System V Application Binary Interface
x86-64™ Architecture Processor Supplement
Draft Version 0.21

Edited by
Jan Hubicka¹, Andreas Jaeger², Mark Mitchell³

September 13, 2002

¹jh@suse.cz
²aj@suse.de
³mark@codesourcery.com
# Contents

## 1 Introduction

1.1 Differences from the Intel386 ABI ........................................... 4

## 2 Software Installation ............................................................. 6

## 3 Low Level System Information .................................................. 7

3.1 Machine Interface ................................................................. 7

3.1.1 Processor Architecture ....................................................... 7

3.1.2 Data Representation ............................................................ 7

3.2 Function Calling Sequence ....................................................... 10

3.2.1 Registers and the Stack Frame .............................................. 10

3.2.2 The Stack Frame ................................................................. 11

3.2.3 Parameter Passing ............................................................... 11

3.3 Operating System Interface ...................................................... 18

3.3.1 Exception Interface .............................................................. 18

3.3.2 Special Registers ................................................................. 18

3.3.3 Virtual Address Space .......................................................... 18

3.3.4 Page Size ........................................................................... 18

3.3.5 Virtual Address Assignments ............................................... 21

3.4 Process Initialization ............................................................... 22

3.4.1 Auxiliary Vector ................................................................. 22

3.5 Coding Examples ................................................................. 24

3.5.1 Architectural Constraints ...................................................... 24

3.5.2 Position-Independent Function Prologue .................................. 26

3.5.3 Data Objects ................................................................. 26

3.5.4 Function Calls ................................................................. 28

3.5.5 Branching ................................................................. 29

3.5.6 Variable Argument Lists ...................................................... 29

x86-64 ABI Draft 0.21 – September 13, 2002 – 4:57
3.6 DWARF Definition ........................................ 33
   3.6.1 DWARF Release Number ............................ 34
   3.6.2 DWARF Register Number Mapping ................. 34

4 Object Files ............................................. 35
   4.1 ELF Header ........................................... 35
      4.1.1 Machine Information ............................... 35
   4.2 Sections ............................................. 35
   4.3 Symbol Table ....................................... 36
   4.4 Relocation ......................................... 36
      4.4.1 Relocation Types ................................. 36

5 Program Loading and Dynamic Linking .................... 40
   5.1 Program Loading ................................... 40
   5.2 Dynamic Linking .................................... 40
      5.2.1 Program Interpreter ............................ 43
      5.2.2 Initialization and Termination Functions ....... 43

6 Libraries .................................................. 44
   6.1 C Library .......................................... 44
      6.1.1 Global Data Symbols ............................ 44
      6.1.2 Floating Point Environment Functions ......... 44
   6.2 Unwind Library Interface .......................... 45
      6.2.1 Exception Handler Framework .................... 45
      6.2.2 Data Structures ................................ 48
      6.2.3 Throwing an Exception ......................... 50
      6.2.4 Exception Object Management ................. 53
      6.2.5 Context Management ............................. 53
      6.2.6 Personality Routine ............................ 55

7 Development Environment ................................ 60

8 Execution Environment ................................ 61

9 Conventions ............................................. 62
   9.1 GOT pointer and IP relative addressing ............ 62
   9.2 Execution of 32bit programs ......................... 62
   9.3 C++ ................................................ 63
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Hardware Exceptions and Signals</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Floating-Point Exceptions</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>x87 Floating-Point Control Word</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>MXCSR Status Bits</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>x86-64 Identification</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Relocation Types</td>
<td>38</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Scalar Types</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Bit-Field Ranges</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Stack Frame with Base Pointer</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Register Usage</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Parameter Passing Example</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Register Allocation Example</td>
<td>17</td>
</tr>
<tr>
<td>3.7</td>
<td>Virtual Address Configuration</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>Conventional Segment Arrangements</td>
<td>22</td>
</tr>
<tr>
<td>3.9</td>
<td>Auxiliary Vector Types</td>
<td>23</td>
</tr>
<tr>
<td>3.10</td>
<td>Absolute Load and Store (Small Model)</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>Absolute Load and Store (Medium Model)</td>
<td>27</td>
</tr>
<tr>
<td>3.12</td>
<td>Position-Independent Load and Store (Small PIC Model)</td>
<td>28</td>
</tr>
<tr>
<td>3.13</td>
<td>Position-Independent Direct Function Call</td>
<td>28</td>
</tr>
<tr>
<td>3.14</td>
<td>Position-Independent Indirect Function Call</td>
<td>29</td>
</tr>
<tr>
<td>3.15</td>
<td>Register Save Area</td>
<td>30</td>
</tr>
<tr>
<td>3.16</td>
<td>va_list Type Declaration</td>
<td>31</td>
</tr>
<tr>
<td>3.17</td>
<td>Sample Implementation of va_arg(l, int)</td>
<td>33</td>
</tr>
<tr>
<td>3.18</td>
<td>DWARF Register Number Mapping</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Relocatable Fields</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Global Offset Table</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Procedure Linkage Table</td>
<td>41</td>
</tr>
</tbody>
</table>

# Revision History

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 x86-64 architecture\footnote{The architecture specification is available on the web at \url{http://www.x86-64.org/documentation}.} is an extension of the x86 architecture. Any processor implementing the x86-64 architecture specification will also provide compatibility 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 x86-64 ABI may provide support for executing programs that are designed to execute in these compatibility modes. The x86-64 ABI does not apply to such programs; this document applies only programs running in the “long” mode provided by the x86-64 architecture.

Except where otherwise noted, the x86-64 architecture ABI follows the conventions described in the Intel386 ABI. Rather than replicate the entire contents of the Intel386 ABI, the x86-64 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.\footnote{See section \ref{sec:abi} for details on C++ ABI.}

\section{1.1 Differences from the Intel386 ABI}

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

\begin{itemize}
\item Sizes of fundamental data types.
\end{itemize}
- 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.

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.

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. Although a long

1The 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 ia32 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.

2Initial implementations of the x86-64 architecture are expected to support operations on the __float128 type only via software emulation.

3This type is the x87 double extended precision data type.
## Figure 3.1: Scalar Types

<table>
<thead>
<tr>
<th>Type</th>
<th>C</th>
<th>sizeof (bytes)</th>
<th>Alignment (bytes)</th>
<th>x86-64 Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Bool†</td>
<td>1</td>
<td>1</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>signed byte</td>
<td></td>
</tr>
<tr>
<td>signed char</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
<td>1</td>
<td>unsigned byte</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>signed twobyte</td>
<td></td>
</tr>
<tr>
<td>signed short</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned short</td>
<td></td>
<td>2</td>
<td>unsigned twobyte</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>signed fourbyte</td>
<td></td>
</tr>
<tr>
<td>signed int</td>
<td>4</td>
<td>4</td>
<td>signed fourbyte</td>
<td></td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>4</td>
<td>unsigned fourbyte</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
<td>signed eightbyte</td>
<td></td>
</tr>
<tr>
<td>signed long</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>signed long long</td>
<td></td>
<td>8</td>
<td>signed eightbyte</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td></td>
<td>8</td>
<td>unsigned eightbyte</td>
<td></td>
</tr>
<tr>
<td>unsigned long long</td>
<td></td>
<td>8</td>
<td>unsigned eightbyte</td>
<td></td>
</tr>
<tr>
<td>__int128††</td>
<td>16</td>
<td>16</td>
<td>signed sixteenbyte</td>
<td></td>
</tr>
<tr>
<td>signed __int128††</td>
<td>16</td>
<td>16</td>
<td>signed sixteenbyte</td>
<td></td>
</tr>
<tr>
<td>unsigned __int128††</td>
<td>16</td>
<td>16</td>
<td>unsigned sixteenbyte</td>
<td></td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>any-type *</td>
<td>8</td>
<td>8</td>
<td>unsigned eightbyte</td>
<td></td>
</tr>
<tr>
<td>any-type (*)()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>single (IEEE)</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>double (IEEE)</td>
<td></td>
</tr>
<tr>
<td>long double</td>
<td></td>
<td></td>
<td>80-bit extended (IEEE)</td>
<td></td>
</tr>
<tr>
<td>__float128††</td>
<td>16</td>
<td>16</td>
<td>128-bit extended (IEEE)</td>
<td></td>
</tr>
<tr>
<td>Packed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m64††</td>
<td>8</td>
<td>8</td>
<td>MMX and 3DNow!</td>
<td></td>
</tr>
<tr>
<td>__m128††</td>
<td>16</td>
<td>16</td>
<td>SSE and SSE-2</td>
<td></td>
</tr>
</tbody>
</table>

† 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 x86-64 architecture 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.

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.

4No other changes required.

Bit-Fields

Amend the description of bit-field ranges as follows:

<table>
<thead>
<tr>
<th>Bit-field Type</th>
<th>Width $w$</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed long</td>
<td>1 to 64</td>
<td>$-2^{w-1}$ to $2^{w-1} - 1$</td>
</tr>
<tr>
<td>long</td>
<td></td>
<td>0 to $2^w - 1$</td>
</tr>
<tr>
<td>unsigned long</td>
<td></td>
<td>0 to $2^w - 1$</td>
</tr>
</tbody>
</table>

4The 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 x86-64 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. 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. 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.
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 $(\%rsp - 8)$ 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.\footnote{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.}

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.\footnote{Locations within 128 bytes can be addressed using one-byte displacements.} 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...
sections.

**Definitions** We first define a number of classes to classify arguments. The classes are corresponding to x86-64 register classes and defined as:

**INTEGER** This class consists of integral types that fit into one of the general purpose registers (\%rax–%r15).

**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 in the most significant half of it.

**X87, X87UP** These classes consists of types that will be passed 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 in memory via the stack.

**Classification** The size of each argument gets rounded up to eight bytes.\(^9\)

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.

- 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

\(^9\)Therefore the stack will always be eightbyte aligned.
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.

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 structure or union type, or contains unaligned fields, it has class MEMORY.
2. Both eightbytes get initialized to class NO_CLASS.
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 or X87UP 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 preceded by SSE, it is converted to SSE.

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.

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.
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.\(^\text{12}\)

3. If the class is SSE, the next available SSE register is used, the registers are taken in the order from `%xmm0` to `%xmm7`.\(^\text{13}\)

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 or X87UP, it is passed in memory.

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\(^\text{13}\)) 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`\(^\text{14}\) 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.

\(^{12}\)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.

\(^{13}\)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.
### Figure 3.4: Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Preserved across function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register</td>
<td>No</td>
</tr>
<tr>
<td>%rbx</td>
<td>callee-saved register; optionally used as base pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rcx</td>
<td>used to pass 4th integer argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdx</td>
<td>used to pass 3rd argument to functions; 2nd return register</td>
<td>No</td>
</tr>
<tr>
<td>%rsp</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rbp</td>
<td>callee-saved register; optionally used as frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rsi</td>
<td>used to pass 2nd argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdi</td>
<td>used to pass 1st argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r8</td>
<td>used to pass 5th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r9</td>
<td>used to pass 6th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r10</td>
<td>temporary register, used for passing a function’s static chain pointer</td>
<td>No</td>
</tr>
<tr>
<td>%r11</td>
<td>temporary register</td>
<td>No</td>
</tr>
<tr>
<td>%r12–r15</td>
<td>callee-saved registers</td>
<td>Yes</td>
</tr>
<tr>
<td>%xmm0–%xmm1</td>
<td>used to pass and return floating point arguments</td>
<td>No</td>
</tr>
<tr>
<td>%xmm2–%xmm7</td>
<td>used to pass floating point arguments</td>
<td>No</td>
</tr>
<tr>
<td>%xmm8–%xmm15</td>
<td>temporary registers</td>
<td>No</td>
</tr>
<tr>
<td>%mmx0–%mmx7</td>
<td>temporary registers</td>
<td>No</td>
</tr>
<tr>
<td>%st0</td>
<td>temporary register; used to return long double arguments</td>
<td>No</td>
</tr>
<tr>
<td>%st1–%st7</td>
<td>temporary registers</td>
<td>No</td>
</tr>
<tr>
<td>%fs</td>
<td>Reserved for system use (as thread specific data register)</td>
<td>No</td>
</tr>
</tbody>
</table>
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`.\footnote{We currently discuss changing this to `%rdi` as return register to avoid one move.}

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.

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

   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

<table>
<thead>
<tr>
<th>General Purpose Registers</th>
<th>Floating Point Registers</th>
<th>Stack Frame Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi: e</td>
<td>%xmm0: s.d</td>
<td>0: ld</td>
</tr>
<tr>
<td>%rsi: f</td>
<td>%xmm1: m</td>
<td>16: j</td>
</tr>
<tr>
<td>%rdx: s.a, s.b</td>
<td>%xmm2: n</td>
<td>24: k</td>
</tr>
<tr>
<td>%rcx: g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8: h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9: i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.3 Operating System Interface

3.3.1 Exception Interface
As the x86-64 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 x86-64 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 x86-64 does not define any instructions that generate a bounds check fault in long mode.

3.3.2 Special Registers
The x86-64 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 5.3 and 5.4.

3.3.3 Virtual Address Space
Although the x86-64 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 0x00007fffffff. 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.

---

\(^ {16}\)0x0000ffff fffffffffff is not a canonical address and cannot be used.
Table 3.1: Hardware Exceptions and Signals

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

Table 3.2: Floating-Point Exceptions

<table>
<thead>
<tr>
<th>Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPE_FLTDIV</td>
<td>floating-point divide by zero</td>
</tr>
<tr>
<td>FPE_FLTOVF</td>
<td>floating-point overflow</td>
</tr>
<tr>
<td>FPEFLTUND</td>
<td>floating-point underflow</td>
</tr>
<tr>
<td>FPEFLTRES</td>
<td>floating-point inexact result</td>
</tr>
<tr>
<td>FPEFLTINV</td>
<td>invalid floating-point operation</td>
</tr>
</tbody>
</table>
### Table 3.3: x87 Floating-Point Control Word

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>0</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>PC</td>
<td>11</td>
<td>Double extended precision</td>
</tr>
<tr>
<td>PM</td>
<td>1</td>
<td>Precision masked</td>
</tr>
<tr>
<td>UM</td>
<td>1</td>
<td>Underflow masked</td>
</tr>
<tr>
<td>OM</td>
<td>1</td>
<td>Overflow masked</td>
</tr>
<tr>
<td>ZM</td>
<td>1</td>
<td>Zero divide masked</td>
</tr>
<tr>
<td>DM</td>
<td>1</td>
<td>Denormal operand masked</td>
</tr>
<tr>
<td>IM</td>
<td>1</td>
<td>Invalid operation masked</td>
</tr>
</tbody>
</table>

### Table 3.4: MXCSR Status Bits

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZ</td>
<td>0</td>
<td>Do not flush to zero</td>
</tr>
<tr>
<td>RC</td>
<td>0</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>PM</td>
<td>1</td>
<td>Precision masked</td>
</tr>
<tr>
<td>UM</td>
<td>1</td>
<td>Underflow masked</td>
</tr>
<tr>
<td>OM</td>
<td>1</td>
<td>Overflow masked</td>
</tr>
<tr>
<td>ZM</td>
<td>1</td>
<td>Zero divide masked</td>
</tr>
<tr>
<td>DM</td>
<td>1</td>
<td>Denormal operand masked</td>
</tr>
<tr>
<td>IM</td>
<td>1</td>
<td>Invalid operation masked</td>
</tr>
<tr>
<td>DAZ</td>
<td>0</td>
<td>Denormals are not zero</td>
</tr>
</tbody>
</table>
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.

![Figure 3.7: Virtual Address Configuration]

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

3.4.1 Auxiliary Vector

The x86-64 ABI uses the following auxiliary vector types.
### Figure 3.9: Auxiliary Vector Types

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

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 i386 ABI.
3.5.1 Architectural Constraints

The x86-64 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

x86-64 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-independent 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)

| extern int src; | .extern src |
| extern int dst; | .extern dst |
| extern int *ptr; | .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 Model)

| extern int src; | .extern src |
| extern int dst; | .extern dst |
| extern int *ptr; | .extern ptr |
| .text | |
| dst = src; | movabs1 src, %eax |
| movabs1 %eax, dst | |
| ptr = &dst; | movabsq $dst,%rdx |
| movabsq %rdx, ptr | |
| *ptr = src; | movabsq ptr,%rdx |
| movabs1 src,%eax |
| movl %eax, (%rdx) | |
3.5.4 Function Calls
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 x86-64 architecture because some arguments are passed in registers. Portable C programs must use the header files `<stdarg.h>` or `<varargs.h>` in order to handle variable argument lists.

When a function taking variable-arguments is called, `%rax` must be set to eight times the number of floating point parameters passed to the function in SSE registers.

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 3.15.
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\footnote{This fact may be determined either by exploring types used by the \texttt{va_arg} macro, or by the fact that the named arguments already are exhausted the argument registers entirely.}, the register save area may be omitted entirely.

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

\begin{figure}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Register & Offset \\
\hline
\texttt{%rdi} & 0 \\
\texttt{%rsi} & 8 \\
\texttt{%rdx} & 16 \\
\texttt{%rcx} & 24 \\
\texttt{%r8} & 32 \\
\texttt{%r9} & 40 \\
\texttt{%xmm0} & 48 \\
\texttt{%xmm1} & 64 \\
\ldots & \\
\texttt{%xmm15} & 288 \\
\hline
\end{tabular}
\caption{Register Save Area}
\end{figure}

\textbf{The \texttt{va_list} Type}

The \texttt{va_list} type is an array containing a single element of one structure containing the necessary information to implement the \texttt{va_arg} macro. The C definition of \texttt{va_list} type is given in figure 3.16.
Figure 3.16: va_list Type Declaration

```c
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.

[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:

   \[ l->gp\_offset > 48 - num\_gp \times 8 \]

or

   \[ l->fp\_offset > 304 - num\_fp \times 16 \]

then go to step 4.

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:

   \[ l->gp\_offset = l->gp\_offset + num\_gp \times 8 \]
   \[ l->fp\_offset = l->fp\_offset + num\_fp \times 16. \]

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:

   \[ l->overflow\_arg\_area + \text{sizeof(type)} \]

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 3.17 is a sample implementation of the `va_arg` macro.

![Figure 3.17: Sample Implementation of `va_arg(l, int)`](image)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl l-&gt;gp_offset,%eax</td>
<td>Is register available?</td>
</tr>
<tr>
<td>cmpl $48, %eax</td>
<td>If not, use stack</td>
</tr>
<tr>
<td>jae stack</td>
<td></td>
</tr>
<tr>
<td>leal $8(%rax), %edx</td>
<td>Next available register</td>
</tr>
<tr>
<td>addq l-&gt;reg_save_area,%rax</td>
<td>Address of saved register</td>
</tr>
<tr>
<td>movl %edx, l-&gt;gp_offset</td>
<td>Update gp_offset</td>
</tr>
<tr>
<td>jmp fetch</td>
<td></td>
</tr>
</tbody>
</table>

**stack:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq l-&gt;overflow_arg_area,%rax</td>
<td>Address of stack slot</td>
</tr>
<tr>
<td>leaq 8(%rax), %rdx</td>
<td>Next available stack slot</td>
</tr>
<tr>
<td>movq %rdx, l-&gt;overflow_arg_area</td>
<td>Update</td>
</tr>
</tbody>
</table>

**fetch:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl (%rax), %eax</td>
<td>Load argument</td>
</tr>
</tbody>
</table>

### 3.6 DWARF Definition

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the x86-64 processor family. The x86-64 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.

---

18 This section is structured in a way similar to the psABI for PowerPC
3.6.1 DWARF Release Number

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the x86-64 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 3.18 outlines the register number mapping for the x86-64 processor family.

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

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.
Chapter 4

Object Files

4.1 ELF Header

4.1.1 Machine Information

For file identification in e_ident, the x86-64 architecture requires the following values.

<table>
<thead>
<tr>
<th>Position</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_ident[EI_CLASS]</td>
<td>ELFCLASS64</td>
</tr>
<tr>
<td>e_ident[EI_DATA]</td>
<td>ELFDATA2LSB</td>
</tr>
</tbody>
</table>

Processor identification resides in the ELF header’s e_machine member and must have the value EM_X86_64.

4.2 Sections

No changes required.

1The value of this identifier is 62.
4.3 Symbol Table

No changes required.

4.4 Relocation

4.4.1 Relocation Types

The x86-64 ABI adds one additional field:

---

Figure 4.1: Relocatable Fields

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>word8</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>word16</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>word32</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>word64</td>
<td>0</td>
</tr>
</tbody>
</table>

---
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 x86-64 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 x86-64 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 x86-64 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 mem-

ory 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 relo-

cated (computed using r_offset).

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

The x86-64 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Field</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_X86_64_NONE</td>
<td>0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>R_X86_64_64</td>
<td>1</td>
<td>word64</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_PC32</td>
<td>2</td>
<td>word32</td>
<td>S + A - P</td>
</tr>
<tr>
<td>R_X86_64_GOT32</td>
<td>3</td>
<td>word32</td>
<td>G + A</td>
</tr>
<tr>
<td>R_X86_64_PLT32</td>
<td>4</td>
<td>word32</td>
<td>L + A - P</td>
</tr>
<tr>
<td>R_X86_64_COPY</td>
<td>5</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>R_X86_64_GLOB_DAT</td>
<td>6</td>
<td>word64</td>
<td>S</td>
</tr>
<tr>
<td>R_X86_64_JUMP_SLOT</td>
<td>7</td>
<td>word64</td>
<td>S</td>
</tr>
<tr>
<td>R_X86_64_RELATIVE</td>
<td>8</td>
<td>word64</td>
<td>B + A</td>
</tr>
<tr>
<td>R_X86_64_GOTPCREL</td>
<td>9</td>
<td>word32</td>
<td>G + GOT + A - P</td>
</tr>
<tr>
<td>R_X86_64_32</td>
<td>10</td>
<td>word32</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_32S</td>
<td>11</td>
<td>word32</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_16</td>
<td>12</td>
<td>word16</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_PC16</td>
<td>13</td>
<td>word16</td>
<td>S + A - P</td>
</tr>
<tr>
<td>R_X86_64_8</td>
<td>14</td>
<td>word8</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_PC8</td>
<td>15</td>
<td>word8</td>
<td>S + A - P</td>
</tr>
</tbody>
</table>

The special semantics for most of these relocation types are identical to those used for the Intel386 architecture.  

The R_X86_64_GOTPCREL relocation has different semantics from the i386 R_I386_GOTPC relocation. In particular, because the x86-64 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

---

2 Even though the x86-64 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 x86-64 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 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.
the GOT where the symbol’s address is given and the location where the relocation is applied.

The \texttt{R}_{X86\_64\_32} and \texttt{R}_{X86\_64\_32S} relocations truncate the computed value to 32-bits. The linker must verify that the generated value for the \texttt{R}_{X86\_64\_32} (\texttt{R}_{X86\_64\_32S}) relocation zero-extends (sign-extends) to the original 64-bit value.

A program or object file using \texttt{R}_{X86\_64\_8}, \texttt{R}_{X86\_64\_16}, \texttt{R}_{X86\_64\_PC16} or \texttt{R}_{X86\_64\_PC8} relocations is not conformant to this ABI, these relocations are only added for documentation purposes. The \texttt{R}_{X86\_64\_16}, and \texttt{R}_{X86\_64\_8} relocations truncate the computed value to 16-bits resp. 8-bits.
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

```c
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 x86-64 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 \texttt{name1}, which transfers control to the label \texttt{.PLT1}.

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

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

6. After pushing the relocation index, the program then jumps to \texttt{.PLT0}, the first entry in the procedure linkage table. The \texttt{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 \texttt{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 \texttt{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 x86-64 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 x86-64 ABI.

6.1.2 Floating Point Environment Functions

ISO C 99 defines the floating point environment functions from <fenv.h>. Since x86-64 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 cases. 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 expected to be provided by any x86-64 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:

- exceptions, as defined by languages that support them (such as C++)

---

1 The overall structure and the external interface is derived from the IA-64 UNIX System V ABI
• “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_RaiseException 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 x86-64 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:

```c
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:

```c
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_RaiseException

_Unwind_RaiseException

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:

UNUSED_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.

UNUSED_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 notic-
ing the NULL stack pointer). It may return this reason code if it cannot handle end-of-stack.

_UCRC_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 _UCRC_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 _UCRC_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 _UCRC_NO_REASON or _UCRC_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 3.18.

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.

_U 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).

_U 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.

_U 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 un-
winding, the function returns the start of the procedure fragment containing the
call site in the current stack frame.

6.2.6 Personality Routine

_Unwind_Reason_Code (*__personality_routine)
    (int version,
     _Unwind_Action actions,
     uint64 exceptionClass,
     struct _Unwind_Exception *exceptionObject,
     struct _Unwind_Context *context);

The personality routine is the function in the C++ (or other language) run-
time 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 per-
          form, as a bit mask. The possible actions are described below.

exceptionClass  An 8-byte identifier specifying the type of the thrown ex-
                 ception. 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:

```c
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 callee-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

9.1 GOT pointer and IP relative addressing

A basic difference between the i386 ABI and the x86-64 ABI is the way the GOT table is found. The i386 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 x86-64 processor family introduces a new IP-relative addressing mode which is used in this ABI instead of using a dedicated register.

On x86-64 the GOT table contains 64 bit entries.

9.2 Execution of 32bit programs

The x86-64 is able to execute 64 bit x86-64 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 x86-64, will use lib64 subdirectories for the libraries, e.g /lib64 and /usr/lib64. Programs conforming to Intel386 ABI and to the x86-64 ABI will share directories like /usr/bin. In particular, there will be no bin64 directory.

This chapter is used to document some features special to the x86-64 ABI. The different sections might be moved to another place or removed completely.
9.3 C++

For the C++ ABI we will use the ia64 C++ ABI and instantiate it appropriately. The current draft of that ABI is available at:

http://www.codesourcery.com/cxx-abi/
Appendix A

x86-64 Linux Kernel Conventions

This chapter is informative only.

A.1 Calling Conventions

The Linux x86-64 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.