Chapter 3

Low Level System Information

3.1 Machine Interface

3.1.1 Processor Architecture

3.1.2 Data Representation

Within this specification, the term *byte* refers to a 8-bit object, the term *twobyte* refers to a 16-bit object, the term *fourbyte* refers to a 32-bit object, the term *eightbyte* refers to a 64-bit object, and the term *sixteenbyte* refers to a 128-bit object.[1]

Fundamental Types

Figure 3.1 shows the correspondence between ISO C's scalar types and the processor's. `__int128`, `__float80`, `__float128`, `__m64`, `__m128`, `__m256` and `__m512` types are optional.

The `__float128` type uses a 15-bit exponent, a 113-bit mantissa (the high order significant bit is implicit) and an exponent bias of 16383.[2]

The `long double` type uses a 15 bit exponent, a 64-bit mantissa with an explicit high order significant bit and an exponent bias of 16383.[3] Although a `long`

[1]The Intel386 ABI uses the term *halfword* for a 16-bit object, the term *word* for a 32-bit object, the term *doubleword* for a 64-bit object. But most IA-32 processor specific documentation define a *word* as a 16-bit object, a *doubleword* as a 32-bit object, a *quadword* as a 64-bit object and a *double quadword* as a 128-bit object.

[2]Initial implementations of the AMD64 architecture are expected to support operations on the `__float128` type only via software emulation.

[3]This type is the x87 double extended precision data type.

Figure 3.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	AMD64 Architecture
Integral	`_Bool`†	1	1	boolean

	`char`
	`signed char`
		1	1	signed byte

	`unsigned char`	1	1	unsigned byte

	`short`
	`signed short`
		2	2	signed twobyte

	`unsigned short`	2	2	unsigned twobyte

	`int`
	`signed int`
	`enum`†††
		4	4	signed fourbyte

	`unsigned int`	4	4	unsigned fourbyte

	`long (LP64)`
	`signed long (LP64)`
		8	8	signed eightbyte

	`unsigned long (LP64)`	8	8	unsigned eightbyte

	`long (ILP32)`
	`signed long (ILP32)`
		4	4	signed fourbyte

	`unsigned long (ILP32)`	4	4	unsigned fourbyte

	`long long`
	`signed long long`
		8	8††††	signed eightbyte

	`unsigned long long`	8	8††††	unsigned eightbyte
	`__int128`†† `signed __int128`††	16 16	16 16	signed sixteenbyte signed sixteenbyte
	`unsigned __int128`††	16	16	unsigned sixteenbyte
Pointer	*any-type* `* (LP64)` *any-type* `(*)() (LP64)`	8	8	unsigned eightbyte
	any-type `* (ILP32)` *any-type* `(*)() (ILP32)`	4	4	unsigned fourbyte
Floating-point	`float`	4	4	single (IEEE-754)
	`double`	8	8††††	double (IEEE-754)
	`__float80`†† `long double`†††††	16 16	16 16	80-bit extended (IEEE-754) 80-bit extended (IEEE-754)
	`__float128`†† `long double`†††††	16 16	16 16	128-bit extended (IEEE-754) 128-bit extended (IEEE-754)
Decimal-floating-point	`_Decimal32`	4	4	32bit BID (IEEE-754R)
	`_Decimal64`	8	8	64bit BID (IEEE-754R)
	`_Decimal128`	16	16	128bit BID (IEEE-754R)
Packed	`__m64`††	8	8	*MMX* and 3DNow!
	`__m128`††	16	16	SSE and SSE-2
	`__m256`††	32	32	AVX
	`__m512`††	64	64	AVX-512

† This type is called `bool` in C++.
†† These types are optional.
††† C++ and some implementations of C permit enums larger than an int. The underlying type is bumped to an unsigned int, long int or unsigned long int, in that order.
†††† The `long long`, `signed long long`, `unsigned long long` and `double` types have 4-byte alignment in the Intel386 ABI.
††††† The `long double` type is 128-bit, the same as the `__float128` type, on the Android™ platform. More information on the Android™ platform is available from `http://www.android.com/`.

double requires 16 bytes of storage, only the first 10 bytes are significant. The remaining six bytes are tail padding, and the contents of these bytes are undefined.

The __int128 type is stored in little-endian order in memory, i.e., the 64 low-order bits are stored at a a lower address than the 64 high-order bits.

A null pointer (for all types) has the value zero.

The type size_t is defined as unsigned long for LP64 and unsigned int for ILP32.

Booleans, when stored in a memory object, are stored as single byte objects the value of which is always 0 (false) or 1 (true). When stored in integer registers (except for passing as arguments), all 8 bytes of the register are significant; any nonzero value is considered true.

Like the Intel386 architecture, the AMD64 architecture in general does not require all data accesses to be properly aligned. Misaligned data accesses are slower than aligned accesses but otherwise behave identically. The only exceptions are that __m128, __m256 and __m512 must always be aligned properly.

Aggregates and Unions

Structures and unions assume the alignment of their most strictly aligned component. Each member is assigned to the lowest available offset with the appropriate alignment. The size of any object is always a multiple of the object‘s alignment.

An array uses the same alignment as its elements, except that a local or global array variable of length at least 16 bytes or a C99 variable-length array variable always has alignment of at least 16 bytes.[4]

Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

Bit-Fields

C struct and union definitions may include bit-fields that define integral values of a specified size.

The ABI does not permit bit-fields having the type __m64, __m128, __m256 or __m512. Programs using bit-fields of these types are not portable.

[4]The alignment requirement allows the use of SSE instructions when operating on the array. The compiler cannot in general calculate the size of a variable-length array (VLA), but it is expected that most VLAs will require at least 16 bytes, so it is logical to mandate that VLAs have at least a 16-byte alignment.

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed char`	1 to 8	-2^{w-1} to $2^{w-1}-1$
`char`		0 to $2^{w}-1$
`unsigned char`		0 to $2^{w}-1$
`signed short`	1 to 16	-2^{w-1} to $2^{w-1}-1$
`short`		0 to $2^{w}-1$
`unsigned short`		0 to $2^{w}-1$
`signed int`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`int`		0 to $2^{w}-1$
`unsigned int`		0 to $2^{w}-1$
`signed long (LP64)`	1 to 64	-2^{w-1} to $2^{w-1}-1$
`long (LP64)`		0 to $2^{w}-1$
`unsigned long (LP64)`		0 to $2^{w}-1$
`long (ILP32)`	1 to 32	0 to $2^{w}-1$
`unsigned long (ILP32)`		0 to $2^{w}-1$
`signed long long`	1 to 64	-2^{w-1} to $2^{w-1}-1$
`long long`		0 to $2^{w}-1$
`unsigned long long`		0 to $2^{w}-1$

Bit-fields that are neither signed nor unsigned always have non-negative values. Although they may have type char, short, int, or long (which can have negative values), these bit-fields have the same range as a bit-field of the same size with the corresponding unsigned type. Bit-fields obey the same size and alignment rules as other structure and union members.

Also:

- bit-fields are allocated from right to left
- bit-fields must be contained in a storage unit appropriate for its declared type
- bit-fields may share a storage unit with other struct / union members

Unnamed bit-fields' types do not affect the alignment of a structure or union.

3.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nevertheless, it is recommended that all functions use the standard calling sequence when possible.

3.2.1 Registers

The AMD64 architecture provides 16 general purpose 64-bit registers. In addition the architecture provides 16 SSE registers, each 128 bits wide and 8 x87 floating point registers, each 80 bits wide. Each of the x87 floating point registers may be referred to in *MMX*/3DNow! mode as a 64-bit register. All of these registers are global to all procedures active for a given thread.

Intel AVX (Advanced Vector Extensions) provides 16 256-bit wide AVX registers (`%ymm0` - `%ymm15`). The lower 128-bits of `%ymm0` - `%ymm15` are aliased to the respective 128b-bit SSE registers (`%xmm0` - `%xmm15`). Intel AVX-512 provides 32 512-bit wide SIMD registers (`%zmm0` - `%zmm31`). The lower 128-bits of `%zmm0` - `%zmm31` are aliased to the respective 128b-bit SSE registers (`%xmm0` - `%xmm31`[5]). The lower 256-bits of `%zmm0` - `%zmm31` are aliased to the respective 256-bit AVX registers (`%ymm0` - `%ymm31`[6]). For purposes of parameter passing and function return, `%xmmN`, `%ymmN` and `%zmmN` refer to the same register. Only one of them can be used at the same time. We use vector register to refer to either SSE, AVX or AVX-512 register. In addition, Intel AVX-512 also provides 8 vector mask registers (`%k0` - `%k7`), each 64-bit wide.

This subsection discusses usage of each register. Registers `%rbp`, `%rbx` and `%r12` through `%r15` "belong" to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function.[7] If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

[5] `%xmm15` - `%xmm31` are only available with Intel AVX-512.

[6] `%ymm15` - `%ymm31` are only available with Intel AVX-512.

[7] Note that in contrast to the Intel386 ABI, `%rdi`, and `%rsi` belong to the called function, not the caller.

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

The CPU shall be in x87 mode upon entry to a function. Therefore, every function that uses the *MMX* registers is required to issue an `emms` or `femms` instruction after using *MMX* registers, before returning or calling another function. [8] The direction flag `DF` in the `%rFLAGS` register must be clear (set to "forward" direction) on function entry and return. Other user flags have no specified role in the standard calling sequence and are *not* preserved across calls.

The control bits of the `MXCSR` register are callee-saved (preserved across calls), while the status bits are caller-saved (not preserved). The x87 status word register is caller-saved, whereas the x87 control word is callee-saved.

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Figure 3.3 shows the stack organization.

The end of the input argument area shall be aligned on a 16 (32 or 64, if `__m256` or `__m512` is passed on stack) byte boundary. In other words, the value (`%rsp` $+ 8$) is always a multiple of 16 (32 or 64) when control is transferred to the function entry point. The stack pointer, `%rsp`, always points to the end of the latest allocated stack frame. [9]

[8] All x87 registers are caller-saved, so callees that make use of the *MMX* registers may use the faster `femms` instruction.

[9] The conventional use of `%rbp` as a frame pointer for the stack frame may be avoided by using

The 128-byte area beyond the location pointed to by `%rsp` is considered to be reserved and shall not be modified by signal or interrupt handlers.[10] Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

3.2.3 Parameter Passing

After the argument values have been computed, they are placed either in registers or pushed on the stack. The way how values are passed is described in the following sections.

Definitions We first define a number of classes to classify arguments. The classes are corresponding to AMD64 register classes and defined as:

INTEGER This class consists of integral types that fit into one of the general purpose registers.

SSE The class consists of types that fit into a vector register.

SSEUP The class consists of types that fit into a vector register and can be passed and returned in the upper bytes of it.

X87, X87UP These classes consists of types that will be returned via the x87 FPU.

COMPLEX_X87 This class consists of types that will be returned via the x87 FPU.

NO_CLASS This class is used as initializer in the algorithms. It will be used for padding and empty structures and unions.

MEMORY This class consists of types that will be passed and returned in memory via the stack.

`%rsp` (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (`%rbp`) available.

[10]Locations within 128 bytes can be addressed using one-byte displacements.

Classification The size of each argument gets rounded up to eightbytes.[11]

The basic types are assigned their natural classes:

- Arguments of types (signed and unsigned) `_Bool`, `char`, `short`, `int`, `long`, `long long`, and pointers are in the INTEGER class.
- Arguments of types `float`, `double`, `_Decimal32`, `_Decimal64` and `__m64` are in class SSE.
- Arguments of types `__float128`, `_Decimal128` and `__m128` are split into two halves. The least significant ones belong to class SSE, the most significant one to class SSEUP.
- Arguments of type `__m256` are split into four eightbyte chunks. The least significant one belongs to class SSE and all the others to class SSEUP.
- Arguments of type `__m512` are split into eight eightbyte chunks. The least significant one belongs to class SSE and all the others to class SSEUP.
- The 64-bit mantissa of arguments of type `long double`
- The 64-bit mantissa of arguments of type `long double` belongs to class X87, the 16-bit exponent plus 6 bytes of padding belongs to class X87UP.
- Arguments of type `__int128` offer the same operations as INTEGERs, yet they do not fit into one general purpose register but require two registers. For classification purposes `__int128` is treated as if it were implemented as:

  ```
  typedef struct {
    long low, high;
  } __int128;
  ```

  with the exception that arguments of type `__int128` that are stored in memory must be aligned on a 16-byte boundary.
- Arguments of `complex T` where `T` is one of the types `float or double` are treated as if they are implemented as:

[11]Therefore the stack will always be eightbyte aligned.

```
struct complexT {
  T real;
  T imag;
};
```

- A variable of type `complex long double` is classified as type COMPLEX_X87.

The classification of aggregate (structures and arrays) and union types works as follows:

1. If the size of an object is larger than eight eightbytes, or it contains unaligned fields, it has class MEMORY [12].
2. If a C++ object is non-trivial for the purpose of calls, as specified in the C++ ABI [13], it is passed by invisible reference (the object is replaced in the parameter list by a pointer that has class INTEGER) [14].
3. If the size of the aggregate exceeds a single eightbyte, each is classified separately. Each eightbyte gets initialized to class NO_CLASS.
4. Each field of an object is classified recursively so that always two fields are considered. The resulting class is calculated according to the classes of the fields in the eightbyte:
   (a) If both classes are equal, this is the resulting class.
   (b) If one of the classes is NO_CLASS, the resulting class is the other class.

[12]The post merger clean up described later ensures that, for the processors that do not support the `__m256` type, if the size of an object is larger than two eightbytes and the first eightbyte is not SSE or any other eightbyte is not SSEUP, it still has class MEMORY. This in turn ensures that for processors that do support the `__m256` type, if the size of an object is four eightbytes and the first eightbyte is SSE and all other eightbytes are SSEUP, it can be passed in a register. This also applies to the `__m512` type. That is for processors that support the `__m512` type, if the size of an object is eight eightbytes and the first eightbyte is SSE and all other eightbytes are SSEUP, it can be passed in a register, otherwise, it will be passed in memory.

[13]See section 9.1 for details on C++ ABI.

[14]An object whose type is non-trivial for the purpose of calls cannot be passed by value because such objects must have the same address in the caller and the callee. Similar issues apply when returning an object from a function. See C++17 [class.temporary] paragraph 3.

   (c) If one of the classes is MEMORY, the result is the MEMORY class.
   (d) If one of the classes is INTEGER, the result is the INTEGER.
   (e) If one of the classes is X87, X87UP, COMPLEX_X87 class, MEMORY is used as class.
   (f) Otherwise class SSE is used.

5. Then a post merger cleanup is done:

   (a) If one of the classes is MEMORY, the whole argument is passed in memory.
   (b) If X87UP is not preceded by X87, the whole argument is passed in memory.
   (c) If the size of the aggregate exceeds two eightbytes and the first eightbyte isn't SSE or any other eightbyte isn't SSEUP, the whole argument is passed in memory.
   (d) If SSEUP is not preceded by SSE or SSEUP, it is converted to SSE.

Passing Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:

1. If the class is MEMORY, pass the argument on the stack.

2. If the class is INTEGER, the next available register of the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` is used[15].

3. If the class is SSE, the next available vector register is used, the registers are taken in the order from `%xmm0` to `%xmm7`.

4. If the class is SSEUP, the eightbyte is passed in the next available eightbyte chunk of the last used vector register.

5. If the class is X87, X87UP or COMPLEX_X87, it is passed in memory.

When a value of type `_Bool` is returned or passed in a register or on the stack, bit 0 contains the truth value and bits 1 to 7 shall be zero[16].

[15]Note that `%r11` is neither required to be preserved, nor is it used to pass arguments. Making this register available as scratch register means that code in the PLT need not spill any registers

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
`%rbx`	callee-saved register	Yes
`%rcx`	used to pass 4th integer argument to functions	No
`%rdx`	used to pass 3rd argument to functions; 2nd return register	No
`%rsp`	stack pointer	Yes
`%rbp`	callee-saved register; optionally used as frame pointer	Yes
`%rsi`	used to pass 2nd argument to functions	No
`%rdi`	used to pass 1st argument to functions	No
`%r8`	used to pass 5th argument to functions	No
`%r9`	used to pass 6th argument to functions	No
`%r10`	temporary register, used for passing a function's static chain pointer	No
`%r11`	temporary register	No
`%r12-r14`	callee-saved registers	Yes
`%r15`	callee-saved register; optionally used as GOT base pointer	Yes
`%xmm0–%xmm1`	used to pass and return floating point arguments	No
`%xmm2–%xmm7`	used to pass floating point arguments	No
`%xmm8–%xmm15`	temporary registers	No
`%mm0–%mm7`	temporary registers	No
`%k0–%k7`	temporary registers	No
`%st0,%st1`	temporary registers; used to return `long double` arguments	No
`%st2–%st7`	temporary registers	No
`%fs`	Reserved for system (as thread specific data register)	No
`mxcsr`	SSE2 control and status word	partial
`x87 SW`	x87 status word	No
`x87 CW`	x87 control word	Yes

If there are no registers available for any eightbyte of an argument, the whole argument is passed on the stack. If registers have already been assigned for some eightbytes of such an argument, the assignments get reverted.

Once registers are assigned, the arguments passed in memory are pushed on the stack in reversed (right-to-left[17]) order.

For calls that may call functions that use varargs or stdargs (prototype-less calls or calls to functions containing ellipsis (. . .) in the declaration) `%al`[18] is used as hidden argument to specify the number of vector registers used. The contents of `%al` do not need to match exactly the number of registers, but must be an upper bound on the number of vector registers used and is in the range 0–8 inclusive.

When passing `__m256` or `__m512` arguments to functions that use varargs or stdarg, function prototypes must be provided. Otherwise, the run-time behavior is undefined.

Returning of Values The returning of values is done according to the following algorithm:

1. Classify the return type with the classification algorithm.

2. If the type has class MEMORY, then the caller provides space for the return value and passes the address of this storage in `%rdi` as if it were the first argument to the function. In effect, this address becomes a "hidden" first argument. This storage must not overlap any data visible to the callee through other names than this argument.

   On return `%rax` will contain the address that has been passed in by the caller in `%rdi`.

3. If the class is INTEGER, the next available register of the sequence `%rax`, `%rdx` is used.

when computing the address to which control needs to be transferred. `%al` is used to indicate the number of vector arguments passed to a function requiring a variable number of arguments. `%r10` is used for passing a function's static chain pointer.

[16]Other bits are left unspecified, hence the consumer side of those values can rely on it being 0 or 1 when truncated to 8 bit.

[17]Right-to-left order on the stack makes the handling of functions that take a variable number of arguments simpler. The location of the first argument can always be computed statically, based on the type of that argument. It would be difficult to compute the address of the first argument if the arguments were pushed in left-to-right order.

[18]Note that the rest of `%rax` is undefined, only the contents of `%al` is defined.

4. If the class is SSE, the next available vector register of the sequence `%xmm0`, `%xmm1` is used.

5. If the class is SSEUP, the eightbyte is returned in the next available eightbyte chunk of the last used vector register.

6. If the class is X87, the value is returned on the X87 stack in `%st0` as 80-bit x87 number.

7. If the class is X87UP, the value is returned together with the previous X87 value in `%st0`.

8. If the class is COMPLEX_X87, the real part of the value is returned in `%st0` and the imaginary part in `%st1`.

As an example of the register passing conventions, consider the declarations and the function call shown in Figure 3.5. The corresponding register allocation is given in Figure 3.6, the stack frame offset given shows the frame before calling the function.

Figure 3.5: Parameter Passing Example

```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;
__m512 z;

extern void func (int e, int f,
                  structparm s, int g, int h,
                  long double ld, double m,
                  __m256 y,
                  __m512 z,
                  double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, z, n, i, j, k);
```

Figure 3.6: Register Allocation Example

General Purpose Registers	Floating Point Registers	Stack Frame Offset
`%rdi:  e`	`%xmm0:  s.d`	`0:   ld`
`%rsi:  f`	`%xmm1:  m`	`16:  j`
`%rdx:  s.a,s.b`	`%ymm2:  y`	`24:  k`
`%rcx:  g`	`%zmm3:  z`	
`%r8:   h`	`%xmm4:  n`	
`%r9:   i`		