, it will display the "Press any key to continue..." message on the console, and the program will wait until the user presses a key before proceeding further.

1. 21.

21. WriteInt:

Use WriteInt to write a 32-bit signed integer to the console window in decimal format. Pass the integer in EAX.

```
mov eax, -12345
call WriteInt ; displays: "-12345"
```

In this code, you are performing the following steps:

mov eax, -12345: This instruction moves the signed 32-bit integer value -12345 into the EAX register. It's a negative integer represented in two's complement form.

call WriteInt: Here, you are calling the WriteInt procedure. This procedure takes a 32-bit signed integer in the EAX register and writes it to the console window in decimal format with a leading sign if it's negative. In this case, since you loaded -12345 into EAX, it will display "-12345" on the console.

So, when you run this code, it will display the signed integer -12345 on the console window.

1. 22.

22. WriteString:

Use WriteString to write a null-terminated string to the console window. Pass the offset of the string in EDX.

```
.data
    greeting BYTE "Hello, Assembly Programmer!", 0
.code
    mov edx, OFFSET greeting
    call WriteString
```

In this part of the code:

mov edx, OFFSET greeting: This line sets the EDX register to the offset of the greeting string. In other words, it tells the program where the string is located in memory.

call WriteString: Here, you are calling the WriteString procedure. This procedure takes the offset of a string (in this case, the offset of the greeting string) and writes the content of the string to the console window.

So, the code you provided is essentially displaying the "Hello, Assembly Programmer!" message on the console window using the WriteString procedure.

1. ***23.***

23. WriteToFile:

This procedure is used to write the contents of a buffer to an output file.

- Parameters: EAX (file handle), EDX (buffer address), ECX (number of bytes to write).
- Returns the number of bytes written if successful; otherwise, an error code.

```
.data
    fileHandle DWORD ?
    buffer BYTE "This is a test.", 0

.code
    mov eax, fileHandle
    mov edx, OFFSET buffer
    mov ecx, LENGTHOF buffer - 1 ; Length excluding the null terminator
    call WriteToFile
```

1. 24.

24. WriteWindowsMsg:

This procedure writes a string containing the most recent error generated by your application to the Console window when executing a system function. It is often used to display error messages to the user.

```
.code
    ; Some code that might generate an error
    ; ...

    ; After an error occurs, call WriteWindowsMsg to display the error message
    call WriteWindowsMsg
```

================================

Implementing the procedures

================================

```
; 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

    ; Clear the screen
    call Clrscr

    ; 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                ; prompt the user
    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

    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
```

The program you've provided is a demonstration of using various procedures from the Irvine32 library to perform tasks like clearing the screen, displaying memory content, reading and displaying integers, and generating random numbers. Here's a breakdown of the program:

Setup and Definitions: The program starts with comments indicating its purpose and includes the Irvine32.inc library.

It defines constants COUNT, BlueTextOnGray, and DefaultColor.

An array named arrayD is declared with four signed doubleword integers. There's also a prompt message.

Main Procedure (main PROC):

- It sets the text color to blue on a light gray background using SetTextColor.

- Clears the screen using Clrscr.
- Displays the content of the arrayD array using the DumpMem procedure.
- It then enters a loop (L1) to ask the user to input signed integers COUNT times.
- For each input, it:
- Prompts the user to enter an integer using WriteString. Reads the integer input into EAX using ReadInt. Displays the entered integer in decimal, hexadecimal, and binary formats using WriteInt, WriteHex, and WriteBin. After the loop, it returns the console window to default colors using WaitMsg, and clears the screen again.
- Random Number Generation Procedures (Rand1 and Rand2):
- Rand1 generates ten pseudo-random unsigned integers in the range 0 to 4,294,967,294 using Random32.
- Rand2 generates ten pseudo-random signed integers in the range -50 to +49 using RandomRange.
- Main Procedure (continued):
- After defining the random number generation procedures, the program calls Randomize to initialize the random number generator.
- It then calls both Rand1 and Rand2 procedures to generate and display random integers.

Program End:

- The program ends with END main.
- This program is a comprehensive example of how to use various Irvine32 library procedures to perform tasks related to console input and output, as well as random number generation. It also demonstrates how to format and display data in different formats (decimal, hexadecimal, and binary).

```
; Library Test #3: Performance Timing (TestLib3.asm)
; Calculate the elapsed execution time of a nested loop.

include Irvine32.inc

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

.code

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

    ; Save the starting time
    call GetMSeconds
    mov startTime, eax

    ; Start the outer loop
    mov ecx, OUTER_LOOP_COUNT
L1:
    call innerLoop
    loop L1

    ; Calculate the elapsed time
    call GetMSeconds
    sub eax, startTime
```

```
    ; Calculate the elapsed time
    call GetMSeconds
    sub eax, startTime

    ; Display the elapsed time
    mov edx, OFFSET msg2  ; "Elapsed milliseconds: "
    call WriteString
    call WriteDec          ; Write the milliseconds
    call Crlf

    exit
main ENDP

innerLoop PROC
    push ecx              ; Save current ECX value
    mov ecx, 0FFFFFFFh    ; Set the loop counter
L1:
    mul eax               ; Use up some cycles
    mul eax
    mul eax
    loop L1              ; Repeat the inner loop
    pop ecx               ; Restore ECX's saved value
    ret
innerLoop ENDP
END main
```

Certainly, let's delve into the additional information you provided regarding the Irvine64 library and some of its key procedures:

WriteHex64: This procedure is used to display the contents of the RAX register as a 64-bit hexadecimal integer. It's a convenient way to view the value of a 64-bit integer in hexadecimal format. Unlike some other procedures, it doesn't return any value; its purpose is to output the hexadecimal representation to the console.

WriteHexB: WriteHexB is a versatile procedure that displays the contents of the RAX register as a hexadecimal integer in different formats, depending on the display size specified in the RBX register. You can pass 1, 2, 4, or 8 in RBX to indicate whether you want to display the value as a 1-byte, 2-byte, 4-byte, or 8-byte hexadecimal integer, respectively. Like WriteHex64, it doesn't return a value but is used for outputting hexadecimal representations.

WriteString: This procedure is used to display a null-terminated ASCII string. You pass the 64-bit offset of the string in RDX, and it will output the string to the console. Like the previous procedures, it also doesn't return a value.

The Irvine64 library you mentioned is designed for 64-bit assembly programming. It's worth noting that in 64-bit assembly, the calling conventions and register usage can be different from 32-bit assembly.

For example, some registers like RAX, RCX, RDX, R8, R9, R10, and R11 may not be preserved across function calls, while others like RBX, RBP, RDI, RSI, R12, R13, R14, and R15 are usually preserved. Understanding these conventions is important when writing 64-bit assembly code.

This library, while smaller than its 32-bit counterpart, still provides essential tools for interactive programming in a 64-bit environment.

As you progress in your 64-bit assembly programming journey, you may find it beneficial to expand this library with your own code to suit your specific needs and applications.

Didn't we deal with those earlier on?

You are correct; some of the procedures mentioned, such as WriteHex64, WriteHexB, and WriteString, are similar to their 32-bit counterparts that we discussed earlier.

However, the key difference here is that these procedures are designed for 64-bit assembly programming, as indicated by the use of the RAX, RBX, RDX, and other 64-bit registers.

In the previous discussions, we primarily focused on 32-bit assembly and the Irvine32 library.

The additional information you provided pertains to the Irvine64 library, which is tailored for 64-bit assembly programming.

While the basic concepts and functionality of these procedures may be similar, the implementation and usage may differ due to the nuances of 64-bit assembly, including register usage and calling conventions.

IRVINE 64

To call a subroutine in 64-bit assembly, you need to place input parameters in registers and then use the CALL instruction. For example:

```
mov rax, 12345678h
call WriteHex64
```

In this example, WriteHex64 is a subroutine being called with rax as an input parameter.

Using the PROTO Directive: To inform the assembler about external procedures (subroutines) that you plan to call but are not defined within your program, you should use the PROTO directive at the top of your program.

This helps the assembler understand the calling conventions and parameter types for these external procedures. For instance:

```
ExitProcess PROTO    ; Located in the Windows API
WriteHex64 PROTO    ; Located in the Irvine64 library
```

In this code, ExitProcess and WriteHex64 are identified as external procedures, and the PROTO directive provides information about their usage.

These guidelines are essential for proper procedure calls in 64-bit assembly with the Irvine64 library, ensuring that the correct calling conventions are followed and that the assembler can generate the appropriate code for calling external procedures.

The additional content you've mentioned explains the x64 calling convention, which is used in 64-bit programs on the x86-64 architecture, such as Windows API functions and C/C++ programs. Here are the key points from this section:

Microsoft x64 Calling Convention: This is a consistent scheme for passing parameters and calling procedures in 64-bit programs. It's followed by C/C++ compilers and the Windows API. Some of its characteristics include:

The CALL instruction subtracts 8 from the **RSP (stack pointer) register** because addresses are 64 bits long.

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 goes in **RCX**, and so on.

The caller's responsibility is to **allocate at least 32 bytes of shadow space** on the runtime stack to optionally save register parameters.

When calling a **subroutine, the stack pointer (RSP) must be aligned on a 16-byte boundary**, considering the 8 bytes pushed by the CALL instruction.

Sample Program: The provided sample program demonstrates how to call a subroutine (AddFour) using the Microsoft x64 calling convention.

```
AddFour PROC
    ; Input parameters: RCX, RDX, R8, R9
    ; Output: RAX contains the sum of the input parameters

    ; Add the input parameters
    mov rax, rcx
    add rax, rdx
    add rax, r8
    add rax, r9

    ret
AddFour ENDP
```

It adds four input parameters placed in RCX, RDX, R8, and R9 and saves the sum in RAX. The program aligns the stack pointer, sets the parameters, and calls the subroutine.

This code defines the "AddFour" subroutine, which takes four input parameters (RCX, RDX, R8, and R9) and calculates their sum, storing the result in RAX. The ret instruction is used to return from the subroutine.

You can include this code in your assembly program to call the "AddFour" subroutine using the Microsoft x64 calling convention, as demonstrated in the previous content.

PROTO Directive: The PROTO directive is used to declare external procedures (subroutines) at the top of your program. It helps the assembler understand how to call these procedures correctly, including their names and parameter types.

Stack Alignment: Ensuring that the stack pointer (RSP) is properly aligned on a 16-byte boundary is crucial when working with the x64 calling convention. This alignment is essential to maintain proper stack integrity during procedure calls.

Usage in Irvine64 Library: It's worth noting that when you work with the Irvine64 library (used in these examples), you don't need to adhere to the Microsoft x64 calling convention for library procedures. It's primarily required when calling Windows API functions or external C/C++ functions.

These explanations provide a detailed understanding of how the Microsoft x64 calling convention works and how to apply it in your 64-bit assembly programs, particularly when interfacing with external functions and libraries.

Certainly, here's a short assembly code snippet that demonstrates the use of the PROTO directive and stack alignment for a simple subroutine:

```
.data
    result QWORD 0   ; Declare a variable to store the result
.code
    main PROC
        sub rsp, 8    ; Align the stack pointer to a 16-byte boundary
        mov rcx, 5    ; First parameter
        mov rdx, 3    ; Second parameter

        call AddNumbers   ; Call the subroutine
        add rsp, 8    ; Restore the stack pointer

        ; The result is now stored in the "result" variable
        ; You can use it as needed
        ; Exit the program
        mov ecx, 0
        call ExitProcess

    AddNumbers PROC
        ; Parameters are already in RCX and RDX
        add rax, rcx   ; Add the first parameter to RAX
        add rax, rdx   ; Add the second parameter to RAX
        mov [result], rax   ; Store the result in the "result" variable
        ret
    AddNumbers ENDP

    END main
```

In this code:

We use the PROTO directive for the ExitProcess procedure, which is an external procedure used to exit the program.

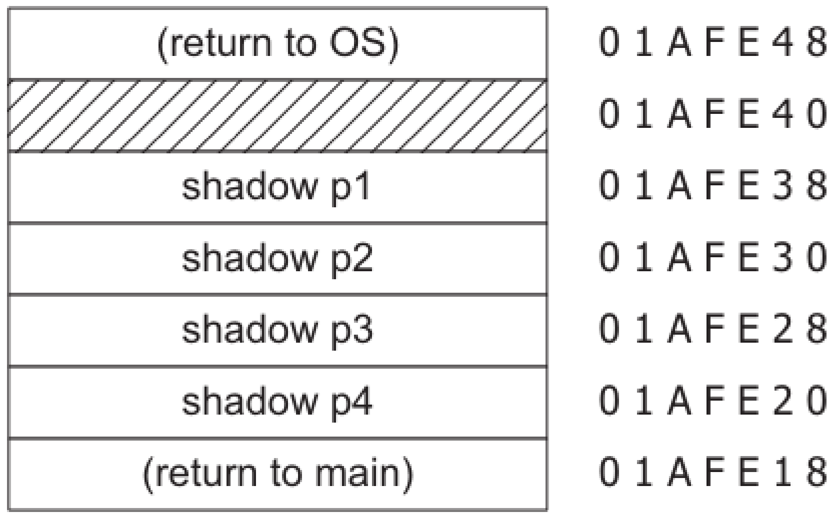
We ensure proper stack alignment by subtracting 8 from rsp at the beginning of the main procedure and then adding 8 to rsp to restore it after the subroutine call.

We call a subroutine AddNumbers with two parameters (rcx and rdx), which adds these parameters and stores the result in the result variable.

Finally, we exit the program using ExitProcess.

Please note that this is a simplified example, and in practice, you would use the Microsoft x64 calling convention for more complex scenarios or when calling external functions.

Figure 5–11 Runtime stack for the CallProc_64 program.



The runtime stack for the CallProc_64 program is a diagram that shows how the stack changes as the program executes. The stack is a region of memory that is used to store function calls, return addresses, and local variables.

The diagram shows the following:

The stack pointer (RSP) starts at address 01AFE48 before the program is called. When the OS calls the program, it subtracts 8 from the stack pointer to push the return address onto the stack.

After line 10 of the program executes, the stack pointer is at address 01AFE40, showing that the stack has been properly aligned on a 16-byte boundary.

After line 11 of the program executes, the stack pointer is at address 01AFE20, showing that 32 bytes of shadow space have been reserved on the stack.

Inside the AddFour procedure, the stack pointer is at address 01AFE18, showing that the caller's return address has been pushed onto the stack.

After AddFour returns, the stack pointer is again at address 01AFE20, the same value it had before calling AddFour. When the program reaches the end of the main procedure, it returns to the OS by executing a RET instruction.

If the program had chosen to execute an ExitProcess instruction instead, it would have been responsible for restoring the stack pointer to the way it was when the main procedure began to execute.

Here is a more detailed explanation of each line in the diagram:

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.)

The RSP register is the stack pointer register. It contains the address of the top of the stack. The CALL instruction pushes the return address onto the stack, which is the address of the instruction that will be executed after the called function returns.

After line 10 executed, RSP = 01AFE40, showing that the stack was properly aligned on a 16-byte boundary.

The stack must be aligned on a 16-byte boundary for performance reasons. The stack is aligned by pushing a dummy value onto the stack before calling a function and popping the dummy value off the stack after the function returns.

After line 11 executed, RSP = 01AFE20, showing that 32 bytes of shadow space were reserved at addresses 01AFE20 through 01AFE3F.

Shadow space is used to store the shadow registers of a function. Shadow registers are used to save the values of the callee-saved registers before a function is called and restore them after the function returns.

Inside the AddFour procedure, RSP = 01AFE18, showing that the caller's return address had been pushed on the stack. When a function is called, the caller's return address is pushed onto the stack.

This is so that the function knows where to return to when it is finished.

After AddFour returned, RSP again was equal to 01AFE20, the same value it had before calling AddFour. When a function returns, the stack pointer is restored to the value it had before the function was called.

This is done by popping the caller's return address off the stack.

Overall, the runtime stack diagram for the CallProc_64 program shows how the stack is used to store function calls, return addresses, and local variables.

QUESTIONS

====================================

QUESTIONS

====================================

Question 1: Which instruction pushes all of the 32-bit general-purpose registers on the stack?

Answer 1: The instruction that pushes all of the 32-bit general-purpose registers on the stack is PUSHA.

Question 2: Which instruction pushes the 32-bit EFLAGS register on the stack?

Answer 2: The instruction that pushes the 32-bit EFLAGS register on the stack is PUSHFD.

Question 3: Which instruction pops the stack into the EFLAGS register?

Answer 3: The instruction that pops the stack into the EFLAGS register is POPFD.

Question 4: Challenge: Another assembler (called NASM) permits the PUSH instruction to list multiple specific registers. Why might this approach be better than the PUSHAD instruction in MASM? Here is a NASM example: PUSH EAX EBX ECX

Answer 4: NASM's approach of allowing the PUSH instruction to list multiple specific registers can be better in some cases because it provides more flexibility. It allows you to choose which registers to push onto the stack, whereas PUSHA in MASM pushes all the general-purpose registers. This can save stack space and execution time when you only need to save a subset of registers.

Question 5: Challenge: Suppose there were no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as push eax.

Answer 5: If there were no PUSH instruction, you could achieve the same result as PUSH EAX using the following two instructions:

```
sub esp, 4
mov [esp], eax
```

Question 6: (True/False): The RET instruction pops the top of the stack into the instruction pointer.

Answer 6: False. The RET instruction pops the return address from the stack into the instruction pointer (EIP).

Question 7: (True/False): Nested procedure calls are not permitted by the Microsoft assembler unless the NESTED operator is used in the procedure definition.

Answer 7: False. Nested procedure calls are permitted without the need for the NESTED operator in the Microsoft assembler.

Question 8: (True/False): In protected mode, each procedure call uses a minimum of 4 bytes of stack space.

Answer 8: False. In protected mode, each procedure call doesn't necessarily use a minimum of 4 bytes of stack space. The actual stack space used depends on the number of parameters and local variables.

Question 9: (True/False): The ESI and EDI registers cannot be used when passing 32-bit parameters to procedures.

Answer 9: False. The ESI and EDI registers can be used when passing 32-bit parameters to procedures.

Question 10: (True/False): The ArraySum procedure (Section 5.2.5) receives a pointer to any array of doublewords.

Answer 10: False. The ArraySum procedure from Section 5.2.5 expects a pointer to an array of doublewords specifically.

Question 11: (True/False): The USES operator lets you name all registers that are modified within a procedure.

Answer 11: True. The USES operator lets you specify all registers that are modified within a procedure.

Question 12: (True/False): The USES operator only generates PUSH instructions, so you must code POP instructions yourself.

Answer 12: True. The USES operator generates PUSH instructions, so you need to code the corresponding POP instructions yourself.

Question 13: (True/False): The register list in the USES directive must use commas to separate the register names.

Answer 13: True. The register list in the USES directive must use commas to separate the register names.

Question 14: Which statement(s) in the ArraySum procedure (Section 5.2.5) would have to be modified so it could accumulate an array of 16-bit words? Create such a version of ArraySum and test it.

Answer 14: To accumulate an array of 16-bit words, you would need to modify the mov eax, [esi] and add esi, 4 statements to work with 16-bit words, like this:

```
mov ax, [esi]
add esi, 2
```

Question 15: What will be the final value in EAX after these instructions execute? push 5 push 6 pop eax pop eax

Answer 15: EAX will equal 5 after these instructions execute. The second pop eax instruction will overwrite the previous value of EAX.

Question 16: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: push 10
3: push 20
4: call Ex2Sub
5: pop eax
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex2Sub PROC
11: pop eax
12: ret
13: Ex2Sub ENDP
```

Answer 16: a. EAX will equal 10 on line 6.

Question 17: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov eax,30
3: push eax
4: push 40
5: call Ex3Sub
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex3Sub PROC
11: pusha
12: mov eax,80
13: popa
14: ret
15: Ex3Sub ENDP
```

Answer 17: d. The program will halt with a runtime error on Line 11 because there's no matching pop for the pusha instruction.

Question 18: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov eax,40
3: push offset Here
4: jmp Ex4Sub
5: Here:
6: mov eax,30
7: INVOKE ExitProcess,0
8: main ENDP

10: Ex4Sub PROC
11: ret
12: Ex4Sub ENDP
```

Answer 18: b. The program will halt with a runtime error on Line 4 because there's no matching pop for the push instruction.

Question 19: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov edx,0
3: mov eax,40
4: push eax
5: call Ex5Sub
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex5Sub PROC
11: pop eax
12: pop edx
13: push eax
14: ret
15: Ex5Sub ENDP
```

Answer 19: a. EDX will equal 40 on line 6.

Question 20: What values will be written to the array when the following code executes?

```
.data
array DWORD 4 DUP(0)

.code
```

In the provided code, you've declared an array named array with four double word (DWORD) elements, and you've initialized each element to 0 using 4 DUP(0). Therefore, when this code is executed, the array will contain the following values:

array[0] will be 0.

array[1] will be 0.

array[2] will be 0.

array[3] will be 0.

So, all elements in the array will have the value 0.

GOD LEVEL QUESTIONS

Exercise 1: Draw Text Colors

```
include Irvine32.inc

.data
    colors BYTE 2, 4, 6, 14    ; Colors: Green, Red, Yellow, White
    message BYTE "Hello, Colors!",0

.code
main PROC
    mov ecx, 4   ; Number of colors
    mov esi, 0   ; Index for colors array

loop_colors:
    mov eax, colors[esi]
    call SetTextColor
    mov edx, OFFSET message
    call WriteString
    call Crlf

    inc esi
    loop loop_colors

    call WaitMsg   ; Wait for a key press
    call Clrscr    ; Clear the screen
    call ExitProcess
main ENDP
END main
```

Exercise 2: Linking Array Items

```
.data
    start DWORD 1
    chars BYTE 'H', 'A', 'C', 'E', 'B', 'D', 'F', 'G'
    links DWORD 0, 4, 5, 6, 2, 3, 7, 0
    outputArray BYTE 8 DUP(?) ; To store the characters in order
.code
    main PROC
        mov edi, OFFSET outputArray  ; Destination for output characters
        mov esi, start  ; Start index
        mov ecx, 8     ; Number of characters to locate
        traverse_links:
            mov al, chars[esi]  ; Load character
            mov [edi], al       ; Store it in outputArray
            inc edi             ; Move to the next position in outputArray
            ; Get the next link index
            mov eax, esi
            mov ebx, 4          ; Size of DWORD (4 bytes)
            mul ebx             ; Multiply esi by 4
            mov esi, links[eax] ; Get the next link index

            loop traverse_links  ; Repeat for all characters

        ; Display the characters in outputArray
        mov edx, OFFSET outputArray
        call WriteString
        call Crlf

        call WaitMsg  ; Wait for a key press
        call ExitProcess
    main ENDP
END main
```

Exercise 3: Simple Addition (1)

```
include Irvine32.inc
.data
    prompt1 BYTE "Enter the first integer: ", 0
    prompt2 BYTE "Enter the second integer: ", 0
    resultMsg BYTE "The sum is: ", 0
    buffer1 DWORD ?
    buffer2 DWORD ?
    result DWORD ?
.code
    main PROC
        call Clrscr
        mov edx, OFFSET prompt1
        call WriteString
        call ReadInt
        mov buffer1, eax

        mov edx, OFFSET prompt2
        call WriteString
        call ReadInt
        mov buffer2, eax

        ; Add the integers
        mov eax, buffer1
        add eax, buffer2
        mov result, eax

        mov edx, OFFSET resultMsg
        call WriteString
        mov eax, result
        call WriteInt

        call WaitMsg  ; Wait for a key press
        call ExitProcess
    main ENDP
END main
```

Exercise 4: Simple Addition (2)

```
include Irvine32.inc
.data
    prompt1 BYTE "Enter the first integer: ", 0
    prompt2 BYTE "Enter the second integer: ", 0
    resultMsg BYTE "The sum is: ", 0
    buffer1 DWORD ?
    buffer2 DWORD ?
    result DWORD ?

.code
    main PROC
        mov ecx, 3  ; Repeat the process three times
        loop_repeat:
            call Clrscr

            mov edx, OFFSET prompt1
            call WriteString
            call ReadInt
            mov buffer1, eax

            mov edx, OFFSET prompt2
            call WriteString
            call ReadInt
            mov buffer2, eax

            ; Add the integers
            mov eax, buffer1
            add eax, buffer2
            mov result, eax
```

```

            mov edx, OFFSET resultMsg
            call WriteString
            mov eax, result
            call WriteInt

            call WaitMsg  ; Wait for a key press

            dec ecx
            jnz loop_repeat  ; Repeat the process three times

        call ExitProcess
    main ENDP
END main
```

Exercise 5: BetterRandomRange Procedure

```
include Irvine32.inc

.data

.code
BetterRandomRange PROC
    ; Input: EBX = Lower bound (M)
    ;        EAX = Upper bound (N)
    ; Output: EAX = Random number between M and N-1

    sub eax, ebx   ; Calculate the range (N - M)
    add eax, 1     ; Include the upper bound itself
    call RandomRange
    add eax, ebx   ; Offset the result by M (lower bound)
    ret
BetterRandomRange ENDP

main PROC
    mov ecx, 50  ; Repeat 50 times
    loop_repeat:
        call Clrscr

        mov ebx, -300 ; Lower bound
        mov eax, 100  ; Upper bound
        call BetterRandomRange

        ; Display the randomly generated value
        mov edx, eax
        call WriteInt
        call Crlf

        call WaitMsg  ; Wait for a key press
        dec ecx
        jnz loop_repeat  ; Repeat the process 50 times

    call ExitProcess
main ENDP
END main
```

These solutions demonstrate the requested functionality for each exercise. Feel free to modify them, as you do your practice to be a better programmer.

Exercise 1: Draw Text Colors Part 2

Write a program that displays the same string in four different colors, using a loop. Call the Set-TextColor procedure from the book's link library. Any colors may be chosen, but you may find it easiest to change the foreground color.

```
INCLUDE Irvine32.inc

.data
message DB "Colorful Text",0

.code
main PROC
   mov ecx, 4 ; Number of times to display the text in different colors
   mov esi, 1 ; Color index

colorLoop:
   call SetTextColor
   mov edx, OFFSET message
   call WriteString
   call Crlf

   inc esi ; Move to the next color
   loop colorLoop

   exit
main ENDP

SetTextColor PROC
   ; Set the text color based on the value in ESI
   mov eax, esi
   switch eax
      case 1
         call SetTextColorRed
      case 2
         call SetTextColorBlue
      case 3
         call SetTextColorGreen
      case 4
         call SetTextColorYellow
   endSwitch
   ret
SetTextColor ENDP

SetTextColorRed PROC
```

```
   mov eax, Red
   call SetTextColor
   ret
SetTextColorRed ENDP

SetTextColorBlue PROC
   mov eax, Blue
   call SetTextColor
   ret
SetTextColorBlue ENDP

SetTextColorGreen PROC
   mov eax, Green
   call SetTextColor
   ret
SetTextColorGreen ENDP

SetTextColorYellow PROC
   mov eax, Yellow
   call SetTextColor
   ret
SetTextColorYellow ENDP

Yellow = 14
Green = 10
Blue = 9
Red = 12

END main
```