System V Application Binary Interface
Intel386 Architecture Processor Supplement
Version 1.0

Edited by
H.J. Lu[1], David L Kreitzer[2], Milind Girkar[3], Zia Ansari[4]

Based on

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Edited by

H.J. Lu[5], Michael Matz[6], Milind Girkar[7], Jan Hubička[8], Andreas Jaeger[9], Mark Mitchell[10]

February 3, 2015

[1]hongjiu.lu@intel.com
[2]david.l.kreitzer@intel.com
[3]milind.girkar@intel.com
[4]zia.ansari@intel.com
[5]hongjiu.lu@intel.com
[6]matz@suse.de
[7]milind.girkar@intel.com
[8]jh@suse.cz
[9]aj@suse.de
[10]mark@codesourcery.com

Contents

List of Tables

List of Figures

Revision History

1.0 — 2015-02-03 Reformat table of Returning Values.

0.1 — 2015-01-19 Initial release.

Chapter 1

About this Document

This document is a supplement to the existing Intel386 System V Application Binary Interface (ABI) document available at `http://www.sco.com/developers/devspecs/abi386-4.pdf`, which describes the Linux IA-32 ABI for processors compatible with the Intel386 Architecture.

Intel processors released after the Pentium processors (Pentium 4, Intel Core, and later), have introduced new architecture features, particularly new registers and corresponding instructions to operate on the registers, like the MMX, Intel SSE(1-4), and Intel AVX instruction set extensions. The C/C++ programming languages have evolved to allow programmers to use new data types (for example, `__m64`, `__m128`, and `__m256`). Many compilers (including the Intel compiler and GCC) have supported these data types for some time. Other features in tools (for example, the decimal floating point types, 64-bit integers, exception handling, and so on) have also been developed since the original ABI was written.

This document describes the conventions and constraints on the implementation of these new features for interoperability between various tools.

1.1 Scope

This document describes the conventions on the new C/C++ language types (including alignment and parameter passing conventions), the relocation symbols in the object binary, and the exception handling mechanism for Intel386 architecture. Some of this work has been discussed before `http://groups.google.com/group/ia32-abi` or `http://www.akkadia.org/drepper/tls.pdf`. The C++ object model that is expected to be followed is described in http://www.codesourcery.com/public/cxx-abi/abi.html. In particular, this document specifies the information that compilers have to generate and the library routines that do the frame unwinding for exception handling.

1.2 Related Information

Links to useful documents:

- System V Application Binary Interface, Intel386TMArchitecture Processor Supplement Fourth Edition: http://www.sco.com/developers/devspecs/abi386-4.pdf
- System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.5: http://www.x86-64.org/documentation/abi.pdf
- Discussion of Intel processor extensions: http://groups.google.com/group/ia32-abi
- ELF Handling of Thread-Local Storage: http://www.akkadia.org/drepper/tls.pdf
- Thread-Local Storage Descriptors for IA32 and AMD64/EM64T: http://people.redhat.com/aoliva/writeups/TLS/RFC-TLSDESC-x86.txt
- Itanium C++ ABI, Revision 1.86: http://www.codesourcery.com/public/cxx-abi/abi.html

Chapter 2

Low Level System Information

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

2.1 Machine Interface

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

2.1.1 Data Representation

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

Fundamental Types

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

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

Table 2.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	_Bool†	1	1	boolean
	char signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	short signed short	2	2	signed twobyte
	unsigned short	2	2	unsigned twobyte
	int signed int enum†††	4	4	signed fourbyte
	unsigned int	4	4	unsigned fourbyte
	long signed long	4	4	signed fourbyte
	unsigned long	4	4	unsigned fourbyte
	long long signed long long	8	4	signed eightbyte
	unsigned long long	8	4	unsigned eightbyte
Pointer	*any-type* * *any-type* (*)()	4	4	unsigned fourbyte
Floating-point	float	4	4	single (IEEE-754)
	double long double††††	8	4	double (IEEE-754)
	__float80†† long double††††	12	4	80-bit extended (IEEE-754)
	__float128††	16	16	128-bit extended (IEEE-754)
Complex Floating-point	_Complex float	8	4	complex single (IEEE-754)
	_Complex double _Complex long double††††	16	4	complex double (IEEE-754)
	_Complex __float80†† _Complex long double††††	24	4	complex 80-bit extended (IEEE-754)
	_Complex __float128††	32	16	complex 128-bit extended (IEEE-754)
Decimal-floating-point	_Decimal32	4	4	32bit BID (IEEE-754R)
	_Decimal64	8	8	64bit BID (IEEE-754R)
	_Decimal128	16	16	128bit BID (IEEE-754R)
Packed	__m64††	8	8	*MMX* and 3DNow!
	__m128††	16	16	SSE and SSE-2
	__m256††	32	32	AVX

† This type is called `bool` in C++.
†† These types are optional.
††† C++ and some implementations of C permit enums larger than an int. The underlying type is bumped to an unsigned int, long int or unsigned long int, in that order.
†††† The `long double` type is 64-bit, the same as the `double` type, on the Android™ platform. More information on the Android™ platform is available from `http://www.android.com/`.

The 128-bit floating-point type uses a 15-bit exponent, a 113-bit mantissa (the high order significant bit is implicit) and an exponent bias of 16383.[2]

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

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

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

Booleans, when stored in a memory object, are stored as single byte objects the value of which is always 0 (`false`) or 1 (`true`). When stored in integer registers (except for passing as arguments), all 4 bytes of the register are significant; any nonzero value is considered `true`.

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

Aggregates and Unions

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

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

2.2 Function Calling Sequence

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

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

---

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

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

2.2.1 Registers

The Intel386 architecture provides 8 general purpose 32-bit registers. In addition the architecture provides 8 SSE registers, each 128 bits wide and 8 x87 floating point registers, each 80 bits wide. Each of the x87 floating point registers may be referred to in *MMX* mode as a 64-bit register. All of these registers are global to all procedures active for a given thread.

Intel AVX (Advanced Vector Extensions) provides 8 256-bit wide AVX registers (`%ymm0` - `%ymm7`). The lower 128-bits of `%ymm0` - `%ymm7` are aliased to the respective 128b-bit SSE registers (`%xmm0` - `%xmm7`). For purposes of parameter passing and function return, `%xmmN` and `%ymmN` refer to the same register. Only one of them can be used at the same time. We use vector register to refer to either SSE or AVX register.

The CPU shall be in x87 mode upon entry to a function. Therefore, every function that uses the *MMX* registers is required to issue an `emms` or `femms` instruction after using *MMX* registers, before returning or calling another function. [4] The direction flag `DF` in the `%EFLAGS` register must be clear (set to "forward" direction) on function entry and return. Other user flags have no specified role in the standard calling sequence and are *not* preserved across calls.

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

2.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Table 2.2 shows the stack organization.

The end of the input argument area shall be aligned on a 16 (32, if `__m256` is passed on stack) byte boundary. In other words, the value $(\texttt{\%esp} + 4)$ is always a multiple of 16 (32) when control is transferred to the function entry point. The stack pointer, `%esp`, always points to the end of the latest allocated stack frame. [5]

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

[5] The conventional use of `%ebp` as a frame pointer for the stack frame may be avoided by using `%esp` (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (`%ebp`) available.

Table 2.2: Stack Frame with Base Pointer

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

2.2.3 Parameter Passing and Returning Values

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

Passing Parameters

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

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

The exceptions to parameters passed on stack are as follows:

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

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

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

Returning Values

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

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

Returning Values in Memory

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

As an example of the register passing conventions, consider the declarations and the function call shown in Table 2.5. The corresponding register allocation is given in Table 2.6, the stack frame layout given in Table 2.7 shows the frame before calling the function.

Table 2.3: Register Usage

Register	Usage	Preserved across function calls
%eax	scratch register; also used to return integer and pointer values from functions; also stores the address of a returned struct or union	No
%ebx	callee-saved register; also used to hold the GOT pointer when making function calls via the PLT	Yes
%ecx	scratch register	No
%edx	scratch register; also used to return the upper 32bits of some 64bit return types	No
%esp	stack pointer	Yes
%ebp	callee-saved register; optionally used as frame pointer	Yes
%esi	callee-saved register	yes
%edi	callee-saved register	yes
%xmm0, %ymm0	scratch registers; also used to pass and return __m128, __m256 parameters	No
%xmm1–%xmm2, %ymm1–%ymm2	scratch registers; also used to pass __m128, __m256 parameters	No
%xmm3–%xmm7, %ymm3–%ymm7	scratch registers	No
%mm0	scratch register; also used to pass and return __m64 parameter	No
%mm1–%mm2	used to pass __m64 parameters	No
%mm3–%mm7	scratch registers	No
%st0	scratch register; also used to return float, double, long double, __float80 parameters	No
%st1–%st7	scratch registers	No
%gs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

Table 2.4: Return Value Locations for Fundamental Data Types

Type	C	Return Value Location
Integral	_Bool char signed char unsigned char	%al The upper 24 bits of %eax are undefined. The caller must not rely on these being set in a predefined way by the called function.
	short signed short unsigned short	%ax The upper 16 bits of %eax are undefined. The caller must not rely on these being set in a predefined way by the called function.
	int signed int enum unsigned int long signed long unsigned long	%eax
	long long signed long long unsigned long long	%edx:%eax The most significant 32 bits are returned in %edx. The least significant 32 bits are returned in %eax.
Pointer	*any-type* * *any-type* (*)()	%eax
Floating-point	float	%st0
	double	%st0
	long double	%st0
	__float80	%st0
	__float128	memory
Complex floating-point	_Complex float	%edx:%eax The real part is returned in %eax. The imaginary part is returned in %edx.
	_Complex double	memory
	_Complex long double	memory
	_Complex __float80	memory
	_Complex __float128	memory
Decimal-floating-point	_Decimal32	%eax
	_Decimal64	%edx:%eax The most significant 32 bits are returned in %edx. The least significant 32 bits are returned in %eax.
	_Decimal128	memory
Packed	__m64	%mm0
	__m128	%xmm0
	__m256	%ymm0

Table 2.5: Parameter Passing Example

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

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

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

Table 2.6: Register Allocation for Parameter Passing Example

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

Table 2.7: Stack Layout at the Call

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

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

2.2.4 Variable Argument Lists

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

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

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

2.3 Process Initialization

2.3.1 Initial Stack and Register State

Special Registers

The Intel386 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 2.8 and 2.9.

Table 2.8: x87 Floating-Point Control Word

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

Table 2.9: MXCSR Status Bits

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

The EFLAGS register contains the system flags, such as the direction flag and the carry flag. The low 16 bits (FLAGS portion) of EFLAGS are accessible by application software. The state of them at process initialization is shown in table 2.10.

Table 2.10: EFLAGS Bits

Field	Value	Note
DF	0	Direction forward
CF	0	No carry
PF	0	Even parity
AF	0	No auxiliary carry
ZF	0	No zero result
SF	0	Unsigned result
OF	0	No overflow occurred

Stack State

This section describes the machine state that exec (BA_OS) creates for new processes. Various language implementations transform this initial program state to the state required by the language standard.

For example, a C program begins executing at a function named `main` declared as:

```
extern int main ( int argc , char *argv[ ] , char* envp[ ] );
```

where

argc is a non-negative argument count

argv is an array of argument strings, with `argv[argc] == 0`

envp is an array of environment strings, terminated by a null pointer.

When `main()` returns its value is passed to `exit()` and if that has been over-ridden and returns, `_exit()` (which must be immune to user interposition).

The initial state of the process stack, i.e. when `_start` is called is shown in table 2.11.

Table 2.11: Initial Process Stack

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 fourbyte
Auxiliary vector entries ...		2 fourbytes each
0		fourbyte
Environment pointers ...		1 fourbyte each
0	`4+4*argc+%esp`	fourbyte
Argument pointers	`4+%esp`	argc fourbytes
Argument count	`%esp`	fourbyte
Undefined	Low Addresses	

Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block and they need not be compactly allocated.

Only the registers listed below have specified values at process entry:

%ebp The content of this register is unspecified at process initialization time, but the user code should mark the deepest stack frame by setting the frame pointer to zero.

%esp The stack pointer holds the address of the byte with lowest address which is part of the stack. It is guaranteed to be 16-byte aligned at process entry.

%edx a function pointer that the application should register with `atexit` (BA_OS).

It is unspecified whether the data and stack segments are initially mapped with execute permissions or not. Applications which need to execute code on the stack or data segments should take proper precautions, e.g., by calling `mprotect()`.

2.3.2 Thread State

New threads inherit the floating-point state of the parent thread and the state is private to the thread thereafter.

2.3.3 Auxiliary Vector

The auxiliary vector is an array of the following structures (ref. table 2.12), interpreted according to the `a_type` member.

Table 2.12: `auxv_t` Type Definition

```
typedef struct
{
    int a_type;
    union {
        long a_val;
        void *a_ptr;
        void (*a_fnc)();
    } a_un;
} auxv_t;
```

The Intel386 ABI uses the auxiliary vector types defined in table 2.13.

Table 2.13: Auxiliary Vector Types

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_NOTELF	10	a_val
AT_UID	11	a_val
AT_EUID	12	a_val
AT_GID	13	a_val
AT_EGID	14	a_val
AT_PLATFORM	15	a_ptr
AT_HWCAP	16	a_val
AT_CLKTCK	17	a_val
AT_SECURE	23	a_val
AT_BASE_PLATFORM	24	a_ptr
AT_RANDOM	25	a_ptr
AT_HWCAP2	26	a_val
AT_EXECFN	31	a_ptr

AT_NULL The auxiliary vector has no fixed length; instead its last entry's `a_type` member has this value.

AT_IGNORE This type indicates the entry has no meaning. The corresponding value of `a_un` is undefined.

AT_EXECFD At process creation the system may pass control to an interpreter program. When this happens, the system places either an entry of type `AT_EXECFD` or one of type `AT_PHDR` in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR The system may create the memory image of the application program before passing control to the interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image.

AT_PHENT The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ If present, this entry's a_val member gives the system page size, in bytes.

AT_BASE The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See "Program Header" in the System V ABI for more information about the base address.

AT_FLAGS If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero.

AT_ENTRY The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

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.

AT_PLATFORM The a_ptr member of this entry points to a string containing the platform name.

AT_HWCAP The `a_val` member of this entry contains an bitmask of CPU features. It mask to the value returned by CPUID 1.EDX.

AT_CLKTCK The `a_val` member of this entry contains the frequency at which times() increments.

AT_SECURE The `a_val` member of this entry contains one if the program is in secure mode (for example started with suid). Otherwise zero.

AT_BASE_PLATFORM The `a_ptr` member of this entry points to a string identifying the base architecture platform (which may be different from the platform).

AT_RANDOM The `a_ptr` member of this entry points to 16 securely generated random bytes.

AT_HWCAP2 The `a_val` member of this entry contains the extended hardware feature mask. Currently it is 0, but may contain additional feature bits in the future.

AT_EXECFN The `a_ptr` member of this entry is a pointer to the file name of the executed program.

2.4 DWARF Definition

This section[8] defines the Debug With Arbitrary Record Format (DWARF) debugging format for the Intel386 processor family. The Intel386 ABI does not define a debug format. However, all systems that do implement DWARF on Intel386 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 3, January, 2006, Free Standards Group, DWARF Standard Committee. It's available at: `http://www.dwarfstd.org/`.

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

2.4.1 DWARF Release Number

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the Intel386 registers. In addition, the DWARF Version 3 specification requires processor-specific address class codes to be defined.

2.4.2 DWARF Register Number Mapping

Table 2.14[9] outlines the register number mapping for the Intel386 processor family.[10]

2.5 Stack Unwind Algorithm

The stack frames are not self descriptive and where stack unwinding is desirable (such as for exception handling) additional unwind information needs to be generated. The information is stored in an allocatable section `.eh_frame` whose format is identical to `.debug_frame` defined by the DWARF debug information standard, see *DWARF Debugging Information Format*, with the following extensions:

Position independence In order to avoid load time relocations for position independent code, the FDE CIE offset pointer should be stored relative to the start of CIE table entry. Frames using this extension of the DWARF standard must set the CIE identifier tag to 1.

Outgoing arguments area delta To maintain the size of the temporarily allocated outgoing arguments area present on the end of the stack (when using `push` instructions), operation `GNU_ARGS_SIZE` (`0x2e`) can be used. This operation takes a single `uleb128` argument specifying the current size. This information is used to adjust the stack frame when jumping into the exception handler of the function after unwinding the stack frame. Additionally the CIE Augmentation shall contain an exact specification of the encoding used. It is recommended to use a PC relative encoding whenever possible and adjust the size according to the code model used.

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

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

Table 2.14: DWARF Register Number Mapping

Register Name	Number	Abbreviation
General Purpose Register EAX	0	%eax
General Purpose Register ECX	1	%ecx
General Purpose Register EDX	2	%edx
General Purpose Register EBX	3	%ebx
Stack Pointer Register ESP	4	%esp
Frame Pointer Register EBP	5	%ebp
General Purpose Register ESI	6	%esi
General Purpose Register EDI	7	%edi
Return Address RA	8	
Flag Register	9	%EFLAGS
Reserved	10	
Floating Point Registers 0–7	11-18	%st0–%st7
Reserved	19-20	
Vector Registers 0–7	21-28	%xmm0–%xmm7
MMX Registers 0–7	29-36	%mm0–%mm7
Media Control and Status	39	%mxcsr
Segment Register ES	40	%es
Segment Register CS	41	%cs
Segment Register SS	42	%ss
Segment Register DS	43	%ds
Segment Register FS	44	%fs
Segment Register GS	45	%gs
Reserved	46-47	
Task Register	48	%tr
LDT Register	49	%ldtr
Reserved	50-92	

Table 2.15: Pointer Encoding Specification Byte

Mask	Meaning
0x1	Values are stored as `uleb128` or `sleb128` type (according to flag 0x8)
0x2	Values are stored as 2 bytes wide integers (`udata2` or `sdata2`)
0x3	Values are stored as 4 bytes wide integers (`udata4` or `sdata4`)
0x4	Values are stored as 8 bytes wide integers (`udata8` or `sdata8`)
0x8	Values are signed
0x10	Values are PC relative
0x20	Values are text section relative
0x30	Values are data section relative
0x40	Values are relative to the start of function

CIE Augmentations: The augmentation field is formated according to the augmentation field formating string stored in the CIE header.

The string may contain the following characters:

z Indicates that a `uleb128` is present determining the size of the augmentation section.

L Indicates the encoding (and thus presence) of an LSDA pointer in the FDE augmentation.

The data filed consist of single byte specifying the way pointers are encoded. It is a mask of the values specified by the table 2.15.

The default DWARF3 pointer encoding (direct 4-byte absolute pointers) is represented by value 0.

R Indicates a non-default pointer encoding for FDE code pointers. The formating is represented by a single byte in the same way as in the 'L' command.

P Indicates the presence and an encoding of a language personality routine in the CIE augmentation. The encoding is represented by a single byte in the same way as in the 'L' command followed by a pointer to the personality function encoded by the specified encoding.

When the augmentation is present, the first command must always be '`z`' to allow easy skipping of the information.

In order to simplify manipulation of the unwind tables, the runtime library provide higher level API to stack unwinding mechanism, for details see section 4.1.

Chapter 3

Object Files

3.1 Sections

3.1.1 Special Sections

Table 3.1: Special sections

Name	Type	Attributes
.eh_frame	SHT_PROGBITS	SHF_ALLOC

.eh_frame This section holds the unwind function table. The contents are described in Section 3.1.2 of this document.

3.1.2 EH_FRAME sections

The call frame information needed for unwinding the stack is output into one section named .eh_frame. An .eh_frame section consists of one or more subsections. Each subsection contains a CIE (Common Information Entry) followed by varying number of FDEs (Frame Descriptor Entry). A FDE corresponds to an explicit or compiler generated function in a compilation unit, all FDEs can access the CIE that begins their subsection for data. If the code for a function is not one contiguous block, there will be a separate FDE for each contiguous sub-piece.

If an object file contains C++ template instantiations there shall be a separate CIE immediately preceding each FDE corresponding to an instantiation.

Using the preferred encoding specified below, the `.eh_frame` section can be entirely resolved at link time and thus can become part of the text segment.

`EH_PE` encoding below refers to the pointer encoding as specified in the enhanced LSB Chapter 7 for `Eh_Frame_Hdr`.

Table 3.2: Common Information Entry (CIE)

Field	Length (byte)	Description
Length	4	Length of the CIE (not including this 4-byte field)
CIE id	4	Value 0 for `.eh_frame` (used to distinguish CIEs and FDEs when scanning the section)
Version	1	Value One (1)
CIE Augmentation String	string	Null-terminated string with legal values being "" or 'z' optionally followed by single occurrances of 'P', 'L', or 'R' in any order. The presence of character(s) in the string dictates the content of field 8, the Augmentation Section. Each character has one or two associated operands in the AS (see table 3.3 for which ones). Operand order depends on position in the string ('z' must be first).
Code Align Factor	uleb128	To be multiplied with the "Advance Location" instructions in the Call Frame Instructions
Data Align Factor	sleb128	To be multiplied with all offsets in the Call Frame Instructions
Ret Address Reg	1/uleb128	A "virtual" register representation of the return address. In Dwarf V2, this is a byte, otherwise it is uleb128. It is a byte in gcc 3.3.x
Optional CIE Augmentation Section	varying	Present if Augmentation String in Augmentation Section field 4 is not 0. See table 3.3 for the content.
Optional Call Frame Instructions	varying	

Table 3.3: CIE Augmentation Section Content

Char	Operands	Length (byte)	Description
z	size	uleb128	Length of the remainder of the Augmentation Section
P	personality_enc	1	Encoding specifier - preferred value is a pc-relative, signed 4-byte
	personality routine	(encoded)	Encoded pointer to personality routine (actually to the PLT entry for the personality routine)
R	code_enc	1	Non-default encoding for the code-pointers (FDE members `initial_location` and `address_range` and the operand for `DW_CFA_set_loc`) - preferred value is pc-relative, signed 4-byte
L	lsda_enc	1	FDE augmentation bodies may contain LSDA pointers. If so they are encoded as specified here - preferred value is pc-relative, signed 4-byte possibly indirect thru a GOT entry

Table 3.4: Frame Descriptor Entry (FDE)

Field	Length (byte)	Description
Length	4	Length of the FDE (not including this 4-byte field)
CIE pointer	4	Distance from this field to the nearest preceding CIE (the value is subtracted from the current address). This value can never be zero and thus can be used to distinguish CIE's and FDE's when scanning the `.eh_frame` section
Initial Location	var	Reference to the function code corresponding to this FDE. If 'R' is missing from the CIE Augmentation String, the field is an 8-byte absolute pointer. Otherwise, the corresponding `EH_PE` encoding in the CIE Augmentation Section is used to interpret the reference
Address Range	var	Size of the function code corresponding to this FDE. If 'R' is missing from the CIE Augmentation String, the field is an 8-byte unsigned number. Otherwise, the size is determined by the corresponding `EH_PE` encoding in the CIE Augmentation Section (the value is always absolute)
Optional FDE Augmentation Section	var	Present if CIE Augmentation String is non-empty. See table 3.5 for the content.
Optional Call Frame Instructions	var	

Table 3.5: FDE Augmentation Section Content

Char	Operands	Length (byte)	Description
z	length	uleb128	Length of the remainder of the Augmentation Section
L	LSDA	var	LSDA pointer, encoded in the format specified by the corresponding operand in the CIE's augmentation body. (only present if length > 0).

The existence and size of the optional call frame instruction area must be computed based on the overall size and the offset reached while scanning the preceding fields of the CIE or FDE.

The overall size of a `.eh_frame` section is given in the ELF section header. The only way to determine the number of entries is to scan the section until the end, counting entries as they are encountered.

3.2 Symbol Table

The `STT_GNU_IFUNC` [1] symbol type is optional. It is the same as `STT_FUNC` except that it always points to a function or piece of executable code which takes no arguments and returns a function pointer. If an `STT_GNU_IFUNC` symbol is referred to by a relocation, then evaluation of that relocation is delayed until load-time. The value used in the relocation is the function pointer returned by an invocation of the `STT_GNU_IFUNC` symbol.

The purpose of the `STT_GNU_IFUNC` symbol type is to allow the run-time to select between multiple versions of the implementation of a specific function. The selection made in general will take the currently available hardware into account and select the most appropriate version.

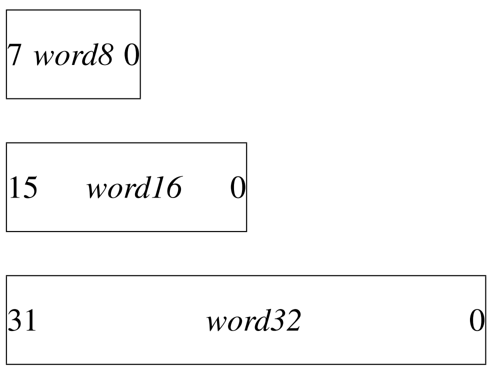
[1] It is specified in **ifunc.txt** at `http://sites.google.com/site/x32abi/documents`

3.3 Relocation

3.3.1 Relocation Types

Figure 3.3.1 shows the allowed relocatable fields.

Figure 3.1: Relocatable Fields



word8	This specifies a 8-bit field occupying 1 byte.
word16	This specifies a 16-bit field occupying 2 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the Intel386 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 Intel386 architecture.

The following notations are used for specifying relocations in table 3.6:

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

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

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

GOT Represents the address of the global offset table.

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

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

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

Z Represents the size of the symbol whose index resides in the relocation entry.

Table 3.6: Relocation Types

Name	Value	Field	Calculation
R_386_NONE	0	none	none
R_386_32	1	*word32*	S + A
R_386_PC32	2	*word32*	S + A - P
R_386_GOT32	3	*word32*	G + A - GOT
R_386_PLT32	4	*word32*	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	*word32*	S
R_386_JUMP_SLOT	7	*word32*	S
R_386_RELATIVE	8	*word32*	B + A
R_386_GOTOFF †	9	*word32*	S + A - GOT
R_386_GOTPC	10	*word32*	GOT + A - P
R_386_TLS_TPOFF	14	*word32*	
R_386_TLS_IE	15	*word32*	
R_386_TLS_GOTIE	16	*word32*	
R_386_TLS_LE	17	*word32*	
R_386_TLS_GD	18	*word32*	
R_386_TLS_LDM	19	*word32*	
R_386_16	20	*word16*	S + A
R_386_PC16	21	*word16*	S + A - P
R_386_8	22	*word8*	S + A
R_386_PC8	23	*word8*	S + A - P
R_386_TLS_GD_32	24	*word32*	
R_386_TLS_GD_PUSH	25	*word32*	
R_386_TLS_GD_CALL	26	*word32*	
R_386_TLS_GD_POP	27	*word32*	
R_386_TLS_LDM_32	28	*word32*	
R_386_TLS_LDM_PUSH	29	*word32*	
R_386_TLS_LDM_CALL	30	*word32*	
R_386_TLS_LDM_POP	31	*word32*	
R_386_TLS_LDO_32	32	*word32*	
R_386_TLS_IE_32	33	*word32*	
R_386_TLS_LE_32	34	*word32*	
R_386_TLS_DTPMOD32	35	*word32*	
R_386_TLS_DTPOFF32	36	*word32*	
R_386_TLS_TPOFF32	37	*word32*	
R_386_SIZE32	38	*word32*	Z + A
R_386_TLS_GOTDESC	39	*word32*	
R_386_TLS_DESC_CALL	40	none	none
R_386_TLS_DESC	41	*word32*	
R_386_IRELATIVE	42	*word32*	indirect (B + A)

A program or object file using R_386_8, R_386_16, R_386_PC16 or R_386_PC8 relocations is not conformant to this ABI, these relocations are only added for documentation purposes. The R_386_16, and R_386_8 relocations truncate the computed value to 16-bits and 8-bits respectively.

The relocations R_386_TLS_TPOFF, R_386_TLS_IE, R_386_TLS_GOTIE, R_386_TLS_LE, R_386_TLS_GD, R_386_TLS_LDM, R_386_TLS_GD_32, R_386_TLS_GD_PUSH, R_386_TLS_GD_CALL, R_386_TLS_GD_POP, R_386_TLS_LDM_32, R_386_TLS_LDM_PUSH, R_386_TLS_LDM_CALL, R_386_TLS_LDM_POP, R_386_TLS_LDO_32, R_386_TLS_IE_32, R_386_TLS_LE_32, R_386_TLS_DTPMOD32, R_386_TLS_DTPOFF32 and R_386_TLS_TPOFF32 are listed for completeness. They are part of the Thread-Local Storage ABI extensions and are documented in the document called "ELF Handling for Thread-Local Storage"[2]. The relocations R_386_TLS_GOTDESC, R_386_TLS_DESC_CALL and R_386_TLS_DESC are also used for Thread-Local Storage, but are not documented there as of this writing. A description can be found in the document "Thread-Local Storage Descriptors for IA32 and AMD64/EM64T"[3].

R_386_IRELATIVE is similar to R_386_RELATIVE except that the value used in this relocation is the program address returned by the function, which takes no arguments, at the address of the result of the corresponding R_386_RELATIVE relocation.

One use of the R_386_IRELATIVE relocation is to avoid name lookup for the locally defined STT_GNU_IFUNC symbols at load-time. Support for this relocation is optional, but is required for the STT_GNU_IFUNC symbols.

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

[3]This document is currently available via http://people.redhat.com/aoliva/writeups/TLS/RFC-TLSDESC-x86.txt

Chapter 4

Libraries

4.1 Unwind Library Interface

This section defines the Unwind Library interface[1], expected to be provided by any Intel386 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,
_Unwind_GetCFA
```

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

In addition, two data types 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.

4.1.1 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++)
- "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 `%eip` and register state, and then unwinding a frame to a new `%eip` 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 `%eip` and register state, and then unwinding a frame to a new `%eip` 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 *presumptive* exception handling (correcting the exceptional condition and resuming execution at the point where it was raised). While C++ does not support presumptive 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 re-throw 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 re-throwing the exception or throwing a new one in C++.

The API specified by the Intel386 psABI for implementing this framework is described in the following sections.

4.1.2 Data Structures

Reason Codes

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

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

The interpretations of these codes are described below.

Exception Header

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

```
typedef void (*_Unwind_Exception_Cleanup_Fn)
    (_Unwind_Reason_Code reason,
     struct _Unwind_Exception *exc);
struct _Unwind_Exception {
    uint64                          exception_class;
    _Unwind_Exception_Cleanup_Fn    exception_cleanup;
    uint32                          private_1;
    uint32                          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 GNUC), 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 re-throwing 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
```

4.1.3 Throwing an Exception

_Unwind_RaiseException

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

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

Possibilities are:

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

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

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

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

_Unwind_ForcedUnwind

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

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

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

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

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

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

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

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

Example: longjmp_unwind()

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

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

_Unwind_Resume

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

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

_Unwind_Resume should not be used to implement re-throwing. To the unwinding runtime, the catch code that re-throws was a handler, and the previous unwinding session was terminated before entering it. Re-throwing 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.

4.1.4 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.

4.1.5 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

```
uint32 _Unwind_GetGR
  (struct _Unwind_Context *context, int index);
```

This function returns the 32-bit value of the given general register. The register is identified by its index as given in table 2.14.

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

_Unwind_SetGR

```
void _Unwind_SetGR
  (struct _Unwind_Context *context,
   int index,
   uint32 new_value);
```

This function sets the 32-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 %eax and %edx should be used. These scratch registers are reserved for passing arguments between the personality routine and the landing pads.

_Unwind_GetIP

```
uint32 _Unwind_GetIP
  (struct _Unwind_Context *context);
```

This function returns the 32-bit value of the instruction pointer (IP).

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

_Unwind_SetIP

```
void _Unwind_SetIP
  (struct _Unwind_Context *context,
   uint32 new_value);
```

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

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

_Unwind_GetLanguageSpecificData

```
uint32 _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

```
uint32 _Unwind_GetRegionStart
  (struct _Unwind_Context *context);
```

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

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

_Unwind_GetCFA

```
uint32 _Unwind_GetCFA
  (struct _Unwind_Context *context);
```

This function returns the 32-bit Canonical Frame Address which is defined as the value of `%esp` at the call site in the previous frame. This value is guaranteed to be correct any time the context has been passed to a personality routine or a stop function.

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

Parameters

The personality routine parameters are as follows:

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

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

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

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

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

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

Personality Routine Actions

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

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

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

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

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

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

Transferring Control to a Landing Pad

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

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

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

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

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

The landing pad may receive other arguments, for instance a switch value indicating the type of the exception. Two scratch registers are reserved for this use (%eax and %edx).

Rules for Correct Inter-Language Operation

The following rules must be observed for correct operation between languages and/or run times 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++ run times).

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 re-throw 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 re-throw.

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

Chapter 5

Conventions

[1]

[1] This chapter is used to document some features special to the Intel386 ABI. The different sections might be moved to another place or removed completely.

5.1 C++

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

Index