Intel x86 Function-call Conventions - Assembly View

One of the "big picture" issues in looking at compiled C code is the function-calling conventions. These are the methods that a calling function and a called function agree on how parameters and return values should be passed between them, and how the stack is used by the function itself. The layout of the stack constitutes the "stack frame", and knowing how this works can go a long way to decoding how something works.

In C and modern CPU design conventions, the stack frame is a chunk of memory, allocated from the stack, at run-time, each time a function is called, to store its automatic variables. Hence nested or recursive calls to the same function, each successively obtain their own separate frames.

Physically, a function's stack frame is the area between the addresses contained in esp, the stack pointer, and ebp, the frame pointer (base pointer in Intel terminology). Thus, if a function pushes more values onto the stack, it is effectively growing its frame.

This is a very low-level view: the picture as seen from the C/C++ programmer is illustrated elsewhere:

• Unixwiz.net Tech Tip: Intel x86 Function-call Conventions - C Programmer's View

For the sake of discussion, we're using the terms that the Microsoft Visual C compiler uses to describe these conventions, even though other platforms may use other terms.

__cdecl (pronounced *see-DECK-'ll* rhymes with "heckle")

This convention is the most common because it supports semantics required by the C language. The C language supports variadic functions (variable argument lists, à la **printf**), and this means that the *caller* must clean up the stack after the function call: the called function has no way to know how to do this. It's not terribly optimal, but the C language semantics demand it.

__stdcall

Also known as **__pascal**, this requires that each function take a fixed number of parameters, and this means that the *called* function can do argument cleanup in one place rather than have this be scattered throughout the program in every place that calls it. The Win32 API primarily uses __stdcall.

It's important to note that these are merely *conventions*, and any collection of cooperating code can agree on nearly anything. There are other conventions (passing parameters in registers, for instance) that behave differently, and of course the optimizer can make mincemeat of any clear picture as well.

Our focus here is to provide an overview, and not an authoritative definition for these conventions.

Register use in the stack frame

In both **__cdecl** and **__stdcall** conventions, the same set of three registers is involved in the function-call frame:

%ESP - Stack Pointer

This 32-bit register is implicitly manipulated by several CPU instructions (PUSH, POP, CALL, and RET among others), it always points to the last element **used** on the stack (not the first *free* element): this means that the PUSH and POP operations would be specified in pseudo-C as:

```
*--ESP = value; // push

value = *ESP++; // pop
```

The "Top of the stack" is an occupied location, not a free one, and is at the **lowest** memory address.

%EBP - Base Pointer

This 32-bit register is used to reference all the function parameters and local variables in the current stack frame. Unlike the %esp register, the base pointer is manipulated only *explicitly*. This is sometimes called the "Frame Pointer".

%EIP - Instruction Pointer

This holds the address of the next CPU instruction to be executed, and it's saved onto the stack as part of the **CALL** instruction. As well, any of the "jump" instructions modify the %EIP directly.

Assembler notation

Virtually everybody in the Intel assembler world uses the Intel notation, but the GNU C compiler uses what they call the "AT&T syntax" for backwards compatibility. This seems to us to be a really dumb idea, but it's a fact of life.

There are minor notational differences between the two notations, but by far the most annoying is that the AT&T syntax *reverses* the source and destination operands. To move the immediate value 4 into the EAX register:

```
mov $4, %eax          // AT&T notation
mov eax, 4            // Intel notation
```

More recent GNU compilers have a way to generate the Intel syntax, but it's not clear if the GNU assembler takes it. In any case, we'll use the Intel notation exclusively.

There are other minor differences that are not of much concern to the reverse engineer.

Calling a __cdecl function

The best way to understand the stack organization is to see each step in calling a function with the __cdecl conventions. These steps are taken automatically by the compiler, and though not all of them are used in every case (sometimes no parameters, sometimes no local variables, sometimes no saved registers), but this shows the overall mechanism employed.

- **Push parameters onto the stack, from right to left**

  Parameters are pushed onto the stack, one at a time, from right to left. Whether the parameters are *evaluated* from right to left is a different matter, and in any case this is unspecified by the language and code should ***never*** rely on this. The calling code must keep track of how many bytes of parameters have been pushed onto the stack so it can clean it up later.

- **Call the function**

  Here, the processor pushes contents of the %EIP (instruction pointer) onto the stack, and it points to the first byte *after* the CALL instruction. After this finishes, the caller has lost control, and the callee is in charge. This step does not change the %ebp register.

- **Save and update the %ebp**

  Now that we're in the new function, we need a new local stack frame pointed to by %ebp, so this is done by saving the current %ebp (which belongs to the previous function's frame) and making it point to the top of the stack.

  ```
  push ebp
  mov  ebp, esp    // ebp « esp
  ```

  Once %ebp has been changed, it can now refer directly to the function's arguments as **8(%ebp)**, **12(%ebp)**.

  Note that **0(%ebp)** is the old base pointer and **4(%ebp)** is the old instruction pointer, but this applies to **near** calls only - **far** calls include segment registers too, but these are uncommon in real programs.

- **Save CPU registers used for temporaries**

  If this function will use any CPU registers, it has to save the old values first lest it walk on data used by the calling functions. Each register to be used is pushed onto the stack one at a time, and the compiler must remember what it did so it can unwind it later.

- **Allocate local variables**

  The function may choose to use local stack-based variables, and they are allocated here simply by decrementing the stack pointer by the amount of space required. This is *always* done in four-byte chunks.

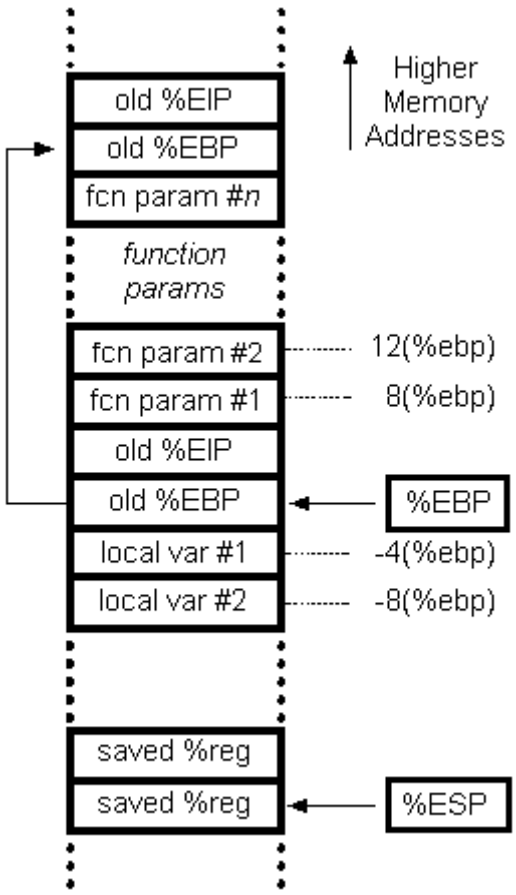
  Now, the local variables are located on the stack between the %ebp and %esp registers, and though it would be possible to refer to them as offsets from either one, by convention the %ebp register is used. This means that **-4(%ebp)** refers to the first local variable.

- **Perform the function's purpose**

  At this point, the stack frame is set up correctly, and this is represented by the diagram to the right. All the parameters and locals are offsets from the **%ebp** register:

16(%ebp)	- third function parameter
12(%ebp)	- second function parameter
8(%ebp)	- first function parameter
4(%ebp)	- old %EIP (the function's "return address")
0(%ebp)	- old %EBP (previous function's base pointer)

-4(%ebp)	- first local variable
-8(%ebp)	- second local variable
-12(%ebp)	- third local variable



The function is free to use any of the registers that had been saved onto the stack upon entry, but it *must not* change the stack pointer or all Hell will break loose upon function return.

- **Release local storage**

  When the function allocates local, temporary space, it does so by decrementing from the stack point by the amount space needed, and this process must be reversed to reclaim that space. It's usually done by *adding* to the stack pointer the same amount which was subtracted previously, though a series of **POP** instructions could achieve the same thing.

- **Restore saved registers**

  For each register saved onto the stack upon entry, it must be restored from the stack in reverse order. If the "save" and "restore" phases don't match exactly, catastrophic stack corruption *will* occur.

- **Restore the old base pointer**

  The first thing this function did upon entry was save the caller's **%ebp** base pointer, and by restoring it now (popping the top item from the stack), we effectively discard the entire local stack frame and put the caller's frame back in play.

- **Return from the function**

  This is the last step of the called function, and the **RET** instruction pops the old %EIP from the stack and jumps to that location. This gives control back to the calling function. Only the stack pointer and instruction pointers are modified by a subroutine return.

- **Clean up pushed parameters**

  In the __cdecl convention, the *caller* must clean up the parameters pushed onto the stack, and this is done either by popping the stack into don't-care registers (for a few parameters) or by adding the parameter-block size to the stack pointer directly.

__cdecl -vs- __stdcall

The **__stdcall** convention is mainly used by the Windows API, and it's a bit more compact than **__cdecl**. The main difference is that any given function has a hard-coded set of parameters, and this cannot vary from call to call like it can in C (no "variadic functions").

Because the size of the parameter block is fixed, the burden of cleaning these parameters off the stack can be shifted to the *called* function, instead of being done by the *calling* function as in __cdecl. There are several effects of this:

1. the code is a tiny bit smaller, because the parameter-cleanup code is found once — in the called function itself — rather than in every place the function is called. These may be only a few bytes per call, but for commonly-used functions it can add up. This presumably means that the code may be a tiny bit faster as well.
2. calling the function with the wrong number of parameters is *catastrophic* - the stack will be badly misaligned, and general havoc will surely ensue.
3. As an offshoot of #2, Microsoft Visual C takes special care of functions that are B{__stdcall}. Since the number of parameters is known at compile time, the compiler encodes the parameter byte count *in the symbol name itself*, and this means that calling the function wrong leads to a link error.

   For instance, the function **int foo(int a, int b)** would generate — at the assembler level — the symbol "**_foo@8**", where "8" is the number of bytes expected. This means that not only will a call with 1 or 3 parameters not resolve (due to the size mismatch), but neither will a call expecting the __cdecl parameters (which looks for **_foo**). It's a clever mechanism that avoids a lot of problems.

Variations and Notes

The x86 architecture provides a number of built-in mechanisms for assisting with frame management, but they don't seem to be commonly used by C compilers. Of particular interest is the **ENTER** instruction, which handles most of the function-prolog code.

```
ENTER 10,0          PUSH ebp
                     MOV  ebp, esp
                     SUB  esp, 10
```

We're pretty sure these are functionally equivalent, but our 80386 processor reference suggests that the **ENTER** version is more compact (6 bytes -vs- 9) but slower (15 clocks -vs- 6). The newer processors are probably harder to pin down, but somebody has probably figured out that **ENTER** is slower. Sigh.
