## APPENDIX B

# Calling Convention Summary

This appendix summarizes the conventions that must be observed in order to successfully call an x86 assembly language function from a Visual C++ function. It includes synopses of volatile and non-volatile registers, function arguments and return values, and control flags. Chapters 2, 4, and 18 cover the calling conventions in greater detail. The Visual C++ documentation can also be consulted for further information.

## Volatile and Non-Volatile Registers

Table B-1 lists the volatile and non-volatile registers for Visual C++ functions. A called function must preserve the contents of any non-volatile register that it uses. Win32 functions can use the `push` and `pop` instructions to preserve the contents of a non-volatile general-purpose register. Win64 functions must preserve non-volatile general-purpose and XMM registers in accordance with the 64-bit prolog rules, which are discussed in Chapter 18. Both Win32 and Win64 functions can use the volatile registers without any restrictions.

***Table B-1.*** *Visual C++ Volatile and Non-Volatile Registers*

| | C++ Win32 | C++ Win64 |
|---|---|---|
| Non-volatile, general-purpose registers | EBP, EBX, ESI, EDI | RBP, RBX, RSI, RDI, R12-R15 |
| Volatile, general-purpose registers | EAX, ECX, EDX | RAX, RCX, RDX, R8-R11 |
| Volatile x87 FPU registers | ST(0)-ST(7) | ST(0)-ST(7) |
| Volatile MMX registers | MM0-MM7 | MM0-MM7 |
| Non-volatile XMM registers | None | XMM6-XMM15 |
| Volatile XMM registers | XMM0-XMM7 | XMM0-XMM5 |
| Volatile YMM registers (bits 255:128) | YMM0-YMM7 | YMM0-YMM15 |

Functions that use registers MM0-MM7 must include an `emms` instruction before returning. Functions that use the YMM registers should include a `vzeroupper` instruction before any epilog code.

# Function Arguments and Return Values

Table B-2 summarizes the conventions that Visual C++ uses for function arguments and return values.

***Table B-2.*** *Visual C++ Function Argument and Return Values*

| | C++ Win32 | C++ Win64 |
|---|---|---|
| Function arguments (Visual C++ defaults) | Caller pushes arguments onto stack from right to left (`__cdecl` convention) | Arg0: RCX or XMM0<br>Arg1: RDX or XMM1<br>Arg2: R8 or XMM2<br>Arg3: R9 or XMM3<br>Additional arguments are pushed onto the stack from right to left<br>RCX-R8 used for integers.<br>XMM0-XMM3 used for floating-point<br>All 8-, 16-, and 32-bit wide argument values are right-justified in a register or on the stack |
| Integer return values (8-, 16-, and 32-bit) | EAX | RAX |
| Integer return values (64-bit) | EDX:EAX | RAX |
| Pointer return value | EAX | RAX |
| Floating-point return value (`float`, `double`) | ST(0)<br>x87 FPU register stack must be empty otherwise | XMM0 |
| Return of structure by value (1, 2, 4, or 8 bytes) | EDX:EAX (right-justified) | RAX (right-justified) |

(*continued*)

***Table B-2.*** (*continued*)

| | C++ Win32 | C++ Win64 |
|---|---|---|
| Return of structure by value (other sizes) | Memory location specified by hidden pointer at [EBP+8]; called function must return hidden pointer in EAX | Memory location specified by hidden pointer in RCX; called function must return hidden pointer in RAX |
| Stack pointer alignment | 4-byte boundary | 8-byte boundary for leaf functions<br>16-byte boundary for non-leaf functions |

Functions that return a structure by value using a hidden pointer must reference their arguments differently. For example, consider the function `extern "C" MyStruct Foo(int a, int b)`. A Win32 implementation of this function can access the hidden structure pointer at memory location `[ebp+8]`, argument `a` at `[ebp+12]` and argument `b` at `[ebp+16]`. The Win64 version passes these arguments in registers RCX, RDX, and R8, respectively. The Visual C++ documentation contains additional information regarding functions that return a structure by value.

# Control Registers and Flags

Table B-3 lists the default values for the x87 FPU control word, x86-SSE control and status register (MXCSR), and string direction flag. All of the control flags are non-volatile; their contents must be preserved by a called function. Programs that need to unmask one or more floating-point exceptions can define a custom floating-point exception handler using the Visual C++ library function `_fpieee_flt`.

***Table B-3.*** *Default Values for Control Registers and Flags (Win32 and Win64)*

| Control Register or Flag | Default Bit Values |
|---|---|
| X87 FPU control word | Bits [5:0] = 1 (all exceptions masked)<br>Bits [7] = 1 (reserved)<br>Bits [9:8] = 10 (53 bits of precision)<br>Bits [11:10] = 00 (round to nearest)<br>Bits [12] = 0 (infinity control disabled)<br>Bits [15:13] = 0 (reserved) |
| MXCSR | Bits [5:0] = status bits (volatile)<br>Bits [6] = 0 (denormals-are-zero disabled)<br>Bits [12:7] = 1 (all exceptions masked)<br>Bits [14:13] = 00 (round to nearest)<br>Bits [15] = 0 (flush-to-zero disabled) |
| Direction flag (EFLAGS.DF) | EFLAGS.DF = 0 (auto increment) |