WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

x86 calling conventions

This article describes the **calling conventions** used when programming **x86** architecture microprocessors.

Calling conventions describe the interface of called code:

- The order in which atomic (scalar) parameters, or individual parts of a complex parameter, are allocated
- How parameters are passed (pushed on the stack, placed in registers, or a mix of both)
- Which registers the called function must preserve for the caller (also known as: callee-saved registers or non-volatile registers)
- How the task of preparing the stack for, and restoring after, a function call is divided between the caller and the callee

This is intimately related with the assignment of sizes and formats to programming-language types. Another closely related topic is name mangling, which determines how symbol names in the code are mapped to symbol names used by the linker. Calling conventions, type representations, and name mangling are all part of what is known as an application binary interface (ABI).

There are subtle differences in how various compilers implement these conventions, so it is often difficult to interface code which is compiled by different compilers. On the other hand, conventions which are used as an API standard (such as stdcall) are very uniformly implemented.

Historical background

Prior to microcomputers, the machine manufacturer generally provided an operating system and compilers for several programming languages. The calling convention(s) for each platform were those defined by the manufacturer's programming tools.

Early microcomputers before the Commodore Pet and Apple II generally came without an OS or compilers. The IBM PC came with Microsoft's fore-runner to Windows, the Disk Operating System (DOS), but it did not come with a compiler. The only hardware standard for IBM PC-compatible machines was defined by the Intel processors (8086, 80386) and the literal hardware IBM shipped. Hardware extensions and all software standards (save for a BIOS calling convention) were thrown open to market competition.

A multitude of independent software firms offered operating systems, compilers for many programming languages, and applications. Many different calling schemes were implemented by the firms, often mutually exclusive, based on different requirements, historical practices, and programmer creativity.

After the IBM-compatible market shakeout, Microsoft operating systems and programming tools (with differing conventions) predominated, while second-tier firms like Borland and Novell, and open-source projects like GCC, still maintained their own standards. Provisions for interoperability between vendors and products were eventually adopted, simplifying the problem of choosing a viable convention.[1]

Caller clean-up

In these types of calling conventions, the caller cleans the arguments from the stack (resets the state of the stack just as it was before the callee function was called).

cdecl

The **cdecl** (which stands for **C declaration**) is a calling convention for the C programming language and is used by many C compilers for the x86 architecture.[1] In cdecl, subroutine arguments are passed on the stack. If the return values are Integer values or memory addresses they are put into the EAX register by the callee, whereas floating point values are put in the ST0 x87 register. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. The x87 floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function. ST0 must also be empty when not used for returning a value.

In the context of the C programming language, function arguments are pushed on the stack in the right-to-left order, i.e. the last argument is pushed first.

Consider the following C source code snippet:

```
int callee(int, int, int);

int caller(void)
{
    return callee(1, 2, 3) + 5;
}
```

On x86, it might produce the following assembly code (Intel syntax):

```
caller:
    ; make new call frame
    ; (some compilers may produce an 'enter' instruction instead)
    push    ebp       ; save old call frame
    mov     ebp, esp  ; initialize new call frame
    ; push call arguments, in reverse
    ; (some compilers may subtract the required space from the stack pointer,
    ; then write each argument directly, see below.
    ; The 'enter' instruction can also do something similar)
    ; sub esp, 12      : 'enter' instruction could do this for us
    ; mov [ebp-4], 3   : or mov [esp+8], 3
    ; mov [ebp-8], 2   : or mov [esp+4], 2
    ; mov [ebp-12], 1  : or mov [esp], 1
    push    3
    push    2
    push    1
    call    callee    ; call subroutine 'callee'
    add     esp, 12   ; remove call arguments from frame
    add     eax, 5    ; modify subroutine result
                      ; (eax is the return value of our callee,
                      ; so we don't have to move it into a local variable)
    ; restore old call frame
    ; (some compilers may produce a 'leave' instruction instead)
    mov     esp, ebp  ; most calling conventions dictate ebp be callee-saved,
                      ; i.e. it's preserved after calling the callee.
                      ; it therefore still points to the start of our stack frame.
                      ; we do need to make sure
                      ; callee doesn't modify (or restore) ebp, though,
                    ; so we need to make sure
                    ; it uses a calling convention which does this
    pop     ebp         ; restore old call frame
    ret                 ; return
```

The caller cleans the stack after the function call returns.

The **cdecl** calling convention is usually the default calling convention for x86 C compilers, although many compilers provide options to automatically change the calling conventions used. To manually define a function to be cdecl, some support the following syntax:

```
return_type __cdecl func_name();
```

Variations

There are some variations in the interpretation of cdecl. As a result, x86 programs compiled for different operating system platforms and/or by different compilers can be incompatible, even if they both use the "cdecl" convention and do not call out to the underlying environment.

In regard to how to return values, some compilers return simple data structures with a length of 2 registers or less in the register pair EAX:EDX, and larger structures and class objects requiring special treatment by the exception handler (e.g., a defined constructor, destructor, or assignment) are returned in memory. To pass "in memory", the caller allocates memory and passes a pointer to it as a hidden first parameter; the callee populates the memory and returns the pointer, popping the hidden pointer when returning.[2]

In Linux, GCC sets the *de facto* standard for calling conventions. Since GCC version 4.5, the stack must be aligned to a 16-byte boundary when calling a function (previous versions only required a 4-byte alignment).[1][3]

A version of **cdecl** is described in System V ABI for i386 systems.[4]

syscall

This is similar to cdecl in that arguments are pushed right-to-left. EAX, ECX, and EDX are not preserved. The size of the parameter list in doublewords is passed in AL.

Syscall is the standard calling convention for 32 bit OS/2 API.

optlink

Arguments are pushed right-to-left. The three first (leftmost) arguments are passed in EAX, EDX, and ECX and up to four floating-point arguments are passed in ST0 through ST3, although space for them is reserved in the argument list on the stack. Results are returned in EAX or ST0. Registers EBP, EBX, ESI, and EDI are preserved.

Optlink is used by the IBM VisualAge compilers.

Callee clean-up

In these conventions, the callee cleans up the arguments from the stack. Functions which use these conventions are easy to recognize in ASM code because they will unwind the stack after returning. The x86 ret instruction allows an optional 16-bit parameter that specifies the number of stack bytes to release after returning to the caller. Such code looks like this:

```
ret 12
```

Conventions entitled **fastcall** or **register** have not been standardized, and have been implemented differently, depending on the compiler vendor.[1] Typically register based calling conventions pass one or more arguments in registers which reduces the number of memory accesses required for the call and thus make them usually faster.

pascal

Based on the Borland Pascal programming language's calling convention, the parameters are pushed on the stack in left-to-right order (opposite of cdecl), and the callee is responsible for removing them from the stack.

Returning the result works as follows:

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), EAX (32-bit values), or DX:AX (32-bit values on 16-bit systems).
- Real values are returned in DX:BX:AX.
- Floating point (8087) values are returned in ST0.
- Pointers are returned in EAX on 32-bit systems and in AX in 16-bit systems.
- Strings are returned in a temporary location pointed by the @Result symbol.

This calling convention was common in the following 16-bit APIs: OS/2 1.x, Microsoft Windows 3.x, and Borland Delphi version 1.x. Modern versions of the Windows API use stdcall, which still has the callee restoring the stack as in the Pascal convention, but the parameters are now pushed right to left.

stdcall

The stdcall[5] calling convention is a variation on the Pascal calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the _cdecl calling convention. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX register.

stdcall is the standard calling convention for the Microsoft Win32 API and for Open Watcom C++.

Microsoft fastcall

Microsoft __fastcall convention (aka __msfastcall) passes the first two arguments (evaluated left to right) that fit, into ECX and EDX.[6] Remaining arguments are pushed onto the stack from right to left. When the compiler compiles for IA64 or AMD64, it ignores the __fastcall keyword (or any other calling convention keyword aside from __vectorcall) and uses the default 64-bit calling convention instead.

Other compilers like GCC,[7] Clang,[8] and ICC provide similar "fastcall" calling conventions, although they are not necessarily compatible with each other or with Microsoft fastcall.[9]

Consider the following C snippet:

```
__attribute__((fastcall)) void printnums(int num1, int num2, int num3){
    printf("The numbers you sent are: %d %d %d", num1, num2, num3);
}

int main(){
    printnums(1, 2, 3);
    return 0;
}
```

x86 decompilation of the main function will look like (in Intel syntax):

```
main:
    ; stack setup
    push ebp
    mov ebp, esp
    push 3 ; immediate 3 (third argument is pushed to the stack)
    mov edx, 0x2 ; immediate 2 (second argument) is copied to edx register.
    mov ecx, 0x1 ; immediate 1 (first argument) is copied to ecx register.
    call printnums
    mov eax, 0 ; return 0
    leave
    retn
```

The first two arguments are passed in the left to right order, and the third argument is pushed on the stack. There is no stack cleanup, as stack cleanup is performed by the callee. The disassembly of the callee function is:

```
printnums:
    ; stack setup
    push ebp
    mov ebp, esp
    sub esp, 0x08
    mov [ebp-0x04], ecx    ; in x86, ecx = first argument.
    mov [ebp-0x08], edx    ; arg2
    push [ebp+0x08]        ; arg3 is pushed to stack.
    push [ebp-0x08]        ; arg2 is pushed
    push [ebp-0x04]        ; arg1 is pushed
    push 0x8065d67         ; "The numbers you sent are %d %d %d"
    call printf
    ; stack cleanup
    add esp, 0x10
    nop
    leave
    retn 0x04
```

As the two arguments were passed through the registers and only one parameter was pushed in the stack, the pushed value is being cleared by the retn instruction, as int is 4 bytes in size in x86 systems.

Microsoft vectorcall

In Visual Studio 2013, Microsoft introduced the `__vectorcall` calling convention in response to efficiency concerns from game, graphic, video/audio, and codec developers. The scheme allows for larger vector types (`float`, `double`, `__m128`, `__m256`) to be passed in registers as opposed to on the stack.[10]

For IA-32 and x64 code, __vectorcall is similar to __fastcall and the original x64 calling conventions respectively, but extends them to support passing vector arguments using SIMD registers. In IA-32, the integer values are passed as usual, and the first six SIMD (XMM/YMM0-5) registers hold up to six floating-point, vector, or HVA values sequentially from left to right, regardless of actual positions caused by, e.g. an int argument appearing between them. In x64, however, the rule from the original x64 convention still apply, so that XMM/YMM0-5 only hold floating-point, vector, or HVA arguments when they happen to be the first through the sixth.[11]

__vectorcall adds support for passing homogeneous vector aggregate (HVA) values, which are composite types (structs) consisting solely of up to four identical vector types, using the same six registers. Once the registers have been allocated for vector type arguments, the unused registers are allocated to HVA arguments from left to right. The positioning rules still apply. Resulting vector type and HVA values are returned using the first four XMM/YMM registers.[11]

The Clang compiler and the Intel C++ Compiler also implement vectorcall.[12] ICC has a similar, earlier convention called __regcall;[13] it is also supported by Clang.[14]

Borland register

Evaluating arguments from left to right, it passes three arguments via EAX, EDX, ECX. Remaining arguments are pushed onto the stack, also left to right.[15] It is the default calling convention of the 32-bit compiler of Delphi, where it is known as *register*. This calling convention is also used by Embarcadero's C++Builder, where it is called *__fastcall*.[16] In this compiler, Microsoft's *fastcall* can be used as *__msfastcall*.[17]

GCC and Clang can be made to use a similar calling convention by using __stdcall with the regparm function attribute or the -mregparm=3 switch. (The stack order is inverted.) It is also possible to produce a caller clean-up variant using cdecl or extend this to also use SSE registers.[18] A cdecl-based version is used by the Linux kernel on i386 since version 2.6.20 (released February 2007).[19]

Watcom register

Watcom does not support the *__fastcall* keyword except to alias it to null. The register calling convention may be selected by command line switch. (However, IDA uses *__fastcall* anyway for uniformity.)

Up to 4 registers are assigned to arguments in the order EAX, EDX, EBX, ECX. Arguments are assigned to registers from left to right. If any argument cannot be assigned to a register (say it is too large) it, and all subsequent arguments, are assigned to the stack. Arguments assigned to the stack are pushed from right to left. Names are mangled by adding a suffixed underscore.

Variadic functions fall back to the Watcom stack based calling convention.

The Watcom C/C++ compiler also uses the #pragma aux[20] directive that allows the user to specify their own calling convention. As its manual states, "Very few users are likely to need this method, but if it is needed, it can be a lifesaver".

TopSpeed / Clarion / JPI

The first four integer parameters are passed in registers eax, ebx, ecx and edx. Floating point parameters are passed on the floating point stack – registers st0, st1, st2, st3, st4, st5 and st6. Structure parameters are always passed on the stack. Additional parameters are passed on the stack after registers are exhausted. Integer values are returned in eax, pointers in edx and floating point types in st0.

safecall

In Delphi and Free Pascal on Microsoft Windows, the safecall calling convention encapsulates COM (Component Object Model) error handling, thus exceptions aren't leaked out to the caller, but are reported in the HRESULT return value, as required by COM/OLE. When calling a safecall function from Delphi code, Delphi also automatically checks the returned HRESULT and raises an exception if necessary.

The safecall calling convention is the same as the stdcall calling convention, except that exceptions are passed back to the caller in EAX as a HResult (instead of in FS:[0]), while the function result is passed by reference on the stack as though it were a final "out" parameter. When calling a Delphi function from Delphi this calling convention will appear just like any other calling convention, because although exceptions are passed back in EAX, they are automatically converted back to proper exceptions by the caller. When using COM objects created in other languages, the HResults will be automatically raised as exceptions, and the result for Get functions is in the result rather than a parameter. When creating COM objects in Delphi with safecall, there is no need to worry about HResults, as exceptions can be raised as normal but will be seen as HResults in other languages.

```
function function_name(a: DWORD): DWORD; safecall;
```

Returns a result and raises exceptions like a normal Delphi function, but it passes values and exceptions as though it was:

```
function function_name(a: DWORD; out Result: DWORD): HResult; stdcall;
```

Either caller or callee clean-up

thiscall

This calling convention is used for calling C++ non-static member functions. There are two primary versions of **thiscall** used depending on the compiler and whether or not the function uses a variable number of arguments.

For the GCC compiler, **thiscall** is almost identical to **cdecl**: The caller cleans the stack, and the parameters are passed in right-to-left order. The difference is the addition of the **this** pointer, which is pushed onto the stack last, as if it were the first parameter in the function prototype.

On the Microsoft Visual C++ compiler, the **this** pointer is passed in ECX and it is the *callee* that cleans the stack, mirroring the **stdcall** convention used in C for this compiler and in Windows API functions. When functions use a variable number of arguments, it is the caller that cleans the stack (cf. **cdecl**).

The **thiscall** calling convention can only be explicitly specified on Microsoft Visual C++ 2005 and later. On any other compiler *thiscall* is not a keyword. (However, disassemblers, such as IDA, must specify it. So IDA uses keyword __*thiscall* for this.)

Register preservation

Another part of a calling convention is which registers are guaranteed to retain their values after a subroutine call.

Caller-saved (volatile) registers

According to the Intel ABI to which the vast majority of compilers conform, the EAX, EDX, and ECX are to be free for use within a procedure or function, and need not be preserved.

As the name implies, these general-purpose registers usually hold temporary (volatile) information, that can be overwritten by any subroutine.

Therefore, it is the caller's responsibility to push each of these registers onto the stack, if it would like to restore their values after a subroutine call.

Callee-saved (non-volatile) registers

The other registers are used to hold long-lived values (non-volatile), that should be preserved across calls.

In other words, when the caller makes a procedure call, it can expect that those registers will hold the same value after the callee returns.

Thus, making it the callee's responsibility to both save (push at the beginning) and restore (pop accordingly) them before returning to the caller. As in the previous case, this practice should only be done on registers that the callee changes.

x86-64 calling conventions

x86-64 calling conventions take advantage of the additional register space to pass more arguments in registers. Also, the number of incompatible calling conventions has been reduced. There are two in common use.

Microsoft x64 calling convention

The Microsoft x64 calling convention[21][22] is followed on Windows and pre-boot UEFI (for long mode on x86-64). The first four arguments are placed onto the registers. That means RCX, RDX, R8, R9 (in that order) for integer, struct or pointer arguments, and XMM0, XMM1, XMM2, XMM3 for floating point arguments. Additional arguments are pushed onto the stack (right to left). Integer return values (similar to x86) are returned in RAX if 64 bits or less. Floating point return values are returned in XMM0. Parameters less than 64 bits long are not zero extended; the high bits are not zeroed.

Structs and unions with sizes that match integers are passed and returned as if they were integers. Otherwise they are replaced with a pointer when used as an argument. When an oversized struct return is needed, another pointer to a caller-provided space is prepended as the first argument, shifting all other arguments to the right by one place.[23]

When compiling for the x64 architecture in a Windows context (whether using Microsoft or non-Microsoft tools), stdcall, thiscall, cdecl, and fastcall all resolve to using this convention.

In the Microsoft x64 calling convention, it is the caller's responsibility to allocate 32 bytes of "shadow space" on the stack right before calling the function (regardless of the actual number of parameters used), and to pop the stack after the call. The shadow space is used to spill RCX, RDX, R8, and R9,[24] but must be made available to all functions, even those with fewer than four parameters.

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile (caller-saved).[25]

The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile (callee-saved).[25]

For example, a function taking 5 integer arguments will take the first to fourth in registers, and the fifth will be pushed on top of the shadow space. So when the called function is entered, the stack will be composed of (in ascending order) the return address, followed by the shadow space (32 bytes) followed by the fifth parameter.

In x86-64, Visual Studio 2008 stores floating point numbers in XMM6 and XMM7 (as well as XMM8 through XMM15); consequently, for x86-64, user-written assembly language routines must preserve XMM6 and XMM7 (as compared to x86 wherein user-written assembly language routines did not need to preserve XMM6 and XMM7). In other words, user-written assembly language routines must be updated to save/restore XMM6 and XMM7 before/after the function when being ported from x86 to x86-64.

Starting with Visual Studio 2013, Microsoft introduced the `__vectorcall` calling convention which extends the x64 convention.

System V AMD64 ABI

The calling convention of the System V AMD64 ABI is followed on Solaris, Linux, FreeBSD, macOS,[26] and is the de facto standard among Unix and Unix-like operating systems. The OpenVMS Calling Standard on x86-64 is based on the System V ABI with some extensions needed for backwards compatibility.[27] The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9 (R10 is used as a static chain pointer in case of nested functions[28]:21), while XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for the first floating point arguments.[28]:22 As in the Microsoft x64 calling convention, additional arguments are passed on the stack.[28]:22 Integer return values up to 64 bits in size are stored in RAX while values up to 128 bit are stored in RAX and RDX. Floating-point return values are similarly stored in XMM0 and XMM1.[28]:25 The wider YMM and ZMM registers are used for passing and returning wider values in place of XMM when they exist.[28]:26,55

Argument Register Overview

Argument Type	Registers
Integer/Pointer Arguments 1-6	RDI, RSI, RDX, RCX, R8, R9
Floating Point Arguments 1-8	XMM0 - XMM7
Excess Arguments	Stack
Static chain pointer	R10

Struct and union parameters with sizes of two (eight in case of only SSE fields) pointers or fewer that are aligned on 64-bit boundaries are decomposed into "eightbytes" and each one is classified and passed as a separate parameter.[28]:24 Otherwise they are replaced with a pointer when used as an argument. Struct and union return types with sizes of two pointers or fewer are returned in RAX and RDX (or XMM0 and XMM1). When an oversized struct return is needed, another pointer to a caller-provided space is prepended as the first argument, shifting all other arguments to the right by one place, and the value of this pointer is returned in RAX.[28]:27

If the callee wishes to use registers RBX, RSP, RBP, and R12–R15, it must restore their original values before returning control to the caller. All other registers must be saved by the caller if it wishes to preserve their values.[28]:16

For leaf-node functions (functions which do not call any other function(s)), a 128-byte space is stored just beneath the stack pointer of the function. The space is called the **red zone**. This zone will not be clobbered by any signal or interrupt handlers. Compilers can thus use this zone to save local variables. Compilers may omit some instructions at the starting of the function (adjustment of RSP, RBP) by using this zone. However, other functions may clobber this zone. Therefore, this zone should only be used for leaf-node functions. `gcc` and `clang` offer the `-mno-red-zone` flag to disable red-zone optimizations.

If the callee is a variadic function, then the number of floating point arguments passed to the function in vector registers must be provided by the caller in the AL register.[28]:55

Unlike the Microsoft calling convention, a shadow space is not provided; on function entry, the return address is adjacent to the seventh integer argument on the stack.

List of x86 calling conventions

This is a list of x86 calling conventions.[1] These are conventions primarily intended for C/C++ compilers (especially the 64-bit part below), and thus largely special cases. Other languages may use other formats and conventions in their implementations.

Archi- tecture	Name	Operating system, compiler	Parameters		Stack cleanup	Notes
			Registers	Stack order		
8086	cdecl			RTL (C)	Caller	
	Pascal			LTR (Pascal)	Callee	
	fastcall (non-member)	Microsoft	AX, DX, BX	LTR (Pascal)	Callee	Return pointer in BX.
	fastcall (member function)	Microsoft	AX, DX	LTR (Pascal)	Callee	this on stack low address. Return pointer in AX.
	fastcall	Turbo C[29]	AX, DX, BX	LTR (Pascal)	Callee	this on stack low address. Return pointer on stack high address.
		Watcom	AX, DX, BX, CX	RTL (C)	Callee	Return pointer in SI.
IA-32	cdecl	Unix-like (GCC)		RTL (C)	Caller	When returning struct/class, the calling code allocates space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address. Stack aligned on 16-byte boundary due to a bug.
	cdecl	Microsoft		RTL (C)	Caller	When returning struct/class, ▪ Plain old data (POD) return values 32 bits or smaller are in the EAX register ▪ POD return values 33–64 bits in size are returned via the EAX:EDX registers. ▪ Non-POD return values or values larger than 64-bits, the calling code will allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address. Stack aligned on 4-byte boundary.

	stdcall	Microsoft		RTL (C)	Callee	Also supported by GCC.
	fastcall	Microsoft	ECX, EDX	RTL (C)	Callee	Return pointer on stack if not member function. Also supported by GCC.
	register	Delphi and Free Pascal	EAX, EDX, ECX	LTR (Pascal)	Callee	
	thiscall	Windows (Microsoft Visual C++)	ECX	RTL (C)	Callee	Default for member functions.
	vectorcall	Windows (Microsoft Visual C++)	ECX, EDX, [XY]MM0–5	RTL (C)	Callee	Extended from fastcall. Also supported by ICC and Clang.[11]
		Watcom compiler	EAX, EDX, EBX, ECX	RTL (C)	Callee	Return pointer in ESI.
x86-64	Microsoft x64 calling convention[21]	Windows (Microsoft Visual C++, GCC, Intel C++ Compiler, Delphi), UEFI	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3	RTL (C)	Caller	Stack aligned on 16 bytes. 32 bytes shadow space on stack. The specified 8 registers can only be used for parameters 1 through 4. For C++ classes, the hidden `this` parameter is the first parameter, and is passed in RCX.[30]
	vectorcall	Windows (Microsoft Visual C++, Clang, ICC)	RCX/[XY]MM0, RDX/[XY]MM1, R8/[XY]MM2, R9/[XY]MM3 + [XY]MM4–5	RTL (C)	Caller	Extended from MS x64.[11]
	System V AMD64 ABI[28]	Solaris, Linux, BSD, macOS, OpenVMS (GCC, Intel C++ Compiler, Clang, Delphi)	RDI, RSI, RDX, RCX, R8, R9, [XYZ]MM0–7	RTL (C)	Caller	Stack aligned on 16 bytes boundary. 128 bytes red zone below stack. The kernel interface uses RDI, RSI, RDX, R10, R8 and R9. In C++, `this` is the first parameter.

References

Footnotes

1. Agner Fog (2010-02-16). *Calling conventions for different C++ compilers and operating systems* (http://agner.org/optimize/calling_conventions.pdf) (PDF).
2. de Boyne Pollard, Jonathan (2010). "The gen on function calling conventions" (http://jdebp.info/FGA/function-calling-conventions.html#cdecl). *Frequently Given Answers*.
3. "GCC Bugzilla – Bug 40838 - gcc shouldn't assume that the stack is aligned" (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=40838#c82). 2009.

4. "SYSTEM V APPLICATION BINARY INTERFACE Intel 386 Architecture Processor Supplement Fourth Edition" (http://www.sco.com/developers/devspecs/abi386-4.pdf) (PDF).
5. "__stdcall (C++)" (https://web.archive.org/web/20080410091312/http://msdn2.microsoft.com/en-us/library/zxk0tw93%28vs.71%29.aspx). *MSDN*. Microsoft. Archived from the original (http://msdn2.microsoft.com/en-us/library/zxk0tw93(vs.71).aspx) on 2008-04-10. Retrieved 2019-02-13.
6. "__fastcall" (http://msdn.microsoft.com/en-us/library/6xa169sk.aspx). MSDN. Retrieved 2013-09-26.
7. Ohse, Uwe. "gcc attribute overview: function fastcall" (http://www.ohse.de/uwe/articles/gcc-attributes.html#func-fastcall). ohse.de. Retrieved 2010-09-27.
8. "Attributes in Clang: fastcall" (https://clang.llvm.org/docs/AttributeReference.html#fastcall). *Clang Documentation*. 2022. Retrieved 15 December 2022.
9. Patocka, Mikulas (11 August 2009). "Fastcall calling convention is incompatible with Windows" (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=41013). Retrieved 15 December 2022.
10. "Introducing 'Vector Calling Convention' " (http://blogs.msdn.com/b/vcblog/archive/2013/07/12/introducing-vector-calling-convention.aspx). MSDN. 11 July 2013. Retrieved 2014-12-31.
11. "__vectorcall" (http://msdn.microsoft.com/en-us/library/dn375768.aspx). MSDN. Retrieved 2014-12-31.
12. "Attributes in Clang: vectorcall" (https://clang.llvm.org/docs/AttributeReference.html#vectorcall). *Clang Documentation*. 2022. Retrieved 15 December 2022.
13. "C/C++ Calling Conventions" (https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/c-c-calling-conventions.html). 16 December 2022. Retrieved 15 December 2022.
14. "_vectorcall and __regcall demystified" (https://software.intel.com/en-us/articles/vectorcall-and-regcall-demystified). *software.intel.com*. 7 June 2017.
15. "Program Control: Register Convention" (http://docwiki.embarcadero.com/RADStudio/en/Program_Control#Register_Convention). docwiki.embarcadero.com. 2010-06-01. Retrieved 2010-09-27.
16. "_fastcall, __fastcall" (http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Fastcall,_fastcall). docwiki.embarcadero.com.
17. "__msfastcall" (http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Msfastcall). docwiki.embarcadero.com.
18. "x86 Function Attributes" (https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html). *Using the GNU Compiler Collection (GCC)*.
19. "i386: always enable regparm" (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a1a70c25bed75ed36ed48bbe18b9029428d2452d).
20. "Calling_Conventions: Specifying_Calling_Conventions_the_Watcom_Way" (http://wiki.openwatcom.org/index.php/Calling_Conventions#Specifying_Calling_Conventions_the_Watcom_Way). openwatcom.org. 2010-04-27. Retrieved 2018-08-31.
21. "x64 Software Conventions: Calling Conventions" (https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2017). msdn.microsoft.com. 2010. Retrieved 2010-09-27.
22. "x64 Architecture" (http://msdn.microsoft.com/en-us/library/windows/hardware/ff561499.aspx). msdn.microsoft.com. 6 January 2023.
23. "x64 Calling Convention: Return Values" (https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2017#return-values). *docs.microsoft.com*. Retrieved 2020-01-17.
24. "x64 Software Conventions - Stack Allocation" (http://msdn.microsoft.com/en-us/library/ew5tede7(v=VS.90).aspx). Microsoft. Retrieved 2010-03-31.

25. "Caller/Callee Saved Registers" (https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019#callercallee-saved-registers). *Microsoft Docs*. Microsoft. 18 May 2022.
26. "x86-64 Code Model" (https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html). *Mac Developer Library*. Apple Inc. Archived (https://web.archive.org/web/20160310135710/https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html) from the original on 2016-03-10. Retrieved 2016-04-06. "The x86-64 environment in OS X has only one code model for user-space code. It is most similar to the small PIC model defined by the x86-64 System V ABI."
27. "VSI OpenVMS Calling Standard" (https://vmssoftware.com/docs/VSI_CALLING_STD.pdf) (PDF). *vmssoftware.com*. May 2020. Retrieved 2020-12-21.
28. H.J. Lu; Michael Matz; Milind Girkar; et al., eds. (2023-05-23). "System V Application Binary Interface: AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0" (https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build) (PDF). *GitLab*. 1.0.
29. *Borland C/C++ version 3.1 User Guide* (http://www.bitsavers.org/pdf/borland/borland_C++/Borland_C++_Version_3.1_Users_Guide_1992.pdf) (PDF). Borland. 1992. pp. 158, 189–191.
30. "Register Usage" (https://docs.microsoft.com/en-us/cpp/build/register-usage). *Microsoft Docs*. Microsoft. Retrieved 15 September 2017.

Other sources

- "SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement" (https://www.sco.com/developers/devspecs/abi386-4.pdf) (PDF) (4th ed.). The Santa Cruz Operation, Inc. 1997-03-19. {{cite journal}}: Cite journal requires |journal= (help)
- Nemanja Trifunovic (2001-07-22). Sean Ewington (ed.). "Calling Conventions Demystified" (http://www.codeproject.com/Articles/1388/Calling-Conventions-Demystified). *The Code Project*.
- Stephen J. Friedl. "Intel x86 Function-call Conventions — Assembly View" (http://www.unixwiz.net/techtips/win32-callconv-asm.html). *Steve Friedl's Unixwiz.net Tech Tips*.
- "Visual Studio 2010 — Visual C++ Calling Convention" (http://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx). *MSDN Library*. Microsoft. 2010.
- Andreas Jonsson (2005-02-13). "Calling conventions on the x86 platform" (http://www.angelcode.com/dev/callconv/callconv.html).
- Raymond Chen (2004-01-02). "The history of calling conventions, part 1" (https://devblogs.microsoft.com/oldnewthing/20040102-00/?p=41213). *The Old New Thing*.
- Raymond Chen (2004-01-07). "The history of calling conventions, part 2" (https://devblogs.microsoft.com/oldnewthing/20040107-00/?p=41183). *The Old New Thing*.
- Raymond Chen (2004-01-08). "The history of calling conventions, part 3" (https://devblogs.microsoft.com/oldnewthing/20040108-00/?p=41163). *The Old New Thing*.
- Raymond Chen (2004-01-13). "The history of calling conventions, part 4: ia64" (https://devblogs.microsoft.com/oldnewthing/20040113-00/?p=41073). *The Old New Thing*.
- Raymond Chen (2004-01-14). "The history of calling conventions, part 5; amd64" (https://devblogs.microsoft.com/oldnewthing/20040114-00/?p=41053). *The Old New Thing*.

Further reading

- Jonathan de Boyne Pollard (2010). "The gen on function calling conventions" (http://jdebp.in fo/FGA/function-calling-conventions.html). *Frequently Given Answers*.

- Kip R. Irvine (2011). "Advanced Procedures (Chapter 8)". *Assembly Language for x86 Processors* (6th ed.). Prentice Hall. ISBN 978-0-13-602212-1.
- *Borland C/C++ version 3.1 User Guide* (http://bitsavers.informatik.uni-stuttgart.de/pdf/borland/borland_C++/Borland_C++_Version_3.1_Users_Guide_1992.pdf) (PDF). Borland. 1992. pp. 158, 189–191.
- Thomas Lauer (1995). "The New __stdcall Calling Sequence". *Porting to Win32: A Guide to Making Your Applications Ready for the 32-Bit Future of Windows*. Springer. ISBN 978-0-387-94572-9.

-