


__vectorcall

Article • 10/17/2022

Microsoft Specific

The `__vectorcall` calling convention specifies that arguments to functions are to be passed in registers, when possible. `__vectorcall` uses more registers for arguments than `__fastcall` or the default `x64 calling convention` use. The `__vectorcall` calling convention is only supported in native code on x86 and x64 processors that include Streaming SIMD Extensions 2 (SSE2) and above. Use `__vectorcall` to speed functions that pass several floating-point or SIMD vector arguments and perform operations that take advantage of the arguments loaded in registers. The following list shows the features that are common to the x86 and x64 implementations of `__vectorcall`. The differences are explained later in this article.

 Expand table

Element	Implementation
C name-decoration convention	Function names are suffixed with two "at" signs (@@) followed by the number of bytes (in decimal) in the parameter list.
Case-translation convention	No case translation is performed.

Using the `/Gv` compiler option causes each function in the module to compile as `__vectorcall` unless the function is a member function, is declared with a conflicting calling convention attribute, uses a `vararg` variable argument list, or has the name `main`.

You can pass three kinds of arguments by register in `__vectorcall` functions: *integer type* values, *vector type* values, and *homogeneous vector aggregate* (HVA) values.

An integer type satisfies two requirements: it fits in the native register size of the processor—for example, 4 bytes on an x86 machine or 8 bytes on an x64 machine—and it's convertible to an integer of register length and back again without changing its bit representation. For example, any type that can be promoted to `int` (`long long` on x64)—for example, a `char` or `short`—or that can be cast to `int` (`long long` on x64) and back to its original type without change is an integer type. Integer types include pointer, reference, and `struct` or `union` types of 4 bytes (8 bytes on x64) or less. On x64 platforms, larger `struct` and `union` types are passed by reference to memory allocated by the caller; on x86 platforms, they are passed by value on the stack.

A vector type is either a floating-point type—for example, a `float` or `double`—or an SIMD vector type—for example, `__m128` or `__m256`.

An HVA type is a composite type of up to four data members that have identical vector types. An HVA type has the same alignment requirement as the vector type of its members. This is an example of an HVA `struct` definition that contains three identical vector types and has 32-byte alignment:

C++

```
typedef struct {
    __m256 x;
    __m256 y;
    __m256 z;
} hva3;    // 3 element HVA type on __m256
```

Declare your functions explicitly with the `__vectorcall` keyword in header files to allow separately compiled code to link without errors. Functions must be prototyped to use `__vectorcall`, and can't use a `vararg` variable length argument list.

A member function may be declared by using the `__vectorcall` specifier. The hidden `this` pointer is passed by register as the first integer type argument.

On ARM machines, `__vectorcall` is accepted and ignored by the compiler. On ARM64EC, `__vectorcall` is unsupported and rejected by the compiler.

For non-static class member functions, if the function is defined out-of-line, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for class non-static members, the calling convention specified during declaration is assumed at the point of definition. Given this class definition:

```
C++

struct MyClass {
    void __vectorcall mymethod();
};
```

this:

```
C++

void MyClass::mymethod() { return; }
```

is equivalent to this:

```
C++

void __vectorcall MyClass::mymethod() { return; }
```

The `__vectorcall` calling convention modifier must be specified when a pointer to a `__vectorcall` function is created. The next example creates a `typedef` for a pointer to a `__vectorcall` function that takes four `double` arguments and returns an `__m256` value:

```
C++

typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

For compatibility with previous versions, `_vectorcall` is a synonym for `__vectorcall` unless compiler option [/Za \(Disable language extensions\)](#) is specified.

`__vectorcall` convention on x64

The `__vectorcall` calling convention on x64 extends the standard x64 calling convention to take advantage of additional registers. Both integer type arguments and vector type arguments are mapped to registers based on position in the argument list. HVA arguments are allocated to unused vector registers.

When any of the first four arguments in order from left to right are integer type arguments, they are passed in the register that corresponds to that position—RCX, RDX, R8, or R9. A hidden `this` pointer is treated as the first integer type argument. When an HVA argument in one of the first four arguments can't be passed in the available registers, a

reference to caller-allocated memory is passed in the corresponding integer type register instead. Integer type arguments after the fourth parameter position are passed on the stack.

When any of the first six arguments in order from left to right are vector type arguments, they are passed by value in SSE vector registers 0 to 5 according to argument position. Floating-point and `__m128` types are passed in XMM registers, and `__m256` types are passed in YMM registers. This differs from the standard x64 calling convention, because the vector types are passed by value instead of by reference, and additional registers are used. The shadow stack space allocated for vector type arguments is fixed at 8 bytes, and the `/homeparams` option does not apply. Vector type arguments in the seventh and later parameter positions are passed on the stack by reference to memory allocated by the caller.

After registers are allocated for vector arguments, the data members of HVA arguments are allocated, in ascending order, to unused vector registers XMM0 to XMM5 (or YMM0 to YMM5, for `__m256` types), as long as there are enough registers available for the entire HVA. If not enough registers are available, the HVA argument is passed by reference to memory allocated by the caller. The stack shadow space for an HVA argument is fixed at 8 bytes with undefined content. HVA arguments are assigned to registers in order from left to right in the parameter list, and may be in any position. HVA arguments in one of the first four argument positions that are not assigned to vector registers are passed by reference in the integer register that corresponds to that position. HVA arguments passed by reference after the fourth parameter position are pushed on the stack.

Results of `__vectorcall` functions are returned by value in registers when possible. Results of integer type, including structs or unions of 8 bytes or less, are returned by value in RAX. Vector type results are returned by value in XMM0 or YMM0, depending on size. HVA results have each data element returned by value in registers XMM0:XMM3 or YMM0:YMM3, depending on element size. Result types that don't fit in the corresponding registers are returned by reference to memory allocated by the caller.

The stack is maintained by the caller in the x64 implementation of `__vectorcall`. The caller prolog and epilog code allocates and clears the stack for the called function. Arguments are pushed on the stack from right to left, and shadow stack space is allocated for arguments passed in registers.

Examples:

```
C++

// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
// This example creates an annotated assembly listing in
// crt_vc64.asm.

#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}
```

```

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
// Passes b by element in [XMM0:XMM1];
// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0, YMM2, YMM4, YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    f = example4(1, 2.0f, h4, d, 5);
    i = example5(1, h2, 3, h4, 5);
}

```

```
h4 = example6(h2, h4, c, h2);
}
```

__vectorcall convention on x86

The `__vectorcall` calling convention follows the `__fastcall` convention for 32-bit integer type arguments, and takes advantage of the SSE vector registers for vector type and HVA arguments.

The first two integer type arguments found in the parameter list from left to right are placed in ECX and EDX, respectively. A hidden `this` pointer is treated as the first integer type argument, and is passed in ECX. The first six vector type arguments are passed by value through SSE vector registers 0 to 5, in the XMM or YMM registers, depending on argument size.

The first six vector type arguments in order from left to right are passed by value in SSE vector registers 0 to 5. Floating-point and `__m128` types are passed in XMM registers, and `__m256` types are passed in YMM registers. No shadow stack space is allocated for vector type arguments passed by register. The seventh and subsequent vector type arguments are passed on the stack by reference to memory allocated by the caller. The limitation of compiler error [C2719](#) does not apply to these arguments.

After registers are allocated for vector arguments, the data members of HVA arguments are allocated in ascending order to unused vector registers XMM0 to XMM5 (or YMM0 to YMM5, for `__m256` types), as long as there are enough registers available for the entire HVA. If not enough registers are available, the HVA argument is passed on the stack by reference to memory allocated by the caller. No stack shadow space for an HVA argument is allocated. HVA arguments are assigned to registers in order from left to right in the parameter list, and may be in any position.

Results of `__vectorcall` functions are returned by value in registers when possible. Results of integer type, including structs or unions of 4 bytes or less, are returned by value in EAX. Integer type structs or unions of 8 bytes or less are returned by value in EDX:EAX. Vector type results are returned by value in XMM0 or YMM0, depending on size. HVA results have each data element returned by value in registers XMM0:XMM3 or YMM0:YMM3, depending on element size. Other result types are returned by reference to memory allocated by the caller.

The x86 implementation of `__vectorcall` follows the convention of arguments pushed on the stack from right to left by the caller, and the called function clears the stack just before it returns. Only arguments that are not placed in registers are pushed on the stack.

Examples:

```
C++

// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;    // 4 element HVA type on __m256
```

```

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
}

```

```
e = example2(1, b, 3, d, e, 6.0f, 7);  
d = example3(1, h2, 3, 4, 5);  
f = example4(1, 2.0f, h4, d, 5);  
i = example5(1, h2, 3, h4, 5);  
h4 = example6(h2, h4, c, h2);  
}
```

End Microsoft Specific

See also

[Argument Passing and Naming Conventions](#)

[Keywords](#)

Feedback

Was this page helpful?



[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)