5.
Calling conventions
for different C++ compilers and operating systems

By Agner Fog. Technical University of Denmark.
Copyright © 2004 - 2015. Last updated 2015-12-23.

Contents

1 Introduction 3
2 The need for standardization 5
3 Data representation 6
4 Data alignment 8
5 Stack alignment 9
6 Register usage 10
6.1 Can floating point registers be used in 64-bit Windows? 13
6.2 YMM vector registers 14
6.3 ZMM vector registers 15
6.4 Register usage in kernel code 15
7 Function calling conventions 16
7.1 Passing and returning objects 20
7.2 Passing and returning SIMD types 23
8 Name mangling 25
8.1 Microsoft name mangling 29
8.2 Borland name mangling 34
8.3 Watcom name mangling 35
8.4 Gnu2 name mangling 36
8.5 Gnu3-4 name mangling 38
8.6 Intel name mangling for Windows 40
8.7 Intel name mangling for Linux 41
8.8 Symantec and Digital Mars name mangling 41
8.9 Codeplay name mangling 41
8.10 Other compilers 42
8.11 Turning off name mangling with extern "C" 42
8.12 Conclusion 43
9 Exception handling and stack unwinding 43
10 Initialization and termination functions 44
11 Virtual tables and runtime type identification 44
12 Communal data 45
13 Memory models 45
13.1 16-bit memory models 45
13.2 32-bit memory models 46
13.3 64-bit memory models in Windows 46
13.4 64-bit memory models in Linux and BSD 46
13.5 64-bit memory models in Intel-based Mac (Darwin) 46
14 Relocation of executable code 47
14.1 Import tables 49
15 Object file formats 49
15.1 OMF format 49
15.2 COFF format 50
15.3 ELF format 51
15.4 Mach-O format 51
15.5 a.out format 52
15.6 Comparison of object file formats 52
15.7 Conversion between object file formats 52
15.8 Intermediate file formats 52

16 Debug information 53
17 Data endian-ness 53
18 Predefined macros 53
19 Available C++ Compilers 55
19.1 Microsoft 55
19.2 Borland 55
19.3 Watcom 55
19.4 Gnu 55
19.5 Clang 55
19.6 Digital Mars 55
19.7 Codeplay 55
19.8 Intel 55
20 Literature 56
20.1 ABI's for Unix, Linux, BSD and Mac OS X (Intel-based). 56
20.2 ABIs for Windows 56
20.3 Object file format specifications 57
21 Copyright notice 57
22 Acknowledgments 57

1 Introduction

This is the fifth in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.

2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.

3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.

5. Calling conventions for different C++ compilers and operating systems.

The latest versions of these manuals are always available from www.agner.org/optimize. Copyright conditions are listed on page 57 below.

The present manual describes differences between various C++ compilers that affect binary compatibility, such as data storage, function calling conventions, and name mangling. The function calling methods, name mangling schemes, etc. are described in detail for each compiler.

The purposes of publishing this information are:

- Point out incompatibilities between compilers.

- Make new compilers compatible with old ones.

- Solve compatibility problems between function libraries produced by different compilers.

- Facilitate linking different programming languages together.

- Facilitate the making of assembly subroutines that are compatible with multiple compilers and multiple operating systems.

- Solve compatibility problems for data stored in binary files.

- Facilitate the construction of debugging, profiling and disassembly tools.

- Facilitate the construction of object file conversion utilities.

- Provoke compiler vendors to use open standards.

- Inspire future standardization.

Hardware platforms covered:

- x86 microprocessors with 16 bit, 32 bit and 64 bit architectures from Intel, AMD, VIA and possibly other vendors.

The IA64 architecture, which is implemented in Intel's Itanium processor, is not compatible with the x86 architecture, and is not covered in this report.

Operating systems covered:

- DOS, 16 bit.
- Windows, 16 bit, 32 bit and 64 bit.
- Linux, 32 bit and 64 bit.
- FreeBSD etc. 32 bit and 64 bit.
- Mac OS X, Intel based, 32 bit and 64 bit.

C++ compilers tested:

- Borland, 16 bit v. 3.0 and 5.0
- Microsoft, 16 bit, v. 8.0
- Watcom, 16 bit v. 1.2
- Borland 32 bit v. 5.0
- Microsoft, 32 bit, v. 9.0 and 13.10
- Gnu, 32 bit, v. 2.95, 3.3.3, 4.1.0 and several other versions under Linux, FreeBSD and Windows.
- Watcom, 32 bit, v. 1.2
- Symantec, 32 bit, v. 7.5
- Digital Mars, 32 bit, v. 8.3.8
- Codeplay VectorC, 32bit, v. 2.1.7
- Intel, 32 bit for Windows and Linux, v. 8.1 and 9.1
- Microsoft, 64 bit, v. 14.00
- Gnu, 64 bit, v. 3.3.3 and 4.1.0 (Linux and FreeBSD)
- Intel, 64 bit for Windows and Linux, v. 8.1 and 9.1

This document provides information that is typically difficult to find. The documentation of calling conventions and binary interfaces of compilers and operating systems is often shamefully poor and sometimes completely absent. Name mangling schemes are rarely documented. For example, it is stated in Microsoft Knowledge Base that "Microsoft does not publish the algorithm its compilers use for name decoration because it may change in the future." (article number Q126845). However, the name mangling scheme has not changed much from the old 16-bit compiler to the newest 32-bit and 64-bit compilers, and it is unlikely to be changed in the future because of compatibility requirements.

As most of the information given here is based on my own experiments, it is obviously not authoritative, and it is not guaranteed to be accurate or complete. This document tells how things are, not how they are supposed to be. Some details appear to be the haphazard consequences of how compilers happen to be implemented rather than results of careful planning. Calling "conventions" may not be the most appropriate term in this case, but it may be necessary to copy the quirks of existing compilers when full compatibility is desired.

I have no knowledge about whether any information provided here is protected by patents or other legal restrictions, but I have found no specific patent markings on the compilers.

I have gathered this information mainly by converting C++ code to assembly. All the compilers I have tested are capable of converting C++ to assembly, either directly or via object files. The reader is encouraged to do your own research, if necessary, to get additional information needed or to clarify any questions you may have. The easiest way of doing this research is to make the compiler convert a C++ test file to assembly. Other possible methods are to use object file dump utilities, disassembly utilities, or provoke error messages from a linker. If you find any errors in this document then please let me know.

Please note that I don't have the time and resources to help people with their programming problems. If you Email me with such questions, you will not get any answer. You may send your questions to appropriate internet forums instead.

2 The need for standardization

In the days of the old DOS operating system, it was often possible to combine development tools from different vendors with few compatibility problems. With 32-bit Windows, the situation has gone completely out of hand. Different compilers use different data representations, different function calling conventions, and different object file formats. While static link libraries have traditionally been considered compiler-specific, the widespread use of dynamic link libraries (DLL's) has made the distribution of function libraries in binary form more common. Unfortunately, the standardization of data representation and calling conventions that would make DLL's compatible is still lacking.

In the Linux, BSD and Mac operating systems, there are fewer compatibility problems because a more or less official standard is defined. Most of this standard is followed by Gnu C++ version 3.x and later. Earlier versions of the Gnu compiler are not compatible with this.

Fortunately, there is a growing recognition of the need for standardization of application binary interfaces (ABI's). The ABI's for the new 64-bit operating systems are specified in much more detail than we have seen in older operating systems. However, some of these ABI's still lack specification of name mangling schemes and other details. Traditionally, compiler vendors have not published or standardized their name mangling schemes. A common excuse was that the object files would not be compatible anyway because of differences in data formats and calling conventions. Now that data formats and calling conventions are specified in the ABI's, there is no excuse any more for not publishing and standardizing name mangling schemes as well. It is my hope that this document will be a contribution towards this end.

Compilers and other development tools is an area where *de facto* standards play an important role. Almost all compilers for UNIX-like x86 platforms are designed to be compatible with the Gnu compiler. And the calling "conventions" of the Microsoft compiler has almost become a *de facto* standard for the Windows operating system. The C++ compilers from Intel, Symantec, Digital Mars and Codeplay are all designed to be binary compatible with Microsoft's C++ compiler, despite the fact that Microsoft has refused to publish important details. At least some of these compiler makers have relied on reverse engineering for obtaining the necessary information. There is a pressing need for publishing the relevant standards, and the present document is my contribution towards this end.

It is highly recommended that designers of development tools follow all available standards. Where no official standard exists, use an existing compiler for reference. Use the Microsoft compiler as a reference for Windows systems and the Gnu compiler as a reference for UNIX-like systems. For features that are not supported by these compilers, use the Intel compiler for reference. The calling conventions of these compilers may be considered *de facto* standards for Windows and UNIX platforms.

3 Data representation

Table 1. Data sizes

segment word size	16 bit			32 bit						64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
__int64				8	8			8	8	8	8	8	8
enum	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	16	16	16
__m64				8	8				8		8	8	8
__m128				16	16				16	16	16	16	16
__m256					32				32		32		32
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

Table 1 shows how many bytes of storage various objects use for different compilers.

Differences in data representation can cause problems when exchanging binary data files between programs, when exchanging data with a DLL compiled with a different compiler, and when porting C++ code that relies on a specific data format.

Bool

The type `bool` typically uses one byte of storage where all bits are significant. 0 indicates false and all other values indicate `true`. Most compilers will always store the value `true` as 1. The ABI for 64 bit Linux/BSD specifies that other values than 0 and 1 are allowed only for function parameters and returns, not for memory objects. The opposite would be more logical since the most likely source of Booleans with other values than 0 and 1 is uninitialized memory objects.

The optimal convention would be to never allow other values than 0 and 1. This would make it possible to implement Boolean expressions without the use of expensive branch instructions except where the evaluation of the second operand of `&&` or `||` has side effects. None of the compilers I have tested take advantage of the fact that the only possible values are 0 and 1, even if the performance could be improved significantly by relying on this fact.

Integers

Signed integers are stored in 2-complement representation. The size is 8, 16, 32 or 64 bits, except in bitfields that can have other sizes.

Floating point numbers

Floating point numbers are stored according to the IEEE-754 standard. The most significant bit of the mantissa is explicit (=1) in `long double` and implicit in `float` and `double`.

The x86 architecture specifies 10 bytes for `long double`. Microsoft compilers don't support this precision, but store `long double` as `double`, using 8 bytes. Other compilers use more than 10 bytes for the sake of alignment. The extra bytes are unused, even if subsequent objects would fit into this unused space. The 32-bit and 64-bit Intel compilers for Windows store long double as 8 bytes by default for compatibility with the Microsoft compiler. Use the option `/Qlong-double` to get 16 bytes long double in Intel compilers.

Member pointers

A class data member pointer basically contains the offset of the member relative to the beginning of the object. A member function pointer basically contains the address of the member function.

Data member pointers and member function pointers may use extra storage in the general case in order to account for rare cases of multiple inheritance etc. The minimum value in table 1 applies to simple cases, the maximum value applies to the case where the compiler doesn't have any information about the class other than its name. Some compilers have options to cover this case in different ways. The extra information is stored in ways that are poorly documented and poorly standardized. The "Itanium C++ ABI" includes more detailed information about the representation of member pointers. This information may apply to other platforms as well. More information on the implementation of member pointers in different compilers can be found in "Member Function Pointers and the Fastest Possible C++ Delegates", by Don Clugston, www.codeproject.com/cpp/FastDelegate.asp

Borland compilers add an offset of 1 to data member pointers in order to distinguish a pointer to the first data member from a `NULL` pointer, represented by 0. The other compilers have no offset, but represent a `NULL` data member pointer by the value -1.

1 and 2-byte types in Gnu compiler

Gnu compilers always zero-extend or sign-extend function return values to 32 bits if the values are less than 32 bits in order to conform to a certain interpretation of the C standard. The 64 bit Gnu compiler sign-extends signed values to 32 bits rather than to 64 bits. The extension to 32 bits appears to be completely superfluous since the calling function will repeat the zero-extension or sign-extension operation if needed rather than relying on the higher bits being valid.

Arrays and strings

Arrays are stored as consecutive objects in memory. No information about the size of the array is included. Multidimensional arrays are stored with the last index as least significant, i.e. changing fastest. Arrays are passed to functions as pointers without copying. C-style strings are stored as arrays with a terminating element of 0.

Most programming languages other than C and C++ store arrays and strings in ways that include a specification of the size.

Composite objects

Objects of structures and classes are stored by placing the data members consecutively in memory. Unused bytes may be inserted between elements and after the last element, if needed, for the sake of alignment. The requirements for alignment are described below.

Additional information for virtual tables and runtime type identification may be added, as described in chapter 11.

4 Data alignment

Table 2. Alignment of static data

segment word size	16 bit			32 bit					64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
1 byte char	1	1	1	1	4	1	1	1	4	1	4	1	4
2 byte int	2	2	2	4	4	2	2	2	4	4	4	2	4
4 byte int	2	2	4	4	4	4	4	4	4	4	4	4	4
8 byte int	2	2	8	8	8	4	8	8	8	8	8	8	8
float	2	2	4	4	4	4	4	4	4	4	4	4	4
double	2	2	8	8	8	4	8	8	8	8	8	8	8
long double	2	2	8		16	4	8	4	4		16	16	16
__m64				8	8				8		8	8	8
__m128				16	16				16	16	16	16	16
__m256					32				32		32		32
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	2	2	2										
big array	2	1-2	2-8	4-8	512	1-4	2-8	32	32	4-8	256	32	32
big structure	2	1	2	4	32	1	8	32	32	4	32	32	32

Table 2 shows the default alignment in bytes of static data. The alignment affects performance, but not compatibility.

Table 3. Alignment of structure members

segment word size	16 bit			32 bit					64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
1 byte char	1	1	1	1	1	1	1	1	1	1	1	1	1
2 byte int	2	1	2	2	2	1	2	2	2	2	2	2	2
4 byte int	2	1	2	4	4	1	4	4	4	4	4	4	4
8 byte int	2	1	2	8	8	1	8	4,8	8	8	8	8	8
float	2	1	2	4	4	1	4	4	4	4	4	4	4
double	2	1	2	8	8	1	8	8	8	8	8	8	8
long double	2	1	2		16	1	8	16	16		16	16	16
__m64					8				8		8		8
__m128					16				16		16		16
__m256					32				32		32		32
pointer	2	1	2	4	4	1	4	4	4	8	8	8	8
far pointer	2	1	2										

Table 3 shows the alignment in bytes of data members of structures and classes. The compiler will insert unused bytes, as required, between members to obtain this alignment. The compiler will also insert unused bytes at the end of the structure so that the total size of the structure is a multiple of the alignment of the element that requires the highest alignment. Many compilers have options to change the default alignments.

Differences in structure member alignment will cause incompatibility between different programs or modules accessing the same data and when data are stored in binary files.

The programmer can avoid such compatibility problems by ordering the structure members so that no unused bytes need to be inserted. Likewise, the padding at the end of the structure may be specified explicitly by inserting dummy members of the required size. The size of the virtual table pointer, if any, must be taken into account (see chapter 11).

5 Stack alignment

The stack pointer must be aligned by the stack word size at all times. Some systems require a higher alignment.

The Gnu compiler version 3.x and later for 32-bit Linux and Mac OS X makes the stack pointer aligned by 16 at every function call instruction. Consequently it can rely on `ESP` = 12 modulo 16 at every function entry. This alignment is not consistently implemented. It is specified in the Mac OS ABI, but nowhere else. The stack is not aligned when compiling with option `-Os` or `-mpreferred-stack-boundary=2`, but apparently the Gnu compiler erroneously relies on the stack being aligned by 16 despite these options. The Intel compiler (v. 9.1.038) for 32 bit Linux does not have the same alignment. (I have submitted bug reports to Gnu and Intel about this in 2006. In 2009 Intel added a `-falign-stack=assume-16-byte` option to ICC version 11.0 to fix the problem).

The stack is aligned by 4 in 32-bit Windows.

The 64 bit systems keep the stack aligned by 16. The stack word size is 8 bytes, but the stack must be aligned by 16 before any call instruction. Consequently, the value of the stack

pointer is always 8 modulo 16 at the entry of a procedure. A procedure must subtract an odd multiple of 8 from the stack pointer before any call instruction. A procedure can rely on these rules when storing XMM data that require 16-byte alignment. This applies to all 64 bit systems (Windows, Linux, BSD).

Where at least one function parameter of type `__m256` is transferred on the stack, Unix systems (32 and 64 bit) align the parameter by 32 and the called function can rely on the stack being aligned by 32 before the call (i.e. the stack pointer is 32 minus the word size modulo 32 at the function entry). This does not apply if the parameter is transferred in a register.

Various methods for aligning the stack are described in Intel's application note AP 589 "Software Conventions for Streaming SIMD Extensions", "Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel® C/C++ Compiler", and "IA-32 Intel ® Architecture Optimization Reference Manual".

6 Register usage

Table 4. Register usage

	16 bit DOS, Windows	32 bit Windows	32 bit Linux, BSD, Mac OS	64 bit Windows	64 bit Linux, BSD, Mac OS
scratch registers	AX, BX, CX, DX, ES, ST(0)-ST(7)	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, K0-K7	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, K0-K7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), K0-K7, XMM0-XMM5, All YMM/ZMM registers except lower 128 bits of XMM6-XMM15	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7) K0-K7, XMM0-XMM15, YMM0-YMM15 ZMM0-ZMM31
callee-save registers	SI, DI, BP, DS	EBX, ESI, EDI, EBP	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, XMM6-XMM15, but nothing beyond the lower 128 bits of vector registers	RBX, RBP, R12-R15
registers for parameter transfer	see table 5	see table 5	see table 5	RCX, RDX, R8, R9, XMM0-XMM3, YMM0-YMM3, ZMM0-ZMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7, YMM0-YMM7
registers for return	AX, DX, ST(0)	EAX, EDX, ST(0), XMM0, YMM0, ZMM0	EAX, ST(0), XMM0, YMM0, ZMM0	RAX, ST(0), XMM0, YMM0, ZMM0	RAX, RDX, ST(0), XMM0, YMM0, ZMM0

The rules for register usage depend on the operating system, as shown in table 4. Scratch registers are registers that can be used for temporary storage without restrictions (also called caller-save or volatile registers). Callee-save registers are registers that you have to save before using them and restore after using them (also called non-volatile registers). You can rely on these registers having the same value after a call as before the call. For example, a function using EBP may look like this

```
FunctionUsingEBP PROC NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, 52
        ...
        mov     eax, [ebp+8]
        push    eax
        call    AnotherFunction
        mov     esp, ebp
        pop     ebp
        ret
FunctionUsingEBP ENDP
```

Here, EBP is saved on the stack in the beginning of the function and restored in the end. The code relies on EBP being unchanged after the call to AnotherFunction. EAX is also used, but doesn't have to be saved.

It is more efficient to use registers for transferring parameters to a function and for receiving the return value than to store these values on the stack. Some calling conventions use certain registers for parameter transfer, but the rules for which registers to use are compiler-specific in 16-bit and 32-bit systems. In 64-bit systems, the use of registers for parameter transfer is standardized. All systems use registers for return values if the returned object fits into the registers that are assigned for this purpose. See the next chapter for details.

Segment registers

You only have to care about segment registers in 16-bit mode. DS has to be saved and restored if you change it. ES can be changed freely. In DOS programs, ES can have any value. In 16-bit Windows, ES can only have values that are valid segment descriptors. It is not allowed to use ES for other purposes.

In 32-bit and 64-bit mode, it is not allowed to change any segment register, not even temporarily. CS, DS, ES and SS all point to the flat segment group. FS is used for a thread environment block in Windows and for thread specific data in Linux. GS is used for a processor control region in 64-bit Windows. It is unused but reserved in 32-bit Windows. It is probably unused in 32-bit Linux.

Arithmetic flags

The rules for the arithmetic flags (zero flag, carry flag, etc.) are the same as for scratch registers. These flags need not be saved. Some programming languages (not C++) use the carry flag for Boolean returns.

Direction flag

The rules for the direction flag is the same in all systems. The direction flag is cleared by default. If the direction flag is set, then it must be cleared again before any call or return. Some compilers and subroutine libraries rely on the direction flag always being clear (Microsoft, Watcom, Digital Mars) while other systems use the double-safe strategy of always leaving the direction flag cleared, but not relying on receiving it cleared (Borland, Gnu).

There is a slight possibility that some programmers may have ignored the rule for the direction flag. Therefore, it may be wise to use the double-safe strategy and clear the

direction flag before using it if the code will be linked together with modules from unreliable sources.

Interrupt flag

It is not allowed to turn off the interrupt flag in programs running in multi-user systems because this would make it possible to steal unlimited amounts of CPU time from other processes. It may be possible to turn off the interrupt flag in console mode programs running under Windows 98 and earlier operating systems without network. But since programs written for old operating systems are likely to be run under newer operating systems, it is reasonable to say that it is never possible to turn off the interrupt flag in application programs.

Floating point registers

The floating point registers `ST(0)`-`ST(7)` need not be saved. The register stack must be emptied before any call or return, except for registers used for return values. The 64-bit Microsoft compiler does not use `ST(0)`-`ST(7)`.

MMX registers

The `MM0`-`MM7` registers are aliased on the lower 64 bits of the floating point x87 registers `ST(0)`-`ST(7)`. There is no callee save rule, so the MMX registers can be used freely in a function that doesn't use the floating point registers. The register set must be left in x87 mode. Therefore, it is required to issue an `EMMS` instruction (or `FEMMS`) before calling any other (ABI compliant) function and before returning. Unfortunately, not all compilers do so. Therefore, the use of MMX registers and the `__m64` type should be avoided if possible. The 64-bit Microsoft compiler does not use `MM0`-`MM7`.

Floating point control word and MXCSR register

The floating point control word and bit 6-15 of the `MXCSR` register must be saved and restored before any call or return by any procedure that needs to modify them, except for procedures that have the purpose of changing these.

Deviating from the conventions

It is possible to deviate from the register usage conventions in an isolated section of code as long as all interfaces to other parts of the code conform to the conventions. Some compilers do this in a process known as whole program optimization. Any deviation from the conventions must be well documented. Deviations from good programming practice are justified only if a significant gain in speed can be obtained.

ABI for 64 bit Windows has been changed

Early versions of the 64 bit Windows ABI specified that only the lower 64 bits of `XMM6`-`XMM15` have callee-save status while later versions specify that all 128 bits must be saved. An MSDN document dated June 14, 2004 specifies the now-obsolete rule while a later version dated February 18, 2005 specifies the new rule without comments on the change. The change is mentioned in Intel compiler manuals. The 2005 standard is supported by Intel C++ compiler version 8.1.015 and later. My tests show that Microsoft compiler version 14.00.2228.2 uses the obsolete convention, while version 14.00.40310.41 uses the new convention. I have no information about Microsoft compiler versions between these two numbers.

Microsoft 16-bit compiler

The 16-bit Microsoft compiler returns `float` and `double` through a static memory location pointed to by `AX`. `long double` is returned in `ST(0)`.

Watcom compiler

The Watcom compiler doesn't conform to the register usage conventions in table 4. The only scratch register is EAX. All other general purpose registers are callee-save, except for EBX, ECX, EDX when used for parameter transfer, and ESI when used for return pointer. (In 16-bit mode, ES is also a scratch register). It is possible to specify any other register usage by the use of pragmas in the Watcom compiler.

How many registers should be callee-save?

I have never seen a study of the optimal ratio of caller-save to callee-save registers. Scratch registers are preferred for temporary values that do not have to be saved across a function call. Functions that do not call any other functions (leaf functions) and functions that have a low probability of calling other functions (effective leaf functions) will prefer to use scratch registers. If a function has more than one call to other functions or calls another function inside a loop, and if it needs to store values of local variables across these function calls, then the function becomes simpler by using callee-save registers. If the called functions need to use the same registers, then there is no advantage in speed, but possibly in size. If the called functions can use other registers, then there is an advantage in speed as well. Since leaf functions are the most likely ones to be speed-critical, it is reasonable to have as many scratch registers as are typically needed in a leaf function. Functions that call other functions, on the other hand, are likely to have more variables and thus need more registers. Balancing these considerations, I would expect the optimal fraction of scratch registers to be between a half and two thirds for architectures that have few registers, and somewhat lower if there are plenty of registers.

Some compilers have capabilities for whole-program-optimization, and we can expect such features to become more common in the future. If the compiler has information about the register needs of both caller and callee at the same time, then it can allocate different registers to the two functions so that no registers need to be saved. In this case, the optimal solution is to define callee-save registers only for system functions, device drivers and library functions.

The size of vector registers will be increased to 512 bits in the future AVX-512 instruction set, and probably increased later to 1024 or 2048 bits. These extensions will use automatic zero-extension of the 256-bit YMM registers. It is therefore not useful to have callee-save status for registers that can be expected to be bigger in future instruction sets if compatibility with existing code is needed.

6.1 Can floating point registers be used in 64-bit Windows?

There has been widespread confusion about whether 64-bit Windows allows the use of the floating point registers ST(0)-ST(7) and the MM0 - MM7 registers that are aliased upon these. One early technical document found at Microsoft's website says "x87/MMX registers are unavailable to Native Windows64 applications" (Rich Brunner: Technical Details Of Microsoft® Windows® For The AMD64 Platform, Dec. 2003). An AMD document says: "64-bit Microsoft Windows does not strongly support MMX and 3Dnow! instruction sets in the 64-bit native mode" (Porting and Optimizing Multimedia Codecs for AMD64 architecture on Microsoft® Windows®, July 21, 2004). A document in Microsoft's MSDN says: "A caller must also handle the following issues when calling a callee: [...] Legacy Floating-Point Support: The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are volatile. That is, these legacy floating-point stack registers do not have their state preserved across context switches" (MSDN: Kernel-Mode Driver Architecture: Windows DDK: Other Calling Convention Process Issues. Preliminary, June 14, 2004; February 18, 2005). This description is nonsense because it confuses saving registers across function calls and saving registers across context switches. Some versions of the Microsoft assembler ml64 (e.g. v. 8.00.40310) gives the following message when attempts are made to use floating point registers in 64 bit mode: "error A2222: x87 and MMX instructions disallowed; legacy FP state not saved in Win64".

However, a public discussion forum quotes the following answers from Microsoft engineers regarding this issue: "From: Program Manager in Visual C++ Group, Sent: Thursday, May 26, 2005 10:38 AM. It does preserve the state. It's the DDK page that has stale information, which I've requested it to be changed. Let them know that the OS does preserve state of x87 and MMX registers on context switches." and "From: Software Engineer in Windows Kernel Group, Sent: Thursday, May 26, 2005 11:06 AM. For user threads the state of legacy floating point is preserved at context switch. But it is not true for kernel threads. Kernel mode drivers can not use legacy floating point instructions." (www.planetamd64.com/index.php?showtopic=3458&st=100).

The issue has finally been resolved with the long overdue publication of a more detailed ABI for x64 Windows in the form of a document entitled "x64 Software Conventions", well hidden in the bin directory (not the help directory) of some compiler packages. This document says: "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." The same text has later appeared at the Microsoft website (msdn2.microsoft.com/en-us/library/a32tsf7t(VS.80).aspx).

My tests indicate that these registers are saved correctly during task switches and thread switches in 64-bit mode, even in an early beta version of x64 Windows.

The Microsoft C++ compiler version 14.0 never uses these registers in 64-bit mode, and doesn't support long double precision. The Intel C++ compiler for x64 Windows supports long double precision and `__m64` in version 9.0 and later, while earlier versions do not.

The conclusion is that it is safe to use floating point registers and MMX registers in 64-bit Windows, except in kernel mode drivers.

6.2 YMM vector registers

The 128-bit XMM registers are extended to 256-bit YMM registers in the AVX instruction set. The use of YMM registers is supported in Windows 7, Windows Server 2008 R2 and Linux kernel version 2.6.30 and later.

A preliminary ABI published by Intel (see literature p. 56) is supported by operating systems and compilers. The YMM registers do not have callee-save status, except for the lower half of YMM6-YMM15 in 64-bit Windows, where XMM6-XMM15 have callee-save status. None of the vector registers have callee save status in Linux.

The corresponding vector types are named `__m256`, `__m256d`, `__m256i`. These should preferably be aligned by 32, but some systems allow alignment by 16. The System V ABI for 64-bit Unix systems requires alignment by 32. The System V ABI for 32-bit Unix does not mention __m256, but tests show that it is aligned by 32. Apparently, Windows allows alignment by 16. In both Windows and Linux, these registers are aligned by 32 when transferred on the stack as function parameters.

Functions that use YMM registers should issue the instruction `VZEROUPPER` or `VZEROALL` before calling any ABI compliant function and before returning to any ABI compliant function. `VZEROUPPER` is used if the ABI specifies that some of the XMM registers must be preserved (64-bit Windows) or if an XMM register is used for parameter transfer or return value. `VZEROALL` can optionally be used instead of `VZEROUPPER` in other cases. Obviously, neither `VZEROUPPER` nor `VZEROALL` is used before calling a function that uses full YMM registers for parameter transfer or before returning from a function that uses the entire `YMM0` register for return value. Failure to use `VZEROUPPER` or `VZEROALL` will result in poor performance but no error. See manual 2: "Optimizing subroutines in assembly language" for

an explanation, and the discussion in Intel Software Network Forum:
http://software.intel.com/en-us/forums/showthread.php?t=59291

6.3 ZMM vector registers

The size of vector registers will be increased to 512 bits in the future AVX-512 instruction set, and probably later increased to 1024 or 2048 bits.

The AVX-512 instruction set increases the number of vector registers in 64-bit mode to 32 registers named ZMM0 - ZMM31. In 32-bit mode, there are only 8 ZMM registers. The 512-bit ZMM registers are stored to memory locations aligned by 64, according to the ABI's, although unaligned access is possible.

In addition, there are eight new mask registers named k0 - k7. These registers are specified as 64 bits, but only 16 bits are used in AVX-512. This means that these mask registers are expected to grow to 64 bits in future extensions, but not beyond 64 bits.

None of the mask registers have callee save status in Linux or Windows. The AVX-512 instruction set has no instruction for storing all 64 bits of a mask register. Therefore, it is impossible to use these registers in an interrupt handler in a way that is compatible with future extensions, unless the entire register state is saved.

The mask registers are treated as integers in function calls. Integer registers are used when mask registers are specified as function parameters or return.

The future Intel memory protection instructions, MPX, adds another four new registers, BND0-BND3 of 2x64 bits each.

6.4 Register usage in kernel code

Register use is more restricted in device drivers and kernel code than in application code. The operating system may save time by not saving all registers during interrupts and task switches.

The `FXSAVE` and `FXRSTOR` instructions can save the x87, MMX and XMM registers during a task switch, but not the YMM or ZMM registers. Instead, it is necessary to use the `XSAVE` and `XRESTOR` instructions for saving the YMM/ZMM registers and future larger registers during a task switch. It is not sufficient to store each vector register individually because this would be incompatible with future extensions of the register size. Any instruction that writes to a YMM/ZMM register will clear all bits beyond 256/512 in future register extensions larger than 256/512 bits. Using `XSAVE` and `XRESTOR` is the only way of saving the vector registers that is compatible with future extensions. The operating system must use CPUID to determine the necessary size of the save buffer.

Interrupt service routines under Windows and Linux

Interrupt service routines should be fast. It would take too much time to save and restore all registers in an interrupt service routine. The use of x87 and vector registers is prohibited in interrupt service routines in most operating systems, including Linux and 32-bit Windows. It may be possible to use the XMM registers, but not the YMM registers, in interrupt service routines in 64-bit Windows, but it is not recommended to use XMM registers here because of the extra cost of changing the YMM register state.

Device drivers under Windows

The x87 floating point registers can be used in device drivers in 32-bit Windows if saved and restored with `KeSaveFloatingPointState` and `KeRestoreFloatingPointState`. It is strictly prohibited to use x87 registers and MMX registers in 64-bit Windows device drivers.

XMM registers can be used in Windows device drivers. In 32-bit Windows it is necessary to save these registers using `KeSaveFloatingPointState` and `KeRestoreFloatingPointState` or `KeSaveExtendedProcessorState` and `KeRestoreExtendedProcessorState`. In 64-bit Windows device drivers it is sufficient to obey the general register usage rules when using XMM registers.

YMM registers can be used in 32-bit and 64-bit Windows device drivers if saved and restored with `KeSaveExtendedProcessorState` and `KeRestoreExtendedProcessorState`. These functions will allocate a buffer of sufficient size in case of future register extensions.

Device drivers under Linux

Linux systems use lazy saving of floating point registers and vector registers. This means that these registers are not saved and restored on every task switch. Instead they are saved/restored on the first access after a task switch. This method saves time in case no more than one thread uses these registers. The lazy saving scheme is not supported in kernel mode. Any device driver that attempts to use these registers improperly will cause an exception that will probably make the system crash. A device driver that needs to use vector registers must first save these registers by calling the function `kernel_fpu_begin()` and restore the registers by calling `kernel_fpu_end()` before returning or sleeping. These functions also prevent pre-emptive interruption of the device driver which could otherwise mess up the registers. `kernel_fpu_begin()` saves all floating point registers and vector registers if available.

There is no red zone in 64-bit Linux kernel mode.

The programmer should be aware of these restrictions if calling any other library than the system kernel libraries from a device driver.

7 Function calling conventions

Table 5. Function calling conventions

segment word size	calling convention, operating system, compiler	parameters in registers	parameter order on stack	stack cleanup by	comments
16 bit	`cdecl`		C	caller	
	`pascal`		Pascal	function	
	`fastcall` Microsoft (non-member)	`ax`, `dx`, `bx`	Pascal	function	return pointer in `bx`
	`fastcall` Microsoft (member function)	`ax`, `dx`	Pascal	function	`this` on stack low address. return pointer in `ax`
	`fastcall` Borland	`ax`, `dx`, `bx`	Pascal	function	`this` on stack low address. return ptr on stack high addr.
	Watcom	`ax`, `dx`, `bx`, `cx`	C	function	return pointer in `si`
32 bit	`cdecl`		C	caller	
	`stdcall`		C	function	
	`pascal`		Pascal	function	
	Gnu		C	hybrid	Stack possibly aligned by 16. See p. 9
	`fastcall` Microsoft	`ecx`, `edx`	C	function	return pointer on stack if not member function

	fastcall Gnu	ecx, edx	C	function	
	fastcall Borland	eax, edx, ecx	Pascal	function	
	thiscall Microsoft	ecx	C	function	default for member functions
	Watcom	eax, edx, ebx, ecx	C	function	return pointer in esi
64 bit	Windows (Microsoft, Intel)	rcx/zmm0, rdx/zmm1, r8/zmm2, r9/zmm3	C	caller	Stack aligned by 16. 32 bytes shadow space on stack. The specified registers can only be used for parameter number 1, 2, 3 and 4, respectively.
	Linux, BSD, Mac (Gnu, Intel)	rdi, rsi, rdx, rcx, r8, r9, zmm0-7	C	caller	Stack aligned by 16. Red zone below stack.

The way of transferring parameters to a function is not always as well standardized as we would wish, as table 5 shows. In many cases it is possible to specify a particular calling convention in a C++ declaration, for example:

```
int __stdcall SomeFunction (float a);
```

In 16 bit and 32 bit mode, we have the same calling conventions in different operating systems, but some differences between different brands of compilers, especially for the __fastcall convention and for member functions. In 64 bit mode, the different operating systems use different calling conventions, but I would not expect differences between different compilers because all details are defined in the official ABI's.

The entries in table 5 need some explanations. Segment word size defines the hardware platform. 16 bit refers to DOS and Windows 3.x and earlier. 32 bit refers to Windows 95 and later, Linux and BSD for the 32-bit x86 processors. 64 bit refers to Windows, Linux and BSD for the x64 processor architecture.

Calling convention is the name of the calling convention. __cdecl, __stdcall, __pascal and __fastcall can be specified explicitly in C++ function declarations for compilers that support these conventions. __cdecl is the default for applications and static libraries. __stdcall is the default for system calls (including Windows API calls) and recommended for library DLL's in 32-bit Windows. __thiscall is used by default in Microsoft compilers for member functions in 16 and 32 bit mode. Microsoft, Borland, Watcom and Gnu are brands of compilers. Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu. Symantec, Digital Mars and Codeplay compilers are compatible with Microsoft. In 64 bit mode, there is one default calling convention for each operating system, while other calling conventions are rare in 64 bit mode.

Parameters in registers specifies which registers are used for transferring parameters. ecx, edx means that the first parameter goes into ecx, the second parameter goes into edx, and subsequent parameters are stored on the stack. Parameter types that do not fit into the registers are stored on the stack. In general, all integer types, bool, enum and pointers can be transferred in the general purpose registers. References are treated as identical to pointers in all respects. Arrays are transferred as pointers. Float and double types are transferred in XMM registers in 64 bit mode, otherwise on the stack. Long doubles, structures, classes and unions may be transferred on the stack or through pointers if they do not fit into registers. The rules for deciding whether an object is transferred in registers, on the stack, or through a pointer are explained below. Where no register is specified in table 5, all parameters go on the stack. Return parameters are returned in registers as specified in chapter 6. Composite objects are returned as specified below.

Parameter order on stack. The Pascal order means that the first parameter has the highest address on the stack and the last parameter has the lowest address, immediately above the return address. If parameters are put on the stack by push instructions then the first parameter is pushed first because the stack grows downwards. The C order is opposite: The first parameter has the lowest address, immediately above the return address, and the last parameter has the highest address. This method was introduced with the C language in order to make it possible to call a function with a variable number of parameters, such as `printf`.

Each parameter must take a whole number of stack entries. If a parameter is smaller than the stack word size then the rest of that stack entry is unused. Likewise, if a parameter is transferred in a register that is too big, then the rest of that register is unused.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), then parameters of type `float` are converted to `double`, `char` and `short int` are converted to `int`.

Stack cleanup by. Specifies whether the stack space used by parameters is freed by the caller or by the called function. If *n* bytes of stack space is used for parameters and the called function has the responsibility for stack cleanup, then this function must return with a `ret n` instruction, otherwise `ret 0`. The 32-bit Gnu compiler uses a hybrid of these two methods: An object return pointer (see below) must be removed from the stack by the called function, all other parameters are removed by the caller.

If stack cleanup is the responsibility of the caller, and if speed is important, then it may be advantageous for the caller to leave the stack pointer where it is after the call and put parameters for a subsequent function call on the stack by `mov` instructions rather than by `push` instructions.

Further rules

Member functions (Applies to all C++ compilers and operating systems). All member functions receive a pointer to the object as an implicit parameter, known as `this` in C++. This pointer comes before the explicit parameters, usually as the first parameter. Constructors must return `this` in the return register.

Returning objects (Applies to all compilers and operating systems). Objects that do not fit into the return registers are returned to a storage space supplied by the caller. The caller must supply a return pointer as an implicit parameter to the called function if this is necessary. The same pointer is returned in the return register. The rules for deciding whether an object is returned in registers or through a return pointer are explained below for each platform.

A member function that returns an object can have two implicit parameters, a return pointer and a `this` pointer. In Microsoft compilers and 64 bit Windows, the `this` pointer is the first parameter, the return pointer is the second parameter, and all explicit parameters come thereafter. In Borland and Gnu compilers and in 64 bit Linux and BSD, the return pointer is the first parameter, the `this` pointer is the second parameter, and all explicit parameters come thereafter (this order is compatible with C).

Prolog and epilog. Some systems have specific rules for how the function prolog and epilog should be constructed in order to support stack unwinding. See chapter 9.

64 bit Windows has more rules. The first parameter goes into `rcx` or `xmm0`; the second parameter goes into `rdx` or `xmm1`, etc. This means that if the first parameter is a float in `xmm0` and the second parameter is an integer, then the latter goes into `rdx`, while `rcx` is unused. The maximum number of parameters that can be transferred in registers is four in total, not four integers plus four floats.

The caller must reserve 32 bytes of stack space as "shadow space" for register parameters, even if there are no parameters. The 32 bytes of shadow space come immediately after the return address. Any parameters on the stack come after the 32 empty bytes. The intended purpose of the shadow space is as a "home address" for the register parameters which the called function can use for storing the register parameters in case the registers are used for something else. The caller does not need to put anything into the shadow space. Since the shadow space is owned by the called function, it is safe to use these 32 bytes of shadow space for any purpose by the called function. Even a function without parameters can rely on having 32 bytes of storage space after the return address.

The shadow space is often used by compilers for storing the register parameters. I haven't seen a compiler using the shadow space for anything else, although it would be perfectly legal to do so.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), and a parameter of type `double` is passed in an XMM register (`float` is converted to `double`), then the corresponding integer register must contain the same value (not converted to `int`). This does not apply to parameters passed on the stack.

64 bit Linux, BSD and Mac. This system has six integer registers and eight XMM registers for parameter transfer. This means that a maximum of 14 parameters can be transferred in registers in 64 bit Linux, BSD and Mac, while 64 bit Windows allows only 4. There is no shadow space on the stack. Instead there is a "red zone" below the stack pointer that can be used for temporary storage. The red zone is the space from `[rsp-128]` to `[rsp-8]`. A function can rely on this space being untouched by interrupt and exception handlers (except in kernel code). It is therefore safe to use this space for temporary storage as long as you don't do any `push` or `call` instructions. Everything stored in the red zone is destroyed by function calls. The red zone is not available in Windows.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), then `rax` must indicate the number of XMM registers used for parameter transfer. Valid values are 0 - 8. A value of `rax` that is higher than the actual number of XMM registers used is allowed as long as it doesn't exceed 8.

`sysenter` calls use `r10` instead of `rcx` for parameter transfer and `rax` for function number.

Hot patching support

Hot patching is a mechanism in Windows that allows any function to be replaced by a security patch without restarting the process that uses the function. If support for hot patching is desired then there must be at least 6 unused bytes before the function entry, and the first instruction in the function must be at least two bytes long. In 32-bit Windows, the compiler may insert a 2-bytes NOP (`MOV EDI,EDI`) in the beginning of the function. In 64-bit Windows the compiler inserts a REX.W prefix before the first instruction if it is a push instruction to make it two bytes long.

7.1 Passing and returning objects

Table 6. Methods for passing structure, class and union objects

segment word size	16 bit			32 bit						64 bit	
compiler	Microsoft	Borland	Watcom	Microsoft	Borland	Borland	Gnu v.3 or later		Watcom	Windows	Linux, BSD, Mac
calling convention	all	all		all	default	fastcall	default	fastcall	default		
max number of integer registers used for transfer of an object	0	0	2	0	0	1	0	1	1	1	2
max number of XMM registers used for transfer of an object	0	0	0	0	0	0	0	0	0	0	2
simple structure, class or union	S	S	I	S	S	I	S	I	I	IZ	R
size not a power of 2	S	S	S	S	S	S	S	S	S	PI	R
contains mixed int and f.p.	S	S	S	S	S	S	S	S	S	IZ	R
contains long double	S	S	S	S	S	S	S	S	S	IZ	S
has member function	S	S	I	S	S	I	S	I	I	IZ	R
has constructor	S	S	I	S	S	S	S	I	I	IZ	R
has copy constructor	S	S	PI	S	S	S	PS	PI	PI	PI	PI
has destructor	S	S	PI	S	S	I	PS	PI	PI	IZ	PI
has virtual	S	S	PI	S	S	S	PS	PI	PI	PI	PI
has inheritance	S	S	I	S	S	I	S	I	I	IZ	R
has no data	S	S	I	S	S	I	S	I	I	IZ	S

Symbols:

S: Copy of entire object transferred on the stack.

PI: Temporary copy referenced by pointer in register. If no vacant register, use PS.

PS: Temporary copy referenced by pointer on stack.

I: Entire object transferred in integer registers. Use S if too big or not enough vacant registers.

IZ: Entire object transferred in integer register, zero-extended to register size. Use PI if too big. Use PS if no vacant register.

R: Entire object is transferred in integer registers and/or XMM registers if the size is no bigger than 128 bits, otherwise on the stack. Each 64-bit part of the object is transferred in an XMM register if it contains only float or double, or in an integer register if it contains integer types or mixed integer and float. Two consecutive floats can be packed into the lower half of one XMM register. Consecutive doubles are not packed. No more than 64 bits of each XMM register is used. Use S if not enough vacant registers for the entire object. Examples: `int` and `float`: `RDI`, `int` and `double`: `EDI` and `XMM0`, four `float`s: `XMM0` and `XMM1`.

There are several different methods to transfer a parameter to a function if the parameter is a structure, class or union object. A copy of the object is always made, and this copy is transferred to the called function either in registers, on the stack, or by a pointer, as specified in table 6. The symbols in the table specify which method to use. S takes precedence over I and R. PI and PS take precedence over all other passing methods.

As table 6 tells, an object cannot be transferred in registers if it is too big or too complex. For example, an object that has a copy constructor cannot be transferred in registers because the copy constructor needs an address of the object. The copy constructor is called by the caller, not the callee.

Objects passed on the stack are aligned by the stack word size, even if higher alignment would be desired. Objects passed by pointers are not aligned by any of the compilers studied, even if alignment is explicitly requested. The 64bit Windows ABI requires that objects passed by pointers be aligned by 16.

An array is not treated as an object but as a pointer, and no copy of the array is made, except if the array is wrapped into a structure, class or union.

The 64 bit compilers for Linux differ from the ABI (version 0.97) in the following respects: Objects with inheritance, member functions, or constructors can be passed in registers. Objects with copy constructor, destructor or virtual are passed by pointers rather than on the stack.

The Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu.

Table 7. Methods for returning structure, class and union objects (except intrinsic vectors __m64, __m128, __m256, __m512)

segment word size	**16 bit**			**32 bit**					**64 bit**	
compiler	Microsoft	Borland	Watcom	Microsoft	Borland	Watcom	Gnu except Mac OS	Mac OS	Windows	Linux, BSD, Mac
max number of integer registers used for return	0	2	2	2	1	1	2	2	1	2
max number of ZMM registers used for return	0	0	0	0	0	0	0	0	0	2
max number of f.p. registers used for return	0	0	0	0	0	0	0(1)	0	0	1
simple structure, class or union	P	I	I	I	I	I	I	I	I	R
bigger than max registers	P	PF	PSI	PS	P	PSI	P	P	P	P
size not a power of 2	P	PF	PSI	PS	P	PSI	P	I	P	R,X
contains only f.p.	P	I	I	I	I	I	P(F 1)	I?	I	X
contains mixed int and f.p.	P	PF	PSI	I	P	PSI	I	I	I	R
contains mixed float and double	P	PF	PSI	PS	P	PSI	P	P	P	X
contains only one long double	P	PF	PSI	I	P	PSI	P(F 1)	P	I	F
contains only one SIMD type				P			P	P	P	Y
contains mixed long double and other	P	PF	PSI	PS	P	PSI	P	P	P	P
has member function	P	I	I	I	I	I	P(I)	I	I	R
has inheritance	P	I	I	P	I	I	P(I)	I	P	R
has constructor	P	PF	I	P	P	I	P(I)	I	P	R
has copy constructor	P	PF	PA X	P	P	PA X	P	P	P	P
has destructor	P	PF	PA X	P	P	PA X	P	P	P	P
has virtual	P	PF	PA	P	P	PA	P	P	P	P

			X			X				
has no data	P	I	I	0	I	I	P(0)	P	0	I

Symbols:

I	Returned in integer registers
X	Returned in `XMM` registers
Y	Returned in `XMM0` or `YMM0` or `ZMM0`
F	Returned in `ST(0)` register
F1	If one `float`, `double` or `long double`, use `ST(0)`, otherwise I.
R	The entire object is returned in integer registers `RAX` and `RDX`, and/or `XMM` registers `XMM0` and `XMM1`, if the size is no bigger than 128 bits, otherwise on the stack. Each 64-bit part of the object is transferred in an `XMM` register if it contains only `float` or `double`, or in an integer register if it contains integer types or mixed integer and float. Two consecutive `float`'s can be packed into the lower half of one `XMM` register. Consecutive `double`'s are not packed. No more than 64 bits of each `XMM` register is used. Use P if not enough vacant registers. Examples: `int` and `float`: `RAX`, `int` and `double`: `EAX` and `XMM0`, four `float`s: `XMM0` and `XMM1`.
P	Pointer to temporary memory space passed to function. Pointer may be passed in register if `fastcall` or 64-bit mode, otherwise on stack. Same pointer is returned in `AX`, `EAX` or `RAX`.
PS	Pointer to temporary memory space passed to function. Pointer is passed on stack, even if `fastcall`. Same pointer is returned in `EAX`.
PF	Far pointer to temporary memory space passed on stack and returned in `DX`:`AX`.
PAX	Pointer to temporary memory space passed to function in `AX`/`EAX` and returned unchanged in `AX`/`EAX`.
PSI	Pointer to temporary memory space passed to function in `SI`/`ESI` and returned unchanged in `AX`/`EAX`.
0	Nothing passed or returned.

A `struct`, `class` or `union` object can be returned from a function in registers only if it is sufficiently small and not too complex. If the object is too complex or doesn't fit into the appropriate registers then the caller must supply storage space for the object and pass a pointer to this space as a parameter to the function. The pointer can be passed in a register or on the stack. The same pointer is returned by the function. The detailed rules are given in table 7.

P and PF take precedence over all other. PS takes precedence over all but P. PAX takes precedence over PSI, which takes precedence over I.

The storage space pointed to by return pointers is aligned by 16 in the 64 bit Gnu compiler and the 64 bit MS compiler. The 32 bit Microsoft compiler can align this space if explicitly requested.

The Gnu compiler version 2.x and some implementations of version 3.x differ, as indicated by the parentheses.

The Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu.

The 64 bit Gnu compiler differs from the ABI (version 0.97) by using only one floating point register for return.

7.2 Passing and returning SIMD types

Table 8. Methods for passing and returning SIMD data types

segment word size	32 bit		64 bit	
operating system	**Windows**	**Linux**	**Windows**	**Linux**
__m64 parameters transferred in registers	3 mmx registers	3 mmx registers	4 g. p. registers	8 xmm registers (0 in gcc v. ≤ 3.3)
alignment of __m64 parameters on stack	4	8	8	8
__m64 returned in register	mm0	mm0	rax	xmm0 (mm0 in gcc v. ≤ 3.3)
__m128 parameters transferred in registers	3 xmm registers	3 xmm registers	transferred by pointer	8 xmm registers
alignment of __m128 parameters on stack	16	16	16	16
__m128 returned in register	xmm0	xmm0	xmm0	xmm0
__m256 parameters transferred in registers	3 ymm registers	3 ymm registers	transferred by pointer	8 ymm registers
alignment of __m256 parameters on stack	16 or 32	32	16 or 32	32
__m256 returned in register	ymm0	ymm0	ymm0	ymm0
__m512 parameters transferred in registers	3 zmm registers	3 zmm registers	transferred by pointer	8 zmm registers
alignment of __m512 parameters on stack	64	64	64	64
__m512 returned in register	zmm0	zmm0	zmm0	zmm0

The types __m64, __m128, __m256 and __m512 define the SIMD data types that fit into the 64-bit mmx registers, the 128-bit xmm registers, the 256-bit ymm registers or the 512-bit zmm registers, respectively. Most compilers supports these types as intrinsic types. There are special rules for passing and returning function parameters of the these types as shown in table 8. Some compilers have options for either treating these types as intrinsic types, using the passing methods defined in table 8, or treating them as structures according to the rules in table 6 and table 7. The types __m128d, __m128i, __m256d, __m256i, __m512d and __m512i are treated in the same way as __m128, __m256 and __m512.

Under 32-bit Windows and 32-bit Linux, the first three parameters of type __m64 are transferred in registers MM0 - MM2. Any additional parameters of type __m64 are transferred on the stack aligned by 4. Under 64-bit Linux, __m64 parameters are passed on the stack. __m64 may not be supported in some 64-bit Windows compilers (see p. 13). Return values of type __m64 are transferred in RAX in 64-bit Windows, and in MM0 on all other platforms.

Under 32-bit Windows and 32-bit Linux, the first three parameters of type __m128 are transferred in registers XMM0 - XMM2. Any additional parameters of type __m128 are transferred on the stack aligned by 16. This alignment is accomplished as follows: If there are more than three parameters of type __m128 then the stack must be aligned by 16 before the call instruction. Consequently, the value of the stack pointer is 12 modulo 16 at the entry of the called function. The parameter space on the stack is padded, if necessary, to align the parameters of type __m128 by 16. In 64-bit Windows, parameters of type __m128 are passed by a pointer to an aligned copy. In 64-bit Linux, the first eight parameters of type __m128 are transferred in registers XMM0 - XMM7. Any additional parameters of type __m128 are transferred on the stack aligned by 16. If there are more than eight parameters of type __m128 then the stack space is padded, if necessary, to align the parameters. Return values

of type `__m128` are transferred in `XMM0` on all platforms. `__m256` and `__m512` parameters are transferred analogously to `__m128` parameters, except for the fact that in some systems they are aligned by 16, in other systems by 32/64, when transferred on the stack.

8 Name mangling

Name mangling (also called name decoration) is a method used by C++ compilers to add additional information to the names of functions and objects in object files. This information is used by linkers when a function or object defined in one module is referenced from another module. Name mangling serves the following purposes:

1. Make it possible for linkers to distinguish between different versions of overloaded functions.

2. Make it possible for linkers to check that objects and functions are declared in exactly the same way in all modules.

3. Make it possible for linkers to give complete information about the type of unresolved references in error messages.

Name mangling was invented to fulfill purpose 1. The other purposes are secondary benefits not fully supported by all compilers.

The minimum information that must be supplied for a function is the name of the function and the types of all its parameters as well as any class or namespace qualifiers. Possible additional information includes the return type, calling convention, etc. All this information is coded into a single ASCII text string which looks cryptic to the human observer. The linker does not have to know what this code means in order to fulfill purpose 1 and 2. It only needs to check if strings are identical.

Each brand of C++ compiler uses its own name mangling scheme. Previously, there was no need to standardize name mangling because the object files produced by different compilers were incompatible anyway for other reasons. However, since data representation, calling conventions, and other details are now being standardized to an increasing degree in the official ABI (Application Binary Interface) standards of new operating systems, there is good reason to standardize name mangling schemes as well.

Unfortunately, few compiler vendors have cared to publish their name mangling schemes. This is the reason why I have studied the name mangling schemes of several different C++ compilers and published the detailed results here.

Compiler vendors typically use the same name mangling scheme on different hardware platforms. Though the information given here has been gathered by investigating compilers for the 16, 32 and 64 bit x86 platforms, it is likely to apply to other platforms as well, and possibly even other programming languages. Not all mangling schemes are covered in this report. There are other schemes used on other platforms, often resembling the schemes described here as Gnu.

The codes used for parameter types, calling conventions, etc. are given in the tables below. The complete syntax for each compiler is given in the following sections. The syntax is written in extended Bacchus Naur notation. For example,

`<a> ::= <b> | [<c>] [<d>]y x`

means that syntax element `<a>` must consist of either syntax element `<b>` or zero or one instance of `<c>` followed by at least `x` and at most `y` instances of `<d>`. Spaces may be included in the syntax specification for the sake of readability, but the coded string cannot contain spaces.

The codes for parameter types etc. are given in the tables below. The syntax details are described in the following sections for each name mangling scheme.

Table 9. Type codes

type	Microsoft	Borland	Watcom	Gnu2	Gnu3-4 ABI v.3	Gnu4 ABI v.4+
void	X	v	v	v	v	v
bool	_N	4bool	q	b	b	b
char	D	c	a	c	c	c
signed char	C	zc	c	Sc	a	a
unsigned char	E	uc	uc	Uc	h	h
short int	F	s	s	s	s	s
unsigned short int	G	us	us	Us	t	t
int	H	i	i	i	i	i
unsigned int	I	ui	ui	Ui	j	j
long int	J	l	l	l	l	l
unsigned long int	K	ul	ul	Ul	m	m
long long (__int64)	_J	j	z	x	x[1]	x[2]
unsigned long long (unsigned __int64)	_K	uj	uz	Ux	y[1]	y[1]
wchar_t	_W (G)	b	w	w	w	w
float	M	f	b	f	f	f
double	N	d	d	d	d	d
long double	O, _T, _Z[3]	g	t	r	e	e
__complex__ float				Jf	Cf	Cf
__complex__ double				Jd	Cd	Cd
__m64	T__m64@@				U8__vectori5__m64	Dv2_i
__m128	T__m128@@				U8__vectorf6__m128	Dv4_f
__m128d	U__m128d@@				U8__vectord7__m128d	Dv2_d
__m128i	T__m128i@@				U8__vectorx7__m128i	Dv2_x
__m256	T__m256@@				U8__vectorf6__m256	Dv8_f
__m256d	U__m256d@@				U8__vectord7__m256d	Dv4_d
__m256i	T__m256i@@				U8__vectorx7__m256i	Dv4_x
__m512	T__m512@@				U8__vectorf6__m512	Dv16_f
__m512d	U__m512d@@				U8__vectord7__m512d	Dv8_d
__m512i	T__m512i@@				U8__vectorx7__m512i	Dv8_x
varargs ...	Z	e	e	e	z	z
const X	X	xX	xX	X	X	X
X *	PEAX[4]	pX	pnX	PX	PX	PX
const X *	PEBX[4]	pxX[5]	pnxX[5]	PCX[5]	PKX[5]	PKX[5]
volatile X *	PECX[4]	pwX[5]	pnyX[5]	PVX[5]	PVX[5]	PVX[5]
const volatile X *	PEDX[4]	pxwX[5]	pnyxX[5]	PCVX[5]	PVKX[5]	PVKX[5]
X * const	QEAX[4]	xpX[5]	pnX	PX	PX	PX
X * volatile	REAX[4]	wpX[5]	pnX	PX	PX	PX
X * const volatile	SEAX[4]	xwpX[5]	pnX	PX	PX	PX
const X * const	QEBX[4]	xpxX[5]	pnxX[5]	PCX[5]	PKX[5]	PKX[5]
X * __restrict	PEIAX[4]			PX	PX	PX
X &	AEAX[4]	rX	rnX	RX	RX	RX
X &&					OX	OX
const X &	AEBX[4]	rxX[5]	rnxX[5]	RCX[5]	RKX[5]	RKX[5]
volatile X &	AECX[4]	rwX[5]	rnyX[5]	RVX[5]	RVX[5]	RVX[5]
const volatile X &	AEDX[4]	rxwX[5]	rnyxX[5]	RCVX[5]	RVKX[5]	RVKX[5]
X[ ] (as global object)	PAX[4,6]		[]X			
X[][8] (as global object)	PAY07X[4,6,7]		[][8]X			
X[][16][5] (as glob obj)	PAY1BA@4X[4,6,7]		[][16][5]X			
X[ ] (as function parameter)	QEAX[4]	pX	pnX	PX	PX	PX
const X[] (as function param.)	QEBX[4]	xpX	pnxX[5]	PCX[5]	PKX	PKX
X[][8] (as function parameter)	QEAY07X[4,7]	pa8$X	pn[8]X	PA7_X	PA8_X	PA8_X

X[][16][5] (as function param.)	?QEAY1BA@4X[4,7]	?pa16$a5$X	?pn[16][5]X	?PA15_A4_X	?PA16_A5_X	?PA16_A5_X
X near *	?PAX[4]	?pX	?pnX			
X far *	?PEX[4]	?nX	?pfX			
X huge *	?PIX[4]	?upX	?phX			
X _seg *		?urX				
X near &	?AAX[4]	?rX	?rnX			
X far &	?AEX[4]	?mX	?rfX			
X huge &	?AIX[4]	?umX	?rhX			
union X	?TX@@	?<LX>[8]X	?$X$$	?G[9]<LX>[8]X	?<LX>[8]X	?<LX>[8]X
struct X	?UX@@	?<LX>[8]X	?$X$$	?G[9]<LX>[8]X	?<LX>[8]X	?<LX>[8]X
class X	?VX@@	?<LX>[8]X	?$X$$	?G[9]<LX>[8]X	?<LX>[8]X	?<LX>[8]X
enum X	?W4X@@	?<LX>[8]X	?$X$$	?<LX>[8]X	?<LX>[8]X	?<LX>[8]X
enum Y::X	?W4X@Y@@	?<LX+LY+1>[8]Y@X	?X:Y$$	?Q2<LY>Y<LX>[8]X	?N<LY>Y<LX>[8]X	?N<LY>Y<LX>[8]X
X (*Y)(W)[10]	?P6AXW@Z[11]	?pqW$X	?pn(W)X[12]	?PFW_X	?PFXWE	?PFXWE
X Y::*V[13]	?PEQY@@X[4,14]	?M<LY>[8]YX	?m$Y$$nX[12]	?PO<LY>[8]Y_X	?M<LY>[8]YX	?M<LY>[8]YX
X (Y::*V)(W)[15]	?P8Y@@EAEXW@Z[4,16]	?M<LY>[8]YqW$X	?m$Y$$n(W)X[1,2]	?PM<LY>YFP<LY>YW_X	?M<LY>[8]YFXWE	?M<LY>[8]YFXWE

notes:

[1] In 64-bit Linux/BSD/Mac, `long int` and `long long` is the same, but with different mangling codes.

[3] The implementation of `long double` in Microsoft compilers has the same precision as double (64 bits) and uses the code O (capital letter O). A different symbol is used when 80 bits precision is supported. The Intel compiler uses the code `_T` when 80 bits is used and O when 64 bits is used. The Symantec/Digital Mars compiler uses the code `_Z` for 80 bits.

[4]. The `E` symbol is a pointer base symbol according to table 13. It is only used in 64 bit mode. The `A`, `B` etc. symbol is the storage class of the target, according to table 10.

[5]. The letter before `X` is the storage class of the target, according to table 10. The letter before `p` is the storage class of the pointer itself.

[6]. See page 30 for a comment.

[7]. After `QAY` follows: the number of dimensions minus 1, then each dimension except the first one. These numbers are all coded in the way described in table 18 page 34.

[8]. `<LX>` = length of name `X`, `<LY>` = length of name `Y`, as decimal numbers.

[9]. The `G` prefix is only used when a `union`, `struct` or `class` appears as a function parameter. It is not used with pointers or references to these or when the type appears as a template argument. The `G` comes before `Q2` if the name has a qualifying namespace.

[10]. `Y` is a pointer to a function with argument type `W` and return type `X`. In the code columns, `X` and `W` represent the codes for types `X` and `W`.

[11]. Replace `6` with `7` if far. `A` represents the calling convention, using table 16. It may be followed by a return type modifier code from table 12.

[12]. Insert any return type modifier or target modifier (table 12) before the return type, and any member function access code (table 15) before `(`.

[13]. `V` is a pointer to a data member of class `Y` of type `X`. In the code columns, `X` and `Y` represent the codes for types `X` and `Y`, `<LY>` represents the length of the name `Y`.

[14]. `Q` qualifies the target. Replace with `R` if const, `S` if volatile, `T` if const volatile.

[15]. `V` is a pointer to a function member of class `Y` with argument type `W` and return type `X`. In the code columns, `X` ,`Y` and `W` represent the codes for types `X`, `Y` and `W`, `<LY>` represents the length of the name `Y`.

[16]. Replace `8` with `9` if far. `A` represents a member function access code, using table 15. It is omitted in 16-bit mode. The second `E` represents the calling convention, using table 16. It may be followed by a return type modifier code from table 12.

Table 10. Storage class codes

storage class	Microsoft	Borland	Watcom	Gnu2	Gnu3
(default)	A		n		
near	A		n		
const	B	x	nx	C	K
volatile	C	w	ny	V	V
const volatile	D	xw	nyx	CV	VK

far	E		f		
const far	F		fx		
volatile far	G		fy		
const volatile far	H		fyx		
huge	I		h		
__unaligned	F				
__restrict	I				

Example: `const int a;`

Table 11. Function distance codes

calling distance	Microsoft	Borland	Watcom	Gnu2	Gnu3
near	Y or Q		n		
far	Z or R		f		

Example: `void far Function1 (int x);`

Table 12. Storage class codes for return

storage class	Microsoft	Borland	Watcom	Gnu2	Gnu3
default	?A				
const	?B		x		
volatile	?C		y		
const volatile	?D		yx		

Example: `const int Function2 (int x);`

Table 13. Pointer base codes

pointer base	Microsoft
segment relative (16 and 32 bit mode)	
absolute (64 bit mode)	E
based (64 bit mode)	M

Table 14. Member function modifier codes (Microsoft only)

storage or call type	private	protected	public
default	A	I	Q
far	B	J	R
static	C	K	S
static far	D	L	T
virtual	E	M	U
virtual far	F	N	V

Example: `public: virtual int Class1::MemberFunction3 (int x);`

Table 15. Member function access codes

storage for `this` target	Microsoft	Borland	Watcom	Gnu2	Gnu3
default	A				
const	B	x	.x		
volatile	C	w	.y		
const volatile	D	xw	.yx		

Example: `int Class1::MemberFunction4 (int x) const;`

Table 16. Function calling convention codes

calling convention	Microsoft	Borland	Watcom	Gnu2	Gnu3-4
__cdecl	A[17]				
__pascal	C	(uppercase)			
__fortran	C	qf			
__thiscall	E				
__stdcall	G	qs		@<size>	
__fastcall	I[17]	qr			
__regcall	E				
__vectorcall					
interrupt	A	qi			

Example: `int __stdcall Function5 (int x);`

notes:

[17]. In 64-bit mode, this calling convention is coded as A.

Table 17. Operator name codes

operator	Microsoft	Borland	Watcom	Gnu2	Gnu3-4
constructor X	?0	$bctr	$ct	_	C1,C2
destructor ~X	?1	$bdtr	$dt	$	D1
operator []	?A	$bsubs	$od	__vc_	ix
operator ()	?R	$bcall	$op	__cl	cl
operator ->	?C	$barow	$oe	__rf_	pt
operator ++X, X++[18]	?E	$binc	$og	__pp_	pp
operator --X, X--[18]	?F	$bdec	$oh	__mm_	mm
operator new	?2	$bnew	$nw	__nw_[19]	nw
operator new[]	?_U	$bnwa	$na	__vn_[19]	na
operator delete	?3	$bdele	$dl	__dl_[19]	dl
operator delete[]	?_V	$bdla	$da	__vd_[19]	da
operator *X	?D	$bind	$of	__ml_	de
operator &X	?I	$badr	$ok	__ad_	ad
operator +X	?H	$badd	$oj	__pl_	ps
operator -X	?G	$bsub	$oi	__mi_	ng
operator !	?7	$bnot	$oc	__nt_	nt
operator ~	?S	$bcmp	$oq	__co_	co
operator ->*	?J	$barwm	$ol	__rm_	pm
operator X * Y	?D	$bmul	$of	__ml_	ml
operator /	?K	$bdiv	$om	__dv_	dv
operator %	?L	$bmod	$on	__md_	rm
operator X + Y	?H	$badd	$oj	__pl_	pl
operator X - Y	?G	$bsub	$oi	__mi_	mi
operator <<	?6	$blsh	$ob	__ls_	ls
operator >>	?5	$brsh	$oa	__rs_	rs
operator <	?M	$blss	$rc	__lt_	lt
operator >	?O	$bgtr	$re	__gt_	gt
operator <=	?N	$bleq	$rd	__le_	le
operator >=	?P	$bgeq	$rf	__ge_	ge
operator ==	?8	$beql	$ra	__eq_	eq
operator !=	?9	$bneq	$rb	__ne_	ne
operator X & Y	?I	$band	$ok	__ad_	an
operator |	?U	$bor	$os	__or_	or
operator ^	?T	$bxor	$or	__er_	eo
operator &&	?V	$bland	$ot	__aa_	aa
operator ||	?W	$blor	$ou	__oo_	oo
operator =	?4	$basg	$aa	__as_	aS
operator *=	?X	$brmul	$ab	__aml_	mL
operator /=	?_0	$brdiv	$ae	__adv_	dV
operator %=	?_1	$brmod	$af	__amd_	rM
operator +=	?Y	$brplu	$ac	__apl_	pL
operator -=	?Z	$brmin	$ad	__ami_	mI
operator <<=	?_3	$brlsh	$ah	__als_	lS
operator >>=	?_2	$brrsh	$ag	__ars_	rS
operator &=	?_4	$brand	$ai	__aad_	aN
operator |=	?_5	$bror	$aj	__aor_	oR
operator ^=	?_6	$brxor	$ak	__aer_	eO
operator ,	?Q	$bcoma	$oo	__cm_	cm
operator TYPE()	?B	$o<L>TYPE[20]	$cv	__op<L>TYPE[20]	cv<L>TYPE[20]
virtual table	?_7				

Example: `Class1 operator + (Class1 & a, Class1 & b);`

notes:

[18]. X++ is distinguished from ++X by a dummy int parameter.

[19]. If the operator is not a class member and there are no extra parameters, i.e. if the built-in operator is replaced, then the full mangled name is replaced by ___builtin_new, ___builtin_vec_new, ___builtin_delete, or ___builtin_vec_delete.

[20]. <L> is the length of the name of TYPE.

8.1 Microsoft name mangling

Microsoft compilers use a name mangling syntax that includes all information needed to check that an object or function is declared in exactly the same way in all modules (except for array sizes). It is also designed to be as short as possible, while allowing case-insensitive linking. The code is unambiguous so that the complete C++ declaration of an

object or function can be recovered from a mangled name. Several other compilers for Windows use the same or almost the same name mangling scheme.

The public mangled name of a global object is composed according to the following syntax:

<public name> ::= ? <name> @ $[<\text{namespace}>@]_0^\infty$ @ 3 <type> <storage class>

The mangled name of a static class member object is:

<public name> ::= ? <name> @ $[<\text{class name}>@]_1^\infty$ @ 2 <type> <storage class>

<name> is the case sensitive C++ name of the object.

<namespace> is any namespace surrounding the object.

<class name> is the class the object belongs to or a namespace. Class names and namespaces are treated as equivalent. In case of nested classes or namespaces, the innermost class or namespace comes first.

<type> is the code for object type, taken from table 9.

<storage class> is any storage modifier, taken from table 10. The default is A. This code is replaced by Q1@ for member pointers and member function pointers, regardless of storage class.

Pointers and references include the <pointer base> and <storage class> code for the target. <pointer base> is a pointer base code according to table 13, which is used only in 64-bit mode. Global pointers and references (including function pointers and member pointers) in 64-bit systems include the <pointer base> twice, both before the <storage class> code of the target and before the <storage class> code of the pointer or reference itself. (Early 64-bit C++ compilers didn't have <pointer base> codes).

Examples:

```
int alpha;
```

is coded as

```
?alpha@@3HA
```

```
char beta[6] = "Hello";
```

is coded as

```
?beta@@3PADA
```

```
double Class1::gamma[10][5];
```

is coded as

```
?gamma@Class1@@2PAY04NA
```

```
int * delta;
```

is coded in 16-bit and 32-bit systems as

```
?delta@@3PAHA
```

and in 64-bit systems as

```
?delta@@3PEAHEA
```

Note that global arrays are coded as pointers (P) while arrays as function parameters are coded as pointer constants (Q). This should have been opposite, since arrays as function parameters are equivalent to non-constant pointers, while global arrays are equivalent to pointer constants. Apparently, this illogical coding has been retained in almost all compilers for the sake of compatibility with legacy code. The first 64-bit C++ compilers used _O instead of P for global arrays, but they soon returned to the P syntax. The consequence of this illogical coding is that an array in one module can be confused with a pointer with the same

name in another module, while pointers and arrays as function parameters are not treated as equivalent even though they are equivalent in the C++ syntax.

The mangled name of a global function is composed according to the following syntax:

<public name> ::= ? <function name> @ $[<\text{namespace}>@]_0^\infty$ @ <near far>
<calling conv> [<stor ret>] <return type> $[<\text{parameter type}>]_1^\infty$ <term> Z

<near far> is Y for near, Z for far. Far calls are only possible in 16-bit mode.

<calling conv> is the calling convention, taken from table 16. The default is A.

<stor ret> defines the storage class of the return, using the codes in table 12. It is omitted for simple types if the storage class is not const or volatile. It is always included if the return type is a struct, class or union.

<return type> is the type returned by the function, taken from table 9.

<parameter type> is the type of each function parameter, taken from table 9.

<term> is @ except if the parameter list is void (X) or ends with ... (Z). In these cases, the @ is omitted because the list is sure to end here.

Example:
```
void Function1 (int a, int * b);
```
is coded as
```
?Function1@@YAXHPAH@Z
```

The mangled name of a class member function is composed according to the following syntax:

<public name> ::= ? <function name> @ $[<\text{class name}>@]_1^\infty$ @ <modif> [<const vol>]
<calling conv> [<stor ret>] <return type> $[<\text{parameter type}>]_1^\infty$ <term> Z

<modif> defines the private, protected, public, static, virtual, near and far modifiers of a member function according to table 14. Far calls are only possible in 16-bit mode.

<const vol> is a member function access code from table 15. The default is A. It is omitted for static member functions.

The default calling convention for non-static member functions in 16 bit mode is C (__pascal), in 32 bit mode it is E (thiscall). In 64 bit mode the only possible calling convention is A. The default calling convention for static member functions is A.

Example:
```
int Class1::MemberFunction(int a, int * b);
```
is coded in 32-bit mode as
```
?MemberFunction@Class1@@QAEHHPAH@Z
```
and in 64-bit mode as
```
?MemberFunction@Class1@@AEAAHHPEAH@Z
```

Constructors, destructors, operators and member operators are coded in the same way as functions, by replacing <function name>@ with the operator name taken from table 17. The return type of constructors and destructors is replaced with @.

Virtual tables are coded as `??_7` [<class name> @]$_1^\infty$ `@6B@`

Template functions and template classes are coded by replacing `<function name>` or `<class name>` by `?$ <name> @` [<template parameter>]$_1^\infty$
where `<name>` is the name of the templated function or class. If the template parameter is a typename or class then `<template parameter>` is a type as defined in table 9. If the template parameter is a constant, then
`<template parameter> ::= $0 <integer>`
where `<integer>` is coded as explained in table 18 below.

Abbreviations for repeated names and parameter types

The name mangling scheme includes two means of shortening mangled names that would contain the same name or type more than once. The first method involves repeated types, the second method involves repeated names.

Abbreviation of repeated types. This method applies to type declarations in a function parameter list or function pointer parameter list. Only types that need more than one character for its code are included in this scheme. This includes pointers, references, arrays, `bool`, `__int64`, `struct`, `class`, `union`, and `enum` parameters. The first such parameter in a parameter list is assigned the number `0`, the second such parameter is assigned the number `1`, and so forth. Simple types that are encoded with a single letter are not assigned a number. Any repeated instance of a type with an assigned number in the parameter list is replaced by the number of the first instance. The maximum number is `9`. If the number would exceed `9` then the repeated instance must use the full declaration. The return type is not included in the type abbreviation scheme.

Example:
```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```
is coded as
```
?ExampleFunction@@YA_NPAHHH0_N1PA_N@Z
```
Here, the return type bool is coded as `_N`. `int*a` is coded as `PAH`, which is assigned the number `0`. `int b` is coded as `H`, and is not assigned a number because it is coded as a single letter. `int c` is also coded as `H` because single letter codes are not abbreviated. `int*d` has the same type as `a`, and is abbreviated to the number `0`. `bool e` is coded as `_N` and is assigned the number `1`. The previous instance of `_N` was a return type, so it cannot be copied. `bool f` has the same type as `e` and is replaced by the number `1`. The `bool` in `bool*g` is not abbreviated because sub-expressions cannot be abbreviated.

If the parameter list contains a function pointer, then the parameter types inside the function pointer type declaration are included in the abbreviation scheme, both as sources that can be assigned numbers and as targets that can be abbreviated. The return type in the function pointer type declaration is not included. If the return type of a function declaration is a function pointer, then the parameters, but not the return type, of this function pointer declaration are included in the type abbreviation scheme of the whole function declaration.

Example:
```
typedef int * (* FunctionPointer) (int * a, int * b);
FunctionPointer WeirdFunction(FunctionPointer x,FunctionPointer y,int*z);
```
is coded as
```
?WeirdFunction@@YAP6APAHPAH0@ZP6APAH00@Z10@Z
```
Here, the code for `int *` is `PAH`, and the code for `FunctionPointer` without abbreviation would be `P6APAHPAHPAH@Z`. The first occurrence of `FunctionPointer` is in the return type of `WeirdFunction`. Within this occurrence, the first occurrence of `PAH` is the return type which is excluded from the abbreviation scheme. The second occurrence of `PAH`, representing `int*a`, is assigned the number `0`. The third occurrence of `PAH`, representing `int*b`, is replaced by `0`. The second occurrence of `FunctionPointer` represents parameter `x`. Within this, the first

occurrence of `PAH` is not abbreviated because it represents the return type of `FunctionPointer`. The next two occurrences of `PAH`, representing `a` and `b` in `x`, are both replaced by the `0` that has already been assigned. The entire sequence representing parameter `x` is thus `P6APAH00@Z`. This sequence is assigned the number `1`. `FunctionPointer y` is simply reduced to `1`, and `int*z` is reduced to `0`.

Abbreviation of repeated names. This method applies to any name that appears inside a declaration, such as structures, classes, unions, enums, and namespaces. If any such name occurs more than once in a mangled name, then all but the first occurrence will be replaced by a number, no matter how short the name is. The number will represent a copy of the name, but not its context or meaning. A name can be copied even if the different occurrences of the name have different meanings (because of namespace or class scope qualifications). The algorithm is as follows: First eliminate any repeated types using the first abbreviation method. Any names that have been eliminated by the type abbreviation method need no further consideration. Then assign numbers to the first occurrence of each name. The first name, which is usually the function's name (except for constructors, destructors, operators and template functions), is assigned the number `0`, the second name is `1`, and so forth. Each repeated name is then eliminated by replacing `<name>@` with the number. If the number would be higher than `9` then the name cannot be eliminated.

Example:
```
Class1 * SomeFunction (Class1 * a, Class2 * b, Class2 * c, Class2 & d);
```
is coded as
```
?SomeFunction@@YAPAVClass1@@PAV1@PAVClass2@@1AAV2@@Z
```
Here, parameters `b` and `c` have the same type, so `Class2 * c` is reduced by the first method, and simply becomes a `1`. The last parameter `Class2 & d` cannot be reduced by the type abbreviation method because `b` and `d` have different types. Neither can the double occurrence of the name `Class1`, because the type abbreviation method doesn't apply to return types. The name abbreviation method now assigns the numbers `SomeFunction` = `0`, `Class1` = `1`, `Class2` = `2`. Now parameter `a` can be changed from `PAVClass1@@` to `PAV1@`, and parameter `d` is changed from `AAVClass2@@` to `AAV2@`.

Templated names and template parameters are isolated from the numbering of names. This means that a name inside a template argument can only be eliminated if there are multiple occurrences of this name within the same templated name. Likewise, templated names are isolated from each other, even if they are identical. In case of nested templates, each sub-template has its own isolated number sequence.

Example:
```
void Class1::MyTemplateFunction<Class1> (Class1*);
```
will be coded as
```
??$MyTemplateFunction@VClass1@@@Class1@@QAEXPAV0@@Z
```
Here, the templated name `MyTemplateFunction<Class1>` is coded as `?$MyTemplateFunction@VClass1@@`. This template has its own number sequence (`MyTemplateFunction` = `0`, `VClass1` = `1`), which is isolated from the rest. The first name in the rest of the code is the representation of the scope `Class1::` coded as `Class1@`. This occurrence of `Class1` gets the number `0`. The parameter `Class1*`, which was first coded as `PAVClass1@@` is now changed to `PAV0@`, where the `0` refers to the name in `Class1::`, not the name in `<Class1>`.

Coding of numbers

Numbers within mangled names are needed for array dimensions, array sizes, and template parameters. These numbers are coded according to the algorithm in table 18. It appears that this algorithm was designed to make the coding as short as possible, rather than making it human readable.

Table 18. Microsoft number encoding

range for N	coding
$1 \le N \le 10$	(N - 1) as a decimal number
$N > 10$	code N as a hexadecimal number without leading zeroes, replace the hexadecimal digits 0 - F by the letters A - P, end with a @
$N = 0$	A@
$N < 0$	? followed by (- N) coded as above

8.2 Borland name mangling

This name mangling scheme is used only by Borland compilers.

The name of a global object without class or namespace qualifiers is not mangled, except for un underscore prefix:

`<public name> ::= _ <name>`

A global object with class or namespace qualifiers is coded as

`<public name> ::=` $[@<\text{class name}>]_1^\infty$ `@ <name>`

where `<class name>` is a class or namespace. In case of nested classes or namespaces, the outermost comes first.

Functions, member functions, constructors, destructors and operators are all coded according to the following syntax:

`<public name> ::= [<template prefix>]` $[@<\text{class name}>]_0^\infty$ `@ <name> $ [<const vol>] q [<calling convention>]` $[<\text{parameter type}>]_1^\infty$

`<template prefix>` is `@` if the function is a template function or member of a template class, otherwise nothing.

`<const vol>` is a member function access code from table 15. It is omitted by default.

`<calling convention>` defines the calling convention as given in table 16. It is omitted by default. There is no distinction between near and far calling.

`<parameter type>` defines each function parameter, using the codes in table 9. The return parameter is not coded.

For constructors, destructors and operators, replace `<name>` by an operator name from table 17.

Template functions have no special encoding other than the `@` prefix, as the template parameters are implied by the function parameter types. The Borland compilers I have tested only support such cases of template functions where the template parameters can be inferred from the function parameters.

Template class member functions and member objects are coded by replacing `<class name>` by

`%<name>` $[\$t<\text{typecode}> \mid \$ii\$<\text{value}>]_1^\infty$ `%`

Global objects of a template class are not mangled if in the global namespace.

Virtual tables are encoded as
`@@<class name>@3`

Type codes that appear more than once in the `<parameter type>` list of a function are abbreviated if the type is a pointer, reference, array, `struct`, `class`, `union`, `enum` or `bool`, but not if it is a simple type with one or two-letter code. All parameters are assigned a number, beginning with 1. The code `t1` repeats the first parameter, `t2` repeats the second parameter, `ta` repeats the 10'th parameter, `tz` repeats parameter number 35. Further parameters cannot be copied. If the parameter list contains a function pointer, then the list of parameters for the target function has its own isolated number sequence, so that type codes within the parameter list of the target function can be abbreviated, but not the return type of the target function. There is no method for abbreviating repeated names that are not part of identical parameter types.

Examples:
```
char beta[6] = "Hello";
```
is coded as
```
_beta
```

```
double Class1::gamma[10][5];
```
is coded as
```
@Class1@gamma
```

```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```
is coded as
```
@ExampleFunction$qpiiit14boolt5p4bool
```

```
void TemplateClass<float>::MemberFunction (TemplateClass<float>*);
```
is coded as
```
@@%TemplateClass$tf%@MemberFunction$qp18%TemplateClass$tf%
```

8.3 Watcom name mangling

This name mangling scheme is used only by Watcom compilers.

The public mangled name of a global object is composed according to the following syntax:

`<public name> ::= W? <name> $` $\left[:<\text{namespace}>\$\right]_0^\infty$ `<storage class> <type>`

where `<storage class>` is taken from table 10 and `<type>` is taken from table 9. `<namespace>` can be any namespace or class qualifier, the innermost first.

Examples:
```
int alpha;
```
is coded as
```
W?alpha$ni
```

```
char beta[6] = "Hello";
```
is coded as
```
W?Beta$npna
```

```
double Class1::gamma[10][5];
```
is coded as
```
W?gamma$:Class1$n[][5]d
```

Functions, member functions, constructors, destructors and operators are coded as follows:

<public name> ::= W? <function name> $ $[:<class name>\$]_1^\infty$ <near far> [<const vol>] ($[<parameter type>]_0^\infty$) [<stor ret>] <return type>

<class name> is a class name or namespace. In case of nested classes or namespaces, the innermost comes first.

<near far> is n for near or f for far. Far calls are only possible in 16 bit mode.

<const vol> is a member access code from table 15. It is omitted by default.

<parameter type> defines the type of each function parameter according to table 9. No <parameter type> is included if the parameter list is (void).

<stor ret> is a return type storage class from table 12. It is omitted by default.

<return type> defines the return type according to table 9.

Example:
```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```
is coded as
```
W?ExampleFunction$n(pniiipniqqpnq)q
```

Constructors, destructors and operators are coded by replacing <function name>$ with a name from table 17. The return type of constructors and destructors is replaced by _.

Names that appear more than once in a mangled code are reduced by replacing all but the first occurrence of a name by a reference to the first occurrence. The first occurrence of each name is assigned a number, starting with 0. A repeated occurrence of a name is then abbreviated by replacing <name>$ by one of the numbers 0 - 9. No replacement is possible if a higher number would be needed. A name can be replaced even if the repeated occurrence has a different meaning or context. There is no method for abbreviating repeated types.

Virtual tables have the funny code
<public name> ::= W?$Wvf <nl> o4: <class name> $$nx[]pn()v
where <nl> is the length of the class name + 4, coded as a two-digit base 36 number with digits 0-9, a-z.

Template classes are coded by replacing <class name>$ by
<name>$:: $[1n<type> | 0<number>]_1^\infty$
where <type> is a template type parameter, and <number> is a template integer parameter, coded as a base-32 number with digits 0-9, a-v, followed by a suffix z if positive or zero, and y if negative. Template functions have no special encoding as the template parameters are implied by the function parameter types. The Watcom compilers I have tested only support such cases of template functions where the template parameters can be inferred from the function parameters.

8.4 Gnu2 name mangling

This mangling scheme is used in Gnu C++ version 2.x.x under several operating systems (Linux, BSD, Windows). Later versions of Gnu C++ use a different scheme described in the next section.

The Gnu2 mangling scheme is a dialect of the scheme used by cfront, one of the oldest C++ tools. Variants of this scheme are widely used in UNIX systems (See: J. R. Levine: Linkers and Loaders. Morgan Kaufmann Publishers, 2000).

The type of a global object is not coded, only class or namespace qualifiers, if any:

```
<public name> ::= [_ <qualifiers list> <list term> ] <name>
```

where

```
<qualifiers list> ::= [<qualifiers count>] [<name length><class name>]∞1

<list term> ::= . | $
```

`<qualifiers count>` is the number of class or namespace qualifiers. It is omitted if the count is 1. It is `Q<number>` if the number is `2` - `9`. It is `Q_<number>_` if the number is more than `9`. All numbers are decimal. Some versions use `.` as list terminator (Red Hat), other versions use `$` (FreeBSD, Cygwin, Mingw32). The namespace `std` is ignored.

```
char beta[6] = "Hello";
```
is coded as
```
beta
```

```
char Namespace1::beta[6];
```
is coded as
`_10Namespace1.beta` or `_10Namespace1$beta`

Functions and member functions are coded as follows

```
<public name> ::= <name> _ _ [ <qualifiers list> | F ] [<parameter type>]∞1
```

The `<qualifiers list>` is replaced with an `F` if there are no class or namespace qualifiers.

`<parameter type>` is the type of each function parameter, as defined by table 9. The return type and function modifiers are not included.

Types that occur more than once in the parameter list can be repeated according to the following rules. Each parameter is assigned a number, beginning with `0`. All parameters are numbered, regardless of whether they are identical to a previous parameter. A repeated occurrence of a parameter is replaced by a reference to the first occurrence if it is a pointer, reference, array or other non-simple type. `bool` is also treated as a non-simple type, while `long double` and `unsigned __int64` are treated as simple types. A repeated occurrence of a non-simple parameter is replaced by `T <first occur>` where `<first occur>` is the number assigned to the first occurrence. If the number is bigger than `9` then `<first occur>` is followed by an `_`. A sequence of identical types can be replaced by `N <count> <first occur>` where `<count>` is the number of identical parameter types to replace and `<first occur>` is the number assigned to the first occurrence. Both `<count>` and `<first occur>` are followed by an `_` if bigger than `9`. For obscure reasons, the compiler uses the `T` replacement rather than the `N` replacement for the first parameter in a sequence of identical parameters if the preceding parameter is not the first occurrence of the same type. There is no method for abbreviating repeated names that are not part of identical parameter types.

Example:
```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```
is coded as
```
ExampleFunction__FPiiiT0bT4Pb
```

Constructors, destructors and operators are coded in the same way as functions with `<name>_` replaced by an operator name from table 17.

Template functions are coded as follows

<public name> ::= <name> _ _ H <numtp> $\left[Z < \text{type parameter} > \mid < \text{type} > < \text{value} > \right]_{\text{numtp}}^{\text{numtp}}$ _ [<qualifiers list>] $\left[< \text{parameter type} > \mid X < \text{temp.par.num} > 1 \right]_1^\infty$ _ <return type>

where <numtp> is the number of template parameters, <type parameter> is a template type parameter, <value> is a template constant parameter of type <type>. <parameter type> in the list of function parameters is replaced by X <temp. par. num.> 1 if, and only if, it is explicitly declared as the same type as a template type parameter. <temp. par. num.> is the number of the template parameter referred to, starting at 0. The return type is included only for template functions.

Template classes are coded by replacing <class name length> <class name> by t <name length> <name> <numtp> $\left[Z < \text{type parameter} > \mid < \text{type} > < \text{value} > \right]_{\text{numtp}}^{\text{numtp}}$

Under Windows, all public names get an additional underscore prefix, for example _ExampleFunction__FPiiiT0bT4Pb

The _ prefix is used only under Windows. It is omitted in Linux and FreeBSD, except possibly if the old a.out object file format is used.

8.5 Gnu3-4 name mangling

This mangling scheme is used in Gnu C++ version 3.x.x and later under several operating systems (Linux, BSD, Mac OS X IE32, Windows) and on several platforms. It is described in "Itanium C++ ABI". The same scheme is used by Intel compilers for Linux and Mac OS. Different variants are available for the Gnu compiler version 4.x.x, specified by the command line parameters -fabi-version=3 and -fabi-version=4, etc. A value for -fabi-version of 4 or more is preferred when parameters of type __m256 are used. A value of 0 gives the latest version.

Earlier versions of Gnu C++ use a different scheme described in the previous section.

The name of a global object without class or namespace qualifiers is not decorated in any way:

<public name> ::= <name>

A global object with class or namespace qualifiers is coded as

<public name> ::= _Z <qualified name>

where

<qualified name> ::= N $\left[< \text{simple name} > \right]_2^\infty$ E

<simple name> ::= <name length> <name>

where <name length> is the length of each name as a decimal number. In case of nested classes or namespaces, the outermost comes first. The object name comes last.

Examples:

```
char beta[6] = "Hello";
```

is coded as

```
beta
```

```
char Namespace1::beta[6];
```

is coded as

```
_ZN10Namespace14betaE
```

There are special abbreviations if the outermost namespace is `std`. If `std` is the only qualifier, use

```
<qualified name> ::= St <simple name>
```

If there are more qualifiers, use

<qualified name> ::= N St $\left[< \text{simple name} > \right]_{2}^{\infty}$ E

If `std` is not the outermost qualifier, then it is treated as any other qualifier, i.e. coded as `3std`.

Functions, member functions, constructors, destructors and operators are all coded according to the following syntax:

<public name> ::= _Z <simple or qualified name> $\left[< \text{parameter type} > \right]_{1}^{\infty}$

```
<simple or qualified name> ::= <simple name> | <qualified name> | <operator name>
```

`<operator name>` is an operator name from table 17. Any classes or namespaces come first in `<qualified name>` and the function name comes last. The abovementioned abbreviations for `std` apply.

Example:

```
bool Example1Function (int a, int * b, bool c, bool d, bool * e);
```

is coded as

```
_Z16Example1FunctioniPibbPb
```

Virtual tables are coded as

```
<public name> ::= _ZTV <simple or qualified name>
```

Template functions and template classes are coded by replacing `<simple name>` by

<simple name> I $\left[< \text{template parameter} > \right]_{1}^{\infty}$ E

```
<template parameter> ::= <type>  |  L <type> <value> E
```

where the first option is for template type parameters and the second option for template constant parameters.

The return type of a template function is included as the first type in the parameter type list. If the template function has no parameters, then the code for void (`v`) is omitted. This is different from non-template functions, where the return parameter is omitted and the void is included.

There is a method for abbreviating repeated names and types. This abbreviation scheme does not distinguish between names and types. The first occurrence of each name or non-simple type is assigned a symbol in the following sequence:

```
S_, S0_, S1_, ... S9_, SA_, SB_, ... SZ_, S10_, S11_, ...
```

These abbreviation symbols are assigned, in the order of occurrence, to the first occurrence of each name of structures, classes, unions, enums and namespaces, but not to the name of the function or object itself. The abbreviation symbols are also assigned to all non-simple types occurring anywhere in the mangled name. A non-simple type is any type that needs

more than one character for its encoding, according to table 9. This scheme also assigns abbreviation symbols to non-simple types that form part of the declaration of a more complex type. For example, the type `Class1**` gets three abbreviation symbols, for `Class1`, `Class1*`, and `Class1**`, respectively. All but the first occurrence of each name or type is replaced by the abbreviation symbol, even if the abbreviation symbol is longer than the original code. If the same name has more than one meaning because of different class or namespace qualifiers, then the occurrences with different meanings are treated as different names.

Template type parameters are included in this abbreviation scheme. A repeated occurrence of a type in `<template parameter>` is abbreviated by e.g. `S0_`. If a function parameter is explicitly declared as the same type as a template parameter, then the first occurrence is replaced by `T_`, `T0_`, etc., where `T_` refers to the first template parameter. A repeated occurrence of the template parameter in the function parameter list is abbreviated using the `S_` scheme.

Example:
```
bool Example2Function (int a, int * b, Class1 & c, Class1 d, Class1 & e);
```
is coded as
```
_Z16Example2FunctioniPiR6Class1S0_S1_
```

Under 32-bit Windows and 32- and 64-bit Mac OS, all public names get an additional underscore prefix, for example
```
__Z16Example2FunctioniPiR6Class1S0_S1_
```

8.6 Intel name mangling for Windows

Intel compilers for 32 bit and 64 bit Windows use the same name mangling scheme as Microsoft.

The Intel compilers can treat the types `__m64` and `__m128` as intrinsic types when the /Qmspp- option is specified. In this case it uses `_K`, which is the code for unsigned `__int64`, to represent `__m64` and `__m128`. This prevents function overloading. This option is deprecated and should be used only when compatibility with legacy code is needed.

Where a function has CPU dispatching, the following suffixes are appended to the (mangled or unmangled) names of CPU-specific functions in order to distinguish the version for each CPU:

Table 19. Intel CPU-specific function name suffixes

CPU	Instruction set	Name suffix	__intel_cpu_indicator
Generic	specified baseline (80386 or higher)	.A	1
Pentium	Pentium	.B	2
Pentium Pro	CMOV	.C	4
Pentium MMX	MMX	.D	8
Pentium II	CMOV and MMX	.E	0x10
Pentium III no XMM	CMOV and MMX	.G	0x40
Pentium III	SSE	.H	0x80
Pentium 4	SSE2	.J	0x200
Pentium M	SSE2	.K	0x400
Pentium 4 w. SSE3	SSE3	.L	0x800
Core 2 duo	Suppl. SSE3	.M	0x1000
Wolfdale	SSE4.1	.N	0x2000
Atom	SSE3	.O	0x4000
Nehalem	SSE4.2 + POPCNT	.P	0x8000

Nehalem	PCLMUL + AES	.Q	0x10000
Sandy Bridge	AVX	.R	0x20000
(unused?)		.S	0x40000
(unused?)		.T	0x80000
(unused?)		.U	0x100000
(unused?)		.V	0x200000
Haswell	AVX2+FMA3+BMI1/2	.W	0x400000
	HLE + RTM	.X	0x800000

The dot (.) is replaced by a $-sign in some versions. `__intel_cpu_indicator` is an internal variable.

8.7 Intel name mangling for Linux

Intel compilers for 32 bit and 64 bit Linux and Mac OS use the same name mangling scheme as Gnu 3.x. In case of CPU dispatching, the suffixes listed above are used.

8.8 Symantec and Digital Mars name mangling

The Symantec and Digital Mars C++ compilers use the same name mangling scheme as Microsoft with very few exceptions. I have found the following differences:

- `long double` has 80 bits precision and is coded as `_Z` in Symantec/Digital Mars compilers. In Microsoft compilers, long double has 64 bits precision (same as double) and is coded as O. Intel compilers use `_T` for 80 bits precision.

- The type `wchar_t` is coded as `_Y`, while Microsoft compilers use `_W`.

- The method for abbreviating types (page 32) applies to `bool` (`_N`), but not to other two-character codes (`_J`, `_K`, `_Y`, `_Z`). It does apply to pointers and references to such types.

- The coding of member function pointers do not have the member function access code and return type modifier code. This may be an obsolete syntax, since it is also missing in 16-bit Microsoft compilers.

- Global arrays have the code `Q` while arrays as function parameters are coded as pointers (`P`) in Symantec and Digital Mars compilers. This is more correct than the coding generated by Microsoft compilers, as explained on page 30.

8.9 Codeplay name mangling

The Codeplay VectorC C++ compiler uses the same name mangling scheme as Microsoft with some exceptions. I have not checked it systematically, but I have found the following differences:

- `long double` is supported in both 64 bits and 80 bits precision. Both are coded as O.

- The method for abbreviating types (page 32) does not apply to two-character codes. It does apply to pointers and references to such types.

- The coding of data member pointers and member function pointers do not include the class name.

- Arrays as function parameters are coded as pointers (`P`).

- Global one-dimensional arrays are coded with the array size. The encoding of the size is slightly different from the Microsoft scheme.

8.10 Other compilers

The PathScale compiler uses Gnu name mangling. The PGI compiler also uses Gnu name mangling, even under Windows.

8.11 Turning off name mangling with extern "C"

Table 20. Function name prefixes with extern "C" declaration

Call type	Microsoft 16 bit	Microsoft 32 bit	Microsoft 64 bit	Digital Mars	Borland 16, 32	Watcom 16, 32	Gnu2 Windows 32	Gnu3 Windows 32	Gnu2, Gnu3 Linux, BSD 32	Gnu3 Linux, BSD 64
(default)	_	_		_	_		_	_		
__cdecl	_	_		_	_	_	_	_		
__stdcall	_	_		_		_	_	_		
__fastcall		@		_	@		_	@		
pascal					UC	UC				
__fortran					_	UC				

Explanation:

_ name has underscore prefix

@ name has @ prefix

UC entire name converted to upper case

Table 21. Function name postfixes with extern "C" declaration

Call type	Microsoft 16 bit	Microsoft 32 bit	Microsoft 64 bit	Digital Mars	Borland 16, 32	Watcom 16, 32	Gnu2 Windows 32	Gnu3 Windows 32	Gnu2, Gnu3 Linux, BSD 32	Gnu3 Linux, BSD 64
(default)						_				
__cdecl										
__stdcall		@S		@S		@S	@S	@S		
__fastcall		@S				_		@S		
pascal										
__fortran										

Explanation:

_ underscore appended after name

@S name followed by @, followed by the combined size of all parameters expressed as the number of bytes pushed on the stack as a decimal number. For __fastcall, register parameters are included by the size they would have if they were transferred on the stack.

The extern "C" attribute on a C++ function turns off name mangling so that the public or external name becomes compatible with the C language. This can be useful for solving problems with incompatible name mangling schemes. In 16 and 32 bit DOS and Windows

systems, however, there is still some name decoration. The public and external names get the prefixes shown in table 20 and the postfixes shown in table 21. This may cause compatibility problems for `__stdcall` and `__fastcall` functions.

The `extern "C"` attribute is only allowed for functions that can be coded in C. Hence, overloaded functions and member functions cannot have the `extern "C"` attribute. When compatibility with all compilers is desired, you may give all functions the `extern "C"` attribute, replace overloaded functions by functions with different names, and replace member functions by `friend` functions.

External functions with the `__declspec(dllimport)` attribute have prefix `__imp_` in all compilers except Borland.

Functions with the names `main` and `WinMain` always have `extern "C"` coding. In addition, some compilers give `WinMain` the `__stdcall` attribute by default.

8.12 Conclusion

Various characteristics of the different name mangling schemes are compared in table 22.

Table 22. Comparison of name mangling schemes

	Microsoft	Borland	Watcom	Gnu2	Gnu3
unambigous and reversible	yes	yes	yes	no	no
includes type of global objects	yes	no	yes	no	no
includes storage class	yes	no	yes	no	no
includes function return type	yes	no	yes	no	no
includes calling convention	yes	yes	no	(no)	(no)
includes function modifiers	yes	few	some	no	no
compact	yes	somewhat	yes	somewhat	yes
allows case insensitive linking	yes	no	no	no	no
human readable	no	yes	yes	yes	yes
non-C characters used	$ @ ?	$ @ %	$?()[]:.	$ or .	none

The Gnu2 and Gnu3 schemes use only characters that are valid in C names in most or all cases. The reason for this is that these schemes have their origin in tools that convert from C++ to C, so the mangled names must be valid C names. This has the disadvantage that the mangled name cannot unambiguously be translated back to the original C++ declaration. The mangled name of a C++ function could in principle be the unmangled name of a variable. This disadvantage is avoided if the mangled code contains characters that are not valid in C++ names. On the other hand, the character set should be restricted to characters that can be generated by common assemblers in order to allow compilation through assembly or linking with assembly language modules. The characters `$ ? @` are allowed in Microsoft and Borland assemblers. The Gnu assembler allows `.` and `$`. The Watcom assembler allows all characters in public symbols.

We will prefer that a name mangling scheme is complete, consistent and compact. It should also be relatively easy for humans to interpret the code, though this requirement conflicts with the desire for compactness. The Microsoft and Gnu3 schemes are the ones that have the most consistent syntax. It is recommended that new compilers use one of these two schemes for the sake of compatibility.

9 Exception handling and stack unwinding

An exception, a thread termination, or a `longjmp` can lead to a process where functions are exited without the normal return being executed. Objects that go out of scope by this process may have destructors that need to be called. In order to find all objects that need to have their destructors called, the system must unwind the stack to trace backwards through

consecutive function calls. Some systems also use stack unwinding for recovering registers saved on the stack after an exception.

If a function has any local objects with destructors and if an exception or `longjmp` or thread termination can occur inside the function or any of its child functions, then this function must support stack unwinding. Some ABI's require that all functions have stack unwinding information if the function saves anything on the stack or calls any other function. The method of stack unwinding is different for different systems. This process often uses stack frames based on `BP/EBP/RBP`. The function prolog must save the old value of the frame pointer and save the value of the stack pointer in the frame pointer register:

```
_FunctionWithFramePointer PROC NEAR
        PUSH    EBP
        MOV     EBP, ESP
        ...
        MOV     ESP, EBP
        POP     EBP
        RET
_FunctionWithFramePointer ENDP
```

Additional information about destructors to be called may be provided either by the function itself or by data in a static data segment designed for only this purpose.

If a structured exception or `longjmp` or thread termination can happen inside a function or any of its child functions, then it is not allowed to use `BP/EBP/RBP` for any other purpose than a frame pointer in systems that rely on `BP` being a frame pointer.

Detailed information about specification of the unwind mechanism for x64 systems can be found in the respective ABI's. See literature, page 56. I don't have the detailed information for 32-bit systems.

10 Initialization and termination functions

A C++ module may contain global objects with constructors that must be called before `main` is called, and destructors to be called after `main` has returned. There may be other initialization and termination tasks to perform, too. For this purpose, the compiler provides a list of pointers to initialization functions and termination functions. These lists of function pointers are stored in separate data segments designed for only this purpose. These lists may contain additional information about the priority or order in which the initialization and termination functions should be called. The names of these segments and the format of these tables are different for different compilers.

11 Virtual tables and runtime type identification

A class with virtual member functions always has a virtual table. This is a table of member function pointers used for finding the right version of a polymorphous function. Each instance of the class has a pointer to the virtual table. Microsoft, Borland and Gnu version 3.x compilers place the pointer to the virtual table at the beginning of the object, while Watcom and Gnu version 2.x compilers place it at the end of the object. This affects the offset of all data members of the class so that member functions may be incompatible between compilers.

Information for runtime type identification is usually stored in connection with the virtual table.

The "Itanium C++ ABI" includes more detailed information about the representation of virtual tables and runtime type identification. This information may apply to other platforms as well.

12 Communal data

Communal data are data that may occur identically in more than one module, but the final executable should contain no more than one instance of this redundant information. The linker may check that all instances are identical and store only one instance in the executable file. Communal data are used for virtual tables, template instantiations, and possibly for global data.

A different use of communal data is for data that may or may not be needed in the final executable. This is typically the case with the non-inlined versions of inlined functions. The compiler does not need this function in the module it is currently compiling, but it may or may not be called from a different module. The compiler can then allow the linker to remove the communal function if it is not needed.

You cannot expect communal data produced by different compilers to be identical or to be identified in the same way in the object files. Assemblers may not support communal data.

13 Memory models

A memory model defines the address ranges and addressing modes for code and data. Different memory models are used for 16-, 32- and 64-bit systems.

13.1 16-bit memory models

The 16-bit DOS operating system has no protection against accessing false addresses and no distinction between physical and logical (virtual) memory addresses. A protected operating system can emulate the 8086 memory space using the virtual 8086 mode of a 32-bit processor. The memory space is divided into segments no bigger than 64 kbytes. The real address of an object is equal to the segment multiplied by 16 plus the 16-bit unsigned offset. The following memory models are used:

Tiny

Code and data is contained in the same segment no bigger than 64 kbytes. The code starts at address 0x100 relative to the segment. The executable file does not have the usual extension `.exe` but instead `.com`.

Small

There are two segments of max. 64 kbytes each, one for code and one for data and stack.

Medium

The code can exceed 64 kbytes. Far function calls are needed. Data and stack limited to one segment of max. 64 kbytes.

Compact

The code is limited to 64 kbytes. The stack is limited to 64 kbytes. Data can exceed 64 kbytes. Far pointers are used for data.

Large

The code can exceed 64 kbytes. Data can exceed 64 kbytes. The stack is limited to 64 kbytes. Far pointers are used for code and data.

Huge

Same as large. A data structure can exceed 64 kbytes by modifying not only the offset but also the segment of a pointer when it is incremented.

Protected

Windows 3.x uses protected memory. It is similar to the above models, but segment registers contain segment selectors rather than physical addresses. Data structures bigger than 64 kbytes can be accessed by adding 8 to the segment descriptor for each 64 kbytes increment or by using a 32-bit offset in case a 32-bit processor is used.

13.2 32-bit memory models

32-bit Windows, Linux, BSD and Intel-based Mac all use the flat memory model. Application code uses only one segment with a maximum size of 2 Gbytes. All pointers use 32-bit signed addresses. Negative addresses are reserved for the operating system kernel and device drivers.

13.3 64-bit memory models in Windows

The combined size of code and static data is limited to 2 Gbytes so that 32-bit self-relative (rip-relative) addresses can be used. The image base (see chapter 14) is usually below 2^{31}, but not always. 32-bit absolute addresses are rarely used. Dynamically allocated data and stack can exceed 2 Gbytes. Pointers are usually 64 bits. 32-bit pointers relative to the image base are sometimes used for arrays and pointer tables. Negative addresses are reserved for the system kernel.

13.4 64-bit memory models in Linux and BSD

Linux x64 small memory model

This is the default memory model in x64 Linux and BSD. Code and static data are limited to 2 Gbytes and are always stored at addresses below 2^{31}. This allows the compiler to use 32-bit signed absolute addresses, typically for addressing static arrays. Dynamically allocated data and stack can exceed 2 Gbytes. Pointers are usually 64 bits.

Linux x64 medium memory model

Static data objects bigger than the "large-data-threshold", typically 64 kbytes, are stored in a large data section which can exceed 2 Gbytes. Code and smaller static data are still limited to addresses below 2^{31}. The compiler option is `-mcmodel=medium`.

Linux x64 large memory model

Code and data can exceed 2 Gbytes. Functions and static data are accessed with 64 bit absolute addresses, which is quite inefficient. The compiler option is `-mcmodel=large`.

Linux x64 position-independent model

This model is used for shared objects (dynamic libraries). The size of each executable or shared object is limited to 2 Gbytes. Functions and data inside the executable are accessed with 32-bit relative addresses. External functions and data are accessed through 64-bit pointers. The compiler option is `-fpic` or `-fpie` (the latter option avoids the use of GOT and PLT tables for accessing local data and functions).

Linux x64 kernel

The system kernel and device drivers use negative addresses between -2^{31} and 0.

13.5 64-bit memory models in Intel-based Mac (Darwin)

The default memory model limits the combined size of code and static data in each executable file to 2 Gbytes so that 32-bit self-relative (rip-relative) addresses can be used.

By default, all code is loaded at addresses above 2^{32}. The address space below 2^{32} (pagezero) is blocked so that any attempt to use 32-bit absolute addresses will generate an error. Dynamically allocated data and stack can exceed 2 Gbytes. Pointers are usually 64 bits. Pointer tables can use 32-bit signed addresses relative to an arbitrary reference point. Certain system functions can be accessed at fixed 64-bit addresses in the so-called commpage.

It is possible to place code at addresses below 2^{31} and reduce the size of pagezero so that 32-bit absolute addresses can be used, but this is rarely done.

14 Relocation of executable code

Most operating systems use the same, or almost the same, file type for executable files and object files. An executable file may need relocation when it is loaded into memory, depending on the load address. The load address or image base is the virtual memory address at which the beginning of the executable file is placed. Relocation is a process where all cross-references using absolute memory addresses in the file are modified according to the load address.

The executable file may be relocated by the linker to a preferred load address. If the preferred load address is not vacant then the executable file has to be relocated again by the loader to another load address.

The most common values for the preferred load address are: 0x400000 for 32-bit and 64-bit Windows and 64-bit Linux systems; 0x8048000 for 32-bit Linux; 0x1000 for 32-bit Mach-O systems; 0x100000000 for 64-bit Mach-O systems. Other positive values can be used as well. The linker can adjust all cross references in the exe file according to the preferred load address. If a process has only one executable file, then the operating system can map the physical memory address at which this file is loaded to the desired virtual address, e.g. 0x400000. A load address less than 2^{31} = 0x80000000 makes it possible to use a small memory model where 32-bit signed absolute addresses can be used in 64-bit systems. Negative addresses are reserved for the operating system kernel.

A dynamically linked library or shared object will need relocation at load time if the preferred load address is occupied by another library. A dynamic library will usually need at least two memory pages: a shared memory page for the code section and possibly read-only data, and a non-shared memory page for the writable data section. The size of a memory page is either 4 kbytes or 2 Mbytes.

Relocation in Windows

The preferred load address for the main executable is traditionally 0x400000, but other addresses are possible. The preferred load address for a DLL differs. A DLL may be loaded at an address higher than 2^{31} in 64-bit mode. There is no compiler option to distinguish between main executables and DLLs. Therefore, 32-bit absolute addresses are rarely used in 64-bit Windows. Instead, code and data within the same executable or DLL are accessed with 32-bit addresses relative to the instruction pointer (rip-relative) or relative to the load address (image-relative).

Multiple processes can use the same virtual memory addresses mapped to different physical addresses. When a DLL is shared between multiple processes, the sections typically have the same virtual addresses in all processes when the DLL is loaded at the preferred address. If relocation is needed in the code section then there will be multiple instances of the code section, while only explicitly shared data sections will be shared.

Relocation in Linux

The preferred load address for the main executable is often 0x8048000 in 32-bit mode and 0x400000 in 64-bit mode.

Shared objects may be loaded at negative addresses in 32-bit mode, but not in 64-bit mode where addresses above 2^{31} are used for shared objects.

When a shared object is shared between multiple processes, the same sections have different virtual addresses in the different processes. The distance between the code section and the data section is the same for all processes so that relative addresses can be used. The code section of a shared object is shared between processes unless it is modified by relocation. A shared object is usually compiled as position-independent code (option `-fpic`) to avoid relocations in the code section.

32-bit absolute addresses are used in the main executable, but not in shared objects in 64-bit mode. A 64-bit shared object must therefore be compiled as position-independent code (option `-fpic` or `-fpie`) to avoid 32-bit absolute addresses.

Shared objects usually have global offset tables (GOT) with pointers to static and global data, and procedure linkage tables (PLT) with pointers to global functions. All global symbols are accessed through these pointer tables when the compiler option `-fpic` is used. In 32-bit mode, the GOT is also used for local variables. The purpose of these tables is to mimic the behavior of static libraries. If a function in the main executable has the same name as a function in the shared object then the version in the main executable will be used, even when called from the shared object. The same applies to global variables. This feature that allows local access inside the shared object to be redirected comes at a high price because all accesses must go through the GOT and PLT tables. See the chapter "Static versus dynamic libraries" in manual 1: "Optimizing software in C++" for tips about how to avoid these pointer tables.

Relocation in BSD

This is similar to Linux.

Relocations in Mac OS X

Position-independent code is used by default even for the main executable, though this is not necessary. It is possible to speed up the main executable in 32-bit mode by making the code position-dependent (option `-fno-pic`).

In 64-bit mode, most addresses use a 32-bit signed offset relative to the instruction pointer or to a reference point.

Shared objects rarely use global offset tables (GOT). Procedure linkage tables are not used for internal references.

When a shared object is shared between multiple processes, the same sections have the same virtual addresses in the different processes.

Relocations in 16-bit systems

16-bit DOS and Windows 3.x systems use segmented memory models where all memory addresses are of the segment:offset kind. Executable files must be relocated when loaded because the operating system cannot map the image to an arbitrary address. Only the segment part of a segment:offset address needs to be relocated. In DOS, the segment address is modified in the executable code. In Windows 3.x the segment descriptor table is modified.

The 16-bit systems do not use the same file format for executable files as for object files.

14.1 Import tables

Almost all executable files contain function calls to functions in other DLLs or shared objects. In most cases, these function calls go via an import table in the executable file. The loader fills the import table with the addresses of the external functions when the program and the DLLs or shared objects are loaded. The implementation of the import table differs among systems.

Some systems allow lazy binding of external references. Lazy binding means that the address of the external function is not inserted in the import table until the first time the function is called. The advantage of lazy binding is that the address of an external function needs not be calculated in case the function is never called. The disadvantage is a considerable delay the first time each external function is called.

It is often possible to use static linking instead of dynamic linking for calls to library functions. The idea of static linking is that the required library functions, but not the entire function library, are copied into the executable file by the linker. The references to these functions can then be resolved at link time rather than at load time.

15 Object file formats

There are at least four different object file formats in common use for x86 platforms. These are OMF, COFF also called PE, ELF and Mach-O format. The a.out format is rarely used any more.

15.1 OMF format

OMF stands for Object Module Format. This format is also called Microsoft 8086 relocatable. The OMF format is used for 16-bit operating systems (DOS, Windows 3.x and earlier). Some compilers (Borland, Watcom, Symantec, Digital Mars) also use OMF format for 32-bit Windows.

This format was originally designed for the segmented memory model of the 16-bit 8088 microprocessor, used in the first IBM PCs. The OMF format allows the resolution of addresses relative to an arbitrary reference frame representing a segment or group of segments. A reference frame starts at an address divisible by 16 and can span 64 kbytes. Reference frames are allowed to overlap. Overlays are supported (i.e. allowing multiple pieces of code to share the same memory space).

The OMF object file consists of a chain of records where the first byte of each record indicates the type of data contained in the record. No record can contain more than 1 kbyte of data.

The OMF format was designed for compactness at the time of the first IBM PCs where the only means of storage was one or two 360 kbytes floppy disk drives. The OMF format allows several different methods for compressing bytes that are zero or repeated, not only in the binary code and data, but also in relocation tables. Repeat-blocks can be nested to an unlimited depth. These compression features make the interpretation of OMF files complicated and error-prone. The method for compressing repeated relocation information is hardly used any more, if it ever was. Repeat-blocks in data segments are still used by some tools. The Microsoft assembler can generate relocations in repeated data although this is discouraged and not supported by all linkers.

The OMF standard also specifies a format for static libraries. The OMF library uses a hash table for listing the public symbols of all modules in the library. Other formats use a simple sorted list or even an unsorted list for the same purpose. The use of a hash table requires

that all linkers and library managers use exactly the same hashing algorithm. Unfortunately, the official definition of the hashing algorithm is not as clear and stringent as one could wish for. The gain in efficiency from using a hash table rather than a sorted list is minimal, and not enough to justify the considerable increase in complexity, in my opinion.

An OMF library can have an optional "extended dictionary" in addition to the hash table. The extended dictionary specifies dependencies between modules in order to facilitate one-pass linking. There are two different and incompatible formats for the extended dictionary. The original IBM/Microsoft format and a proprietary format used by Borland. Extended dictionaries are rarely used.

Limitations:

Segment word size: 16 or 32 bits. Can be mixed.
Segment alignment: Supports only 1, 2, 4, 16, and either 256 or 4096.
Max identifier length: 255 characters.
Max external symbols: 64 k.
Max number of segments: 64 k.
Max segment size: 4 Gbytes.
Max number of modules in library: < 64 k.
Some old linkers limit the size of the library hash table to 251 blocks of 37 buckets each. This limits the number of public symbols in a library to between 251 and 9287, depending on the lengths of the names. Even for linkers that allow more than 251 blocks there are somewhat unpredictable limitations in the hash table.

15.2 COFF format

COFF stands for Common Object File Format. This format was first used in UNIX system V, but later superseded by ELF. A modification of the COFF format is used in Windows. The Windows version of COFF is also called PE (Portable Executable). The 64-bit version is called PE32+. The same format is used for object files and executable files. The COFF format used under windows is a Microsoft adaptation of the COFF format used on certain other platforms.

The COFF format is used for object files in 32-bit and 64-bit Windows by Microsoft, Intel and Gnu compilers. The Codeplay compiler for 32-bit Windows can use the OMF, COFF or ELF32 formats.

The COFF format uses many different data structures, which makes it somewhat complicated. The definition of the data structures is not in agreement with the default alignment rules of modern compilers.

Different object file formats differ in the way self-relative references are implemented, due to the way the CPU calculates relative addresses. The CPU calculates self-relative addresses relative to the value of the instruction pointer after the instruction, not relative to the position of the address field in the instruction code. The distance from the position of the address field (relocation source) to the reference point used by the CPU is 2 in 16-bit code, 4 in 32-bit code, and 4, 5, 6 or 8 in 64-bit code. The COFF and OMF formats have different relocation codes for each of these distances so that the necessary correction can be inserted by the linker. The ELF and Mach-O formats have no inherent recognition of this difference, so that the correction must be inserted as an explicit addend in the object file.

The COFF format allows the specification of image-relative references, which are not available on other platforms. Image-relative references are used in 32-bit mode for debug information. 32-bit image-relative references are used in 64-bit mode for exception handling information and in general for saving space where 64-bit pointers would otherwise be needed.

Limitations:

Segment word size: 32 or 64 bits.
Max number of sections: 32 k.
Max file size: 4 Gbytes.
Max section size: 4 Gbytes.
Max relocations per section: 64 k.
Max library size: 4 Gbytes.

15.3 ELF format

ELF stands for Executable and Linkable Format. This format has replaced older formats like a.out and COFF in Linux and BSD. Gnu tools running under Linux and BSD often accept several other formats, not including the formats used under Windows. Gnu tools running under Windows accept COFF and ELF formats.

The ELF format is designed to be flexible and expandable. The sizes of all data structures are specified explicitly so that they can be expanded without loosing backwards compatibility. This makes the ELF format far more clear and robust than other formats.

Limitations:

Segment word size: 32 or 64 bits.
Max number of sections: 64 k.
Max file size: 4 Gbytes for 32 bits, 2^{64} bytes for 64 bits.
Max section size: 4 Gbytes for 32 bits, 2^{64} bytes for 64 bits.
Max string table size: 4 Gbytes.
Max number of symbols: 16 M for 32 bits, 4 G for 64 bits.
Max library size: 4 Gbytes.

15.4 Mach-O format

Mach-O stands for Mach Object. This format is used in Mac OS and the older Mach OS systems for object files and executable files. The following description applies only to the Intel-based Mac OS X (Darwin) system.

Mach-O object files have only one segment record comprising several section records. Mach-O executable files have several segment records.

The Mach-O format allows the specification of addresses relative to an arbitrary reference point in an arbitrary section. This addressing method, which is not available in the other object file formats, is used for position-independent code.

Position-independent code and lazy binding is used by default by the Gnu compiler for 32-bit Mac OS X. This makes code execution less efficient. Position-independent code is required only for shared objects (dynamic link libraries) in Mac OS X.

Local symbols are referenced by their address only in 32-bit Mach-O files, where the COFF and ELF formats have symbol table entries for local symbols.

Limitations:

Section name length: 16 characters.
Max file size: 4 Gbytes.
Max section size: 16 M for position-indep. code, 4 Gbytes for 32 bits, 2^{64} bytes for 64 bits.
Max number of sections: 16 M.
Max number of symbols: 16 M.
Max library size: 4 Gbytes.

15.5 a.out format

a.out stands for Assembler Output. This format is used in older versions of UNIX and similar systems. The name `a.out` remains as the default name for linker output files, even though these files are not in a.out format any more. Some linkers still support the a.out format.

15.6 Comparison of object file formats

The ELF format stands out as the most consistent, clear, robust and flexible of the object file formats. The other formats are full of patches and appear kludgy in comparison. I would recommend the ELF format for new applications. The OMF format should be used only for 16-bit applications.

15.7 Conversion between object file formats

It is often possible to convert object files from one format to another. A tool for this purpose named `objconv` is available at www.agner.org/optimize. This tool can also be used as a cross-platform library manager. See the `objconv` manual for details and mentioning of other relevant tools.

Conversion of an object file will fail if the file contains references of a type that is not supported by the target format, e.g. image-relative references in COFF files or position-independent code in Mach-O files.

An object file that has been converted from one platform to another will work on the target platform if all calling conventions etc. are the same on both platforms and there is no reference to platform-specific library functions or system functions.

Differences in name-mangling conventions can be fixed by using `extern "C"` declarations or by using `objconv` to change symbol names in the object file.

The use of converted object files can fail to work for many reasons. All the compatibility problems described in the present document should be considered in order to predict whether object file conversion is likely to work. While the calling conventions are almost the same in all 32-bit x86 systems, they are quite different in 64-bit systems. A call stub is needed for converted functions in 64-bit systems.

Conversion of compiler-generated object files should be used only as a last resort when the source code is not available. Conversion of object files made from assembly code can be expected to work if the assembly source is carefully inspected for possible compatibility problems.

15.8 Intermediate file formats

Some compilers have features for optimizing code across function calls and modules (whole program optimization). These compilers use intermediate files containing partially compiled code. The format for these intermediate files is not standardized. It is not even guaranteed to be compatible between different versions of the same compiler. The intermediate file format is therefore not suitable for distributing function libraries.

It would be useful to have a standardized object file format that includes information about which registers each function modifies, in order to optimize register allocation. Such a file format has not been implemented for any of the platforms I have studied.

Java compilers generate an intermediate code called Java bytecode. This code is either interpreted or just-in-time compiled by a Java machine on the target platform. The byte code

is platform independent and needs no translation. Java bytecode is less efficient than compiled machine code.

A similar technology is used by C++, C# and Visual Basic compilers for the Microsoft .NET platform. The bytecode is just-in-time compiled by the .NET runtime framework on the target machine. The bytecode is expected to work on any platform for which a .NET framework exists. .NET bytecode is less efficient than compiled machine code.

16 Debug information

Compilers differ in the way they store debugging information and information for profilers in object files and executable files. Thus, it may not be possible to use the same debugger for different compilers, even on the same platform. The information stored includes names of source files, line numbers, and variable names. I have not studied the details of how debug information is stored.

17 Data endian-ness

All systems based on 16, 32 and 64 bit x86 microprocessors use little-endian data storage, i.e. the least significant byte of a multi-byte data unit is stored at the lowest address. Many other microprocessor platforms use big-endian data storage. This can give rise to compatibility problems when exchanging binary data files between platforms, and when porting C++ programs that explicitly address part of a data object, such as the sign bit of a floating point number.

18 Predefined macros

Most C++ compilers have a predefined macro containing the version number of the compiler. Programmers can use preprocessing directives to check for the existence of these macros in order to detect which compiler the program is compiled on and thereby fix problems with incompatible compilers.

Table 23. Compiler version predefined macros

Compiler	Predefined macro
Borland	`__BORLANDC__`
Codeplay VectorC	`__VECTORC__`
Digital Mars	`__DMC__`
Gnu	`__GNUC__`
Intel	`__INTEL_COMPILER`
Microsoft	`_MSC_VER`
Pathscale	`__PATHSCALE__`
Symantec	`__SYMANTECC__`
Watcom	`__WATCOMC__`

Unfortunately, not all compilers have well-documented macros telling which hardware platform and operating system they are compiling for. The following macros may or may not be defined:

Table 24. Hardware platform predefined macros

Hardware platform	Predefined macro
x86	`_M_IX86`, `__INTEL__`, `__i386__`
x86-64	`_M_X64`, `__x86_64__`, `__amd64`
IA64	`__IA64__`
DEC Alpha	`__ALPHA__`
Motorola Power PC	`__POWERPC__`
Any little endian	`__LITTLE_ENDIAN__`
Any big endian	`__BIG_ENDIAN__`

Table 25. Operating system predefined macros

Operating system	Predefined macro
DOS 16 bit	`__MSDOS__`, `_MSDOS`
Windows 16 bit	`_WIN16`
Windows 32 bit	`_WIN32`, `__WINDOWS__`
Windows 64 bit	`_WIN64`, `_WIN32`
Linux 32 bit	`__unix__`, `__linux__`
Linux 64 bit	`__unix__`, `__linux__`, `__LP64__`, `__amd64`
BSD	`__unix__`, `__BSD__`, `__FREEBSD__`
Mac OS	`__APPLE__`, (`__DARWIN__`, `__MACH__`)
OS/2	`__OS2__`

A more comprehensive list of predefined macros can be found at predef.sf.net.

19 Available C++ Compilers

19.1 Microsoft

Microsoft Visual C++ comes in several different versions. The professional edition, which is relatively expensive, includes integrated development environment and many tools including debugger and profiler. A limited "express" edition is available for free. Command line versions of the C++ compilers and assemblers are available for free in the Platform Software Development Kit (PSDK) from www.microsoft.com.

19.2 Borland

Borland C++ compilers and development tools are available in several different versions. A command line version can be downloaded for free from www.borland.com.

19.3 Watcom

Watcom C++ is no longer sold commercially, but it has been continued as an open source project. The Watcom compiler is available from www.openwatcom.org, including integrated development environment, debugger, profiler, assembler, disassembler, and other tools. The old commercial version is compatible with the new open source version.

Users of this compiler should be aware that register usage and calling conventions differ from other compilers. You must fix these problems by using `#pragma`'s when defining or calling DLL functions and when combining with tools from other vendors.

19.4 Gnu

Gnu C++ is an open source compiler that comes with most distributions of Linux, BSD and Mac OS. Windows versions are available from www.cygwin.com and www.mingw.org. Also available is an assembler (GAS) which uses the AT&T syntax by default.

19.5 Clang

Clang is a front end for the Low Level Virtual Machine (LLVM) open source compiler. It is the default compiler for newer versions of Mac OS, but is also available for Linux and Windows.

19.6 Digital Mars

Digital Mars C++ compiler is a continuation of Zortech C++ and Symantec C++, available from www.digitalmars.com. The compiler package is cheap, and a command line version is available for free. Both Symantec C++ and Digital Mars C++ are binary compatible with Microsoft C++ in most respects, including calling conventions and name mangling.

19.7 Codeplay

The Codeplay VectorC C++ compiler is available from www.codeplay.com. Can be used as a plug-in to older versions of Microsoft Visual Studio or as a command line compiler. 14 days free trial.

19.8 Intel

Intel compilers are available for Windows, Linux and Intel-based Max OS X. This is a commercial compiler with time-limited trial versions. The Intel compilers for Windows are binary compatible with Microsoft compilers. The Intel compilers for Linux are binary compatible with Gnu compilers. www.intel.com.

20 Literature

20.1 ABI's for Unix, Linux, BSD and Mac OS X (Intel-based).

Despite its name, the "Itanium C++ ABI" applies to other hardware platforms than the Itanium, except for a few processor-specific details, though not all x86 compilers conform to this ABI. The "Itanium C++ ABI" contains valuable information about the representation of member pointers, virtual tables, runtime type identification and name mangling, not found anywhere else. Most of this information applies to 32-bit and 64-bit Gnu compilers for x86 platforms. See also refspecs.freestandards.org/.

- SYSTEM V. APPLICATION BINARY INTERFACE. Edition 4.1, 1996. www.caldera.com
- Itanium C++ ABI. Revision 1.86. Draft, 2005. www.codesourcery.com/cxx-abi/
- Itanium C++ ABI: Exception Handling. Revision: 1.22. Draft, 2005. www.codesourcery.com/cxx-abi/abi-eh.html

Linux, BSD, Mac OS 32 bits:

- SYSTEM V. APPLICATION BINARY INTERFACE. Intel386 Architecture Processor Supplement. Fourth Edition.

Mac OS X IA32

- Mac OS X ABI Function Call Guide. 2006-04-04. developer.apple.com/documentation/DeveloperTools/Conceptual/LowLevelABI/ This appears to be the most up-to-date specification of the IA32 ABI. Many of the specifications, but not all, apply to Linux and BSD platforms as well.

Linux and BSD, 64 bits:

- System V Application Binary Interface. AMD64 Architecture Processor Supplement. Draft Version 0.99.5, 2010. www.x86-64.org/.

Amendment for YMM registers:

- Upcoming Intel64 Instruction Set Architecture Extensions - Intel Advanced Vector Extensions (Intel AVX). intel.wingateweb.com/SHchina/published/NGMS002/SP_NGMS002_100r_eng.pdf, 2008.

20.2 ABIs for Windows

Windows, 32 bits:

- C++ Language Reference: Calling Conventions. msdn.microsoft.com, 2006.

Windows, 64 bits:

- x64 Software Conventions. msdn.microsoft.com, 2006.

Amendment for YMM registers:

- Intel® Advanced Vector Extensions Programming Reference. http://software.intel.com/en-us/avx/
- Upcoming Intel®64 Instruction Set Architecture Extensions -Intel®Advanced Vector Extensions (Intel®AVX). Intel Developer Forum 2008.

20.3 Object file format specifications

OMF:

- Tool Interface Standards (TIS): Relocatable Object Module Format (OMF) Specification. Version 1.1. TIS Committee, 1995.

- United States Patent 5408665, 1995. Describes the Borland library extended dictionary format.

COFF:

- Visual Studio, Microsoft Portable Executable and Common Object File Format Specification. Revision 8.0, 2006. download.microsoft.com.

ELF:

- Executable and Linkable Format (ELF). Version1.1. Tool Interface Standards (TIS).

Mach-O:

- Mac OS X ABI Mach-O File Format Reference, 2007. Apple Computer, Inc. developer.apple.com.

21 Copyright notice

This series of five manuals is copyrighted by Agner Fog. Public distribution and mirroring is not allowed. Non-public distribution to a limited audience for educational purposes is allowed. The code examples in these manuals can be used without restrictions. A GNU Free Documentation License shall automatically come into force when I die. See www.gnu.org/copyleft/fdl.html.

22 Acknowledgments

Thank you to the many people who have sent me additional information and corrections for my manuals and provided other kinds of help. This allows me to keep improving these manuals.