x64 Software Conventions

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
This section describes the Visual C++ calling convention methodology for x64, 64-bit extension to the x86 architecture.

The following compiler option helps you optimize your application for x64:

- /favor (Optimize for Architecture Specifics)

See Also
Calling Conventions

Overview of x64 Calling Conventions

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Two important differences between x86 and x64 are the 64-bit addressing capability and a flat set of 16 64-bit registers for general use. Given the expanded register set, x64 uses the __fastcall calling convention and a RISC-based exception-handling model. The `__fastcall` convention uses registers for the first four arguments and the stack frame to pass additional arguments.
The following compiler option helps you optimize your application for x64:

- /favor (Optimize for Architecture Specifics)

Calling convention

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 is a strict one-to-one correspondence between the arguments to a function call and the registers used for those arguments. Any argument that doesn't fit in 8 bytes, or is not 1, 2, 4, or 8 bytes, must be passed by reference. There is no attempt to spread a single argument across multiple registers. The x87 register stack is unused. It may be used by the callee, but must be considered 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. In addition to these registers, RAX, R10, R11, XMM4, and XMM5 are considered volatile. All other registers are non-volatile. Register usage is documented in detail in Register Usage and Caller/Callee Saved Registers.
The caller is responsible for allocating space for parameters to the callee, and must always allocate sufficient space to store four register parameters, even if the callee doesn't take that many parameters. This 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, above the shadow store for the first four, prior to 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 aligned to 16 bytes in order to aid performance. Alignment above 16 bytes must be done manually, but since 16 bytes is a common alignment size for XMM operations, this should work for most code. For more information about structure layout and alignment see Types and Storage. For information about the stack layout, see Stack Usage.

Unwindability

Leaf functions are functions that do not change any non-volatile registers. A non-leaf function may change non-volatile RSP, for example, by calling a function or allocating additional stack space for local variables. In order to recover non-volatile registers when an exception is handled, non-leaf functions must be annotated with static data that 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 be aligned to 16 bytes 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 are not required. For details about the proper structure of function prologs and epilogs, see Prolog and Epilog. For more information about exception handling, and the exception handling and unwinding of pdata and xdata, see Exception Handling (x64).

Types and Storage

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
This section describes the enumeration and storage of data types for the x64 architecture.

- Scalar Types
- Aggregates and Unions
- Structure Alignment Examples
- Bitfields
- Conflicts with the x86 Compiler

Scalar Types

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation. Although the access of data can stem from any alignment, it is recommended that data be aligned on its natural boundary to avoid performance loss (or a multiple thereof). Enums are constant integers and are treated as 32-bit integers. The following table describes the type definition and recommended storage for it as it pertains to alignment using the following alignment values:

- Byte – 8 bits
- Word – 16 bits
- Double Word – 32 bits
- Quad Word – 64 bits
- Octa Word – 128 bits

Scalar Type	C Data Type	Storage Size (in bytes)	Recommended Alignment
INT8	`char`	1	Byte
UINT8	`unsigned char`	1	Byte
INT16	**short**	2	Word
UINT16	**unsigned short**	2	Word
INT32	**int, long**	4	Doubleword

UINT32	**unsigned int, unsigned long**	4	Doubleword
INT64	`__int64`	8	Quadword
UINT64	**unsigned __int64**	8	Quadword
FP32 (single precision)	**float**	4	Doubleword
FP64 (double precision)	**double**	8	Quadword
POINTER	*****	8	Quadword
`__m64`	**struct __m64**	8	Quadword
`__m128`	**struct __m128**	16	Octaword

Aggregates and Unions

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Other types such as arrays, structs, and unions have stricter alignment requirements that ensure consistent aggregate and union storage and data retrieval. Here are the definitions for array, structure, and union:
Array
Contains an ordered group of adjacent data objects. Each object is called an element. All elements within an array have the same size and data type.
Structure
Contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have different data types and sizes. Each data object in a structure is called a member.
Union
An object that holds any one of a set of named members. The members of the named set can be of any type. The storage allocated for a union is equal to the storage required for the largest member of that union, plus any padding required for alignment.
The following table shows the strongly suggested alignment for the scalar members of unions and structures.

Scalar Type	C Data Type	Required Alignment
INT8	`char`	Byte
UINT8	`unsigned char`	Byte
INT16	**short**	Word
UINT16	**unsigned short**	Word
INT32	**int, long**	Doubleword

UINT32	**unsigned int, unsigned long**	Doubleword
INT64	`__int64`	Quadword
UINT64	**unsigned __int64**	Quadword
FP32 (single precision)	**float**	Doubleword
FP64 (double precision)	**double**	Quadword
POINTER	*****	Quadword
`__m64`	**struct __m64**	Quadword
`__m128`	**struct __m128**	Octaword

The following aggregate alignment rules apply:

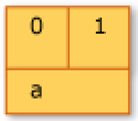
- The alignment of an array is the same as the alignment of one of the elements of the array.
- The alignment of the beginning of a structure or a union is the maximum alignment of any individual member. Each member within the structure or union must be placed at its proper alignment as defined in the previous table, which may require implicit internal padding, depending on the previous member.
- Structure size must be an integral multiple of its alignment, which may require padding after the last member. Since structures and unions can be grouped in arrays, each array element of a structure or union must begin and end at the proper alignment previously determined.
- It is possible to align data in such a way as to be greater than the alignment requirements as long as the previous rules are maintained.

- An individual compiler may adjust the packing of a structure for size reasons. For example /Zp (Struct Member Alignment) allows for adjusting the packing of structures.

Examples of Structure Alignment

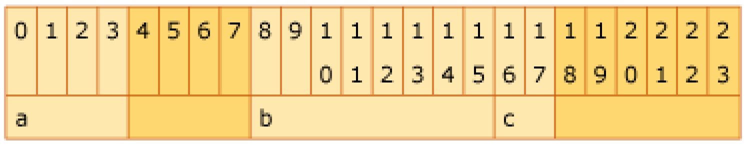
For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation. The following four examples each declare an aligned structure or union, and the corresponding figures illustrate the layout of that structure or union in memory. Each column in a figure represents a byte of memory, and the number in the column indicates the displacement of that byte. The name in the second row of each figure corresponds to the name of a variable in the declaration. The shaded columns indicate padding that is required to achieve the specified alignment.

```
// Total size = 2 bytes, alignment = 2 bytes (word).

_declspec(align(2)) struct {
   short a;     // +0; size = 2 bytes
}
```



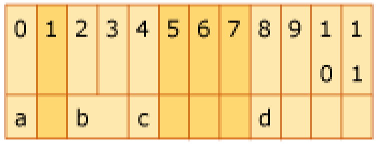
Example 1

```
// Total size = 24 bytes, alignment = 8 bytes (quadword).

_declspec(align(8))  struct {
   int a;       //  +0;  size = 4 bytes
   double b;    //  +8;  size = 8 bytes
   short c;     //  +16; size = 2 bytes
}
```



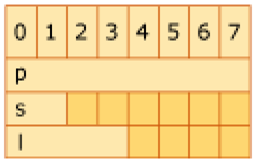
Example 2

```
// Total size = 12 bytes, alignment = 4 bytes (doubleword).

_declspec(align(4))  struct {
   char a;      //  +0; size = 1 byte
   short b;     //  +2; size = 2 bytes
   char c;      //  +4; size = 1 byte
   int d;       //  +8; size = 4 bytes
}
```



Example 3

```
// Total size = 8 bytes, alignment = 8 bytes (quadword).

_declspec(align(8))  union {
   char *p;     //  +0; size = 4 byte
   short s;     //  +0; size = 2 bytes
   long l;      //  +0; size = 4 byte
}
```



Example 4

See Also

Bitfields

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Structure bit fields are limited to 64 bits and can be of type signed int, unsigned int, int64, or unsigned int64. Bit fields that cross the type boundary will skip bits to align the bitfield to the next type alignment. For example, integer bitfields may not cross a 32-bit boundry.

Conflicts with the x86 Compiler

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Data types that are larger than 4 bytes are not automatically aligned on the stack when you use the x86 compiler to compile an application. Because the architecture for the x86 compiler is a 4 byte aligned stack, anything larger than 4 bytes, for example, a 64-bit integer, cannot be automatically aligned to an 8-byte address.
Working with unaligned data has two implications.

- It may take longer to access unaligned locations than it takes to access aligned locations.
- Unaligned locations cannot be used in interlocked operations.

If you require more strict alignment, use `__declspec(align(N)) on your variable declarations`. This causes the compiler to dynamically align the stack to meet your specifications. However, dynamically adjusting the stack at run time may cause slower execution of your application.

Register Usage

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
The x64 architecture provides for 16 general-purpose registers (hereafter referred to as integer registers) as well as 16 XMM/YMM registers available for floating-point use. Volatile registers are scratch registers presumed by the caller to be destroyed across a call. Nonvolatile registers are required to retain their values across a function call and must be saved by the callee if used.
The following table describes how each register is used across function calls:

Register	Status	Use
RAX	Volatile	Return value register
RCX	Volatile	First integer argument
RDX	Volatile	Second integer argument
R8	Volatile	Third integer argument
R9	Volatile	Fourth integer argument

R10:R11	Volatile	Must be preserved as needed by caller; used in syscall/sysret instructions
R12:R15	Nonvolatile	Must be preserved by callee
RDI	Nonvolatile	Must be preserved by callee
RSI	Nonvolatile	Must be preserved by callee
RBX	Nonvolatile	Must be preserved by callee
RBP	Nonvolatile	May be used as a frame pointer; must be preserved by callee
RSP	Nonvolatile	Stack pointer
XMM0, YMM0	Volatile	First FP argument; first vector-type argument when `__vectorcall` is used
XMM1, YMM1	Volatile	Second FP argument; second vector-type argument when `__vectorcall` is used

XMM2, YMM2	Volatile	Third FP argument; third vector-type argument when `__vectorcall` is used
XMM3, YMM3	Volatile	Fourth FP argument; fourth vector-type argument when `__vectorcall` is used
XMM4, YMM4	Volatile	Must be preserved as needed by caller; fifth vector-type argument when `__vectorcall` is used
XMM5, YMM5	Volatile	Must be preserved as needed by caller; sixth vector-type argument when `__vectorcall` is used
XMM6:XMM15, YMM6:YMM15	Nonvolatile (XMM), Volatile (upper half of YMM)	Must be preserved as needed by callee. YMM registers must be preserved as needed by caller.

Calling Convention

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
This section describes the process that one function (caller) makes call into another function (callee).
For prototyped functions, all arguments are converted to the expected callee types before passing.

- Parameter Passing
- Varargs
- Unprototyped Functions
- Return Values
- Caller/Callee Saved Registers
- Function Pointers

- Floating-Point Support for Older Code (Visual C++)
- FpCsr
- MXCSR
- setjmp/longjump

Parameter Passing

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
The first four integer arguments are passed in registers. Integer values are passed (in order left to right) in RCX, RDX, R8, and R9. Arguments five and higher are passed on the stack. All arguments are right-justified in registers. This is done so the callee can ignore the upper bits of the register if need be and can access only the portion of the register necessary.
Floating-point and double-precision arguments are passed in XMM0 – XMM3 (up to 4) with the integer slot (RCX, RDX, R8, and R9) that would normally be used for that cardinal slot being ignored (see example) and vice versa.
__m128 types, arrays and strings are never passed by immediate value but rather a pointer is passed to memory allocated by the caller. Structs/unions of size 8, 16, 32, or 64 bits and __m64 are passed as if they were integers of the same size. Structs/unions other than these 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 will be 16-byte aligned.
Intrinsic functions that do not allocate stack space and do not call other functions can use other volatile registers to pass additional register arguments because there is a tight binding between the compiler and the intrinsic function implementation. This is a further opportunity for improving performance.
The callee has the responsibility of dumping the register parameters into their shadow space if needed.
The following table summarizes how parameters are passed:

Parameter type	How passed
Floating point	First 4 parameters – XMM0 through XMM3. Others passed on stack.
Integer	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.

Parameter type	How passed
Aggregates (8, 16, 32, or 64 bits) and __m64	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (other)	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9
__m128	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9

Example of argument passing 1 – all integers

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

Example of argument passing 2 – all floats

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

Example of argument passing 3 – mixed ints and floats

```
func3(int a, double b, int c, float d);
// a in RCX, b in XMM1, c in R8, d in XMM3
```

Example of argument passing 4 –__m64, __m128, and aggregates

```
func4(__m64 a, _m128 b, struct c, float d);
// a in RCX, ptr to b in RDX, ptr to c in R8, d in XMM3
```

Varargs

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.

If parameters are passed via varargs (for example, ellipsis arguments), then essentially the normal parameter passing applies including spilling the fifth and subsequent arguments. It is again the callee's responsibility to dump arguments that have their address taken. For floating-point values only, both the integer and the floating-point register will contain the float value in case the callee expects the value in the integer registers.

Unprototyped Functions

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
For functions not fully prototyped, the caller will pass 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 will contain the float value in case the callee expects the value in the integer registers.

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

Return Values (C++)

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
A scalar return value that can fit into 64 bits is returned through RAX—this includes __m64 types. 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 be returned by value in RAX, user-defined types must have a length of 1, 2, 4, 8, 16, 32, or 64 bits; no user-defined constructor, destructor, or copy assignment operator; no private or protected non-static data members; no non-static data members of reference type; no base classes; no virtual functions; and no data members that do not also meet these requirements. (This is essentially the definition of a C++03 POD type. Because the definition has changed in the C++11 standard, we do not recommend using `std::is_pod` for this test.) Otherwise, the caller assumes the responsibility of allocating memory and passing a pointer for the return value as the first argument. Subsequent 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

Output

```
__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

Output

```
__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

Output

```
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

Output

```
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

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile and must be considered destroyed on function calls (unless otherwise safety-provable by analysis such as whole program optimization).
The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile and must be saved and restored by a function that uses them.

Function Pointers

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Function pointers are simply pointers to the label of the respective function. There are no table of contents (TOC) requirements for function pointers.

Floating-Point Support for Older Code (Visual C++)

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are preserved across context switches. There is no explicit calling convention for these registers. The use of these registers is strictly prohibited in kernel mode code.

FpCsr

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
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 is set to the following standard values at the start of program execution:

```
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 regarding the non-volatility of the control flags:

1. In functions where the documented purpose of the given function is to modify the nonvolatile FpCsr flags.
2. When it is provably correct that the violation of these rules results in a programs that behaves/means the same as a program where these rules are not violated, for example, through whole-program analysis.

MxCsr

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](). 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 6 status flags, MXCSR[0:5], while the remainder 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:

```
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 regarding 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 is provably correct that the violation of these rules results in a programs that behaves/means the same as a program where these rules are not violated, for example, through whole-program analysis.

No assumptions can be made about the state of the volatile portion of MXCSR across a function boundary, unless specifically described in a function's documentation.

setjmp/longjump

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
When you include setjmpex.h or setjmp.h, all calls to setjmp or longjmp will result in an unwind that invokes destructors and finally calls. This 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.

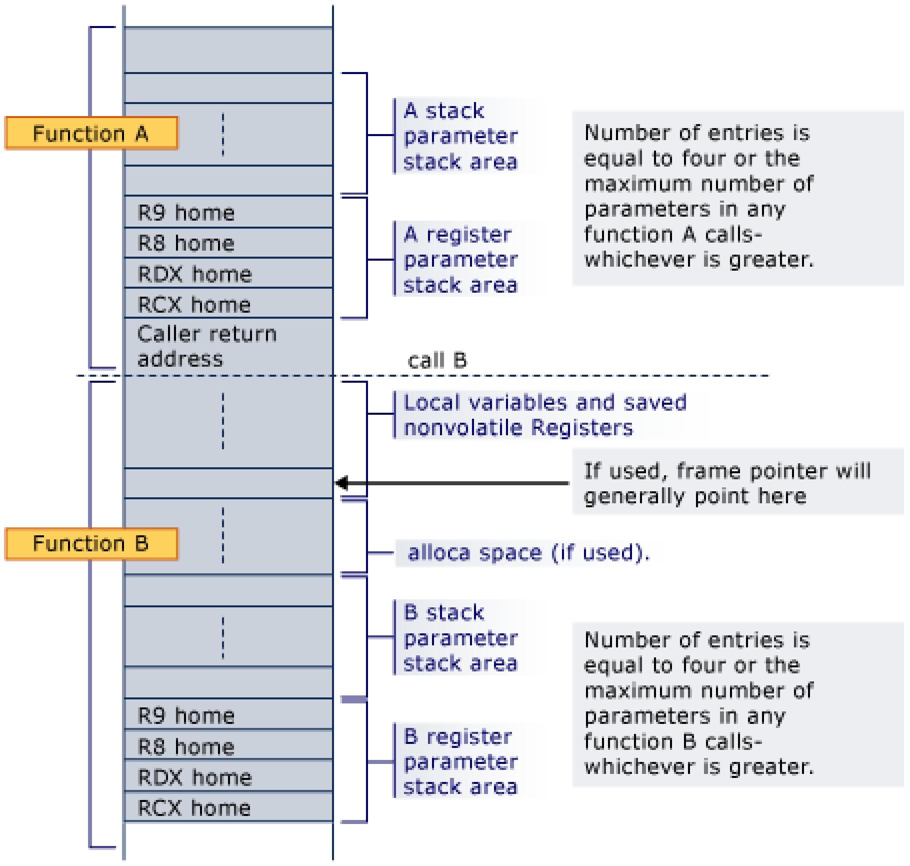
Stack Usage

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
All memory beyond the current address of RSP is considered volatile: The OS, or a debugger, may overwrite this memory during a user debug session, or an interrupt handler. Thus, RSP must always be set before attempting to read or write values to a stack frame.
This section discusses the allocation of stack space for local variables and the **alloca** intrinsic.

- Stack Allocation
- Dynamic Parameter Stack Area Construction
- Function Types
- malloc Alignment
- alloca

Stack Allocation

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
A function's prolog is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.
The parameter area is always at the bottom of the stack (even if alloca is used), so that it will always be adjacent to the return address during any function call. It contains at least four entries, but always enough space to hold all the parameters needed by any function that may be called. Note that space is always allocated for the register parameters, even if the parameters themselves are never homed to the stack; a callee is guaranteed that space has been allocated for all its parameters. Home addresses are required for the register arguments so a contiguous area is available in case the called function needs to take the address of the argument list (va_list) or an individual argument. This area also provides a convenient place to save register arguments during thunk execution and as a debugging option (for example, it makes the arguments easy to find during debugging if they are stored at their home addresses in the prolog code). Even if the called function has fewer than 4 parameters, these 4 stack locations are effectively owned by the called function, and may be used by the called function for other purposes besides saving parameter register values. Thus the caller may not save information in this region of stack across a function call.
If space is dynamically allocated (alloca) in a function, then a nonvolatile register must be used as a frame pointer to mark the base of the fixed part of the stack and that register must be saved and initialized in the prolog. Note that when alloca is used, calls to the same callee from the same caller may have different home addresses for their register parameters.
The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function Types for a certain class of frame functions.
The following is an example of the stack layout where function A calls a non-leaf function B. Function A's prolog has already allocated space for all the register and stack parameters required by B at the bottom of the stack. The call pushes the return address and B's prolog allocates space for its local variables, nonvolatile registers, and the space needed for it to call functions. If B uses alloca, the space is allocated between the local variable/nonvolatile register save area and the parameter stack area.


Function A
A stack parameter stack area
Number of entries is equal to four or the maximum number of parameters in any function A calls-whichever is greater.
R9 home
R8 home
RDX home
RCX home
A register parameter stack area
Caller return address
call B
Local variables and saved nonvolatile Registers
If used, frame pointer will generally point here
Function B
alloca space (if used).
B stack parameter stack area
Number of entries is equal to four or the maximum number of parameters in any function B calls-whichever is greater.
R9 home
R8 home
RDX home
RCX home
B register parameter stack area

When the function B calls another function, the return address is pushed just below the home address for RCX.

Dynamic Parameter Stack Area Construction

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
If a frame pointer is used, the option exists to dynamically create the parameter stack area. This is not currently done in the x64 compiler.

Function Types

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.

There are basically two types of functions. A function that requires a stack frame is called a frame function. A function that does not require a stack frame is called a leaf function.
A frame function is a function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling. It also requires a function table entry. A frame function requires a prolog and an epilog. A frame function can dynamically allocate stack space and can employ a frame pointer. A frame function has the full capabilities of this calling standard at its disposal.
If a frame function does not call another function then it is not required to align the stack (referenced in Section Stack Allocation).
A leaf function is one that does not require a function table entry. It cannot call any functions, allocate space, or save any nonvolatile registers. It is allowed to leave the stack unaligned while it executes.

malloc Alignment

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
malloc is guaranteed to return memory that's suitably aligned for storing any object that has a fundamental alignment and that could fit in the amount of memory that's allocated. A *fundamental alignment* is an alignment that's less than or equal to the largest alignment that's supported by the implementation without an alignment specification. (In Visual C++, this is the alignment that's required for a `double`, or 8 bytes. In code that targets 64-bit platforms, it's 16 bytes.) For example, a four-byte allocation would be aligned on a boundary that supports any four-byte or smaller object.
Visual C++ permits types that have *extended alignment*, which are also known as *over-aligned* types. For example, the SSE types __m128 and `__m256`, and types that are declared by using `__declspec(align(``n``))` where `n` is greater than 8, have extended alignment. Memory alignment on a boundary that's suitable for an object that requires extended alignment is not guaranteed by `malloc`. To allocate memory for over-aligned types, use _aligned_malloc and related functions.

alloca

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
_alloca is required to be 16-byte aligned and additionally required to use a frame pointer.
The stack that is allocated needs to include space below it for parameters of subsequently called functions, as discussed in Stack Allocation.

Prolog and Epilog

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
Every function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling must have a prolog whose address limits are described in the unwind data associated with the respective function table entry (see Exception Handling (x64)). The prolog saves argument registers in their home addresses if required, pushes nonvolatile registers on the stack, allocates the fixed part of the stack for locals and temporaries, and optionally establishes a frame pointer. The associated unwind data must describe the action of the prolog and must provide the information necessary to undo the effect of the prolog code.
If the fixed allocation in the stack is more than one page (that is, greater than 4096 bytes), then it is possible that the stack allocation could span more than one virtual memory page and, therefore, the allocation must be checked before it is actually allocated. A special routine that is callable from the prolog and which does not destroy any of the argument registers is provided for this purpose.
The preferred method for saving nonvolatile registers is to move them onto the stack before the fixed stack allocation. If the fixed stack allocation were performed before the nonvolatile registers were saved, then most probably a 32-bit displacement would be required to address the saved register area (reportedly, pushes of registers are just as fast as moves and should remain so for the foreseeable future in spite of the implied dependency between pushes). Nonvolatile registers can be saved in any order. However, the first use of a nonvolatile register in the prolog must be to save it.
The code for a typical prolog might be:

```
mov       [RSP + 8], RCX
push   R15
push   R14
push   R13
sub      RSP, fixed-allocation-size
lea      R13, 128[RSP]
...
```

This prolog stores the argument register RCX in its home location, saves nonvolatile registers R13-R15, allocates the fixed part of the stack frame, and establishes a frame pointer that points 128 bytes into the fixed allocation area. Using an offset allows more of the fixed allocation area to be addressed with one-byte offsets.
If the fixed allocation size is greater than or equal to one page of memory, then a helper function must be called before modifying RSP. This helper, __chkstk, is responsible for probing the to-be-allocated stack range, to ensure that the stack is extended properly. In that case, the previous prolog example would instead be:

```
mov       [RSP + 8], RCX
push   R15
push   R14
push   R13
mov      RAX,  fixed-allocation-size
call   __chkstk
sub      RSP, RAX
lea      R13, 128[RSP]
...
```

The __chkstk helper will not modify any registers other than R10, R11, and the condition codes. In particular, it will return RAX unchanged and leave all nonvolatile registers and argument-passing registers unmodified.

Epilog code exists at each exit to a function. Whereas there is normally only one prolog, there can be many epilogs. Epilog code trims the stack to its fixed allocation size (if necessary), deallocates the fixed stack allocation, restores nonvolatile registers by popping their saved values from the stack, and returns.

The epilog code must follow a strict set of rules for the unwind code to reliably unwind through exceptions and interrupts. This reduces the amount of unwind data required, because no extra data is needed to describe each epilog. Instead, the unwind code can determine that an epilog is being executed by scanning forward through a code stream to identify an epilog.

If no frame pointer is used in the function, then the epilog must first deallocate the fixed part of the stack, the nonvolatile registers are popped, and control is returned to the calling function. For example,

```
add      RSP, fixed-allocation-size
pop      R13
pop      R14
pop      R15
ret
```

If a frame pointer is used in the function, then the stack must be trimmed to its fixed allocation prior to the execution of the epilog. This is technically not part of the epilog. For example, the following epilog could be used to undo the prolog previously used:

```
lea      RSP, -128[R13]
; epilogue proper starts here
add      RSP, fixed-allocation-size
pop      R13
pop      R14
pop      R15
ret
```

In practice, when a frame pointer is used, there is no good reason to adjust RSP in two steps, so the following epilog would be used instead:

```
lea      RSP, fixed-allocation-size – 128[R13]
pop      R13
pop      R14
pop      R15
ret
```

These are the only legal forms for an epilog. It must consist of either an `add RSP,constant` or `lea RSP,constant[FPReg]`, followed by a series of zero or more 8-byte register pops and a return or a jmp. (Only a subset of jmp statements are allowable in the epilog. These are exclusively of the class of jmps with ModRM memory references where ModRM mod field value 00. The use of jmps in the epilog with ModRM mod field value 01 or 10 is prohibited. See Table A-15 in the AMD x86-64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions, for more info on the allowable ModRM references.). No other code can appear. In particular, nothing can be scheduled within an epilog, including loading of a return value.

Note that, when a frame pointer is not used, the epilog must use `add RSP,constant` to deallocate the fixed part of the stack. It may not use `lea RSP,constant[RSP]` instead. This restriction exists so the unwind code has fewer patterns to recognize when searching for epilogs.

Following these rules allows the unwind code to determine that an epilog is currently being executed and to simulate execution of the remainder of the epilog to allow recreating the context of the calling function.

Intrinsics and Inline Assembly

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.
One of the constraints for the x64 compiler is to have no inline assembler support. This means that functions that cannot be written in C or C++ will either have to be written as subroutines or as intrinsic functions supported by the compiler. Certain functions are performance sensitive while others are not. Performance-sensitive functions should be implemented as intrinsic functions.
The intrinsics supported by the compiler are described in Compiler Intrinsics.