Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Lecture 26, 27: Advanced Procedures

Local Variables

Stack Parameters and Frames

Recursion and Creating Multimodule Programs

Slides prepared by Kip R. Irvine, Revision date: 07/11/2002

Modified by Dr. Nikolay Metodiev Sirakov on April 30.2005

- Chapter corrections (Web) Assembly language sources (Web)

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)

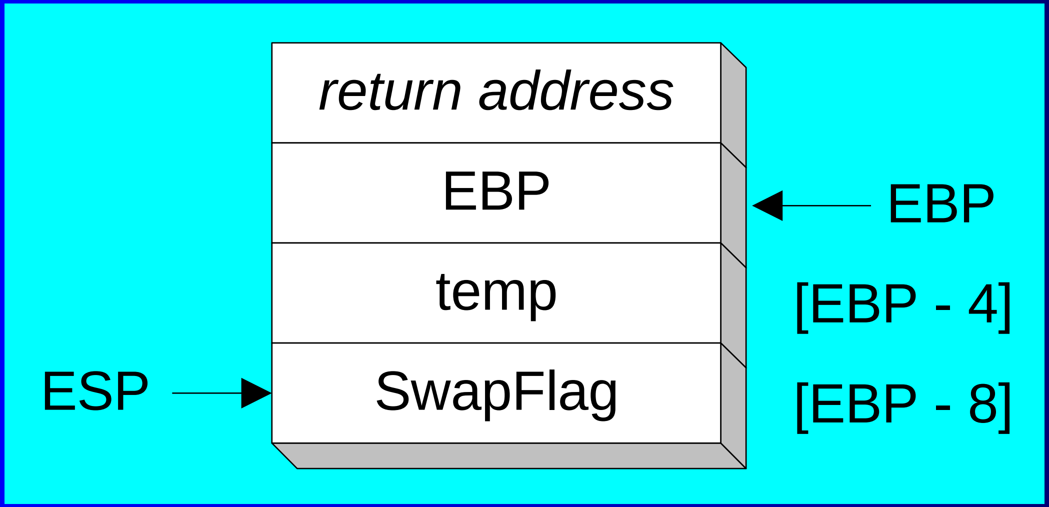
```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

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:



The ADD instruction adds -8 to ESP, moving it downward and Creating an opening in the stack between ESP and EBP for the two local variables.

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

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*]

- *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]
```

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 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.

•C++ programs typically return 32-bit integers from functions in EAX.

PROC Examples

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

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

- 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
```

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
```

Trouble-Shooting Tips

- Save and restore registers when they are modified by a procedure.
  - Except a register that returns a function result
- 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.

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, 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**.

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

Language Specifiers

- C:
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack
- pascal
  - procedure arguments pushed in forward order (left to right)
  - called procedure cleans up the stack
- stdcall
  - procedure arguments pushed on stack in reverse order (right to left)
  - called procedure cleans up the stack

Explicit Access to Stack Parameters

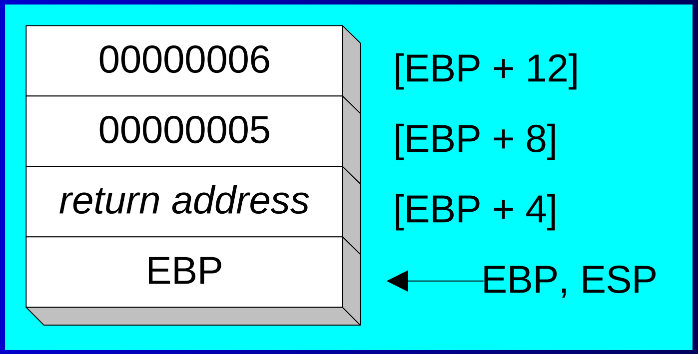
- A procedure can explicitly access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
- EBP is often called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the procedure.
- EBP must be restored to its original value when a procedure returns.

[1] BP in Real-address mode

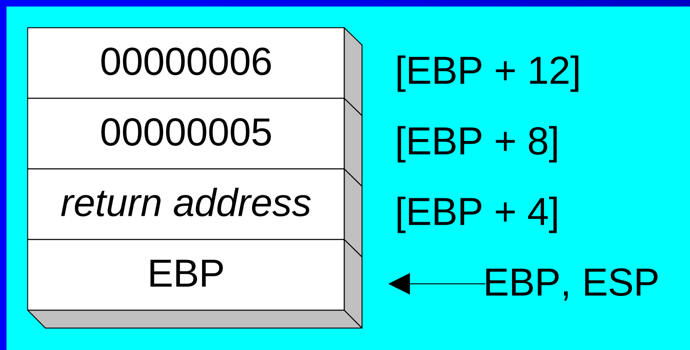
Stack Frame Example (1 of 2)

```
.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
```

Passing Arguments by Reference (1 of 2)

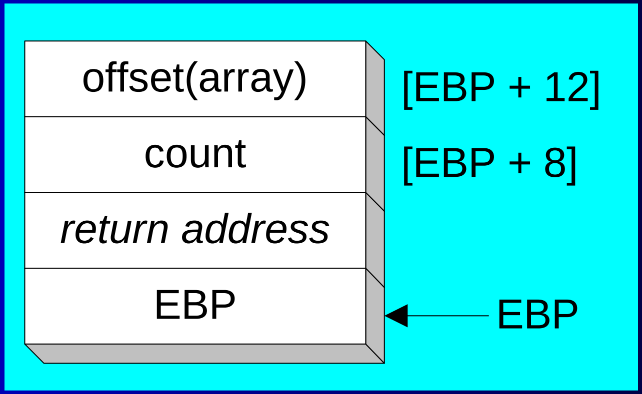
- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
    push OFFSET array
    push COUNT
    call ArrayFill
```

Passing Arguments by Reference (2 of 2)

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

```
ArrayFill PROC
    push ebp
    mov  ebp,esp
    pushad
    mov  esi,[ebp+12]
    mov  ecx,[ebp+8]
    .
    .
```



ESI points to the beginning of the array, so it's easy to use a loop to access each array element. View the complete program.

LEA Instruction

- The LEA instruction returns offsets of both direct and indirect operands.
  - OFFSET operator can only return constant offsets.
- LEA is required when obtaining the offset of a stack parameter or local variable. For example:

```
CopyString PROC,
    count:DWORD
    LOCAL temp[20]:BYTE

    mov edi,OFFSET count        ; invalid operand
    mov esi,OFFSET temp         ; invalid operand
    lea edi,count               ; ok
    lea esi,temp                ; ok
```

Creating Local Variables

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

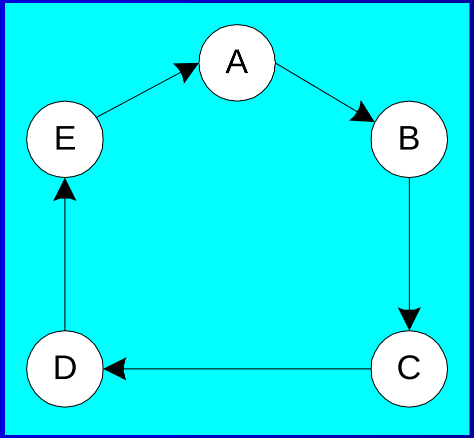
```
MySub PROC
    push ebp
    mov  ebp,esp
    sub  esp,8
    mov  [ebp-4],123456h              ; locA
    mov  [ebp-8],0                    ; locB
    .
    .
```

Recursion

- What is recursion?
- Recursively Calculating a Sum
- Calculating a Factorial

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:



Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

```
CalcSum PROC
    cmp ecx,0             ; check counter value
    jz L2                 ; quit if zero
    add eax,ecx           ; otherwise, add to sum
    dec ecx               ; decrement counter
    call CalcSum          ; recursive call
L2: ret
CalcSum ENDP
```

Stack frame:

Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

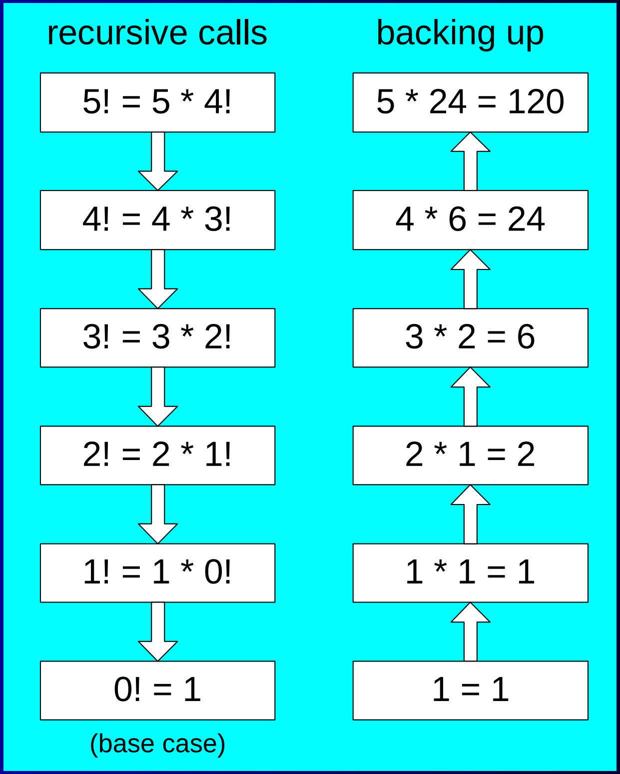
View the complete program

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:

```
int function 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.



Calculating a Factorial (2 of 3)

```
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
```

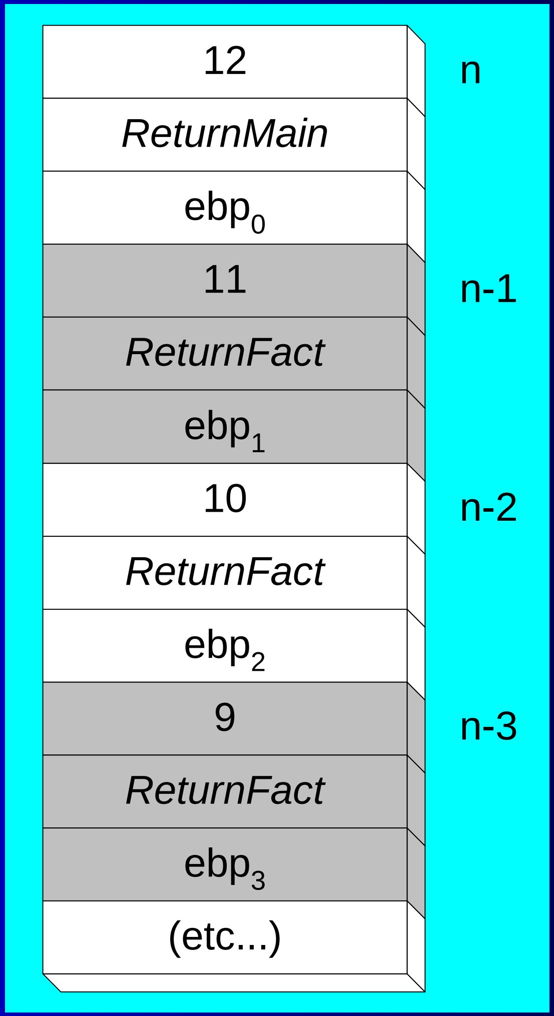
See the program listing

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.



Multimodule Programs

- A multimodule program is a program whose source code has been divided up into separate ASM files.
- Each ASM file (module) is assembled into a separate OBJ file.
- All OBJ files belonging to the same program are linked using the link utility into a single EXE file.
  - This process is called static linking

Advantages

- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
- When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
- A module can be a container for logically related code and data (think object-oriented here...)
  - encapsulation: procedures and variables are automatically hidden in a module unless you declare them public

Creating a Multimodule Program

- Here are some basic steps to follow when creating a multimodule program:
  - Create the main module
  - Create a separate source code module for each procedure or set of related procedures
  - Create an include file that contains procedure prototypes for external procedures (ones that are called between modules)
  - Use the INCLUDE directive to make your procedure prototypes available to each module

INCLUDE File

The sum.inc file contains prototypes for external functions that are not in the Irvine32 library:

```
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    ptrArray:PTR DWORD,          ; points to the array
    arraySize:DWORD              ; size of the array

ArraySum PROTO,
    ptrArray:PTR DWORD,          ; points to the array
    count:DWORD                  ; size of the array

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    theSum:DWORD                 ; sum of the array
```