LOCAL Directive

The LOCAL directive in assembly language is used to declare local variables.

It can be used to declare named local variables of any type, including standard types such as BYTE, DWORD, and PTR WORD, as well as user-defined types such as structures.

The LOCAL directive must be placed on the line immediately following the PROC directive. Its syntax is as follows:

```
LOCAL varlist
```

Where varlist is a list of variable definitions, separated by commas. Each variable definition takes the following form:

label:type where label is the name of the local variable and type is the type of the local variable.

For example, the following assembly language code declares a local variable named var1 of type BYTE:

```
MySub PROC
LOCAL var1:BYTE
```

The following assembly language code declares a doubleword local variable named temp of type DWORD and a variable named SwapFlag of type BYTE:

```
BubbleSort PROC
LOCAL temp:DWORD, SwapFlag:BYTE
```

The following assembly language code declares a PTR WORD local variable named pArray, which is a pointer to a 16-bit integer:

```
Merge PROC
LOCAL pArray:PTR WORD
```

The following assembly language code declares a local variable named TempArray which is an array of 10 doublewords:

```
LOCAL TempArray[10]:DWORD
```

The LOCAL directive reserves stack space for the local variables that it declares. The amount of stack space reserved depends on the type and size of each local variable.

For example, a BYTE variable requires 1 byte of stack space, while a DWORD variable requires 4 bytes of stack space.

Local variables are accessible within the procedure in which they are declared. They are not accessible to other procedures.

It is important to note that the LOCAL directive is not equivalent to the ENTER instruction. The ENTER instruction creates a stack frame for a called procedure, while the LOCAL directive simply declares local variables.

The ENTER instruction must be used in conjunction with the LEAVE instruction to destroy the stack frame.

The LOCAL directive is a convenient and easy-to-use way to declare local variables in assembly language.

It is a good idea to use the LOCAL directive for all local variables, even if they are only used in a single procedure. This will make your code more readable and maintainable.

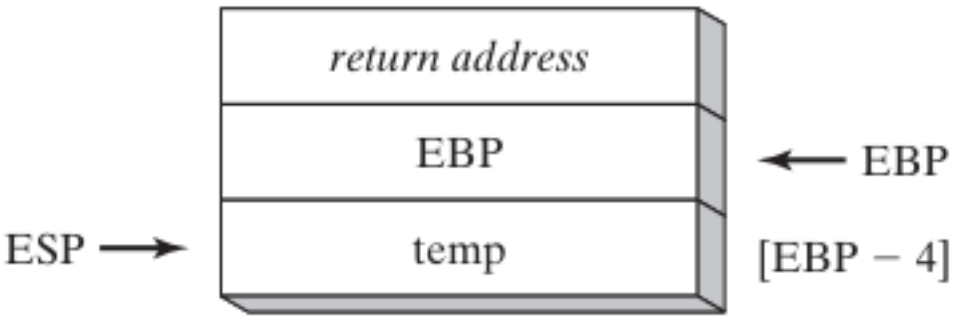
The following is a more in-depth explanation of the MASM code generation for the LOCAL directive:

```
485 Example1 PROC
486     LOCAL temp:DWORD
487     mov
488     eax,temp
489     ret
490 Example1 ENDP
```

MASM generated code:

```
499 push
500 ebp
501 mov
502 ebp,esp
503 add
504 esp,0FFFFFFFCh
505 ; add -4 to ESP
506 mov
507 eax,[ebp-4]
508 leave
509 ret
```

Stack frame diagram:



The MASM code generator works as follows:

It pushes the value of the EBP register onto the stack.

This saves the current value of the base pointer register. It sets the EBP register to the address of the current stack frame.

This makes the base pointer register point to the top of the new stack frame.

It subtracts 4 bytes from the ESP register.

This reserves 4 bytes of stack space for the local variable temp.

It loads the value of the local variable temp into the EAX register.

It calls the leave instruction to destroy the stack frame and restore the ESP register to its value before the enter instruction was executed.

It returns from the procedure.

Image explanation:

The image shows the stack before and after the Example1 procedure has executed.

The Example1 procedure has reserved 4 bytes of stack space for the local variable temp. The local variable temp is now located at the address ebp-4.

The esp register points to the top of the stack.

The ebp register points to the base of the stack frame. The base of the stack frame is the address of the first local variable.

The example1 procedure has no parameters.

Therefore, the ebp register points to the same address before and after the procedure has executed.

Microsoft x64 calling convention

The **Microsoft x64 calling convention** is a set of rules that govern how parameters are passed to and from functions in 64-bit Windows programs. It is used by C and C++ compilers, as well as by the Windows API library.

Here is a summary of the key points of the Microsoft x64 calling

convention:

The first four parameters to a function are passed in registers: RCX, RDX, R8, and R9. Additional parameters are pushed onto the stack in left-to-right order.

Parameters less than 64 bits long are not zero extended, so the upper bits have indeterminate values. The return value from a function is returned in the RAX register, if it is an integer whose size is less than or equal to 64 bits.

Otherwise, the return value is placed on the stack and RCX points to its location. The caller is responsible for allocating at least 32 bytes of shadow space on the runtime stack, so called functions can optionally save the register parameters in this area.

The stack pointer (RSP) must be aligned on a 16-byte boundary when calling a function.

The caller is responsible for removing all parameters and shadow space from the runtime stack after the called function has finished.

Here are some additional details about the Microsoft x64 calling convention:

The CALL instruction subtracts 8 from the RSP register, since addresses are 64 bits long.

The RAX, RCX, RDX, R8, R9, R10, and R11 registers are often altered by functions, so if the calling program wants them preserved, it must push them onto the stack before the function call and pop them off the stack afterwards.

The values of the RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers must be preserved by functions.

The Microsoft x64 calling convention is a complex topic, but it is important to understand it if you are writing 64-bit Windows programs.

Questions

1. (True/False): A subroutine's stack frame always contains the caller's return address and the subroutine's local variables.

Answer: True

Explanation: A subroutine's stack frame is a region of memory on the stack that is used to store information about the subroutine, such as its local variables and the caller's return address. The caller's return address is the address of the instruction in the calling function that will be executed after the subroutine returns.

2. (True/False): Arrays are passed by reference to avoid copying them onto the stack.

Answer: True

Explanation: Arrays are typically passed by reference to functions to avoid copying them onto the stack. This is because arrays can be very large, and copying them onto the stack would be inefficient.

3. (True/False): A subroutine's prologue code always pushes EBP on the stack.

Answer: True

Explanation: The prologue code for a subroutine is the code that is executed at the beginning of the subroutine. The prologue code typically saves the value of the EBP register on the stack. The EBP register is used to point to the base of the current stack frame.

4. (True/False): Local variables are created by adding a positive value to the stack pointer.

Answer: True

Explanation: Local variables are created by adding a positive value to the stack pointer. This value is the size of the local variable in bytes.

5. (True/False): In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 8.

Answer: False

Explanation: In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 4.

6. (True/False): Passing by reference means that an argument's address is stored on the runtime stack.

Answer: True

Explanation: Passing by reference means that an argument's address is stored on the runtime stack. This means that the function can directly access the argument in the calling function.

7. What are the two common types of stack parameters?

Answer: The two common types of stack parameters are:

Value parameters: Value parameters are copied onto the stack when the function is called. When the function returns, the changes made to the parameter on the stack are not reflected in the calling function.

Reference parameters: Reference parameters are passed by address. When a function is called with a reference parameter, the stack contains the address of the parameter in the calling function. Changes made to the parameter in the called function are also reflected in the calling function. Example:

```
void swap_values(int a, int b) {
  // a and b are value parameters

  int temp = a;
  a = b;
  b = temp;
}

void swap_references(int* a, int* b) {
  // a and b are reference parameters

  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  int x = 10;
  int y = 20;

  // Call swap_values() with the value parameters x and y
  swap_values(x, y);

  // x and y will still be 10 and 20, respectively

  // Call swap_references() with the reference parameters &x and &y
  swap_references(&x, &y);

  // x and y will now be 20 and 10, respectively

  return 0;
}
```

Explanation:

In the above example, the swap_values() function takes two value parameters, a and b. When the function is called, the values of a and b are copied onto the stack. The function then swaps the values of the two parameters on the stack. When the function returns, the values of a and b in the calling function are not affected.

The swap_references() function takes two reference parameters, a and b. When the function is called, the stack contains the addresses of the a and b parameters in the calling function. The function then swaps the values of the two parameters in the calling function. When the function returns, the values of a and b in the calling function are affected.