

Exploration 1 - Passing Parameters on the Stack

Passing Parameters on the Stack



As promised, we've progressed far enough to start discussing *passing **parameters** on the **runtime stack***. Things are about to get complicated, so it is important to keep these in mind:

- The stack is just an area of memory.
- Each “place” in the stack has a distinct address, just like any other place in memory.
- The stack is contiguous within the memory region where the stack resides.

It may help to review [Module 5, Exploration 1 - The Runtime Stack](https://canvas.oregonstate.edu/courses/1976334/pages/exploration-1-the-runtime-stack) (<https://canvas.oregonstate.edu/courses/1976334/pages/exploration-1-the-runtime-stack>) prior to embarking on this section.

The Stack Frame (Activation Record) and the Call Stack

As you may recall from [Module 5, Exploration 3 - More on Procedures: Documentation and Modularization](https://canvas.oregonstate.edu/courses/1976334/pages/exploration-3-more-on-procedures-documentation-and-modularization) (<https://canvas.oregonstate.edu/courses/1976334/pages/exploration-3-more-on-procedures-documentation-and-modularization>), an **activation record** (also known as a **stack frame**) of a procedure is a section of the stack containing a procedure's components. To drill down, we'll be adding these components to the stack in the following order:

1. Passed parameters

By **PUSH**ing before the procedure call

2. Return address

By the procedure **CALL**

3. Old value of Base Pointer

By `PUSH EBP` or assembler directive

4. Local Variables

By directly decrementing `ESP` or by assembler directive `LOCAL`

5. Saved Registers

Individually by `PUSH` , as a group by `PUSHAD` , or by assembler directive `USES`

With nested procedures the stack ends up having one activation record on top of another, which is called a call stack (illustrated in this [Call Stack Video \(https://media.oregonstate.edu/media/t/0_i7m1jcdt\)](https://media.oregonstate.edu/media/t/0_i7m1jcdt)). So how is this **stack frame** created?

Passed Parameters

The first part of the **stack frame** pushed on the stack is the parameters. We have previously shown the syntax for passing *input*, *output*, and *input/output* parameters, but to review...

- **Input parameters** *may* be passed as **value parameters** (passed by *value*), as shown for `lowVal` and `highVal` in the example below. These **values**, once on the stack, have no relationship to the memory or register location (or immediate) they were pulled from.
- **Output parameters** and **Input/Output parameters** *must* be passed as **reference parameters** (passed by *reference/address/pointer*), as shown for `sum` in the example below. These **addresses** may be used to change contents in memory.

```
PUSH lowVal      ; Pass value parameter "lowVal"  
PUSH highVal     ; Pass value parameter "highVal"  
PUSH OFFSET sum  ; Pass reference parameter, pointer to "sum"  
CALL sumVals     ; Call the procedure
```

Return Address

The *calling procedure* pushes the return address onto the stack as the first step of the **CALL** instruction calling the procedure. This address will be the value loaded into **EIP** to return from the called procedure.

Base Pointer

Until now we have only used the **stack pointer** **ESP** when addressing the contents of the stack. The stack pointer has some problematic limitations, however, because it is automatically updated to always point at the *top of the stack*. Since we will need to be referencing parameters inside the procedure, perhaps while making additional modifications to the **ESP**, it is vital to have an additional pointer which we can set to point at any particular location on the stack without potentially misaligning the stack.

The **base pointer** **EBP** is used for this purpose. It is sometimes called a *frame pointer* because it points to a static (within the procedure) location in the stack frame. As with any other register, if we want to assign it a new value, the current value must be preserved beforehand. The takeaway from this little discussion is that your first task in writing MASM procedures will always be to preserve **EBP** and then copy **ESP** into it. From now on, whenever you write a procedure, the first two instructions within will be:

```
myProc PROC
    PUSH    EBP           ; Step 1) Preserve EBP
    MOV     EBP, ESP      ; Step 2) Assign static stack-frame pointer
    ; ... All the rest of the procedure
myProc ENDP
```

Local Variables

Local variables will be covered in more detail later in the course, but traditionally higher-level language functions make local variables by reserving space on the system stack (often by directly decrementing **ESP** without pushing anything on the stack). This space is reserved just "above" the base pointer (at lower addresses), but before any registers are saved.

Saved Registers

After making space for any potential local variables, the *called procedure* will push any registers that must be saved onto the stack. The methods for those (`PUSH` vs. `PUSHAD`) were discussed previously in [Module 5, Exploration 1 - The Runtime Stack](https://canvas.oregonstate.edu/courses/1976334/pages/exploration-1-the-runtime-stack)

(<https://canvas.oregonstate.edu/courses/1976334/pages/exploration-1-the-runtime-stack>)_.

More on the RET Instruction

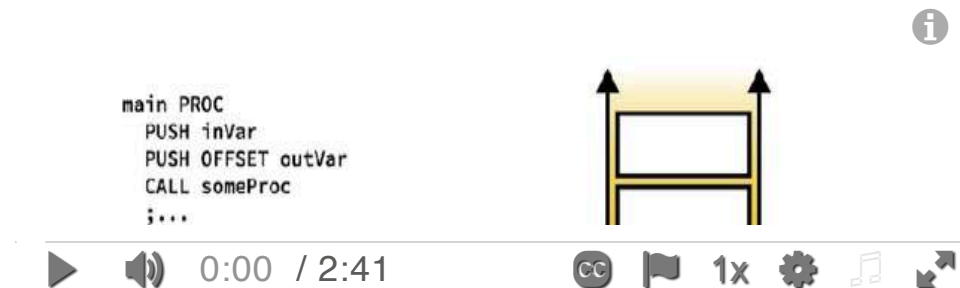
Recall that the `RET` instruction (with no operands) will pop the top of the stack into the **instruction pointer** `EIP`. The optional operand `n` (syntax: `RET [n]`) causes `n` to be added to the **stack pointer** `ESP` after `EIP` is assigned a value. The purpose of this is to de-reference any parameters which had been pushed on the stack. The value of `n` should be equal to the number of *bytes* of parameters which were pushed on the stack before the `CALL` statement.

For example, if one `DWORD` (4 bytes), one `WORD` (2 bytes), and two address `OFFSET`s (4 bytes each) were `PUSH`ed before the `CALL`, this would total $4 + 2 + 4 + 4 = 14$ bytes. So then `RET 14` should be used in the called procedure.

```
main PROC
; ...
PUSH  varA           ; varA is a DWORD, 4 bytes
PUSH  varB           ; varA is a WORD, 2 bytes
PUSH  OFFSET arrayA  ; Address OFFSETs are 4 bytes
PUSH  OFFSET outVarB ; Address OFFSETs are 4 bytes
CALL  someProc       ; Total parameter space = 4 + 2 + 4 + 4
                     = 14 bytes
; ...
main ENDP

someProc PROC
PUSH  EBP
MOV   EBP, ESP      ; Base Pointer
; ...
POP   EBP
RET   14            ; De-reference 4 + 2 + 4 + 4 = 14 bytes
someProc ENDP
```

Note that the top of the stack is popped *before* the additional adjustment is made. Also note that the size of the return address is not counted in this calculation, so `RET` (with no operands) is equivalent to `RET 0` .



This ordering for building the stack frame, the use of the stack for passed parameters, and the method for deconstructing it are collectively known as the **stdcall** calling convention.

Referencing Stack-Passed Parameters

It can be seen from the animation above that the stack-passed parameters will be within the stack frame, but not at the *top of the stack*. We will need some way to get to values that aren't on the top of the stack. The tool we'll use for this is an alternate **addressing mode** . Previously we have referred to data in memory using variable names (*data labels*), which is called **direct addressing**. To meet our new need we will need to use another form of addressing called **base+offset addressing**.

Base+Offset Addressing

Base+Offset is an addressing mode where a **base pointer** is used to establish a known location in memory, and a **constant offset** from that memory location is used to access some information (e.g., a variable's value, a value/address stored on the stack, etc.). Remember, in MASM, square brackets are used to de-reference an address (i.e., de-reference the pointer), to get to the value stored at the address.

- `[reg]` means “contents of memory at the address in `reg`”.
 - We previously used `[ESP]` to refer to the information stored at the top of the stack.
 - If EDX holds an address (e.g. `MOV EDX, OFFSET myVar`) then `[EDX]` would access the *value stored at that address*.
- You may add a constant (named or literal) to the register inside the brackets.
 - Example: `MOV EAX, [EDX + 12]` would take the value stored in memory at the address starting 12 bytes *after* the address stored in `EDX` and save that value in to `EAX`.
- NOTE: Base+Offset addressing generates a *memory reference*, for instruction operand purposes.
 - `[reg]` (e.g. `[EDX]`) and `[reg+constant]` (e.g. `[EDX + 12]`) are both memory references (addresses)
 - Recall there are no *memory-to-memory* instruction formats, so `MOV [EDX], [EAX]` is invalid syntax!
- Often a size is implied (such as moving from a 4-byte register to memory) but some operations require an explicit statement of operand size. More on this later...

Check your knowledge!



Given the following data segment, identify what hex value is put into the destination register for each listed operation. Remember that x86 systems use little endian order, since this will impact storage and reading of anything accessed as a multi-byte data type.

```
.data
myArr1  WORD  11AAh, 33CCh, 55EEh    ; 3-WORD array, in memor
```

y:

```
myArr2 BYTE "Hello There!",0 ; AA 11 CC 33 EE 55
                                     ; BYTE array in memory:
                                     ; 48 65 6C 6C 6F 20 54 68
65 72 65 21 00
.code
main PROC
    MOV     EBX, OFFSET myArr1    ; Address of myArr1 into EBX
    MOV     EDX, OFFSET myArr2    ; Address of myArr2 into EDX
    ;... listed operations here
    exit
main ENDP
```

1. MOV AX, [EBX]

AX = ?h

[Click here to reveal the answer.](#)

2. MOV AX, [EBX+2]

AX = ?h

[Click here to reveal the answer.](#)

3. MOV AL, [EBX+3]

AL = ?h

[Click here to reveal the answer.](#)

4. MOV AL, [EDX]

AL = ?h

[Click here to reveal the answer.](#)

5. MOV AL, [EDX+4]

AL = ?h

[Click here to reveal the answer.](#)

6. MOV AX, [EDX+6]

AX = ?h

[Click here to reveal the answer.](#)

7. Challenge: MOV EAX, [EBX]

EAX = ?h

[Click here to reveal the answer.](#)

8. Challenge: `MOV EAX, [EDX+3]`

EAX = ?h

[Click here to reveal the answer.](#)

9. Challenge: `MOV EAX, [EBX+4]`

EAX = ?h

[Click here to reveal the answer.](#)

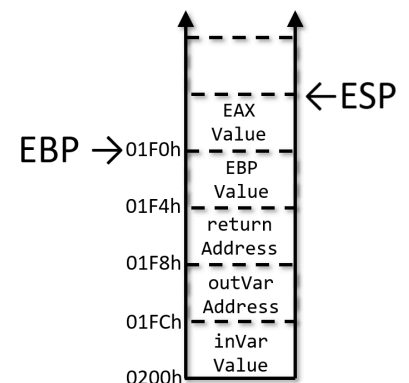
Base+Offset Access to Stack Parameters

When the **stack frame** is established by moving the value of `ESP` into `EBP`, the register `EBP` (called the **base pointer** or **frame pointer**) provides a static pointer to the runtime stack, from which we can offset to explicitly reference any parameters which were passed on the stack. It is vital the new value in `EBP` remain unchanged within the body of the procedure. If it is changed, the Base+Offset references will have to track those changes, which defeats the purpose of having the static pointer. Don't forget that `EBP` must be restored to its original value before execution returns to the calling procedure.

Let's take a look at some example code. You've already seen the animation that creates this stack frame (above). Remember that the return address is pushed onto the stack *after the parameters are pushed*, and the software engineer is responsible for managing the stack.

```
main PROC
    PUSH inVar
    PUSH OFFSET outVar
    CALL someProc
    ; ...
main ENDP

someProc PROC
    PUSH EBP
    MOV EBP, ESP
    PUSH EAX
    ; Execution Point A
    POP EAX
```



Passed Parameters
Diagram


```
POP EBP
RET 8
someProc ENDP
```

Consider the case where, at Execution Point A, we need to move the value of input parameter `inVar` to `EAX` for processing. Using what you know about the stack, and Base+Offset addressing, what code would accomplish this?

Note the value of `EBP` = 01F0h and the address where the value of `inVar` is stored is 01FCh. One way to access this point on the stack using Base+Offset addressing would be first to calculate what offset is necessary (01FCh - 01F0h = 0Ch = 12d), and then to write the move statement appropriately...

```
MOV EAX, [EBP+12] ; Move input parameter inVar to EAX
```

So how do you find that value (12) without knowing the stack addresses? The value is calculated by counting the *number of bytes pushed on the stack after the value you're looking for, up to (and including) the old value of `EBP`*. Looking at the code above, we pushed (in order)...

1. `inVar`'s value (not counted, this is what we're looking for)
2. The OFFSET of `outVar` (this is an address offset, so count 4 bytes)
3. The return address from the called procedure (address pushed by `CALL`, so count 4 bytes)
4. The old value of `EBP` (a 4-byte register, so count 4 bytes)

Knowing that 12 bytes were pushed to the stack *after* the desired value and *before* the stack base pointer `EBP` was set, and remembering that the stack pointer *decrements*, we must first add 12 to the stack address stored in `EBP`, then de-reference it to get the value of `inVar`:

```
MOV EAX, [EBP+12] ; Move the pushed value of inVar to EAX
```

Now expand this and try to do another! What statement would you write to load the OFFSET of `outVar` into the `EDX` register? It's worked out below, so you can check your answer.

1. `inVar`'s value (not counted, pushed before what we're looking for)
2. `OFFSET outVar` (not counted, this is what we're looking for)
3. The return address (address pushed by `CALL`, so count 4 bytes)
4. Old value of `EBP` (4-byte register, so count 4 bytes)

Summing these we get 8, so to move the address offset of `outVar` into `EDX` we would write...

```
MOV     EDX, [EBP+8]    ; Move address of outVar into EDX
```

Notice that neither of these lists accounted for the `PUSH EAX` statement. This is because that instruction happens *after we set the base pointer*, so it does not change our references! That's the key reason for using the base pointer `EBP` to determine the address locations on the stack rather than `ESP`.

This video series will guide you through the process of passing parameters to a called procedure using the runtime stack. The starting point of this video is the

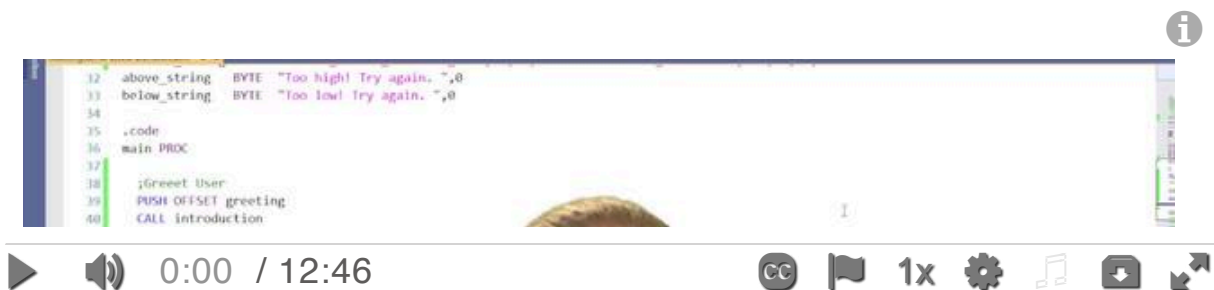
[Guessing Game with Data Validation](https://canvas.oregonstate.edu/courses/1976334/files/107292336?wrap=1)


<https://canvas.oregonstate.edu/courses/1976334/files/107292336?wrap=1>)_ ↓

https://canvas.oregonstate.edu/courses/1976334/files/107292336/download?download_frd=1) we worked on earlier this term.




The result of this first video, and the starting point for the second video, is the **[Guessing Game with Passed Parameters \(Part 1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292399?wrap=1)** (<https://canvas.oregonstate.edu/courses/1976334/files/107292399?wrap=1>)_ [↓](https://canvas.oregonstate.edu/courses/1976334/files/107292399/download?download_frd=1) (https://canvas.oregonstate.edu/courses/1976334/files/107292399/download?download_frd=1) program.



The result of this second video, and the starting point for the third video, is the [Guessing Game with Passed Parameters \(Part 2\)](https://canvas.oregonstate.edu/courses/1976334/files/107292400?wrap=1) (<https://canvas.oregonstate.edu/courses/1976334/files/107292400?wrap=1>)_ 
(https://canvas.oregonstate.edu/courses/1976334/files/107292400/download?download_frd=1) program.



The result of this video series is the [Guessing Game with Passed Parameters Final](https://canvas.oregonstate.edu/courses/1976334/files/107292398?wrap=1)

(<https://canvas.oregonstate.edu/courses/1976334/files/107292398?wrap=1>)_ 
(https://canvas.oregonstate.edu/courses/1976334/files/107292398/download?download_frd=1) program.

Check your knowledge!



Given the following data segment, main, and subprocedure, write statements to copy the listed stack-pushed parameters into the specified registers as if the statements would replace “Execution Point A”.

```
.data
name      BYTE    "Margaret Hamilton",0
year      WORD    2020
```

```
birthday    DWORD    08171936
calcAge     WORD     ?
```

```
.code
main PROC
    PUSH OFFSET name
    PUSH year
    PUSH birthday
    PUSH OFFSET calcAge
    CALL ageCalc
    ;...
main ENDP
```

```
ageCalc PROC
    PUSH EBP
    MOV  EBP, ESP
    PUSH EAX
    PUSH EDX
    ; Execution Point A
    POP  EDX
    POP  EAX
    POP  EBP
    RET  14
ageCalc ENDP
```

NOTE: For more info on Margaret Hamilton, who among other things worked at the MIT Instrumentation Laboratory on software for NASA's Apollo Program (in Assembly!), check out her [Wikipedia Bio](https://en.wikipedia.org/wiki/Margaret_Hamilton_(software_engineer))) ➞

[\(\(https://en.wikipedia.org/wiki/Margaret_Hamilton_\(software_engineer\)\)](https://en.wikipedia.org/wiki/Margaret_Hamilton_(software_engineer)))).

1. Address of into .
- [Click here to reveal the answer.](#)
2. Value of into .
- [Click here to reveal the answer.](#)
3. Value of into .
- [Click here to reveal the answer.](#)
4. Address of into .
- [Click here to reveal the answer.](#)

Stack-passed Parameter Tips

All of these tips are reflected in the code samples above, but direct reminders have been very useful in the past.

- Save and restore registers when they are modified by a procedure using STDCALL style.
 - Pop in the reverse order that you pushed (see code above for example)
 - Exception: A register which is used for a return parameter - note this in the procedure documentation.
- Ensure the runtime stack is balanced by the time the procedure ends (`ESP` should be pointing at the return address before `RET` is executed).
- Ensure the operand for a procedure's `RET` instruction reflects the number of bytes pushed on the stack just before the `CALL` instruction.
- Do not pass an *immediate value* or *value parameter* to a procedure that expects a *reference parameter* (address/pointer).
 - De-referencing a value or immediate as if they were an address will likely attempt to access a region of memory outside the stack segment, which will cause the CPU to issue a special type of error called a *general-protection fault*.






Check your knowledge!



To help you further reinforce the operation of `PUSH`, `POP`, `CALL`, `RET`, and de-referencing addresses, please trace the register and memory contents for the program in the [Module 7 Exploration 1 Worksheet](https://canvas.oregonstate.edu/courses/1976334/files/107292370/download?wrap=1) (<https://canvas.oregonstate.edu/courses/1976334/files/107292370/download?wrap=1>). Fill in the table entry for each *changed* register and memory location. The idea is to trace the execution, showing every change in the indicated registers and memory. The traces for the first two instructions are shown for you, so you can understand the format requested.

[Click here to reveal the answer.](#)

Optional Additional Resources

- The **[Apollo Guidance Computer](https://en.wikipedia.org/wiki/Apollo_Guidance_Computer)**  **[\(https://en.wikipedia.org/wiki/Apollo_Guidance_Computer\)](https://en.wikipedia.org/wiki/Apollo_Guidance_Computer)** was the first silicon integrated circuit based computer, a monumental accomplishment. Check out its Instruction Set!
- **[GuessingGameDataValidation.asm](https://canvas.oregonstate.edu/courses/1976334/files/107292336?wrap=1)**
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292336?wrap=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292336?wrap=1) 
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292336/download?download_frd=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292336/download?download_frd=1)
- **[GuessingGamePassedParametersPt1.asm](https://canvas.oregonstate.edu/courses/1976334/files/107292399?wrap=1)**
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292399?wrap=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292399?wrap=1) 
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292399/download?download_frd=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292399/download?download_frd=1)
- **[GuessingGamePassedParametersPt2.asm](https://canvas.oregonstate.edu/courses/1976334/files/107292400?wrap=1)**
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292400?wrap=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292400?wrap=1) 
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292400/download?download_frd=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292400/download?download_frd=1)
- **[GuessingGamePassedParametersFinal.asm](https://canvas.oregonstate.edu/courses/1976334/files/107292398?wrap=1)**
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292398?wrap=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292398?wrap=1) 
 [\(https://canvas.oregonstate.edu/courses/1976334/files/107292398/download?download_frd=1\)](https://canvas.oregonstate.edu/courses/1976334/files/107292398/download?download_frd=1)