Chapter 2

Low Level System Information

This section describes the low-level system information for the Intel386 System V ABI.

2.1 Machine Interface

The Intel386 processor architecture and data representation are covered in this section.

2.1.1 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

Table 2.1 shows the correspondence between ISO C scalar types and the processor scalar types. `__float80`, `__float128`, `__m64`, `__m128`, `__m256` and `__m512` types are optional.

[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.

Table 2.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	Intel386 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`
	`signed long`
		4	4	signed fourbyte

	`unsigned long`	4	4	unsigned fourbyte

	`long long`
	`signed long long`
		8	4	signed eightbyte

	`unsigned long long`	8	4	unsigned eightbyte

Pointer	*`any-type *`*
	`any-type (*)()`
		4	4	unsigned fourbyte

Floating-point	`float`	4	4	single (IEEE-754)
	`double` `long double`[††††]	8	4	double (IEEE-754)
	`__float80`[††] `long double`[††††]	12	4	80-bit extended (IEEE-754)
	`__float128`[††]	16	16	128-bit extended (IEEE-754)
Complex Floating-point	`_Complex float`	8	4	complex single (IEEE-754)
	`_Complex double` `_Complex long double`[††††]	16	4	complex double (IEEE-754)
	`_Complex __float80`[††] `_Complex long double`[††††]	24	4	complex 80-bit extended (IEEE-754)
	`_Complex __float128`[††]	32	16	complex 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.
[††††] The `long double` type is 64-bit, the same as the `double` type, on the Android™ platform. More information on the Android™ platform is available from `http://www.android.com/`.

The 128-bit floating-point 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 80-bit floating-point type uses a 15 bit exponent, a 64-bit mantissa with an explicit high order significant bit and an exponent bias of 16383.[3]

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

The type `size_t` is defined as `unsigned int`.

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 4 bytes of the register are significant; any nonzero value is considered `true`.

The Intel386 architecture in general does not require all data accesses to be properly aligned. Misaligned data accesses may be slower than aligned accesses but otherwise behave identically. The only exceptions are that `__float128`, `_Complex __float128`, `_Decimal128`, `__m128`, `__m256` and `__m512` must always be aligned properly.

Structures 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.

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

2.2 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.

[2]Initial implementations of the Intel386 architecture are expected to support operations on the 128-bit floating-point type only via software emulation.

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

2.2.1 Registers

The Intel386 architecture provides 8 general purpose 32-bit registers. In addition the architecture provides 8 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* 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 8 256-bit wide AVX registers (`%ymm0` - `%ymm7`). The lower 128-bits of `%ymm0` - `%ymm7` are aliased to the respective 128b-bit SSE registers (`%xmm0` - `%xmm7`). Intel AVX-512 provides 8 512-bit wide SIMD registers (`%zmm0` - `%zmm7`). The lower 128-bits of `%zmm0` - `%zmm7` are aliased to the respective 128b-bit SSE registers (`%xmm0` - `%xmm7`). The lower 256-bits of `%zmm0` - `%zmm7` are aliased to the respective 256-bit AVX registers (`%ymm0` - `%ymm7`). 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.

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. [4] The direction flag `DF` in the `%EFLAGS` 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.

2.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Table 2.2 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 (`%esp` $+ 4$) is always a multiple of 16 (32 or 64) when control is transferred to

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

Table 2.2: Stack Frame with Base Pointer

Position	Contents	Frame
`4n+8(%ebp)`	memory argument fourbyte n	
	...	Previous
`8(%ebp)`	memory argument fourbyte 0	
`4(%ebp)`	return address	
`0(%ebp)`	previous `%ebp` value	
`-4(%ebp)`	unspecified	Current
	...	
`0(%esp)`	variable size	

the function entry point. The stack pointer, `%esp`, always points to the end of the latest allocated stack frame. [5]

2.2.3 Parameter Passing and Returning Values

After the argument values have been computed, they are placed either in registers or pushed on the stack.

Passing Parameters

Most parameters are passed on the stack. Parameters are pushed onto the stack in reverse order - the last argument in the parameter list has the highest address, that is, it is stored farthest away from the stack pointer at the time of the call.

Padding may be needed to increase the size of each parameter to enforce alignment according to the values in Table 2.1. There is an exception for `__m64` and `_Decimal64`, which are treated as having an alignment of four for the purposes of parameter passing. Additional padding may be necessary to ensure that the bottom of the parameter block (closest to the stack pointer) is at an address which is `0 mod 16`, to guarantee proper alignment to the callee.

The exceptions to parameters passed on stack are as follows:

[5]The conventional use of `%ebp` as a frame pointer for the stack frame may be avoided by using `%esp` (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 (`%ebp`) available.

- The first three parameters of type `__m64` are passed in `%mm0`, `%mm1`, and `%mm2`.
- The first three parameters of type `__m128` are passed in `%xmm0`, `%xmm1`, and `%xmm2`.[6]

If parameters of type `__m256` are required to be passed on the stack, the stack pointer must be aligned on a `0 mod 32` byte boundary at the time of the call.

If parameters of type `__m512` are required to be passed on the stack, the stack pointer must be aligned on a `0 mod 64` byte boundary at the time of the call.

Returning Values

Table 2.4 lists the location used to return a value for each fundamental data type. Aggregate types (`structs` and `unions`) are always returned in memory.

Functions that return scalar floating-point values in registers return them on the top of the x87 register stack, that is, `%st0`. It is the responsibility of the calling function to pop this value from the stack regardless of whether or not the value is actually used. Failure to do so results in undefined behavior. An implication of this requirement is that functions returning scalar floating-point values must be properly prototyped. Again, failure to do so results in undefined behavior.

Returning Values in Memory

Some fundamental types and all aggregate types are returned in memory. For functions that return a value in memory, the caller passes a pointer to the memory location where the called function must write the return value. This pointer is passed to called function as an implicit first argument. The memory location must be properly aligned according to the rules in section 2.1.1. In addition to writing the return value to the proper location, the called function is responsible for popping the implicit pointer argument off the stack and storing it in `%eax` prior to returning. The calling function may choose to reference the return value via `%eax` after the function returns.

As an example of the register passing conventions, consider the declarations and the function call shown in Table 2.5. The corresponding register allocation

[6]The SSE, AVX and AVX-512 registers share resources. Therefore, if the first `__m128` parameter gets assigned to `%xmm0` , the first `__m256`/`__m512` parameter after that is assigned to `%ymm1`/`%zmm1` and not `%ymm0`/`%zmm0`.

Table 2.3: Register Usage

Register	Usage	Preserved across function calls
`%eax`	scratch register; also used to return integer and pointer values from functions; also stores the address of a returned struct or union	No
`%ebx`	callee-saved register; also used to hold the GOT pointer when making function calls via the PLT	Yes
`%ecx`	scratch register	No
`%edx`	scratch register; also used to return the upper 32bits of some 64bit return types	No
`%esp`	stack pointer	Yes
`%ebp`	callee-saved register; optionally used as frame pointer	Yes
`%esi`	callee-saved register	yes
`%edi`	callee-saved register	yes
`%xmm0`, `%ymm0`	scratch registers; also used to pass and return `__m128`, `__m256` parameters	No
`%xmm1–%xmm2`, `%ymm1–%ymm2`	scratch registers; also used to pass `__m128`, `__m256` parameters	No
`%xmm3–%xmm7`, `%ymm3–%ymm7`	scratch registers	No
`%mm0`	scratch register; also used to pass and return `__m64` parameter	No
`%mm1–%mm2`	used to pass `__m64` parameters	No
`%mm3–%mm7`	scratch registers	No
`%k0–%k7`	scratch registers	No
`%st0`	scratch register; also used to return `float`, `double`, `long double`, `__float80` parameters	No
`%st1–%st7`	scratch registers	No
`%gs`	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

Table 2.4: Return Value Locations for Fundamental Data Types

Type	C	Return Value Location

Integral	`_Bool`
	`char`
	`signed char`
	`unsigned char`
		`%al`
		The upper 24 bits of `%eax` are undefined. The caller must not rely on these being set in a predefined way by the called function.

	`short`
	`signed short`
	`unsigned short`
		`%ax`
		The upper 16 bits of `%eax` are undefined. The caller must not rely on these being set in a predefined way by the called function.

	`int`
	`signed int`
	`enum`
	`unsigned int`
	`long`
	`signed long`
	`unsigned long`
		`%eax`

	`long long`
	`signed long long`
	`unsigned long long`
		`%edx:%eax`
		The most significant 32 bits are returned in `%edx`. The least significant 32 bits are returned in `%eax`.

Pointer	*`any-type`* `*`
	`any-type` `(*)()`
		`%eax`

Floating-point	`float`	`%st0`
	`double`	`%st0`
	`long double`	`%st0`
	`__float80`	`%st0`
	`__float128`	memory
Complex floating-point	`_Complex float`	`%edx:%eax` The real part is returned in `%eax`. The imaginary part is returned in `%edx`.
	`_Complex double`	memory
	`_Complex long double`	memory
	`_Complex __float80`	memory
	`_Complex __float128`	memory
Decimal-floating-point	`_Decimal32`	`%eax`
	`_Decimal64`	`%edx:%eax` The most significant 32 bits are returned in `%edx`. The least significant 32 bits are returned in `%eax`.
	`_Decimal128`	memory
Packed	`__m64`	`%mm0`
	`__m128`	`%xmm0`
	`__m256`	`%ymm0`
	`__m512`	`%zmm0`

is given in Table 2.6, the stack frame layout given in Table 2.7 shows the frame before calling the function.

Table 2.5: Parameter Passing Example

```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int i;
__m128 v, x, y;
__m256 w, z;

extern structparm func (int i, __m128 v,
                        structparm s, __m256 w,
                        __m128 x, __m128 y,
                        __m256 z);

func (i, v, s, w, x, y, z);
```

Table 2.6: Register Allocation for Parameter Passing Example

Parameter	Location before the call
Return value pointer	(%esp)
i	4(%esp)
v	%xmm0
s	8(%esp)
w	%ymm1
x	%xmm2
y	32(%esp)
z	64(%esp)

Table 2.7: Stack Layout at the Call

Contents	Length	
`z`	32 bytes	
padding	16 bytes	
`y`	16 bytes	
padding	8 bytes	
`s`	16 bytes	
`i`	4 bytes	
Return value pointer	4 bytes	$\longleftarrow$ `%esp` (32-byte aligned)

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[7].

2.2.4 Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that all arguments are passed on the stack, and arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the Intel386 architecture because some arguments are passed in registers. Portable C programs must use the header file `<stdarg.h>` in order to handle variable argument lists.

When a function taking variable-arguments is called, all parameters are passed on the stack, including `__m64`, `__m128` and `__m256`. This rule applies to both named and unnamed parameters. Because parameters are passed differently depending on whether or not the called function takes a variable argument list, it is necessary for such functions to be properly prototyped. Failure to do so results in undefined behavior.

[7]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.