x64 calling convention

Article • 05/18/2022

This section describes the standard processes and conventions that one function (the caller) uses to make calls into another function (the callee) in x64 code.

For more information on the `__vectorcall` calling convention, see __vectorcall.

Calling convention defaults

The x64 Application Binary Interface (ABI) uses a four-register fast-call calling convention by default. Space is allocated on the call stack as a shadow store for callees to save those registers.

There's a strict one-to-one correspondence between a function call's arguments and the registers used for those arguments. Any argument that doesn't fit in 8 bytes, or isn't 1, 2, 4, or 8 bytes, must be passed by reference. A single argument is never spread across multiple registers.

The x87 register stack is unused. It may be used by the callee, but consider it volatile across function calls. All floating point operations are done using the 16 XMM registers.

Integer arguments are passed in registers RCX, RDX, R8, and R9. Floating point arguments are passed in XMM0L, XMM1L, XMM2L, and XMM3L. 16-byte arguments are passed by reference. Parameter passing is described in detail in Parameter passing. These registers, and RAX, R10, R11, XMM4, and XMM5, are considered *volatile*, or potentially changed by a callee on return. Register usage is documented in detail in x64 register usage and Caller/callee saved registers.

For prototyped functions, all arguments are converted to the expected callee types before passing. The caller is responsible for allocating space for the callee's parameters. The caller must always allocate sufficient space to store four register parameters, even if the callee doesn't take that many parameters. This convention simplifies support for unprototyped C-language functions and vararg C/C++ functions. For vararg or unprototyped functions, any floating point values must be duplicated in the corresponding general-purpose register. Any parameters beyond the first four must be stored on the stack after the shadow store before the call. Vararg function details can be found in Varargs. Unprototyped function information is detailed in Unprototyped functions.

Alignment

Most structures are aligned to their natural alignment. The primary exceptions are the stack pointer and `malloc` or `alloca` memory, which are 16-byte aligned to aid performance. Alignment above 16 bytes must be done manually. Since 16 bytes is a common alignment size for XMM operations, this value should work for most code. For more information about structure layout and alignment, see x64 type and storage layout. For information about the stack layout, see x64 stack usage.

Unwindability

Leaf functions are functions that don't change any non-volatile registers. A non-leaf function may change non-volatile RSP, for example, by calling a function. Or, it could change RSP by allocating additional stack space for local variables. To recover non-volatile registers when an exception is handled, non-leaf functions are annotated with static data. The data describes how to properly unwind the function at an arbitrary instruction. This data is stored as *pdata*, or procedure data, which in turn refers to *xdata*, the exception handling data. The xdata contains the unwinding information, and can point to additional pdata or an exception handler function.

Prologs and epilogs are highly restricted so that they can be properly described in xdata. The stack pointer must remain 16-byte aligned in any region of code that isn't part of an epilog or prolog, except within leaf functions. Leaf functions can be unwound simply by simulating a return, so pdata and xdata aren't required. For details about the proper structure of function prologs and epilogs, see x64 prolog and epilog. For more information about exception handling, and the exception handling and unwinding of pdata and xdata, see x64 exception handling.

Parameter passing

By default, the x64 calling convention passes the first four arguments to a function in registers. The registers used for these arguments depend on the position and type of the argument. Remaining arguments get pushed on the stack in right-to-left order.

Integer valued arguments in the leftmost four positions are passed in left-to-right order in RCX, RDX, R8, and R9, respectively. The fifth and higher arguments are passed on the stack as previously described. All integer arguments in registers are right-justified, so the callee can ignore the upper bits of the register and access only the portion of the register necessary.

Any floating-point and double-precision arguments in the first four parameters are passed in XMM0 - XMM3, depending on position. Floating-point values are only placed in the integer registers RCX, RDX, R8, and R9 when there are varargs arguments. For details, see Varargs. Similarly, the XMM0 - XMM3 registers are ignored when the corresponding argument is an integer or pointer type.

__m128 types, arrays, and strings are never passed by immediate value. Instead, a pointer is passed to memory allocated by the caller. Structs and unions of size 8, 16, 32, or 64 bits, and `__m64` types, are passed as if they were integers of the same size. Structs or unions of other sizes are passed as a pointer to memory allocated by the caller. For these aggregate types passed as a pointer, including `__m128`, the caller-allocated temporary memory must be 16-byte aligned.

Intrinsic functions that don't allocate stack space, and don't call other functions, sometimes use other volatile registers to pass additional register arguments. This optimization is made possible by the tight binding between the compiler and the intrinsic function implementation.

The callee is responsible for dumping the register parameters into their shadow space if needed.

The following table summarizes how parameters are passed, by type and position from the left:

Expand table

Parameter type	fifth and higher	fourth	third	second	leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and `__m64`	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
`__m128`, as a pointer	stack	R9	R8	RDX	RCX

Example of argument passing 1 - all integers

C++

```cpp
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack
```

Example of argument passing 2 - all floats

C++

```cpp
func2(float a, double b, float c, double d, float e, float f);
// a in XMM0, b in XMM1, c in XMM2, d in XMM3, f then e pushed on stack
```

Example of argument passing 3 - mixed ints and floats

C++

```cpp
func3(int a, double b, int c, float d, int e, float f);
// a in RCX, b in XMM1, c in R8, d in XMM3, f then e pushed on stack
```

Example of argument passing 4 - `__m64`, `__m128`, and aggregates

C++

```cpp
func4(__m64 a, __m128 b, struct c, float d, __m128 e, __m128 f);
// a in RCX, ptr to b in RDX, ptr to c in R8, d in XMM3,
// ptr to f pushed on stack, then ptr to e pushed on stack
```

Varargs

If parameters are passed via varargs (for example, ellipsis arguments), then the normal register parameter passing convention applies. That convention includes spilling the fifth and later arguments to the stack. It's the callee's responsibility to dump arguments that have their address taken. For floating-point values only, both the integer register and the floating-point register must contain the value, in case the callee expects the value in the integer registers.

Unprototyped functions

For functions not fully prototyped, the caller passes integer values as integers and floating-point values as double precision. For floating-point values only, both the integer register and the floating-point register contain the float value in case the callee expects the value in the integer registers.

C++

```cpp
func1();
func2() {   // RCX = 2, RDX = XMM1 = 1.0, and R8 = 7
   func1(2, 1.0, 7);
}
```

Return values

A scalar return value that can fit into 64 bits, including the `__m64` type, is returned through RAX. Non-scalar types including floats, doubles, and vector types such as __m128, __m128i, __m128d are returned in XMM0. The state of unused bits in the value returned in RAX or XMM0 is undefined.

User-defined types can be returned by value from global functions and static member functions. To return a user-defined type by value in RAX, it must have a length of 1, 2, 4, 8, 16, 32, or 64 bits. It must also have no user-defined constructor, destructor, or copy assignment operator. It can have no private or protected non-static data members, and no non-static data members of reference type. It can't have base classes or virtual functions. And, it can only have data members that also meet these requirements. (This definition is essentially the same as a C++03 POD type. Because the definition has changed in the C++11 standard, we don't recommend using `std::is_pod` for this test.) Otherwise, the caller must allocate memory for the return value and pass a pointer to it as the first argument. The remaining arguments are then shifted one argument to the right. The same pointer must be returned by the callee in RAX.

These examples show how parameters and return values are passed for functions with the specified declarations:

Example of return value 1 - 64-bit result

C++

```cpp
__int64 func1(int a, float b, int c, int d, int e);
// Caller passes a in RCX, b in XMM1, c in R8, d in R9, e pushed on stack,
// callee returns __int64 result in RAX.
```

Example of return value 2 - 128-bit result

C++

```cpp
__m128 func2(float a, double b, int c, __m64 d);
// Caller passes a in XMM0, b in XMM1, c in R8, d in R9,
// callee returns __m128 result in XMM0.
```

Example of return value 3 - user type result by pointer

C++

```cpp
struct Struct1 {
   int j, k, l;    // Struct1 exceeds 64 bits.
};
Struct1 func3(int a, double b, int c, float d);
// Caller allocates memory for Struct1 returned and passes pointer in RCX,
// a in RDX, b in XMM2, c in R9, d pushed on the stack;
// callee returns pointer to Struct1 result in RAX.
```

Example of return value 4 - user type result by value

C++

```
struct Struct2 {
   int j, k;    // Struct2 fits in 64 bits, and meets requirements for return by value.
};
Struct2 func4(int a, double b, int c, float d);
// Caller passes a in RCX, b in XMM1, c in R8, and d in XMM3;
// callee returns Struct2 result by value in RAX.
```

Caller/callee saved registers

The x64 ABI considers the registers RAX, RCX, RDX, R8, R9, R10, R11, and XMM0-XMM5 volatile. When present, the upper portions of YMM0-YMM15 and ZMM0-ZMM15 are also volatile. On AVX512VL, the ZMM, YMM, and XMM registers 16-31 are also volatile. When AMX support is present, the TMM tile registers are volatile. Consider volatile registers destroyed on function calls unless otherwise safety-provable by analysis such as whole program optimization.

The x64 ABI considers registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15, and XMM6-XMM15 nonvolatile. They must be saved and restored by a function that uses them.

Function pointers

Function pointers are simply pointers to the label of the respective function. There's no table of contents (TOC) requirement for function pointers.

Floating-point support for older code

The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are preserved across context switches. There's no explicit calling convention for these registers. The use of these registers is strictly prohibited in kernel mode code.

FPCSR

The register state also includes the x87 FPU control word. The calling convention dictates this register to be nonvolatile.

The x87 FPU control word register gets set using the following standard values at the start of program execution:

Expand table

Register[bits]	Setting
FPCSR[0:6]	Exception masks all 1's (all exceptions masked)
FPCSR[7]	Reserved - 0
FPCSR[8:9]	Precision Control - 10B (double precision)
FPCSR[10:11]	Rounding control - 0 (round to nearest)
FPCSR[12]	Infinity control - 0 (not used)

A callee that modifies any of the fields within FPCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee, unless by agreement the callee expects the modified values.

There are two exceptions to the rules about the non-volatility of the control flags:

- In functions where the documented purpose of the given function is to modify the nonvolatile FPCSR flags.
- When it's provably correct that the violation of these rules results in a program that behaves the same as a program that doesn't violate the rules, for example, through whole-program analysis.

MXCSR

The register state also includes MXCSR. The calling convention divides this register into a volatile portion and a nonvolatile portion. The volatile portion consists of the six status flags, in MXCSR[0:5], while the rest of the register, MXCSR[6:15], is considered nonvolatile.

The nonvolatile portion is set to the following standard values at the start of program execution:

Expand table

Register[bits]	Setting
MXCSR[6]	Denormals are zeros - 0
MXCSR[7:12]	Exception masks all 1's (all exceptions masked)
MXCSR[13:14]	Rounding control - 0 (round to nearest)
MXCSR[15]	Flush to zero for masked underflow - 0 (off)

A callee that modifies any of the nonvolatile fields within MXCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee, unless by agreement the callee expects the modified values.

There are two exceptions to the rules about the non-volatility of the control flags:

- In functions where the documented purpose of the given function is to modify the nonvolatile MXCSR flags.
- When it's provably correct that the violation of these rules results in a program that behaves the same as a program that doesn't violate the rules, for example, through whole-program analysis.

Make no assumptions about the MXCSR register's volatile portion state across a function boundary, unless the function documentation explicitly describes it.

setjmp/longjmp

When you include setjmpex.h or setjmp.h, all calls to setjmp or longjmp result in an unwind that invokes destructors and `__finally` calls. This behavior differs from x86, where including setjmp.h results in `__finally` clauses and destructors not being invoked.

A call to `setjmp` preserves the current stack pointer, non-volatile registers, and MXCSR registers. Calls to `longjmp` return to the most recent `setjmp` call site and resets the stack pointer, non-volatile registers, and MXCSR registers, back to the state as preserved by the most recent `setjmp` call.

See also

x64 software conventions

Feedback