Today’s Topics

Advanced Procedures:
- Stack Parameters
- Local Variables
- Stack Frame
- Recursion
Study Guide

• Section 8.3: Stack Parameters
• Section 8.2: Local Variables
• Section 8.4: Stack Frame
• Section 8.5: Recursion

Overview

• Why study stack parameters and local variables?
  1. Passing stack parameters is more convenient than passing register parameters.
  2. This is how the compilers of high-level language handle parameters and local variables.
Register vs. Stack Parameters

- Register parameters require dedicating a register to each parameter. Stack parameters are more convenient.
- Imagine two possible ways of calling the DumpMem procedure. Clearly the second is easier:

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```  

```
push OFFSET array
push LENGTHOF array
push TYPE array
call DumpMem
```  

INVOKE Directive

- The INVOKE directive is a powerful replacement for Intel's CALL instruction that lets you pass multiple arguments.
- Syntax:
  
  ```
  INVOKE procedureName [, argumentList]
  ```
- Arguments can be:
  - immediate values and integer expressions
  - variable names
  - address and ADDR expressions
  - register names
INVOKE Examples

.data
byteVal BYTE 10
wordVal WORD 1000h
.code
; direct operands:
INVOKE Sub1,byteVal,wordVal

; address of variable:
INVOKE Sub2,ADDR byteVal

; register name, integer expression:
INVOKE Sub3,eax,(10 * 20)

; address expression (indirect operand):
INVOKE Sub4,[ebx]

ADDR Operator

• Returns a near or far pointer to a variable, depending on which memory model your program uses:
  • Small model: returns 16-bit offset
  • Large model: returns 32-bit segment/offset
  • Flat model: returns 32-bit offset
• Simple example:

.data
myWord WORD ?
.code
INVOKE mySub,ADDR myWord
PROC Directive

- The PROC directive declares a procedure with an optional list of parameters.
- Syntax:
  
  \[ \text{label} \text{ PROC} \text{ paramList} \]

- \text{paramList} is a list of parameters separated by commas. Each parameter has the following syntax:

  \[ \text{paramName}:\text{type} \]

\text{type} must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

Sanity Check #1

Remember that PROC and INVOKE are directives, not instructions.

(i.e., CPU does not understand it.)
PROC Examples (1 of 2)

- The AddTwo procedure receives two integers and returns their sum in EAX.
- C++ programs typically return 32-bit integers from functions in EAX.

```
AddTwo PROC,
    val1:DWORD, val2:DWORD
    mov eax,val1
    add eax,val2
    ret
AddTwo ENDP
```

PROC Examples (2 of 2)

FillArray receives a pointer to an array of bytes, a single byte fill value that will be copied to each element of the array, and the size of the array.

```
FillArray PROC,
    pArray:PTR BYTE, fillVal:BYTE
    arraySize:DWORD
    mov ecx,arraySize
    mov esi,pArray
    mov al,fillVal
L1:mov [esi],al
    inc esi
    loop L1
    ret
FillArray ENDP
```
PROTO Directive

- Creates a procedure prototype
- Syntax:
  - `label PROTO paramList`
- Every procedure called by the INVOKE directive must have a prototype
- A complete procedure definition can also serve as its own prototype

Standard configuration: PROTO appears at top of the program listing, INVOKE appears in the code segment, and the procedure implementation occurs later in the program:

```plaintext
MySub PROTO ; procedure prototype
.CODE
.INVOKE MySub ; procedure call
MySub PROC ; procedure implementation
.
MySub ENDP
```
PROTO Example

• Prototype for the ArraySum procedure, showing its parameter list:

```plaintext
ArraySum PROTO,
    ptrArray:PTR DWORD,
    szArray:DWORD
```

Sanity Check #2

So, PROC and INVOKE are directives, not instruction.
How does MASM handle the parameters?
Passing by Value

• When a procedure argument is passed by value, a copy of a 16-bit or 32-bit integer is pushed on the stack. Example:

```
.data
myData WORD 1000h
.code
main PROC
    INVOKE Sub1, myData
```

MASM generates the following code:

```
push myData
call Sub1
```

Passing by Reference

• When an argument is passed by reference, its address is pushed on the stack. Example:

```
.data
myData WORD 1000h
.code
main PROC
    INVOKE Sub1, ADDR myData
```

MASM generates the following code:

```
push OFFSET myData
call Sub1
```
Example: Exchanging Two Integers

The Swap procedure exchanges the values of two 32-bit integers. pValX and pValY do not change values, but the integers they point to are modified.

```
Swap PROC USES eax esi edi,
pValX:PTR DWORD, ; pointer to first integer
pValY:PTR DWORD ; pointer to second integer
mov esi,pValX ; get pointers
mov edi,pValY
mov eax,[esi] ; get first integer
xchg eax,[edi] ; exchange with second
mov [esi],eax ; replace first integer
ret
Swap ENDP
```

Example: Exchanging Two Integers

Will the following work?

```
Swap PROC USES eax esi edi,
 X:DWORD,
 Y:DWORD
mov eax, X ; get first integer
xchg eax, Y ; exchange with second
mov X,eax ; replace first integer
ret
Swap ENDP
```

A DWORD 10
B DWROD 20
INVOKE Swap A, B
INVOKE Swap ADDR A, ADDR B ;for previous slide
Memory Address of Parameter

• Where are the parameters stored in memory?
• Hint: INVOKE is translated into PUSH and CALL.

• Answer: They are stored in stack!

Local Directive

• A local variable is created, used, and destroyed within a single procedure
• The LOCAL directive declares a list of local variables
  – immediately follows the PROC directive
  – each variable is assigned a type
• Syntax:
  LOCAL varlist
Example:

<table>
<thead>
<tr>
<th>MySub PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL var1:BYTE, var2:WORD, var3:SDWORD</td>
</tr>
</tbody>
</table>
Local Variables

Examples:

LOCAL flagVals[20]:BYTE ; array of bytes

LOCAL pArray:PTR WORD ; pointer to an array

myProc PROC, ; procedure
    p1:PTR WORD ; parameter
    LOCAL t1:BYTE, ; local variables
t2:WORD,
t3:DWORD,
t4:PTR DWORD

MASM-Generated Code (1 of 2)

MASM generates the following code:

BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    ...
    ret
BubbleSort ENDP

BubbleSort PROC
    push ebp
    mov ebp,esp
    add esp,0FFFFFFF8h ; add -8 to ESP
    ...
    mov esp,ebp
    pop ebp
    ret
BubbleSort ENDP
MASTM-Generated Code (2 of 2)

Diagram of the stack frame for the BubbleSort procedure:

Stack Frame

• Also known as an activation record
• Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
• Created by the following steps:
  – Calling program pushes arguments on the stack and calls the procedure.
  – The called procedure pushes EBP on the stack, and sets EBP to ESP.
  – If local variables are needed, a constant is subtracted from ESP to make room on the stack.
Explicit Access to Stack Parameters

- A procedure can explicitly access stack parameters using constant offsets from EBP.
  - Example: [EBP + 8]
- EBP is often called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the procedure.
- EBP must be restored to its original value when a procedure returns.

Stack Frame Example (1 of 2)

```
.data
sum DWORD ?
.code
    push 6 ; second argument
    push 5 ; first argument
    call AddTwo ; EAX = sum
    mov sum,eax ; save the sum
AddTwo PROC
    push ebp
    mov ebp,esp
    .
    .
```

```
00000006 [EBP + 12]
00000005 [EBP + 8]
return address [EBP + 4]
EBP ← EBP, ESP
```
Stack Frame Example (2 of 2)

AddTwo PROC
    push ebp
    mov ebp,esp ; base of stack frame
    mov eax,[ebp + 12] ; second argument (6)
    add eax,[ebp + 8] ; first argument (5)
    pop ebp
    ret 8 ; clean up the stack
AddTwo ENDP ; EAX contains the sum

What is Recursion?

- The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:
Calculating a Factorial (1 of 3)

This function calculates the factorial of integer \( n \). A new value of \( n \) is saved in each stack frame:

```c
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

As each call instance returns, the product it returns is multiplied by the previous value of \( n \).

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

0! = 1 (base case)

5 * 24 = 120

4 * 6 = 24

3 * 2 = 6

2 * 1 = 2

1 * 1 = 1

This function calculates the factorial of integer \( n \). A new value of \( n \) is saved in each stack frame:

```c
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Calculating a Factorial (2 of 3)

```asm
Factorial PROC
    push ebp
    mov ebp,esp
    mov eax,[ebp+8] ; get n
    cmp eax,0 ; n < 0?
    ja L1 ; yes: continue
    mov eax,1 ; no: return 1
    jmp L2
L1: dec eax
    push eax ; Factorial(n-1)
    call Factorial
    ; Instructions from this point on execute when
    ; each recursive call returns.
    ReturnFact:
        mov ebx,[ebp+8] ; get n
        mul ebx ; ax = ax * bx
        L2: pop ebp ; return EAX
        ret 4 ; clean up stack
Factorial ENDP
```

See the program listing (Fact.asm)
Calculating a Factorial  (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial.

Each recursive call uses 12 bytes of stack space.