Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Chapter 8:Advanced Procedures

Chapter Overview

- Local Variables
- Stack Parameters
- Stack Frames
- Recursion
- Creating Multimodule Programs

8.2 Local Variables

- Global variables
  - Variables declared in the data segment are static global variables
    - Static: indicate a variable's lifetime
      - The same as the duration of program
    - Global: indicate a variable's visibility
      - Visible from all procedures in the current source code file
- Local variables
  - A variable that is created, used, and destroyed within a single procedure

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:

```
MySub PROC
    LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

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)

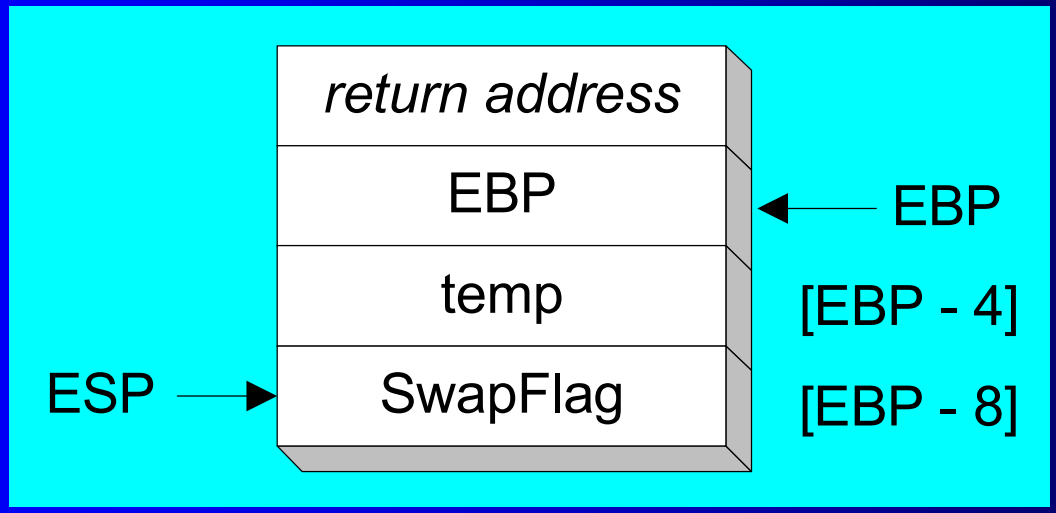
```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

MASM generates the following code:

```
BubbleSort PROC
    push ebp
    mov  ebp,esp
    add  esp,0FFFFFFF8h         ; add -8 to ESP
    . . .
    mov  esp,ebp
    pop  ebp
    ret
BubbleSort ENDP
```

MASM-Generated Code (2 of 2)

Diagram of the stack frame for the BubbleSort procedure:



Reserving Stack Space

- In irvine32.inc file, we have the following statement
  - *.stack 4096*
  - Allocate 4096 bytes of stack space
- Reserve adequate stack space
  - If you plan to create arrays of any size as local variables
- Note, if nested procedure calls are invoked
  - Stack space must be large enough to hold the sum of all local variables active at any point in the program's execution

Reserving Stack Space

- Suppose SUB1 calls SUB2, and SUB2 calls SUB3
- When in SUB3, the program will have the combined local variables from SUB1, SUB2, and SUB3 on the stack
  - There will be 660 bytes used by local variables
  - Plus the two procedure return address (8 bytes)
  - Plus any registers that might have been pushed on the stack

```
Sub1    PROC
    LOCAL    array1[50]: DWORD      ;200 bytes
..
..
SUB2    PROC
    LOCAL    array2[80]: WORD       ;160 bytes
..
..
SUB3    PROC
    LOCAL    array3[300]: BYTE      ;300 bytes
```

8.3 Stack Parameters

- Register vs. Stack Parameters
- INVOKE Directive
- PROC Directive
- PROTO Directive
- Passing by Value or by Reference
- Parameter Classifications
- Example: Exchanging Two Integers
- Trouble-Shooting Tips

Procedure Parameters

- Two types of procedure parameters
  - Register parameters
  - Stack parameters
- ***Register parameters***
  - Optimized for program execution speed
  - Register parameters require dedicating a register to each parameter
    - Otherwise, existing register contents often must be saved
- ***Stack parameters***
  - The required arguments must be pushed on the stack by a calling program
  - Nearly all high-level language uses them

Register vs. Stack Parameters

- Imagine two possible ways of calling the DumpMem procedure. Clearly the second is easier:
  - Stack parameters are more convenient

```
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***
  - Automatically pushes arguments on the stack and calls a procedure
- Syntax:

  INVOKE *procedureName* [, *argumentList*]
- *ArgumentList* is an optional comma-delimited list of procedure arguments
- 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]
```

INVOKE Examples (Cont.)

```
.data
val1 DWORD 12345h
val2 DWORD 23456h
.code
    INVOKE AddTwo,val1,val2
```

Assembler

```
    push   val2
    push   val1
    call   AddTwo
```

ADDR Operator

- Used to pass a pointer in INVOKE directive
  - Passing an address is called passing by reference
- ADDR returns a near or a far pointer, 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
```

```
.data
Array DWORD 20 DUP(?)
.code
INVOKE Swap,
       ADDR Array
       ARRD Array+4
```

PROC Directive

- The PROC directive declares a procedure with an optional list of named parameters.
- Syntax:

  label PROC paramList
- *paramList* is a list of parameters separated by commas. Each parameter has the following syntax:

  paramName:*type*

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.

PROC Examples (1 of 3)

- The AddTwo procedure receives two integers and returns their sum in EAX.

```
AddTwo PROC,
    val1:DWORD, val2:DWORD

    mov eax,val1
    add eax,val2
    ret
AddTwo ENDP
```

PROC Examples (2 of 3)

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
```

PROC Examples (3 of 3)

```
Swap PROC,
    pValX:PTR DWORD,        ; a pointer to doubleword
    pValY:PTR DWORD
    . . .
Swap ENDP
```

```
ReadFile PROC,
    pBuffer:PTR BYTE        ; a pointer to a byte
    LOCAL fileHandle:DWORD  ; a local variable
    . . .
ReadFile ENDP
```

PROTO Directive

- Creates a procedure prototype, including a procedure's name and parameter list
- Syntax:
  - *label* PROTO *paramList*
- It requires a prototype for each procedure called by the INVOKE directive and must appear before INVOKE
- Or A complete procedure definition appears prior to the INVOKE can also serve as its own prototype

PROTO Directive

- 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:

```
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:

```
ArraySum PROTO,
    ptrArray:PTR DWORD,      ; points to the array
    szArray:DWORD            ; array size
```

Passing by Value v.s. Passing by Reference

- Passing by value:
  - A copy of the variable is pushed on the stack by the calling program

- Passing by reference:
  - The address of a variable is passed to a procedure

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
```

Passing by Reference: Example

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

```
Sub2 PROC dataPtr: PTR WORD
    mov  esi, dataptr
    mov  WORD PTR [esi], 0
    ret
Sub2  ENDP
```

Passing Data Structure

- High-level languages use passing by reference when passing a data structure
  - If pass by value, slow down the program and use up precious stack space

Parameter Classifications

- An **input parameter** is data passed by a calling program to a procedure.
  - The called procedure is not expected to modify the corresponding parameter variable, and even if it does, the modification is confined to the procedure itself.
- An **output parameter** is created by passing a pointer to a variable when a procedure is called.
  - The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
- An **input-output parameter** represents a value passed as input to a procedure, which the procedure may modify.
  - The same parameter is then able to return the changed data to the calling program.

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
```

pValX and pValY are input-output parameters

Trouble-Shooting Tips (Cont.)

- Save and restore registers when they are modified by a procedure.

- Wrong operand size: addresses are based on the size of the array elements

```
.data
DoubleArray DWORD 10000h, 20000h
.code
    INVOKE Swap, ADDR [DoubleArray+0], ADDR [DoubleArray+1]
```

It should be [DoubleArray+4] since DWORD is 4 bytes

Trouble-Shooting Tips (cont.)

- When using INVOKE, be careful to pass a pointer to the correct data type.
  - For example, MASM cannot distinguish between a DWORD argument and a PTR BYTE argument.

- Do not pass an immediate value to a procedure that expects a reference parameter.
  - Dereferencing its address will likely cause a general-protection fault.

8.4 Stack Frames

- Memory Models
- Language Specifiers
- Explicit Access to Stack Parameters
- Passing Arguments by Reference
- Creating Local Variables

Stack Frame

- Also known as an ***activation record***
- Area of the stack set aside for a procedure's *return address*, *passed parameters*, *any saved registers*, and *local variables*
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The procedure is called, causing the return address to be pushed on the stack
  - The called procedure pushes EBP on the stack
  - Sets EBP to ESP. From this point, EBP acts as a base reference for all of the procedure parameters
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Memory Models

- A program's memory model determines the number and sizes of code and data segments.
- Real-address mode supports tiny, small, medium, compact, large, and huge models.
- Protected mode supports only the flat model.

Small model: code < 64 KB, data (including stack) < 64 KB.
All offsets are 16 bits.

Flat model: single segment for code and data, up to 4 GB.
All offsets are 32 bits.

.MODEL Directive

- .MODEL directive specifies a program's memory model and model options (language-specifier).
- Syntax:

```
.MODEL memorymodel [,modeloptions]
```

- *memorymodel* can be one of the following:
  - tiny, small, medium, compact, large, huge, or flat
- *modeloptions* includes the language specifier:
  - procedure naming scheme
  - parameter passing conventions

Memory Models

Model	Description
tiny	a single segment, containing both code and data. This model is used by .com programs
small	One code segment and one data segment.
Medium	Multiple code segments and a single data segment
Compact	One code segment and multiple data segments
Large	Multiple code and data segments
Huge	Same as the large model, except that individual data Items may be larger than a single segment
Flat	protected mode. Uses 32-bit offsets for code and data

Language Specifiers

- Determine the calling and naming conventions for procedures and public symbols
- The options are C, BASIC, FORTRAN, PASCAL, SYSCALL, and STDCALL

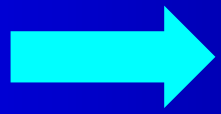
- The C, BASIC, FORTRAN, and PASCAL
  - Enable assembly language programmers to create procedures that are compatible with these languages

- The SYSCALL and STDCALL
  - Variations on the other language specifiers

Language Specifiers (Cont.)

- stdcall
  - Indicate that procedure arguments must be pushed on stack in reverse order (last to first)

```
INVOKE AddTwo,5,6
```



```
push 6          ; second argument
push 5          ; first argument
call AddTwo
```

  - How procedure arguments are removed from the stack after a procedure call
    - Called procedure cleans up the stack
    - A constant must be applied to the RET instruction

```
AddTwo PROC
…
…
ret  8              ; add 8 to the ESP after returning
ADDTWO ENDP
```

Language Specifiers (Cont.)

- stdcall (Cont.)
  - Modify exported procedure name by storing them in the following format

    _name@nn

    - A leading underscore is added to the procedure name
    - An integer follows the @ indicate the number of bytes used by the procedure parameters
    - nn: the number of bytes used by the procedure parameters
  - Example
    - Suppose **MySub** has two doubleword parameters
    - Name is **_MySub@8**

Language Specifiers (Cont.)

- C:
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack by adding a constant to the ESP
  - handle external names in the same way as STDCALL
- pascal
  - procedure arguments are pushed in forward order (left to right)
  - called procedure cleans up the stack
  - procedure names are converted to all uppercase letters

Explicit Access to Stack Parameters

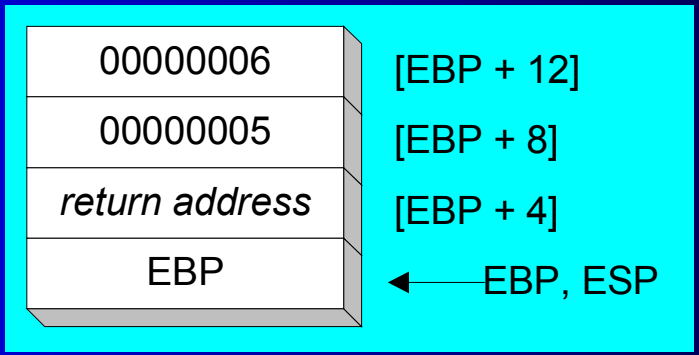
- A procedure can explicitly access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
  - Thus, you have more control to the stack elements than using INVOKE and variable names
- EBP is often called the ***base pointer*** or ***frame pointer*** because it holds the base address of the stack frame.
- EBP does not change value during the procedure.
- EBP must be restored to its original value when a procedure returns.

[1] BP in Real-address mode

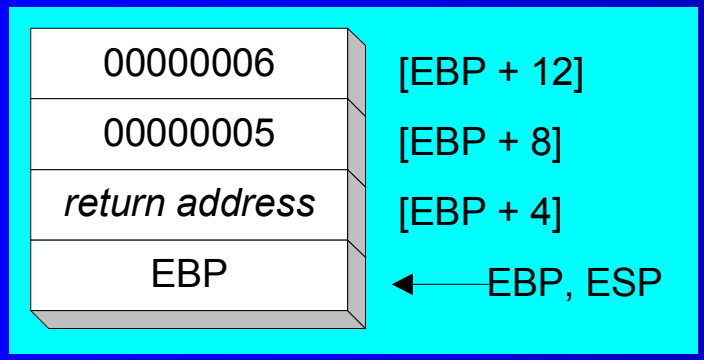
Stack Frame Example (1 of 2)

```
.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
    .
    .
```



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
```

Your turn . . .

- Create a procedure named Difference that subtracts the first argument from the second one. Following is a sample call:

```
push 14                    ; first argument
push 30                    ; second argument
call Difference            ; EAX = 16
```

```
Difference PROC
   push ebp
   mov  ebp,esp
   mov  eax,[ebp + 8]      ; second argument
   sub  eax,[ebp + 12]     ; first argument
   pop  ebp
   ret  8
Difference ENDP
```

Saving and Restoring Registers

- If a procedure modifies the value in some registers
  - It should save these registers and restore them later
- Example: push EDX and restore later
- Note: push EDX does not affect the displacement of parameters from EBP because the stack grows downward without affecting EBP

```
AddTwo PROC
    push ebp
    mov  ebp,esp
    push edx        ; save EDX
    …
    …
    pop  edx        ; restore EDX
    pop  ebp
    ret  8
AddTwo ENDP
```

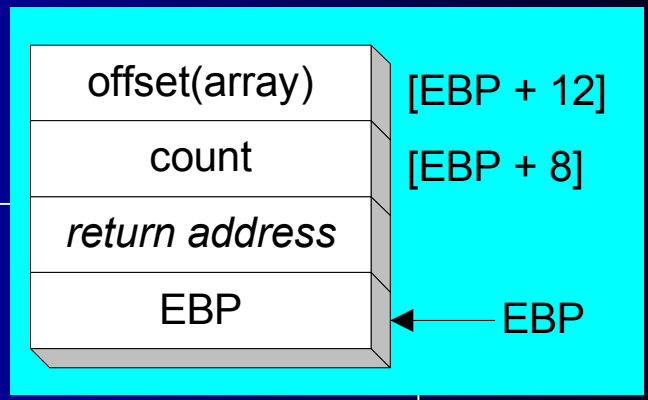
Passing Arguments by Reference (1 of 2)

- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
    push OFFSET array       ; pass by reference
    push COUNT              ; pass by value
    call ArrayFill
```

Passing Arguments by Reference (2 of 2)

In ArrayFill, ArrayFill can reference an array without knowing the array's name:




```
ArrayFill PROC
    push ebp
    mov  ebp,esp
    pushad
    mov  esi,[ebp+12]    ; offset of array
    mov  ecx,[ebp+8]     ; array size
    cmp  ecs, 0          ; ECX < 0
    jle  L2              ; yes: skip over loop
L1:
    mov  eax, 10000h     ; get random 0-FFFFh
    call Randowm_Range   ; from the link library
    mov  [esi], eax      ; indirect addressing
    add  esi, TYPE DWORD  ; point to next element
    loop L1
L2: popad
    pop  ebp
```

LEA Instruction

- The LEA instruction returns offsets of any type of indirect operands.
  - Because indirect operand may use register, thus its offset can only be calculated at run time
  - OFFSET operator, in contrast, can only return constant offsets.

- LEA is required for obtaining the offset of a stack parameter and local variable
  - Since the address is known only at run time.

LEA Instruction: Example

```
FillString PROC   USES eax, esi
    LOCAL string[20]:BYTE
; create and fill a 20-byte string with ASCII digits

    lea esi, string              ; load effective address
    mov ecx, 20
L1: mov eax, 20
    call RandomRange             ; AL = 0…9
    add al, 30h                  ; convert to ASCII character
    mov [esi], al
    add esi, 1                   ; since string element is byte
    Loop L1
    ret
FillString ENDP
```

Following instru. generates an error since string is an indirect operand

mov eax, OFFSET string ;error

Creating Local Variables

- To explicitly create local variables, subtract their total size from ESP.
  - Instead of using LOCAL Directive
- The following example creates and initializes two 32-bit local variables (we'll call them locA and locB):

```
MySub PROC
    push ebp
    mov  ebp,esp
    sub  esp,8
    mov  [ebp-4],123456h          ; locA
    mov  [ebp-8],0                ; locB
    .
    .
```

Creating Local Variables: Example

- The following C++ function declares several local variables named X, Y, name, and Z

```
Void  MySub()
{
    char  X = 'X';
    int   Y = 10;
    char  name[20];
    name[0] = 'B';
    double Z = 1.2;
}
```

Creating Local Variables: Example (Cont.)

- The forgoing C++ code can be easily implemented in assembly language
  - Each stack entry defaults to 32 bits
    - So each variable's storage size in bytes is rounded upward to a multiple of 4
  - Thus, a total of 36 bytes are reserved for local variables

Variable	Bytes	Stack Offset
X	4	EBP-4
Y	4	EBP-8
name	20	EBP-28
Z	8	EBP-36

Creating Local Variables: Example (Cont.)

```
MySub  PROC
    push  ebp
    mov   ebp, esp
    sub   esp, 36                    ; create variables

    mov   BYTE PTR [ebp-4], 'X'      ; X
    mov   DWORD PTR [ebp-8], 10      ; Y
    mov   BYTE PTR [ebp-20], 'Y'     ; name[0]
    mov   DWORD PTR [ebp-32], 3ff33333h    ;Z (high)
    mov   DWORD PTR [ebp-36], 33333333h    ;Z (low)

    mov   esp, ebp                   ; destroy variables
    pop   ebp
    ret
MySub ENDP
```