

# *Defining and Using Procedures*

Procedures are subroutines that can be used to divide programs into separate tasks. This makes the program easier to understand, implement, and test.

In assembly language, a procedure is a block of code that is identified by a name. The procedure can be called from anywhere in the program.



Procedures can be used to encapsulate data and code. This makes the program more modular and reusable.

Assembly language programmers must impose their own formal structure on programs, as it does not have the formal structure found in object-oriented languages.



=====

## ***PROC Directive***

=====

The PROC directive is used to define a procedure in assembly language. A procedure is a named block of code that can be called from anywhere in the program.

To define a procedure, you use the PROC directive followed by the name of the procedure. The procedure then ends with the **ENDP directive**.

For example, the following code defines a procedure called SumOf:

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

This procedure calculates the sum of three 32-bit integers and returns the result in the EAX register.

## ***Calling a Procedure***

To call a procedure, you simply use the name of the procedure. For example, the following code calls the SumOf procedure:

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

call SumOf
```

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

## ***Returning from a Procedure***

To return from a procedure, you use the RET instruction. This instruction causes the CPU to return to the location from where the procedure was called.

The PROC directive is a powerful tool that allows you to define procedures in assembly language. Procedures can be used to divide programs into separate tasks, which makes the program easier to understand, implement, and test.

=====

## ***Call and RET Instructions***

=====

The CALL instruction calls a procedure by pushing the return address on the stack and copying the called procedure's address into the instruction pointer.

When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer.

The following example shows how to call and return from a procedure:

```

.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

```

Now, let's break down and explain the example:

**.code Section:** This section contains the main program and the MyProcedure procedure.

**main PROC:** This is the starting point of the program.

**call MyProcedure:** The CALL instruction is used to call the MyProcedure procedure. When this instruction is executed, it does the following: Pushes the return address onto the stack. This return address is the address of the instruction immediately following the CALL instruction in the main program. Copies the address of the MyProcedure into the instruction pointer (EIP). After the CALL instruction, you can include any code you want to execute after calling the procedure.

**MyProcedure PROC:** This is the beginning of the MyProcedure procedure. Inside MyProcedure, you can include any code specific to this

procedure. This is where you define what the procedure does.

**ret:** The RET instruction is used to return from the procedure. When this instruction is executed, it does the following: Pops the return address from the stack into the instruction pointer (EIP). This effectively "jumps back" to the instruction immediately after the CALL in the main program, continuing execution from there.

**MyProcedure ENDP:** This marks the end of the MyProcedure procedure.

**main ENDP:** This marks the end of the main program.

**END main:** This indicates the end of the assembly program.

In summary, the CALL instruction is used to call a procedure, and the RET instruction is used to return from the procedure to the caller. The return address is stored on the stack during the CALL and retrieved from the stack during the RET, allowing for structured program flow and function calls in assembly language.

=====

## ***Nested Procedure Calls***

=====

A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns.

The following example shows a nested procedure call:

```

.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

```

Now, let's break down and explain the example of nested procedure calls:

**.code Section:** This section contains the main program and two procedures, OuterProcedure and InnerProcedure.

**main PROC:** This is the starting point of the program.

**call OuterProcedure:** The main program calls OuterProcedure using the CALL instruction. When this happens, control is transferred to OuterProcedure.

**OuterProcedure PROC:** Inside OuterProcedure, there is some code specific to this procedure. After executing this code, it calls InnerProcedure using the CALL instruction.

**call InnerProcedure:** The CALL instruction within OuterProcedure invokes InnerProcedure. Control is transferred to InnerProcedure, and

the execution continues there.

**InnerProcedure PROC:** Inside InnerProcedure, there is some code specific to this procedure. After executing its code, InnerProcedure returns using the RET instruction.

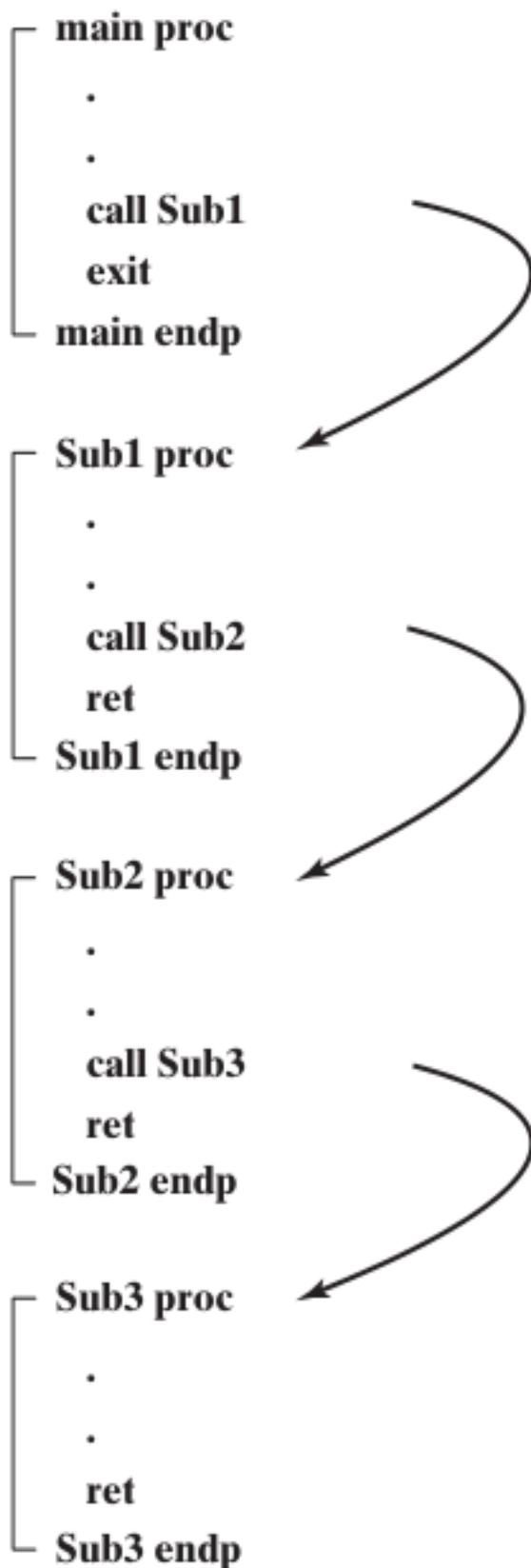
**ret (InnerProcedure):** The RET instruction in InnerProcedure causes control to return to the point immediately after the CALL instruction in OuterProcedure.

**ret (OuterProcedure):** Finally, when OuterProcedure completes its execution, it returns using the RET instruction. This brings control back to the instruction immediately after the CALL instruction in the main program.

The main program can then continue executing after the call to OuterProcedure.

In summary, this example demonstrates nested procedure calls. The main program calls OuterProcedure, which in turn calls InnerProcedure. Each procedure has its own code and can perform specific tasks.

The use of CALL and RET instructions facilitates structured program flow, allowing you to call procedures within procedures and manage control flow effectively.



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:

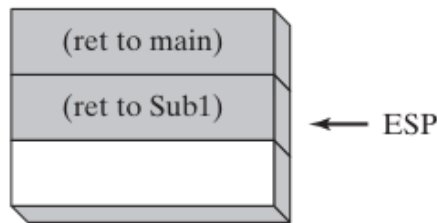
- The main procedure calls Sub1 by executing the CALL Sub1 instruction.



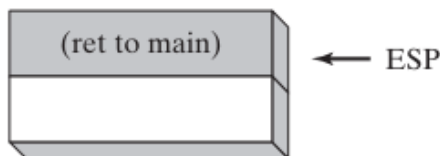
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub1.
- The CALL instruction then loads the address of Sub1 into the instruction pointer.
- The Sub1 procedure begins executing. Sub1 calls Sub2 by executing the 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.

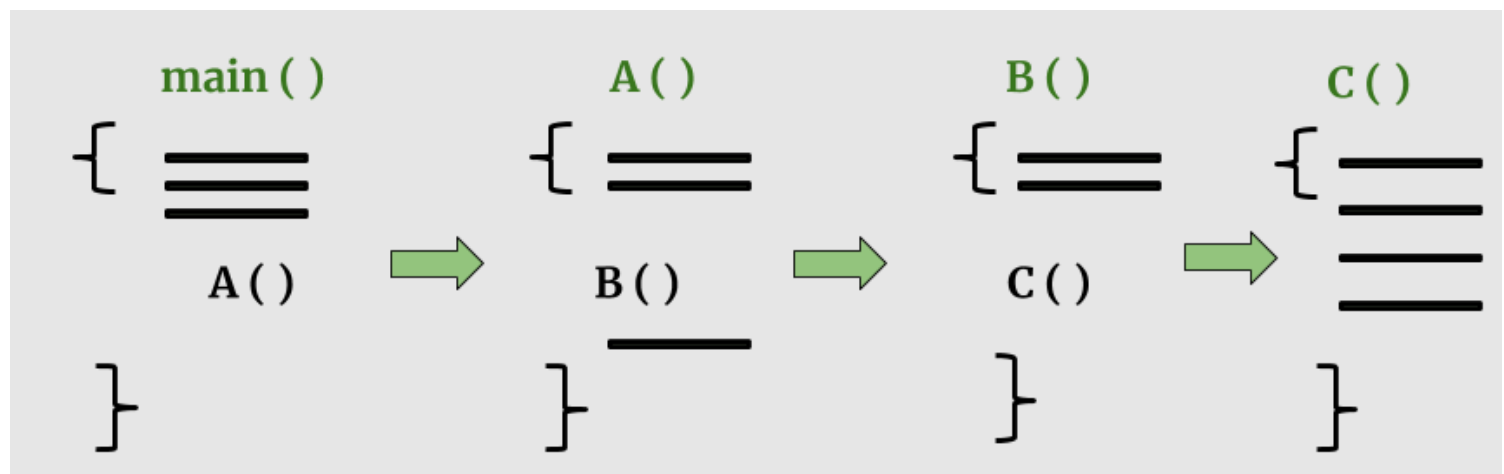
-----

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.