System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.90

Edited by
Jan Hubička[1], Andreas Jaeger[2], Mark Mitchell[3]

December 2, 2003

[1]jh@suse.cz
[2]aj@suse.de
[3]mark@codesourcery.com

Contents

List of Tables

List of Figures

Revision History

0.90 Change DWARF register numbers again; mention that __m128 needs alignment; fix typo in figure 3.3; add some comments on kernel expectations; mention TLS extensions; add example for passing of variable-argument lists; change semantics of %rax in variable-argument lists; improve formatting; mention that X87 class is not used for passing; make /lib64 a Linux specific section; rename x86-64 to AMD64; describe passing of complex types. Special thanks to Andi Kleen, Michal Ludvig, Michael Matz, David O'Brien and Eric Young for their comments.

0.21 Define __int128 as class INTEGER in register passing. Mention that %al is used for variadic argument lists. Fix some textual problems. Thanks to H. Peter Anvin, Bo Thorsen, and Michael Matz.

0.20 — 2002-07-11 Change DWARF register number values of %rbx, %rsi, %rsi (thanks to Michal Ludvig). Fix footnotes for fundamental types (thanks to H. Peter Anvin). Specify size_t (thanks to Bo Thorsen and Andreas Schwab). Add new section on floating point environment functions.

0.19 — 2002-03-27 Set name of Linux dynamic linker, mention %fs. Incorporate changes from H. Peter Anvin <hpa@zytor.com> for booleans and define handling of sub-64-bit integer types in registers.

Chapter 1

Introduction

The AMD64[1] architecture[2] is an extension of the x86 architecture. Any processor implementing the AMD64 architecture specification will also provide compatiblity modes for previous descendants of the Intel 8086 architecture, including 32-bit processors such as the Intel 386, Intel Pentium, and AMD K6-2 processor. Operating systems conforming to the AMD64 ABI may provide support for executing programs that are designed to execute in these compatiblity modes. The AMD64 ABI does not apply to such prorams; this document applies only programs running in the “long” mode provided by the AMD64 architecture.

Except where otherwise noted, the AMD64 architecture ABI follows the conventions described in the Intel386 ABI. Rather than replicate the entire contents of the Intel386 ABI, the AMD64 ABI indicates only those places where changes have been made to the Intel386 ABI.

No attempt has been made to specify an ABI for languages other than C. However, it is assumed that many programming languages will wish to link with code written in C, so that the ABI specifications documented here are relevant.[3]

1.1 Differences from the Intel386 ABI

The most fundamental differences from the Intel386 ABI document are as follows:

[1] AMD64 has been previously called x86-64. The latter name is used in a number of places out of historical reasons instead of AMD64.

[2] The architecture specification is available on the web at `http://www.x86-64.org/documentation`.

[3] See section 9.2 for details on C++ ABI.

- Sizes of fundamental data types.
- Parameter-passing conventions.
- Floating-point computations.
- Removal of the GOT register.
- Use of RELA relocations.

Chapter 2

Software Installation

No changes required.

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. The `__int128`, `__float128`, `__m64` and `__m128` 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 *quardword* 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 signed long long long signed long long	8	8	signed eightbyte
	unsigned long unsigned long long	8 8	8 8	unsigned eightbyte unsigned eightbyte
	__int128†† signed __int128††	16 16	16 16	signed sixteenbyte signed sixteenbyte
	unsigned __int128††	16	16	unsigned sixteenbyte
Pointer	*any-type* * *any-type* (*)()	8	8	unsigned eightbyte
Floating-point	float double long double __float128††	4 8 16 16	4 8 16 16	single (IEEE) double (IEEE) 80-bit extended (IEEE) 128-bit extended (IEEE)
Packed	__m64†† __m128††	8 16	8 16	*MMX* and 3DNow! SSE and SSE-2

† This type is called `bool` in C++.
†† These types are optional.

`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`.

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 or passed as arguments on the stack, 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 access to be properly aligned. Accessing misaligned data will be slower than accessing properly aligned data, but otherwise there is no difference. The only exception here is that `__m128` always has to be aligned properly.

Aggregates and Unions

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

No other changes required.

Bit-Fields

Amend the description of bit-field ranges as follows:

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed long`		-2^{w-1} to $2^{w-1}-1$
`long`	1 to 64	0 to 2^w-1
`unsigned long`		0 to 2^w-1

[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.

The ABI does not permit bitfields having the type `__m64` or `__m128`. Programs using bitfields of these types are not portable.

No other changes required.

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 and the Stack Frame

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 in a running program.

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.[5] 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.

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 before accessing the *MMX* registers.[6] The direction flag in the `%eflags` register must be clear on function entry, and on function return.

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

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

Figure 3.3: Stack Frame with Base Pointer

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

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 byte boundary. In other words, the value ($\texttt{%rsp} - 8$) is always a multiple of 16 when control is transferred to the function entry point. The stack pointer, `%rsp`, always points to the end of the latest allocated stack frame. [7]

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.[8] 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.

3.2.3 Parameter Passing

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

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

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

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 fits into a SSE register.

SSEUP The class consists of types that fit into a SSE register and can be passed and returned in the most significant half 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.

Classification The size of each argument gets rounded up to eightbytes.[9]

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` and `__m64` are in class SSE.
- Arguments of types `__float128` and `__m128` are split into two halves. The least significant ones belong to class SSE, the most significant one to class SSEUP.
- 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.

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

- 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:

```
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 two eightbytes, or in C++, is a non-POD [10] structure or union type, or contains unaligned fields, it has class MEMORY.[11]

2. Both eightbytes get initialized to class NO_CLASS.

[10] The term POD is from the ANSI/ISO C++ Standard, and stands for Plain Old Data. Although the exact definition is technical, a POD is essentially a structure or union that could have been written in C; there cannot be any member functions, or base classes, or similar C++ extensions.

[11] A non-POD object cannot be passed in registers because such objects must have well defined addresses; the address at which an object is constructed (by the caller) and the address at which the object is destroyed (by the callee) must be the same. Similar issues apply when returning a non-POD object from a function.

3. 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.
   (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.
4. Then a post merger cleanup is done:
   (a) If one of the classes is MEMORY, the whole argument is passed in memory.
   (b) If SSEUP is not preceeded by SSE, 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[12].
3. If the class is SSE, the next available SSE 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 upper half of the least used SSE register.
5. If the class is X87, X87UP or COMPLEX_X87, it is passed in memory.

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

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
`%rbx`	callee-saved register; optionally used as base pointer	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-r15`	callee-saved registers	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
`%mmx0–%mmx7`	temporary registers	No
`%st0`	temporary register; used to return `long double` arguments	No
`%st1`	temporary registers; used to return `long double` arguments	No
`%st2–%st7`	temporary registers	No
`%fs`	Reserved for system use (as thread specific data register)	No

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

Once registers are assigned, the arguments passed in memory are pushed on the stack in reversed (right-to-left[13]) 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` [14] is used as hidden argument to specify the number of SSE 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 SSE registers used and is in the range 0–8 inclusive.

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.

   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.
4. If the class is SSE, the next available SSE register of the sequence `%xmm0`, `%xmm1` is used.
5. If the class is SSEUP, the eightbyte is passed in the upper half of the last used SSE register.
6. If the class is X87, the value is returned on the X87 stack in `%st0` as 80-bit x87 number.

[13]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.

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

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;

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

func (e, f, s, g, h, ld, l, m, 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`	`%xmm2: n`	`24: k`
`%rcx: g`		
`%r8: h`		
`%r9: i`		

3.3 Operating System Interface

3.3.1 Exception Interface

As the AMD64 manuals describe, the processor changes mode to handle *exceptions,* which may be synchronous, floating-point/coprocessor or asynchronous. Synchronous and floating-point/coprocessor exceptions, being caused by instruction execution, can be explicitly generated by a process. This section, therefore, specifies those exception types with defined behavior. The AMD64 architecture classifies exceptions as *faults*, *traps*, and *aborts*. See the Intel386 ABI for more information about their differences.

Hardware Exception Types

The operating system defines the correspondence between hardware exceptions and the signals specified by `signal` (BA_OS) as shown in table 3.1. Contrary to the i386 architecture, the AMD64 does not define any instructions that generate a bounds check fault in long mode.

3.3.2 Special Registers

The AMD64 architecture defines floating point instructions. At process startup the two floating point units, SSE2 and x87, both have all floating-point exception status flags cleared. The status of the control words is as defined in tables 3.3 and 3.4.

Table 3.1: Hardware Exceptions and Signals

Number	Exception name	Signal
0	divide error fault	SIGFPE
1	single step trap/fault	SIGTRAP
2	nonmaskable interrupt	none
3	breakpoint trap	SIGTRAP
4	overflow trap	SIGSEGV
5	(reserved)	
6	invalid opcode fault	SIGILL
7	no coprocessor fault	SIGFPE
8	double fault abort	none
9	coprocessor overrun abort	SIGSEGV
10	invalid TSS fault	none
11	segment no present fault	none
12	stack exception fault	SIGSEGV
13	general protection fault/abort	SIGSEGV
14	page fault	SIGSEGV
15	(reserved)	
16	coprocessor error fault	SIGFPE
other	(unspecified)	SIGILL

Table 3.2: Floating-Point Exceptions

Code	Reason
FPE_FLTDIV	floating-point divide by zero
FPE_FLTOVF	floating-point overflow
FPE_FLTUND	floating-point underflow
FPE_FLTRES	floating-point inexact result
FPE_FLTINV	invalid floating-point operation

Table 3.3: x87 Floating-Point Control Word

Field	Value	Note
RC	0	Round to nearest
PC	11	Double extended precision
PM	1	Precision masked
UM	1	Underflow masked
OM	1	Overflow masked
ZM	1	Zero divide masked
DM	1	Denormal operand masked
IM	1	Invalid operation masked

Table 3.4: MXCSR Status Bits

Field	Value	Note
FZ	0	Do not flush to zero
RC	0	Round to nearest
PM	1	Precision masked
UM	1	Underflow masked
OM	1	Overflow masked
ZM	1	Zero divide masked
DM	1	Denormal operand masked
IM	1	Invalid operation masked
DAZ	0	Denormals are not zero

3.3.3 Virtual Address Space

Although the AMD64 architecture uses 64-bit pointers, implementations are only required to handle 48-bit addresses. Therefore, conforming processes may only use addresses from `0x00000000 00000000` to `0x00007fff ffffffff`[15].

No other changes required.

3.3.4 Page Size

Systems are permitted to use any power-of-two page size between 4KB and 64KB, inclusive.

No other changes required.

3.3.5 Virtual Address Assignments

Conceptually processes have the full address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration dependent amount of virtual space.
- The system reserves a configuration dependent amount of space per process.
- A process whose size exceeds the system's available combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amount.

[15] 0x0000ffff ffffffff is not a canonical address and cannot be used.

Figure 3.7: Virtual Address Configuration

0xffffffffffffffff	Reserved system area ...	End of memory
0x80000000000	... Dynamic segments	
0	... Process segments	Beginning of memory

Although applications may control their memory assignments, the typical arrangement appears in figure 3.8.

Figure 3.8: Conventional Segment Arrangements

0x80000000000	... Dynamic segments
	Stack segment ...
	... Data segments
0x400000	... Text segments
0	Unmapped

3.4 Process Initialization

3.4.1 Auxiliary Vector

The AMD64 ABI uses the following auxiliary vector types.

Figure 3.9: Auxiliary Vector Types

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_NOTELF	10	a_val
AT_UID	11	a_val
AT_EUID	12	a_val
AT_GID	13	a_val
AT_EGID	14	a_val

The entries that are different than in the Intel386 ABI are specified as follows:

AT_NOTELF The `a_val` member of this entry is non-zero if the program is in another format than ELF.

AT_UID The `a_val` member of this entry holds the real user id of the process.

AT_EUID The `a_val` member of this entry holds the effective user id of the process.

AT_GID The `a_val` member of this entry holds the real group id of the process.

AT_EGID The `a_val` member of this entry holds the effective group id of the process.

3.5 Coding Examples

The following sections show only the difference to the Intel386 ABI.

3.5.1 Architectural Constraints

The AMD64 architecture usually does not allow to encode arbitrary 64-bit constants as immediate operand of the instruction. Most instructions accept 32-bit immediates that are sign extended to the 64-bit ones. Additionally the 32-bit operations with register destinations implicitly perform zero extension making loads of 64-bit immediates with upper half set to 0 even cheaper.

Additionally the branch instructions accept 32-bit immediate operands that are sign extended and used to adjust instruction pointer. Similarly an instruction pointer relative addressing mode exists for data accesses with equivalent limitations.

In order to improve performance and reduce code size, it is desirable to use different code models depending on the requirements.

Code models define constraints for symbolic values that allow the compiler to generate better code. Basically code models differ in addressing (absolute versus position independent), code size, data size and address range. We define only a small number of code models that are of general interest:

Small code model The virtual address of code executed is known at link time. Additionally all symbols are known to be located in the virtual addresses in the range from 0 to $2^{31} - 2^{10} - 1$.

This allows the compiler to encode symbolic references with offsets in the range from -2^{31} to 2^{10} directly in the sign extended immediate operands, with offsets in the range from 0 to $2^{31} + 2^{10}$ in the zero extended immediate operands and use instruction pointer relative addressing for the symbols with offsets in the range -2^{10} to 2^{10}.

This is the fastest code model and we expect it to be suitable for the vast majority of programs.

Kernel code model The kernel of an operating system is usually rather small but runs in the negative half of the address space. So we define all symbols to be in the range from $2^{64} - 2^{31}$ to $2^{64} - 2^{10}$.

This code model has advantages similar to those of the small model, but allows encoding of zero extended symbolic references only for offsets from 2^{31} to $2^{31} + 2^{10}$. The range offsets for sign extended reference changes to 0–$2^{31} + 2^{10}$.

Medium code model The medium code model does not make any assumptions about the range of symbolic references to data sections. Size and address of the text section have the same limits as the small code model.

This model requires the compiler to use `movabs` instructions to access static data and to load addresses into register, but keeps the advantages of the small code model for manipulation of addresses to the text section (specially needed for branches).

Large code model The large code model makes no assumptions about addresses and sizes of sections.

The compiler is required to use the `movabs` instruction, as in the medium code model, even for dealing with addresses inside the text section. Additionally, indirect branches are needed when branching to addresses whose offset from the current instruction pointer is unknown.

It is possible to avoid the limitation for the text section by breaking up the program into multiple shared libraries, so we do not expect this model to be needed in the foreseeable future.

Small position independent code model (PIC) Unlike the previous models, the virtual addresses of instructions and data are not known until dynamic link time. So all addresses have to be relative to the instruction pointer.

Additionally the maximum distance between a symbol and the end of an instruction is limited to $2^{31} - 2^{10} - 1$, allowing the compiler to use instruction pointer relative branches and addressing modes supported by the hardware for every symbol with an offset in the range -2^{10} to 2^{10}.

Medium position independent code model (PIC) This model is like the previous model, but makes no assumptions about the distance of symbols to the data section.

In the medium PIC model, the instruction pointer relative addressing can not be used directly for accessing static data, since the offset can exceed the limitations on the size of the displacement field in the instruction. Instead an unwind sequence consisting of `movabs`, `lea` and `add` needs to be used.

Large position independent code model (PIC) This model is like the previous model, but makes no assumptions about the distance of symbols.

The large PIC model implies the same limitation as the medium PIC model regarding addressing of static data. Additionally, references to the global offset table and to the procedure linkage table and branch destinations need to be calculated in a similar way.

3.5.2 Position-Independent Function Prologue

AMD64 does not need any function prologue for calculating the global offset table address since it does not have an explicit GOT pointer.

3.5.3 Data Objects

This section describes only objects with static storage. Stack-resident objects are excluded since programs always compute their virtual address relative to the stack or frame pointers.

Because only the `movabs` instruction uses 64-bit addresses directly, depending on the code model either `%rip`-relative addressing or building addresses in registers and accessing the memory through the register has to be used.

For absolute addresses `%rip`-relative encoding can be used in the small model. In the medium model the `movabs` instruction has to be used for accessing addresses.

Position-independend code cannot contain absolute address. To access a global symbol the address of the symbol has to be loaded from the Global Offset Table. The address of the entry in the GOT can be obtained with a `%rip`-relative instruction in the small model.

Figure 3.10: Absolute Load and Store (Small Model)

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.extern  src`
	`.extern  dst`
	`.extern  ptr`
	`.text`

`dst = src;`	`movl     src(%rip), %eax` `movl     %eax, dst(%rip)`
`ptr = &dst;`	`lea      dst(%rip),%rdx` `movq     %rdx, ptr(%rip)`

`*ptr = src;`	`movq     ptr(%rip),%rax`
	`movl     src(%rip),%edx`
	`movl     %edx, (%rax)`

Figure 3.11: Absolute Load and Store (Medium and Large Models)

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.extern  src`
	`.extern  dst`
	`.extern  ptr`
	`.text`

`dst = src;`	`movabsl  src, %eax` `movabsl  %eax, dst`
`ptr = &dst;`	`movabsq  $dst,%rdx` `movabsq  %rdx, ptr`

`*ptr = src;`	`movabsq  ptr,%rdx`
	`movabsl  src,%eax`
	`movl     %eax, (%rdx)`

Figure 3.12: Position-Independend Load and Store (Small and Medium PIC Model)

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.extern src`
	`.extern dst`
	`.extern ptr`
	`.text`

`dst = src;`	`movq src@GOTPCREL(%rip), %rax`
	`movl (%rax), %edx`
	`movq dst@GOTPCREL(%rip), %rax`
	`movl %edx, (%rax)`

`ptr = &dst;`	`movq ptr@GOTPCREL(%rip), %rax`
	`movq dst@GOTPCREL(%rip), %rdx`
	`movq %rdx, (%rax)`

`*ptr = src;`	`movq ptr@GOTPCREL(%rip),%rax`
	`movq (%rax), %rdx`
	`movq src@GOTPCREL(%rip), %rax`
	`movl (%rax), %eax`
	`movl %eax, (%rdx)`

3.5.4 Function Calls

Figure 3.13: Position-Independent Direct Function Call (Small and Medium Model)

```
extern void function ();
function ();
```

```
.globl function
call function@PLT
```

Figure 3.14: Position-Independent Indirect Function Call

`extern void (*ptr) ();`
`extern void name ();`
`ptr = name;`
 	
`(*ptr)();`
	`.globl ptr, name`
	
	`movq ptr@GOTPCREL(%rip), %rax`
	`movq name@GOTPCREL(%rip), %rdx`
	`movq %rdx, (%rax)`
	
	`movq ptr@GOTPCREL(%rip), %rax`
	`call *(%rax)`

3.5.5 Branching

Not done yet.

3.5.6 Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments are passed on the stack, and 2) 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 AMD64 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, `%rax` must be set to the total number of floating point parameters passed to the function in SSE registers.[16]

[16]This implies that the only legal values for `%rax` when calling a function with variable-argument lists are 0 to 8 (inclusive).

Figure 3.15: Parameter Passing Example with Variable-Argument List

```
int a, b;
long double ld;
double m, n;

extern void func (int a, double m,...);

func (a, m, b, ld, n);
```

Figure 3.16: Register Allocation Example for Variable-Argument List

General Purpose Registers	Floating Point Registers	Stack Frame Offset
`%rdi: a`	`%xmm0: m`	`0: ld`
`%rsi: b`	`%xmm1: n`	
`%rax: 2`		

The Register Save Area

The prologue of a function taking a variable argument list and known to call the macro `va_start` is expected to save the argument registers to the *register save area*. Each argument register has a fixed offset in the register save area as defined in the figure 3.17.

Only registers that might be used to pass arguments need to be saved. Other registers are not accessed and can be used for other purposes. If a function is known to never accept arguments passed in registers[17], the register save area may be omitted entirely.

The prologue should use `%rax` to avoid unnecessarily saving XMM registers. This is especially important for integer only programs to prevent the initialization of the XMM unit.

[17] This fact may be determined either by exploring types used by the `va_arg` macro, or by the fact that the named arguments already are exhausted the argument registers entirely.

Figure 3.17: Register Save Area

Register	Offset
%rdi	0
%rsi	8
%rdx	16
%rcx	24
%r8	32
%r9	40
%xmm0	48
%xmm1	64
...	
%xmm15	288

The **va_list** Type

The va_list type is an array containing a single element of one structure containing the necessary information to implement the va_arg macro. The C definition of va_list type is given in figure 3.18.

Figure 3.18: va_list Type Declaration

```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
} va_list[1];
```

The **va_start** Macro

The va_start macro initializes the structure as follows:

reg_save_area The element points to the start of the register save area.

overflow_arg_area This pointer is used to fetch arguments passed on the stack. It is initialized with the address of the first argument passed on the stack, if any, and then always updated to point to the start of the next argument on the stack.

gp_offset The element holds the offset in bytes from `reg_save_area` to the place where the next available general purpose argument register is saved. In case all argument registers have been exhausted, it is set to the value 48 $(6 * 8)$.

fp_offset The element holds the offset in bytes from `reg_save_area` to the place where the next available floating point argument register is saved. In case all argument registers have been exhausted, it is set to the value 304 $(6 * 8 + 16 * 16)$.

The `va_arg` Macro

The algorithm for the generic `va_arg(l, type)` implementation is defined as follows:

1. Determine whether `type` may be passed in the registers. If not go to step 7.

2. Compute `num_gp` to hold the number of general purpose registers needed to pass `type` and `num_fp` to hold the number of floating point registers needed.

3. Verify whether arguments fit into registers. In the case:

   $$\texttt{l->gp\_offset} > 48 - \texttt{num\_gp} * 8$$

   or

   $$\texttt{l->fp\_offset} > 304 - \texttt{num\_fp} * 16$$

   go to step 7.

4. Fetch `type` from `l->reg_save_area` with an offset of `l->gp_offset` and/or `l->fp_offset`. This may require copying to a temporary location in case the parameter is passed in different register classes or requires an alignment greater than 8 for general purpose registers and 16 for XMM registers.

5. Set:

$$\texttt{l->gp\_offset} = \texttt{l->gp\_offset} + \texttt{num\_gp} * 8$$

$$\texttt{l->fp\_offset} = \texttt{l->fp\_offset} + \texttt{num\_fp} * 16.$$

6. Return the fetched `type`.

7. Align `l->overflow_arg_area` upwards to a 16 byte boundary if alignment needed by `type` exceeds 8 byte boundary.

8. Fetch `type` from `l->overflow_arg_area`.

9. Set `l->overflow_arg_area` to:

$$\texttt{l->overflow\_arg\_area} + \texttt{sizeof}(\texttt{type})$$

10. Align `l->overflow_arg_area` upwards to an 8 byte boundary.

11. Return the fetched `type`.

The `va_arg` macro is usually implemented as a compiler builtin and expanded in simplified forms for each particular type. Figure 3.19 is a sample implementation of the `va_arg` macro.

Figure 3.19: Sample Implementation of `va_arg(l, int)`

	movl	`l->gp_offset`, `%eax`	
	cmpl	$48, `%eax`	Is register available?
	jae	stack	If not, use stack
	leal	$8(`%rax`), `%edx`	Next available register
	addq	`l->reg_save_area`, `%rax`	Address of saved register
	movl	`%edx`, `l->gp_offset`	Update `gp_offset`
	jmp	fetch	
stack:	movq	`l->overflow_arg_area`, `%rax`	Address of stack slot
	leaq	8(`%rax`), `%rdx`	Next available stack slot
	movq	`%rdx,l->overflow_arg_area`	Update
fetch:	movl	(`%rax`), `%eax`	Load argument

3.6 DWARF Definition

This section[18] defines the Debug With Arbitrary Record Format (DWARF) debugging format for the AMD64 processor family. The AMD64 ABI does not define a debug format. However, all systems that do implement DWARF shall use the following definitions.

DWARF is a specification developed for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. For more information on DWARF, see *DWARF Debugging Information Format*, revision: Version 2.0.0, July 27, 1993, UNIX International, Program Languages SIG.

3.6.1 DWARF Release Number

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the AMD64 registers. In addition, the DWARF Version 2 specification requires processor-specific address class codes to be defined.

3.6.2 DWARF Register Number Mapping

Table 3.20[19] outlines the register number mapping for the AMD64 processor family.[20]

[18] This section is structured in a way similar to the PowerPC psABI

[19] The table defines Return Address to have a register number, even though the address is stored in 0(`%rsp`) and not in a physical register.

[20] This document does not define mappings for privileged registers.

Figure 3.20: DWARF Register Number Mapping

Register Name	Number	Abbreviation
General Purpose Register RAX	0	`%rax`
General Purpose Register RDX	1	`%rdx`
General Purpose Register RCX	2	`%rcx`
General Purpose Register RBX	3	`%rbx`
General Purpose Register RSI	4	`%rsi`
General Purpose Register RDI	5	`%rdi`
Frame Pointer Register RBP	6	`%rbp`
Stack Pointer Register RSP	7	`%rsp`
Extended Integer Registers 8-15	8-15	`%r8–%r15`
Return Address RA	16	
SSE Registers 0–7	17-24	`%xmm0–%xmm7`
Extended SSE Registers 8–15	25-32	`%xmm8–%xmm15`
Floating Point Registers 0–7	33-40	`%st0–%st7`
MMX Registers 0–7	41-48	`%mm0–%mm7`

Chapter 4

Object Files

4.1 ELF Header

4.1.1 Machine Information

For file identification in `e_ident`, the AMD64 architecture requires the following values.

Table 4.1: AMD64 Identification

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS64`
`e_ident[EI_DATA]`	`ELFDATA2LSB`

Processor identification resides in the ELF header's `e_machine` member and must have the value `EM_X86_64`.[1]

4.2 Sections

No changes required.

[1] The value of this identifier is 62.

4.3 Symbol Table

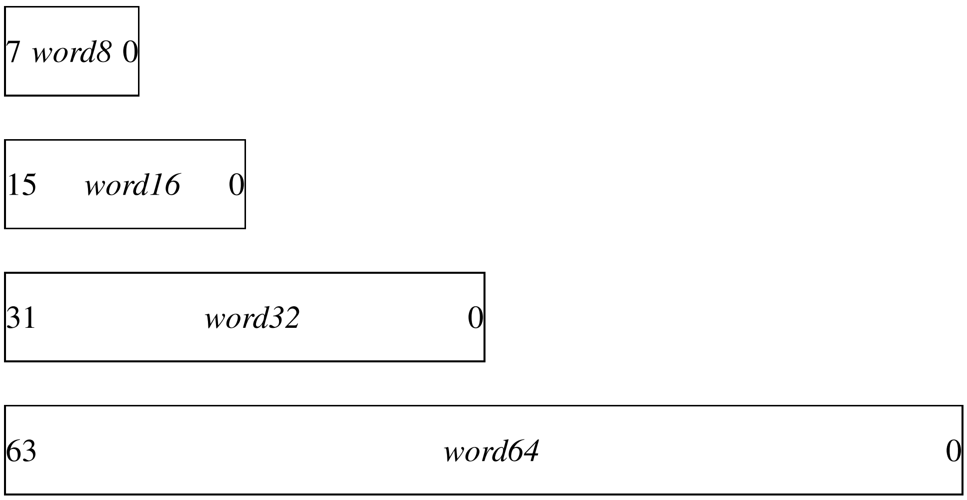
No changes required.

4.4 Relocation

4.4.1 Relocation Types

The AMD64 ABI adds one additional field:

Figure 4.1: Relocatable Fields



word8	This specifies a 8-bit field occupying 1 byte.
word16	This specifies a 16-bit field occupying 2 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the AMD64 architecture.
word32	This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the AMD64 architecture.
word64	This specifies a 64-bit field occupying 8 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the AMD64 architecture.

The following notations are used for specifying relocations in table 4.2:

A Represents the addend used to compute the value of the relocatable field.

B Represents the base address at which a shared object has been loaded into memory during execution. Generally, a shared object is built with a 0 base virtual address, but the execution address will be different.

G Represents the offset into the global offset table at which the relocation entry's symbol will reside during execution.

GOT Represents the address of the global offset table.

L Represents the place (section offset or address) of the Procedure Linkage Table entry for a symbol.

P Represents the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

S Represents the value of the symbol whose index resides in the relocation entry.

The AMD64 ABI architectures uses only `Elf64_Rela` relocation entries with explicit addends. The `r_addend` member serves as the relocation addend.

Table 4.2: Relocation Types

Name	Value	Field	Calculation
R_X86_64_NONE	0	none	none
R_X86_64_64	1	*word64*	S + A
R_X86_64_PC32	2	*word32*	S + A - P
R_X86_64_GOT32	3	*word32*	G + A
R_X86_64_PLT32	4	*word32*	L + A - P
R_X86_64_COPY	5	none	none
R_X86_64_GLOB_DAT	6	*word64*	S
R_X86_64_JUMP_SLOT	7	*word64*	S
R_X86_64_RELATIVE	8	*word64*	B + A
R_X86_64_GOTPCREL	9	*word32*	G + GOT + A - P
R_X86_64_32	10	*word32*	S + A
R_X86_64_32S	11	*word32*	S + A
R_X86_64_16	12	*word16*	S + A
R_X86_64_PC16	13	*word16*	S + A - P
R_X86_64_8	14	*word8*	S + A
R_X86_64_PC8	15	*word8*	S + A - P
R_X86_64_DPTMOD64	16	*word64*	
R_X86_64_DTPOFF64	17	*word64*	
R_X86_64_TPOFF64	18	*word64*	
R_X86_64_TLSGD	19	*word32*	
R_X86_64_TLSLD	20	*word32*	
R_X86_64_DTPOFF32	21	*word32*	
R_X86_64_GOTTPOFF	22	*word32*	
R_X86_64_TPOFF32	23	*word32*	

The special semantics for most of these relocation types are identical to those used for the Intel386 ABI. [2] [3]

[2]Even though the AMD64 architecture supports IP-relative addressing modes, a GOT is still required since the offset from a particular instruction to a particular data item cannot be known by the static linker.

[3]Note that the AMD64 architecture assumes that offsets into GOT are 32-bit values, not 64-bit values. This choice means that a maximum of $2^{32}/8 = 2^{29}$ entries can be placed in the GOT. However, that should be more than enough for most programs. In the event that it is not enough,

The R_X86_64_GOTPCREL relocation has different semantics from the i386 R_I386_GOTPC relocation. In particular, because the AMD64 architecture has an addressing mode relative to the instruction pointer, it is possible to load an address from the GOT using a single instruction. The calculation done by the R_X86_64_GOTPCREL relocation gives the difference between the location in the GOT where the symbol's address is given and the location where the relocation is applied.

The R_X86_64_32 and R_X86_64_32S relocations truncate the computed value to 32-bits. The linker must verify that the generated value for the R_X86_64_32 (R_X86_64_32S) relocation zero-extends (sign-extends) to the original 64-bit value.

A program or object file using R_X86_64_8, R_X86_64_16, R_X86_64_PC16 or R_X86_64_PC8 relocations is not conformant to this ABI, these relocations are only added for documentation purposes. The R_X86_64_16, and R_X86_64_8 relocations truncate the computed value to 16-bits resp. 8-bits.

The relocations R_X86_64_DPTMOD64, R_X86_64_DTPOFF64, R_X86_64_TPOFF64 , R_X86_64_TLSGD , R_X86_64_TLSLD , R_X86_64_DTPOFF32, R_X86_64_GOTTPOFF and R_X86_64_TPOFF32 are listed for completeness. They are part of the Thread-Local Storage ABI extensions and are documented in the document called "ELF Handling for Thread-Local Storage"[4].

---

the linker could create multiple GOTs. Because 32-bit offsets are used, loads of global data do not require loading the offset into a displacement register; the base plus immediate displacement addressing form can be used.

[4]This document is currently available via http://people.redhat.com/drepper/tls.pdf

Chapter 5

Program Loading and Dynamic Linking

5.1 Program Loading

No changes required.

5.2 Dynamic Linking

Dynamic Section

No changes required.

Global Offset Table

The global offset table contains 64-bit addresses.

No other changes required.

Figure 5.1: Global Offset Table

```
extern Elf64_Addr _GLOBAL_OFFSET_TABLE_ [];
```

Function Addresses

No changes required.

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the AMD64 architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables. Unlike Intel386 ABI, this ABI uses the same procedure linkage table for both programs and shared objects.

Figure 5.2: Procedure Linkage Table

```
.PLT0:  pushq  GOT+8(%rip) # GOT[1]
        jmp    *GOT+16(%rip) # GOT[2]
        nop
        nop
        nop
        nop
.PLT1:  jmp    *name1@GOTPCREL(%rip)
        pushq  $index
        jmp    .PLT0
.PLT2:  jmp    *name2@GOTPCREL(%rip)
        pushq  $index
        jmp    .PLT0
        ...
```

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.

2. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file.

3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.

4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.

5. Now the program pushes a relocation index (*index*) on the stack. The relocation index is a 32-bit, non-negative index into the relocation table addressed by the `DT_JMPREL` dynamic section entry. The designated relocation entry will have type `R_X86_64_JUMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry contains a symbol table index that will reference the appropriate symbol, `name1` in the example.

6. After pushing the relocation index, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (GOT+8) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (GOT+16), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.

8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That

is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change the dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_X86_64_JUMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

5.2.1 Program Interpreter

There is one valid program interpreter for programs conforming to the AMD64 ABI:

`/lib/ld64.so.1`
However, Linux puts this in

`/lib64/ld-linux-x86-64.so.2`

5.2.2 Initialization and Termination Functions

The implementation is responsible for executing the initialization functions specified by `DT_INIT`, `DT_INIT_ARRAY`, and `DT_PREINIT_ARRAY` entries in the executable file and shared object files for a process, and the termination (or finalization) functions specified by `DT_FINI` and `DT_FINI_ARRAY`, as specified by the *System V ABI*. The user program plays no further part in executing the initialization and termination functions specified by these dynamic tags.

Chapter 6

Libraries

A further review of the Intel386 ABI is needed.

6.1 C Library

6.1.1 Global Data Symbols

The symbols `_fp_hw`, `__flt_rounds` and `__huge_val` are not provided by the AMD64 ABI.

6.1.2 Floating Point Environment Functions

ISO C 99 defines the floating point environment functions from `<fenv.h>`. Since AMD64 has two floating point units with separate control words, the programming environment has to keep the control values in sync. On the other hand this means that routines accessing the control words only need to access one unit, and the SSE unit is the unit that should be accessed in these casese. The function `fegetround` therefore only needs to report the rounding value of the SSE unit and can ignore the x87 unit.

6.2 Unwind Library Interface

This section defines the Unwind Library interface [1], expected to be provided by any AMD64 psABI-compliant system. This is the interface on which the C++ ABI exception-handling facilities are built. We assume as a basis the Call Frame Information tables described in the DWARF Debugging Information Format document.

This section is meant to specify a language-independent interface that can be used to provide higher level exception-handling facilities such as those defined by C++.

The unwind library interface consists of at least the following routines:
`_Unwind_RaiseException`,
`_Unwind_Resume`,
`_Unwind_DeleteException`,
`_Unwind_GetGR`,
`_Unwind_SetGR`,
`_Unwind_GetIP`,
`_Unwind_SetIP`,
`_Unwind_GetRegionStart`,
`_Unwind_GetLanguageSpecificData`,
`_Unwind_ForcedUnwind`

In addition, two datatypes are defined (`_Unwind_Context` and `_Unwind_Exception`) to interface a calling runtime (such as the C++ runtime) and the above routine. All routines and interfaces behave as if defined `extern "C"`. In particular, the names are not mangled. All names defined as part of this interface have a `"_Unwind_"` prefix.

Lastly, a language and vendor specific personality routine will be stored by the compiler in the unwind descriptor for the stack frames requiring exception processing. The personality routine is called by the unwinder to handle language-specific tasks such as identifying the frame handling a particular exception.

6.2.1 Exception Handler Framework

Reasons for Unwinding

There are two major reasons for unwinding the stack:

[1] The overall structure and the external interface is derived from the IA-64 UNIX System V ABI

- exceptions, as defined by languages that support them (such as C++)
- "forced" unwinding (such as caused by longjmp or thread termination).

The interface described here tries to keep both similar. There is a major difference, however.

- In the case where an exception is thrown, the stack is unwound while the exception propagates, but it is expected that the personality routine for each stack frame knows whether it wants to catch the exception or pass it through. This choice is thus delegated to the personality routine, which is expected to act properly for any type of exception, whether "native" or "foreign". Some guidelines for "acting properly" are given below.
- During "forced unwinding", on the other hand, an external agent is driving the unwinding. For instance, this can be the longjmp routine. This external agent, not each personality routine, knows when to stop unwinding. The fact that a personality routine is not given a choice about whether unwinding will proceed is indicated by the _UA_FORCE_UNWIND flag.

To accommodate these differences, two different routines are proposed. _Unwind_RaiseExcepti performs exception-style unwinding, under control of the personality routines. _Unwind_ForcedUnwind, on the other hand, performs unwinding, but gives an external agent the opportunity to intercept calls to the personality routine. This is done using a proxy personality routine, that intercepts calls to the personality routine, letting the external agent override the defaults of the stack frame's personality routine.

As a consequence, it is not necessary for each personality routine to know about any of the possible external agents that may cause an unwind. For instance, the C++ personality routine need deal only with C++ exceptions (and possibly disguising foreign exceptions), but it does not need to know anything specific about unwinding done on behalf of longjmp or pthreads cancellation.

The Unwind Process

The standard ABI exception handling/unwind process begins with the raising of an exception, in one of the forms mentioned above. This call specifies an exception object and an exception class.

The runtime framework then starts a two-phase process:

- In the *search* phase, the framework repeatedly calls the personality routine, with the `_UA_SEARCH_PHASE` flag as described below, first for the current `%rip` and register state, and then unwinding a frame to a new `%rip` at each step, until the personality routine reports either success (a handler found in the queried frame) or failure (no handler) in all frames. It does not actually restore the unwound state, and the personality routine must access the state through the API.

- If the search phase reports a failure, e.g. because no handler was found, it will call `terminate()` rather than commence phase 2.

  If the search phase reports success, the framework restarts in the *cleanup* phase. Again, it repeatedly calls the personality routine, with the `_UA_CLEANUP_PHASE` flag as described below, first for the current `%rip` and register state, and then unwinding a frame to a new `%rip` at each step, until it gets to the frame with an identified handler. At that point, it restores the register state, and control is transferred to the user landing pad code.

Each of these two phases uses both the unwind library and the personality routines, since the validity of a given handler and the mechanism for transferring control to it are language-dependent, but the method of locating and restoring previous stack frames is language-independent.

A two-phase exception-handling model is not strictly necessary to implement C++ language semantics, but it does provide some benefits. For example, the first phase allows an exception-handling mechanism to *dismiss* an exception before stack unwinding begins, which allows *resumptive* exception handling (correcting the exceptional condition and resuming execution at the point where it was raised). While C++ does not support resumptive exception handling, other languages do, and the two-phase model allows C++ to coexist with those languages on the stack.

Note that even with a two-phase model, we may execute each of the two phases more than once for a single exception, as if the exception was being thrown more than once. For instance, since it is not possible to determine if a given catch clause will rethrow or not without executing it, the exception propagation effectively stops at each catch clause, and if it needs to restart, restarts at phase 1. This process is not needed for destructors (cleanup code), so the phase 1 can safely process all destructor-only frames at once and stop at the next enclosing catch clause.

For example, if the first two frames unwound contain only cleanup code, and the third frame contains a C++ catch clause, the personality routine in phase 1, does not indicate that it found a handler for the first two frames. It must do so for the third frame, because it is unknown how the exception will propagate out of this third frame, e.g. by rethrowing the exception or throwing a new one in C++.

The API specified by the AMD64 psABI for implementing this framework is described in the following sections.

6.2.2 Data Structures

Reason Codes

The unwind interface uses reason codes in several contexts to identify the reasons for failures or other actions, defined as follows:

```
typedef enum {
   _URC_NO_REASON = 0,
   _URC_FOREIGN_EXCEPTION_CAUGHT = 1,
   _URC_FATAL_PHASE2_ERROR = 2,
   _URC_FATAL_PHASE1_ERROR = 3,
   _URC_NORMAL_STOP = 4,
   _URC_END_OF_STACK = 5,
   _URC_HANDLER_FOUND = 6,
   _URC_INSTALL_CONTEXT = 7,
   _URC_CONTINUE_UNWIND = 8
} _Unwind_Reason_Code;
```

The interpretations of these codes are described below.

Exception Header

The unwind interface uses a pointer to an exception header object as its representation of an exception being thrown. In general, the full representation of an exception object is language- and implementation-specific, but it will be prefixed by a header understood by the unwind interface, defined as follows:

```
typedef void (*_Unwind_Exception_Cleanup_Fn)
  (_Unwind_Reason_Code reason,
   struct _Unwind_Exception *exc);
```

```
struct _Unwind_Exception {
   uint64                         exception_class;
   _Unwind_Exception_Cleanup_Fn   exception_cleanup;
   uint64                         private_1;
   uint64                         private_2;
};
```

An `_Unwind_Exception` object must be eightbyte aligned. The first two fields are set by user code prior to raising the exception, and the latter two should never be touched except by the runtime.

The `exception_class` field is a language- and implementation-specific identifier of the kind of exception. It allows a personality routine to distinguish between native and foreign exceptions, for example. By convention, the high 4 bytes indicate the vendor (for instance AMD\0), and the low 4 bytes indicate the language. For the C++ ABI described in this document, the low four bytes are C++\0.

The `exception_cleanup` routine is called whenever an exception object needs to be destroyed by a different runtime than the runtime which created the exception object, for instance if a Java exception is caught by a C++ catch handler. In such a case, a reason code (see above) indicates why the exception object needs to be deleted:

`_URC_FOREIGN_EXCEPTION_CAUGHT = 1` This indicates that a different runtime caught this exception. Nested foreign exceptions, or rethrowing a foreign exception, result in undefined behavior.

`_URC_FATAL_PHASE1_ERROR = 3` The personality routine encountered an error during phase 1, other than the specific error codes defined.

`_URC_FATAL_PHASE2_ERROR = 2` The personality routine encountered an error during phase 2, for instance a stack corruption.

Normally, all errors should be reported during phase 1 by returning from `_Unwind_RaiseException`. However, landing pad code could cause stack corruption between phase 1 and phase 2. For a C++ exception, the runtime should call `terminate()` in that case.

The private unwinder state (`private_1` and `private_2`) in an exception object should be neither read by nor written to by personality routines or other parts of the language-specific runtime. It is used by the specific implementation of the unwinder on the host to store internal information, for instance to remember the final handler frame between unwinding phases.

In addition to the above information, a typical runtime such as the C++ runtime will add language-specific information used to process the exception. This is expected to be a contiguous area of memory after the `_Unwind_Exception` object, but this is not required as long as the matching personality routines know how to deal with it, and the `exception_cleanup` routine de-allocates it properly.

Unwind Context

The `_Unwind_Context` type is an opaque type used to refer to a system-specific data structure used by the system unwinder. This context is created and destroyed by the system, and passed to the personality routine during unwinding.

```
struct _Unwind_Context
```

6.2.3 Throwing an Exception

`_Unwind_RaiseException`

```
_Unwind_Reason_Code _Unwind_RaiseException
  ( struct _Unwind_Exception *exception_object );
```

Raise an exception, passing along the given exception object, which should have its `exception_class` and `exception_cleanup` fields set. The exception object has been allocated by the language-specific runtime, and has a language-specific format, except that it must contain an `_Unwind_Exception` struct (see Exception Header above). `_Unwind_RaiseException` does not return, unless an error condition is found (such as no handler for the exception, bad stack format, etc.). In such a case, an `_Unwind_Reason_Code` value is returned.

Possibilities are:

`_URC_END_OF_STACK` The unwinder encountered the end of the stack during phase 1, without finding a handler. The unwind runtime will not have modified the stack. The C++ runtime will normally call `uncaught_exception()` in this case.

`_URC_FATAL_PHASE1_ERROR` The unwinder encountered an unexpected error during phase 1, e.g. stack corruption. The unwind runtime will not have modified the stack. The C++ runtime will normally call `terminate()` in this case.

If the unwinder encounters an unexpected error during phase 2, it should return `_URC_FATAL_PHASE2_ERROR` to its caller. In C++, this will usually be `__cxa_throw`, which will call `terminate()`.

The unwind runtime will likely have modified the stack (e.g. popped frames from it) or register context, or landing pad code may have corrupted them. As a result, the the caller of `_Unwind_RaiseException` can make no assumptions about the state of its stack or registers.

`_Unwind_ForcedUnwind`

```
typedef _Unwind_Reason_Code (*_Unwind_Stop_Fn)
  (int version,
   _Unwind_Action actions,
   uint64 exceptionClass,
   struct _Unwind_Exception *exceptionObject,
   struct _Unwind_Context *context,
   void *stop_parameter );
  _Unwind_Reason_Code_Unwind_ForcedUnwind
    ( struct _Unwind_Exception *exception_object,
     _Unwind_Stop_Fn stop,
     void *stop_parameter );
```

Raise an exception for forced unwinding, passing along the given exception object, which should have its `exception_class` and `exception_cleanup` fields set. The exception object has been allocated by the language-specific runtime, and has a language-specific format, except that it must contain an `_Unwind_Exception` struct (see Exception Header above).

Forced unwinding is a single-phase process (phase 2 of the normal exception-handling process). The `stop` and `stop_parameter` parameters control the termination of the unwind process, instead of the usual personality routine query. The `stop` function parameter is called for each unwind frame, with the parameters described for the usual personality routine below, plus an additional `stop_parameter`.

When the `stop` function identifies the destination frame, it transfers control (according to its own, unspecified, conventions) to the user code as appropriate without returning, normally after calling `_Unwind_DeleteException`. If not, it should return an `_Unwind_Reason_Code` value as follows:

_URC_NO_REASON This is not the destination frame. The unwind runtime will call the frame's personality routine with the _UA_FORCE_UNWIND and _UA_CLEANUP_PHASE flags set in actions, and then unwind to the next frame and call the stop function again.

_URC_END_OF_STACK In order to allow _Unwind_ForcedUnwind to perform special processing when it reaches the end of the stack, the unwind runtime will call it after the last frame is rejected, with a NULL stack pointer in the context, and the stop function must catch this condition (i.e. by noticing the NULL stack pointer). It may return this reason code if it cannot handle end-of-stack.

_URC_FATAL_PHASE2_ERROR The stop function may return this code for other fatal conditions, e.g. stack corruption.

If the stop function returns any reason code other than _URC_NO_REASON, the stack state is indeterminate from the point of view of the caller of _Unwind_ForcedUnwind. Rather than attempt to return, therefore, the unwind library should return _URC_FATAL_PHASE2_ERROR to its caller.

Example: longjmp_unwind()

The expected implementation of longjmp_unwind() is as follows. The setjmp() routine will have saved the state to be restored in its customary place, including the frame pointer. The longjmp_unwind() routine will call _Unwind_ForcedUnwind with a stop function that compares the frame pointer in the context record with the saved frame pointer. If equal, it will restore the setjmp() state as customary, and otherwise it will return _URC_NO_REASON or _URC_END_OF_STACK.

If a future requirement for two-phase forced unwinding were identified, an alternate routine could be defined to request it, and an actions parameter flag defined to support it.

_Unwind_Resume

```
void _Unwind_Resume
  (struct _Unwind_Exception *exception_object);
```

Resume propagation of an existing exception e.g. after executing cleanup code in a partially unwound stack. A call to this routine is inserted at the end of a landing pad that performed cleanup, but did not resume normal execution. It causes unwinding to proceed further.

`_Unwind_Resume` should not be used to implement rethrowing. To the unwinding runtime, the catch code that rethrows was a handler, and the previous unwinding session was terminated before entering it. Rethrowing is implemented by calling `_Unwind_RaiseException` again with the same exception object.

This is the only routine in the unwind library which is expected to be called directly by generated code: it will be called at the end of a landing pad in a "landing-pad" model.

6.2.4 Exception Object Management

`_Unwind_DeleteException`

```
void _Unwind_DeleteException
  (struct _Unwind_Exception *exception_object);
```

Deletes the given exception object. If a given runtime resumes normal execution after catching a foreign exception, it will not know how to delete that exception. Such an exception will be deleted by calling `_Unwind_DeleteException`. This is a convenience function that calls the function pointed to by the `exception_cleanup` field of the exception header.

6.2.5 Context Management

These functions are used for communicating information about the unwind context (i.e. the unwind descriptors and the user register state) between the unwind library and the personality routine and landing pad. They include routines to read or set the context record images of registers in the stack frame corresponding to a given unwind context, and to identify the location of the current unwind descriptors and unwind frame.

`_Unwind_GetGR`

```
uint64 _Unwind_GetGR
  (struct _Unwind_Context *context, int index);
```

This function returns the 64-bit value of the given general register. The register is identified by its index as given in 3.20.

During the two phases of unwinding, no registers have a guaranteed value.

_Unwind_SetGR

```
void _Unwind_SetGR
  (struct _Unwind_Context *context,
   int index,
   uint64 new_value);
```

This function sets the 64-bit value of the given register, identified by its index as for `_Unwind_GetGR`.

The behavior is guaranteed only if the function is called during phase 2 of unwinding, and applied to an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`. In that case, only registers `%rdi`, `%rsi`, `%rdx`, `%rcx` should be used. These scratch registers are reserved for passing arguments between the personality routine and the landing pads.

_Unwind_GetIP

```
uint64 _Unwind_GetIP
  (struct _Unwind_Context *context);
```

This function returns the 64-bit value of the instruction pointer (IP).

During unwinding, the value is guaranteed to be the address of the instruction immediately following the call site in the function identified by the unwind context. This value may be outside of the procedure fragment for a function call that is known to not return (such as `_Unwind_Resume`).

_Unwind_SetIP

```
void _Unwind_SetIP
  (struct _Unwind_Context *context,
   uint64 new_value);
```

This function sets the value of the instruction pointer (IP) for the routine identified by the unwind context.

The behavior is guaranteed only when this function is called for an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`. In this case, control will be transferred to the given address, which should be the address of a landing pad.

_Unwind_GetLanguageSpecificData

```
uint64 _Unwind_GetLanguageSpecificData
(struct _Unwind_Context *context);
```

This routine returns the address of the language-specific data area for the current stack frame.

This routine is not strictly required: it could be accessed through `_Unwind_GetIP` using the documented format of the DWARF Call Frame Information Tables, but since this work has been done for finding the personality routine in the first place, it makes sense to cache the result in the context. We could also pass it as an argument to the personality routine.

_Unwind_GetRegionStart

```
uint64 _Unwind_GetRegionStart
  (struct _Unwind_Context *context);
```

This routine returns the address of the beginning of the procedure or code fragment described by the current unwind descriptor block.

This information is required to access any data stored relative to the beginning of the procedure fragment. For instance, a call site table might be stored relative to the beginning of the procedure fragment that contains the calls. During unwinding, the function returns the start of the procedure fragment containing the call site in the current stack frame.

The personality routine is the function in the C++ (or other language) runtime library which serves as an interface between the system unwind library and language-specific exception handling semantics. It is specific to the code fragment described by an unwind info block, and it is always referenced via the pointer in the unwind info block, and hence it has no psABI-specified name.

Parameters

The personality routine parameters are as follows:

version Version number of the unwinding runtime, used to detect a mis-match between the unwinder conventions and the personality routine, or to provide backward compatibility. For the conventions described in this document, version will be 1.

actions Indicates what processing the personality routine is expected to perform, as a bit mask. The possible actions are described below.

`exceptionClass` An 8-byte identifier specifying the type of the thrown exception. By convention, the high 4 bytes indicate the vendor (for instance AMD\0), and the low 4 bytes indicate the language. For the C++ ABI described in this document, the low four bytes are C++\0. This is not a null-terminated string. Some implementations may use no null bytes.

`exceptionObject` The pointer to a memory location recording the necessary information for processing the exception according to the semantics of a given language (see the Exception Header section above).

`context` Unwinder state information for use by the personality routine. This is an opaque handle used by the personality routine in particular to access the frame's registers (see the Unwind Context section above).

return value The return value from the personality routine indicates how further unwind should happen, as well as possible error conditions. See the following section.

Personality Routine Actions

The actions argument to the personality routine is a bitwise OR of one or more of the following constants:

```
typedef int _Unwind_Action;
const _Unwind_Action _UA_SEARCH_PHASE = 1;
const _Unwind_Action _UA_CLEANUP_PHASE = 2;
const _Unwind_Action _UA_HANDLER_FRAME = 4;
const _Unwind_Action _UA_FORCE_UNWIND = 8;
```

`_UA_SEARCH_PHASE` Indicates that the personality routine should check if the current frame contains a handler, and if so return `_URC_HANDLER_FOUND`, or otherwise return `_URC_CONTINUE_UNWIND`. `_UA_SEARCH_PHASE` cannot be set at the same time as `_UA_CLEANUP_PHASE`.

`_UA_CLEANUP_PHASE` Indicates that the personality routine should perform cleanup for the current frame. The personality routine can perform this cleanup itself, by calling nested procedures, and return `_URC_CONTINUE_UNWIND`. Alternatively, it can setup the registers (including the IP) for transferring control to a "landing pad", and return `_URC_INSTALL_CONTEXT`.

_UA_HANDLER_FRAME During phase 2, indicates to the personality routine that the current frame is the one which was flagged as the handler frame during phase 1. The personality routine is not allowed to change its mind between phase 1 and phase 2, i.e. it must handle the exception in this frame in phase 2.

_UA_FORCE_UNWIND During phase 2, indicates that no language is allowed to "catch" the exception. This flag is set while unwinding the stack for `longjmp` or during thread cancellation. User-defined code in a catch clause may still be executed, but the catch clause must resume unwinding with a call to `_Unwind_Resume` when finished.

Transferring Control to a Landing Pad

If the personality routine determines that it should transfer control to a landing pad (in phase 2), it may set up registers (including IP) with suitable values for entering the landing pad (e.g. with landing pad parameters), by calling the context management routines above. It then returns `_URC_INSTALL_CONTEXT`.

Prior to executing code in the landing pad, the unwind library restores registers not altered by the personality routine, using the context record, to their state in that frame before the call that threw the exception, as follows. All registers specified as callee-saved by the base ABI are restored, as well as scratch registers `%rdi`, `%rsi`, `%rdx`, `%rcx` (see below). Except for those exceptions, scratch (or caller-saved) registers are not preserved, and their contents are undefined on transfer.

The landing pad can either resume normal execution (as, for instance, at the end of a C++ catch), or resume unwinding by calling `_Unwind_Resume` and passing it the `exceptionObject` argument received by the personality routine. `_Unwind_Resume` will never return.

`_Unwind_Resume` should be called if and only if the personality routine did not return `_Unwind_HANDLER_FOUND` during phase 1. As a result, the unwinder can allocate resources (for instance memory) and keep track of them in the exception object reserved words. It should then free these resources before transferring control to the last (handler) landing pad. It does not need to free the resources before entering non-handler landing-pads, since `_Unwind_Resume` will ultimately be called.

The landing pad may receive arguments from the runtime, typically passed in registers set using `_Unwind_SetGR` by the personality routine. For a landing pad that can call to `_Unwind_Resume`, one argument must be the `exceptionObject` pointer, which must be preserved to be passed to `_Unwind_Resume`.

The landing pad may receive other arguments, for instance a switch value indicating the type of the exception. Four scratch registers are reserved for this use (`%rdi`, `%rsi`, `%rdx`, `%rcx`).

Rules for Correct Inter-Language Operation

The following rules must be observed for correct operation between languages and/or runtimes from different vendors:

An exception which has an unknown class must not be altered by the personality routine. The semantics of foreign exception processing depend on the language of the stack frame being unwound. This covers in particular how exceptions from a foreign language are mapped to the native language in that frame.

If a runtime resumes normal execution, and the caught exception was created by another runtime, it should call `_Unwind_DeleteException`. This is true even if it understands the exception object format (such as would be the case between different C++ runtimes).

A runtime is not allowed to catch an exception if the `_UA_FORCE_UNWIND` flag was passed to the personality routine.

Example: Foreign Exceptions in C++. In C++, foreign exceptions can be caught by a `catch(...)` statement. They can also be caught as if they were of a `__foreign_exception` class, defined in `<exception>`. The `__foreign_exception` may have subclasses, such as `__java_exception` and `__ada_exception`, if the runtime is capable of identifying some of the foreign languages.

The behavior is undefined in the following cases:

- A `__foreign_exception` catch argument is accessed in any way (including taking its address).
- A `__foreign_exception` is active at the same time as another exception (either there is a nested exception while catching the foreign exception, or the foreign exception was itself nested).
- `uncaught_exception()`, `set_terminate()`, `set_unexpected()`, `terminate()`, or `unexpected()` is called at a time a foreign exception exists (for example, calling `set_terminate()` during unwinding of a foreign exception).

All these cases might involve accessing C++ specific content of the thrown exception, for instance to chain active exceptions.

Otherwise, a catch block catching a foreign exception is allowed:

- to resume normal execution, thereby stopping propagation of the foreign exception and deleting it, or
- to rethrow the foreign exception. In that case, the original exception object must be unaltered by the C++ runtime.

A catch-all block may be executed during forced unwinding. For instance, a longjmp may execute code in a `catch(...)` during stack unwinding. However, if this happens, unwinding will proceed at the end of the catch-all block, whether or not there is an explicit rethrow.

Setting the low 4 bytes of exception class to C++\0 is reserved for use by C++ runtimes compatible with the common C++ ABI.

Chapter 7

Development Environment

No changes required.

Chapter 8

Execution Environment

Not done yet.

Chapter 9

Conventions

[1]

9.1 GOT pointer and IP relative addressing

A basic difference between the Intel386 ABI and the AMD64 ABI is the way the GOT table is found. The Intel386 ABI, like (most) other processor specific ABIs, uses a dedicated register (`%ebx`) to address the base of the GOT table. The function prologue of every function needs to set up this register to the correct value. The AMD64 processor family introduces a new IP-relative addressing mode which is used in this ABI instead of using a dedicated register.

On AMD64 the GOT table contains 64-bit entries.

9.2 C++

For the C++ ABI we will use the IA-64 C++ ABI and instantiate it appropriately. The current draft of that ABI is available at:
`http://www.codesourcery.com/cxx-abi/`

[1] This chapter is used to document some features special to the AMD64 ABI. The different sections might be moved to another place or removed completely.

Appendix A

Linux Conventions

This chapter describes some details that are only relevant to GNU/Linux systems and the Linux kernel.

A.1 Execution of 32-bit Programs

The AMD64 processors are able to execute 64-bit AMD64 and also 32-bit ia32 programs. Libraries conforming to the Intel386 ABI will live in the normal places like `/lib`, `/usr/lib` and `/usr/bin`. Libraries following the AMD64, will use `lib64` subdirectories for the libraries, e.g `/lib64` and `/usr/lib64`. Programs conforming to Intel386 ABI and to the AMD64 ABI will share directories like `/usr/bin`. In particular, there will be no `/bin64` directory.

A.2 AMD64 Linux Kernel Conventions

The section is informative only.

A.2.1 Calling Conventions

The Linux AMD64 kernel uses internally the same calling conventions as user-level applications (see section 3.2.3 for details). User-level applications that like to call system calls should use the functions from the C library. The interface between the C library and the Linux kernel is the same as for the user-level applications with the following differences:

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.

2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.

3. The number of the syscall has to be passed in register `%rax`.

4. System-calls are limited to six arguments, no argument is passed directly on the stack.

5. Returning from the `syscall`, register `%rax` contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.

6. Only values of class INTEGER or class MEMORY are passed to the kernel.

A.3 Execution of 32bit programs

A.3.1 Required Processor Features

The AMD64 is able to execute 64 bit AMD64 and also 32 bit IA-32 programs. Libraries conforming to the Intel386 ABI will live in the normal places like `/lib`, `/usr/lib` and `/usr/bin`. Libraries following the AMD64, will use `lib64` subdirectories for the libraries, e.g `/lib64` and `/usr/lib64`. Programs conforming to Intel386 ABI and to the AMD64 ABI will share directories like `/usr/bin`. In particular, there will be no `bin64` directory.

Any program or kernel can expect that a AMD64 processor implements the features mentioned in table A.1. In general a program has to check itself whether those features are available but for AMD64 systems, these should always be available. Tablẽeffeatures uses the names for the processor features as documented in the processor manual.

A.3.2 Miscelleaneous Remarks

Linux Kernel code is not allowed to change the x87 and SSE units. If those are changed by kernel code, they have to be restored properly before sleeping or leaving the kernel. On preemptive kernels also more precautions may be needed.

Figure A.1: Required Processor Features

Feature	Comment
	Features need for programs
fpu	Necessary for `long double`, MMX
tsc	User-visible
cx8	User-visible
cmov	User-visible
mmx	User-visible
sse	User-visible, required for `float`
sse2	User-visible, required for `double`
fxsr	Required for SSE/SSE2
mmxext	User-visible
syscall	For calling the kernel
	Features need in the kernel
pae	This kind of page tables is used
pse	PAE needs PSE.
msr	At least needed to enter long mode
pge	Kernel optimization
pat	Kernel optimization
clflush	Kernel optimization

Index