Itanium C++ ABI: Exception Handling ($Revision: 1.22 $)

Contents

Introduction

In this document, we define the C++ exception handling ABI, at three levels:

I. the base ABI, interfaces common to all languages and implementations;
II. the C++ ABI, interfaces necessary for interoperability of C++ implementations; and
III. the specification of a particular runtime implementation.

This specification is based on the general model described roughly in the Itanium Software Conventions and Runtime Architecture Guide. However, the Level I (base ABI) specification here contradicts that document in some particulars, and is being proposed as a modification. That document describes a framework which can be used by an arbitrary implementation, with a complete definition of the stack unwind mechanism, but no significant constraints on the language-specific processing. In particular, it is not sufficient to guarantee that two object files compiled by different C++ compilers could interoperate, e.g. throwing an exception in one of them and catching it in the other.

In Section I below, we will elaborate missing details from this base document, largely in the form of specifying the APIs to be used in accessing the language-independent stack unwind facilities, namely the unwind descriptor tables and the personality routines. This specification should be implemented by any Itanium psABI-compliant system.

In Section II below, we will specify the API of the C++ exception handling facilities, specifically for raising and catching exceptions. These APIs should be implemented by any C++ system compliant with the Itanium C++ ABI. *Note that the level II and level III specifications are not completed at this time.*

Definitions

The descriptions below make use of the following definitions:

landing pad
:
A section of user code intended to catch, or otherwise clean up after, an exception. It gains control from the exception runtime via the personality routine, and after doing the appropriate processing either merges into the normal user code or returns to the runtime by resuming or raising a new exception.

Base Documents

This document is based on the **C++ ABI for Itanium**, and the Level II specification below is considered to be part of that document (Chapter 4). See Base Documents in that document for further references.

Level I. Base ABI

This section defines the Unwind Library interface, expected to be provided by any Itanium psABI-compliant system. This is the interface on which the C++ ABI exception-handling facilities are built. We assume as a basis the unwind descriptor tables described in the base Itanium Software Conventions & Runtime Architecture Guide. Our focus here will on the APIs for accessing those structures.

It is intended that nothing in this section be specific to C++, though some parts are clearly intended to support C++ features.

The unwinding 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 routines. 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.

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 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 accomodate 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 PC and register state, and then unwinding a frame to a new PC 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 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 PC and register state, and then unwinding a frame to a new PC 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 Itanium psABI for implementing this framework is described in the following sections.

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 it will be prefixed by a header understood by the unwind interface, defined as follows:

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

struct _Unwind_Exception {
```

```
        uint64                          exception_class;
        _Unwind_Exception_Cleanup_Fn exception_cleanup;
        uint64                          private_1;
        uint64                          private_2;
};
```

An **`_Unwind_Exception`** object must be double-word 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 HP\0\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 behaviour.
- **`_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
```

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 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.

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.

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

```
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: 0 to 31 are for the fixed registers, and 32 to 127 are for the stacked registers.

During the two phases of unwinding, only GR1 has a guaranteed value, which is the Global Pointer (GP) of the frame referenced by the unwind context. If the register has its NAT bit set, the behaviour is unspecified.

_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 NAT bit of the given register is reset.

The behaviour 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 GR15, GR16, GR17, GR18 should be used. These scratch registers are reserved for passing arguments between the personality routine and the landing pads.

`_Unwind_GetIP`

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

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

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

`_Unwind_SetIP`

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

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

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

_Unwind_GetLanguageSpecificData

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

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


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

_Unwind_GetRegionStart

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

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

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

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.

1.6.1 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 HP\0\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.

1.6.2 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;
static const _Unwind_Action _UA_SEARCH_PHASE = 1;
static const _Unwind_Action _UA_CLEANUP_PHASE = 2;
static const _Unwind_Action _UA_HANDLER_FRAME = 4;
static 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.

1.6.3 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 GR15, GR16, GR17 and GR18 (see below). Except for those exceptions, scratch (or caller-saved) registers are not preserved, and their contents are undefined on transfer. The accessibility of registers in the frame will be restored to that at the point of call, i.e. the same logical registers will be accessible, but their mappings to physical registers may change. Further, the state of stacked registers beyond the current frame is unspecified, i.e. they may be either in physical registers or on the register stack.

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 (GR15, GR16, GR17 and GR18.)

1.6.4 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.

Level II: C++ ABI

2.1 Introduction

The second level of specification is the minimum required to allow interoperability in the sense described above. This level requires agreement on:

- Standard runtime initialization, e.g. pre-allocation of space for out-of-memory exceptions.
- The layout of the exception object created by a throw and processed by a catch clause.
- When and how the exception object is allocated and destroyed.
- The API of the personality routine, i.e. the parameters passed to it, the logical actions it performs, and any results it returns (either function results to indicate success, failure, or continue, or changes in global or exception object state), for both the phase 1 handler search and the phase 2 cleanup/unwind.
- How control is ultimately transferred back to the user program at a catch clause or other resumption point. That is, will the last personality routine transfer control directly to the user code resumption point, or will it return information to the runtime allowing the latter to do so?
- Multithreading behavior.

2.2 Data Structures

2.2.1 C++ Exception Objects

A complete C++ exception object consists of a header, which is a wrapper around an unwind object header with additional C++ specific information, followed by the thrown C++ exception object itself. The structure of the header is as follows:

```
struct __cxa_exception {
  std::type_info *        exceptionType;
  void (*exceptionDestructor) (void *);
  unexpected_handler      unexpectedHandler;
```

```
    terminate_handler       terminateHandler;
    __cxa_exception *       nextException;

    int                     handlerCount;
    int                     handlerSwitchValue;
    const char *            actionRecord;
    const char *            languageSpecificData;
    void *                  catchTemp;
    void *                  adjustedPtr;

    _Unwind_Exception       unwindHeader;
  };
```

The fields in the exception object are as follows:

- The **`exceptionType`** field encodes the type of the thrown exception. The **`exceptionDestructor`** field contains a function pointer to a destructor for the type being thrown, and may be NULL. These pointers must be stored in the exception object since non-polymorphic and built-in types can be thrown.

- The fields **`unexpectedHandler`** and **`terminateHandler`** contain pointers to the **`unexpected`** and **`terminate`** handlers at the point where the exception is thrown. *The **ISO C++ Final Draft International Standard** [lib.unexpected] (18.6.2.4) states that the handlers to be used are those active immediately after evaluating the throw argument. If destructors change the active handlers during unwinding, the new values are not used until unwinding is complete.*

- The **`nextException`** field is used to create a linked list of exceptions (per thread).

- The **`handlerCount`** field contains a count of how many handlers have caught this exception object. It is also used to determine exception life-time (see Section ??? [was 11.12]).

- The **`handlerSwitchValue`**, **`actionRecord`**, **`languageSpecificData`**, **`catchTemp`**, and **`adjustedPtr`** fields cache information that is best computed during pass 1, but useful during pass 2. By storing this information in the exception object, the cleanup phase can avoid re-examining action records. These fields are reserved for use

of the personality routine for the stack frame containing the handler to be invoked.

- The **`unwindHeader`** structure is used to allow correct operation of exception in the presence of multiple languages or multiple runtimes for the same language. The **`_Unwind_Exception`** type is described in Section 1.2.

By convention, a **`__cxa_exception`** pointer points at the C++ object representing the exception being thrown, immediately following the header. The header structure is accessed at a negative offset from the **`__cxa_exception`** pointer. *This layout allows consistent treatment of exception objects from different languages (or different implementations of the same language), and allows future extensions of the header structure while maintaining binary compatibility.*

Version information is not required, since the general unwind library framework specifies an exception class identifier, which will change should the layout of the exception object change significantly.

2.2.2 Caught Exception Stack

Each thread in a C++ program has access to an object of the following class:

```
struct __cxa_eh_globals {
  __cxa_exception *       caughtExceptions;
  unsigned int            uncaughtExceptions;
};
```

The fields of this structure are defined as follows:

- The **`caughtExceptions`** field is a list of the active exceptions, organized as a stack with the most recent first, linked through the **`nextException`** field of the exception header.

- The **`uncaughtExceptions`** field is a count of uncaught exceptions, for use by the C++ library **`uncaught_exceptions()`** routine.

This information is maintained on a per-thread basis. Thus, **`caughtExceptions`** is a list of exceptions thrown and caught by the current thread, and **`uncaughtExceptions`** is a count of exceptions thrown and not yet caught by the current thread. (This includes rethrown exceptions, which may still have active handlers, but are not considered caught.)

The **`__cxa_eh_globals`** for the current thread can be obtained by using either of the APIs:

- **`__cxa_eh_globals *__cxa_get_globals(void)`**:
  Return a pointer to the **`__cxa_eh_globals`** structure for the current thread, initializing it if necessary.

- **`__cxa_eh_globals *__cxa_get_globals_fast(void)`**:
  Return a pointer to the **`__cxa_eh_globals`** structure for the current thread, assuming that at least one prior call to **`__cxa_get_globals`** has been made from the current thread.

2.3 Standard Runtime Initialization

2.4 Throwing an Exception

This section specifies the process by which the C++ generated code and runtime library throw an exception, transferring control to the unwind library for handling.

2.4.1 Overview of Throw Processing

In broad outline, a possible implementation of the processing necessary to throw an exception includes the following steps:

- *Call **`__cxa_allocate_exception`** to create an exception object (see Section 2.4.2).*

- *Evaluate the thrown expression, and copy it into the buffer returned by **`__cxa_allocate_exception`**, possibly using a copy constructor. If evaluation of the thrown expression exits by throwing an exception, that*

exception will propagate instead of the expression itself. Cleanup code must ensure that ***__cxa_free_exception*** *is called on the just allocated exception object. (If the copy constructor itself exits by throwing an exception,* ***terminate()*** *is called.)*

- *Call* ***__cxa_throw*** *to pass the exception to the runtime library (see Section 2.4.3).* ***__cxa_throw*** *never returns.*

Based on this outline, throwing an object X as in:

```
throw X;
```

will produce code approximating the template:

```
// Allocate -- never throws:
temp1 = __cxa_allocate_exception(sizeof(X));

// Construct the exception object:
#if COPY_ELISION
  [evaluate X into temp1]
#else
  [evaluate X into temp2]
  copy-constructor(temp1, temp2)
  // Landing Pad L1 if this throws
#endif

// Pass the exception object to unwind library:
__cxa_throw(temp1, type_info<X>, destructor<X>); // Never returns

// Landing pad for copy constructor:
L1: __cxa_free_exception(temp1) // never throws
```

The following sections will describe the ABI facilities available to implement these steps, or possible variants, in more detail. One possible variant is to evaluate the thrown expression before allocating the exception object and always copy the value.

2.4.2 Allocating the Exception Object

Storage is needed for exceptions being thrown. This storage must persist while stack is being unwound, since it will be used by the handler, and must be thread-safe. Exception object storage will therefore normally be allocated in the heap, although implementations may provide an emergency buffer to support throwing **`bad_alloc`** exceptions under low memory conditions (see Section 3.3.1).

Memory will be allocated by the **`__cxa_allocate_exception`** runtime library routine. This routine is passed the size of the exception object to be thrown (not including the size of the **`__cxa_exception`** header), and returns a pointer to the temporary space for the exception object. It will allocate the exception memory on the heap if possible. If heap allocation fails, an implementation may use other backup mechanisms (see Section 3.4.1).

If **`__cxa_allocate_exception`** cannot allocate an exception object under these constraints, it calls **`terminate()`**.

```
void *__cxa_allocate_exception(size_t thrown_size);
```

Once space is allocated, a throw expression must initialize the exception object with the thrown value as specified by the C++ Standard.

The temporary space will be freed by **`__cxa_free_exception`** (see Section 2.5.5), which is passed the address returned by the previous **`__cxa_allocate_exception`**.

```
void __cxa_free_exception(void *thrown_exception);
```

These routines are thread-safe (in a multi-threading environment), and may block threads after the maximum number of threads allowed to use the emergency buffer has been reached.

2.4.3 Throwing the Exception Object

After constructing the exception object with the throw argument value, the generated code calls the **`__cxa_throw`** runtime library routine. This routine never returns.

```
void __cxa_throw (void *thrown_exception, std::type_info *tinfo, void (*dest) (void *) );
```

The arguments are:

- The address of the thrown exception object (which points to the throw value, after the header, as specified above).
- A **`std::type_info`** pointer, giving the static type of the throw argument as a **`std::type_info`** pointer, used for matching potential catch sites to the thrown exception.
- A destructor pointer to be used eventually to destroy the object.

The **`__cxa_throw`** routine will do the following:

- Obtain the **`__cxa_exception`** header from the thrown exception object address, which can be computed as follows:

```
__cxa_exception *header = ((__cxa_exception *) thrown_exception - 1);
```

- Save the current **`unexpected_handler`** and **`terminate_handler`** in the **`__cxa_exception`** header.
- Save the **`tinfo`** and **`dest`** arguments in the **`__cxa_exception`** header.
- Set the **`exception_class`** field in the unwind header. This is a 64-bit value representing the ASCII string "XXXXC++\0", where "XXXX" is a vendor-dependent string. That is, for implementations conforming to this ABI, the low-order 4 bytes of this 64-bit value will be "C++\0".
- Increment the **`uncaught_exception`** flag.
- Call **`_Unwind_RaiseException`** in the system unwind library, Its argument is the pointer to the thrown exception, which **`__cxa_throw`** itself received as an argument.

__Unwind_RaiseException begins the process of stack unwinding, described in Section 2.5. In special cases, such as an inability to find a handler, **_Unwind_RaiseException** may return. In that case, **__cxa_throw** will call terminate, assuming that there was no handler for the exception.

2.5 Catching an Exception

2.5.1 Overview of Catch Processing

See Section 3.5 below for an overview of a possible implementation of catch processing. This section specifies the interfaces which must be supported by ABI-compliant exception handling runtime libraries.

2.5.2 The Personality Routine

The personality routine is the function in the C++ runtime library which serves as an interface between the system unwind library and the C++ specific semantics. Its interface is defined by Section 1.6:

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

The interface between the unwind library and the personality routine is specified as part of the Level I interface in Section 1.6. Furthermore, the personality routine is tailored by the implementation to the handlers for the frame in which it works, and to the exception tables for that frame. Therefore, only general behavior is specified here.

Although other approaches are possible, this ABI requires a two-phase unwind process. During the first phase, i.e. with **actions** including the bit **_UA_SEARCH_PHASE**, the personality routine should do nothing to update state, simply searching for a handler and returning **_URC_HANDLER_FOUND** when it finds one. During the second phase, i.e. with **actions** including the bit **_UA_CLEANUP_PHASE**, the personality routine may perform cleanup actions at intermediate

frames, and must transfer to the handler found when **`actions`** includes the bit **`_UA_HANDLER_FRAME`**, which it does by setting up the context and returning **`_URC_INSTALL_CONTEXT`**.

If the exception is not a C++ exception, the C++ personality routine must not catch it, that is it should return _URC_CONTINUE_UNWIND in both phases, after performing any required cleanup actions in the second phase. See the specification in Section 1.6.4.

The Level I specification requires that the personality routine transfer control to a landing pad via the unwind library, enabling the latter to do any final cleanup. It does so by modifying the context record for the current frame, and letting the unwind library transfer control:

- Using **`_Unwind_SetIP`** to set the PC of the current stack frame to the address of the landing pad.
- Using **`_Unwind_SetGR`** to set parameters to the landing pad in the general registers of the current stack frame. Although this is again an unspecified implementation detail, it is suggested that the exception object address be passed in GR15, the switch value in GR16, and any other parameters in GR17 and GR18.
- Once the frame is set, returning **`_URC_INSTALL_CONTEXT`** to the unwind library, which does any cleanup required, installs the context, and transfers control to the landing pad.

Note that any cleanup activity may be implemented as a landing pad that performs only cleanup tasks (no handlers), and calls **`_Unwind_Resume`** when done. In such cases, the personality routine should treat the cleanup landing pad as a handler.

The personality routine works only within the current frame; that is, it returns control to the unwind library for any processing required beyond this frame.

2.5.3 Exception Handlers

For purposes of this ABI, several things are considered *exception handlers*:

- A normal C++ handler, i.e. a catch clause.
- An **`unexpected()`** call, due to a violated exception specification.
- A **`terminate()`** call due to a throw.

Most of the behavior of a handler is implementation-defined. However, in order to allow maintenance of the exception stack, all handlers must make the following calls.

Upon entry, a handler must call:

```
void *__cxa_get_exception_ptr ( void *exceptionObject );
```

This routine returns the adjusted pointer to the exception object. (The adjusted pointer is typically computed by the personality routine during phase 1 and saved in the exception object.)

~~Upon entry,~~ Following initialization of the catch parameter, a handler must call:

```
void *__cxa_begin_catch ( void *exceptionObject );
```

This routine:

- Increment's the exception's handler count.
- Places the exception on the stack of currently-caught exceptions if it is not already there, linking the exception to the previous top of the stack.
- Decrements the uncaught_exception count.
- Returns the adjusted pointer to the exception object.

If the initialization of the catch parameter is trivial (e,g., there is no formal catch parameter, or the parameter has no copy constructor), the calls to **`__cxa_get_exception_ptr()`** and **`__cxa_begin_catch()`** may be combined into a single call to **`__cxa_begin_catch()`**.

When the personality routine encounters a termination condition, it will call **`__cxa_begin_catch()`** to mark the exception as handled and then call **`terminate()`**, which shall not return to its caller.

Upon exit for any reason, a handler must call:

```
void __cxa_end_catch ();
```

This routine:

- Locates the most recently caught exception and decrements its handler count.
- Removes the exception from the caught exception stack, if the handler count goes to zero.
- Destroys the exception if the handler count goes to zero, and the exception was not re-thrown by **`throw`**.

*Collaboration between **`__cxa_rethrow()`** and **`__cxa_end_catch()`** is necessary to handle the last point. Though implementation-defined, one possibility is for **`__cxa_rethrow()`** to set a flag in the **`handlerCount`** member of the exception header to mark an exception being rethrown.*

If a landing pad is going to resume unwinding, e.g. because

- it contains no handlers, just cleanup actions;
- none of its catch handlers matches the exception; or
- the catch handler re-throws the exception,

then it shall do any required cleanup for the current frame before calling **`_Unwind_Resume`** to resume unwinding.

A handler for an arbitrary exception, including a **`terminate_handler`** or **`unwind_handler`**, has no way in standard C++ of determining the type of the exception without resorting to a complete enumeration, which is impractical at best. Since we use **`type_info`** for EH type matching, a user can access this information by calling:

```
std::type_info *__cxa_current_exception_type ();
```

which returns the type of the first caught exception, or null if there are no caught exceptions. This routine is optional; a conforming ABI implementation need not provide it. However, if it is provided, it shall have the behavior specified here.

2.5.4 Rethrowing Exceptions

Rethrowing an exception is possible any time an exception is being handled. Most commonly, that means within a catch clause, but it is also possible to rethrow within an **`unexpected()`** or **`terminate()`** handler.

A catch handler rethrows the exception on top of the caughtExceptions stack by calling:

```
void __cxa_rethrow ();
```

This routine marks the exception object on top of the caughtExceptions stack (in an implementation-defined way) as being rethrown. If the caughtExceptions stack is empty, it calls **`terminate()`** (see [C++FDIS] [except.throw], 15.1.8). It then returns to the handler that called it, which must call **`__cxa_end_catch()`**, perform any necessary cleanup, and finally call **`_Unwind_Resume()`** to continue unwinding.

2.5.5 Finishing and Destroying the Exception

An exception is considered *handled*:

- Immediately after initializing the parameter of the corresponding catch clause (or upon entry to a **`catch(...)`** clause).
- Upon entering **`unexpected()`** or **`terminate()`** due to a throw.

An exception is considered *finished*:

- When the corresponding catch clause exits (normally, by another throw, or by rethrow).
- When **`unexpected()`** exits (by throwing), after being entered due to a throw.

Because an exception can be rethrown and caught within a handler, there can be more than one handler active for an exception. The exception is destroyed when the last (outermost) handler exits by any means other than rethrow. The destruction occurs immediately after destruction of the catch clause parameter, if any.

This lifetime management is performed by the **__cxa_begin_catch** and **__cxa_end_catch** runtime functions, which keep track of what handlers exist for which exceptions. When **__cxa_end_catch** detects that an exception is no longer being thrown or handled, it destroys the exception and frees the memory allocated for it.


Managing an exception lifetime occurs at runtime, because it is impossible to determine statically when it ends. For example:

```
try { throw X(); }
catch (X x)
{
  try { throw; }
  catch(...) {
    if (case1)
      throw;
    else if (case2)
      throw Y();
  }
  if (case3) throw;
}
```

In this example, the lifetime of the thrown exception created in the outermost try block depends on the conditions:

- *In case case1, the initial exception is thrown again, using the same temporary. It survives the outer catch.*
- *In case case2, another exception is being thrown, causing the initial exception to be destroyed as the exception propagates from the outer catch.*
- *In case case3, the exception lives outside both the innermost and outermost catch block. (The effect is the same as case1.)*
- *Otherwise, the exception is finished at the end of the outermost catch block.*

The complete exception object will be destroyed by calling its destructor and **__cxa_free_exception**, as described in Section 2.4.2. This is not normally called directly by the generated code, but rather by the **__cxa_end_catch** routine, which also removes the exception from the caught-exceptions stack.

2.6 Auxiliary Runtime APIs

The C++ ABI includes the APIs of a number of runtime routines to facilitate code generation for recurring situations that need not always produce inline code.

```
extern "C" void __cxa_bad_cast ();
```

Raise a bad_cast exception (lib.bad.cast, 18.5.2). Does not return (normally), and implementations may note this fact, e.g. with pragmas.

```
extern "C" void __cxa_bad_typeid ();
```

Raise a bad_typeid exception (lib.bad.typeid, 18.5.3). Does not return (normally), and implementations may note this fact, e.g. with pragmas.

Level III. Suggested Implementation

3.1 Introduction

The third level is a specification sufficient to allow all compliant C++ systems to share the relevant runtime implementation. It includes, in addition to the Level II specification:

- Format of the C++ language-specific unwind tables.
- APIs of the functions named __allocate_exception, __throw, and __free_exception (and likely others) by HP, or their equivalents.
- API of landing pad code, and of any other entries back into the user code.

- Definition of what HP calls the exception class value.

The vocal attendees at the meeting wish to achieve the third level, and we will attempt to do so. What follows is a subset of the required information; See the HP exception handling specification for elaboration at this time.

3.2 Data Structures

<To Be Supplied>

3.3 Standard Runtime Initialization

3.3.1 Exception Storage

The C++ runtime library shall allocate a static emergency buffer of at least 4K bytes per potential task, up to 64KB. This buffer shall be used only in cases where dynamic allocation of exception objects fails. It shall be allocated in 1KB chunks. At most 16 tasks may use the emergency buffer at any time, for at most 4 nested exceptions, with each exception object (including header) of size at most 1KB. Additional threads are blocked until one of the 16 de-allocates its emergency buffer storage.

The interface to the emergency buffer is implementation-defined, and used only by the exception library.

3.4 Throwing an Exception

This section specifies further implementation details of the process by which the C++ generated code and runtime library throw an exception, transferring control to the unwind library for handling.

3.4.1 Allocating the Exception Object

Memory for an exception object will be allocated by the **`__cxa_allocate_exception`** runtime library routine, with

general requirements as described in Section 2.4.2. If normal allocation fails, then it will attempt to allocate one of the emergency buffers, described in Section 3.3.1, under the following constraints:

- The exception object size, including headers, is under 1KB.
- The current thread does not already hold four buffers.
- There are fewer than 16 other threads holding buffers, or this thread will wait until one of the others releases its buffers before acquiring one.

3.5 Catching an Exception

3.5.1 Overview of Catch Processing

3.5.1.1 Unwinding the Stack

As specified by this ABI, stack unwinding itself is begun by calling ***`__Unwind_RaiseException()`****, and performed by the unwind library (see Section 1.1). We summarize it again here.*

The stack unwind library runs two passes on the stack, as follows:

- *Recover the program counter (PC) in the current stack frame.*
- *Using an unwind table, find information on how to handle exceptions that occur at that PC, and in particular, get the address of the personality routine for that address range. The unwind table is described in [SWCONV].*
- *Call the personality routine (see Section 2.5.2). that will determine if an appropriate handler is found at that level of the stack (in pass 1), and that will determine which particular handler to invoke from the landing pad (in pass 2), as well as the arguments to pass to the landing pad (see Section 3.5.2). The personality routine passes this information back to the unwind library.*

- *In the second phase, the unwind library jumps to the landing pad corresponding to the call (see Section 3.5.2) for each level of the stack being unwound. Landing pad parameters are set as indicated by the personality routine. The landing pad executes compensation code (generated by the back end) to restore the appropriate register and stack state.*

- *Some cleanup code generated by the front-end may then execute, corresponding to the exit of the **try** block. For instance, an automatic variable local to the try block would be destroyed here.*

- *The exception handler may select and execute user-defined code corresponding to C++ catch clauses and other handlers. The generated code may be similar to a switch statement, where the switch value is determined by the runtime based on the exception type (see Section 2.5.3 and Section 3.5.3), and passed in a landing pad argument.*

- *As soon as the runtime determines that the execution will go to a handler, the unwinding process is considered complete for the unwind library. A handler may still rethrow the current exception (see Section 2.5.4) or a different exception, but a new unwind process will occur in both cases. Otherwise, after the code in a handler has executed, execution resumes at the end of the try block which defines this handler.*

- *If none of the possible handlers matches the exception being thrown, the runtime selects a switch value that does not match any switch statement. In that case, control passes through all switch statements, and goes to additional cleanup code, which will call all destructors that need to be called for the current stack frame. For instance, an automatic variable local to the outer block of the function would be destroyed here. This means that the handler must loop through any try blocks enclosing the original one in the process of cleaning up the frame, trying the switch value in each one.*

- *At the end of this current cleanup code, control is transferred back to the unwind library, to unwind one more stack frame (see Section 2.5.3).*

3.5.1.2 Generated Catch Code

For purposes of illustration, suppose we have a call site in a try-catch block such as:

```
try { foo(); }
catch (TYPE1) { ... }
catch (TYPE2) { buz(); }
bar();
```

This might be translated as follows:

```
// In "Normal" area:
foo(); // Call Attributes: Landing Pad L1, Action Record A1
goto E1;
...
E1: // End Label
bar();

// In "Exception" area;
L1: // Landing Pad label
[Back-end generated □compensation□ code]
goto C1;

C1: // Cleanup label
[Front-end generated cleanup code, destructors, etc]
[corresponding to exit of try { } block]
goto S1;

S1: // Switch label
switch(SWITCH_VALUE_PAD_ARGUMENT)
{
    case 1: goto H1; // For TYPE1
    case 2: goto H2; // For TYPE2
    //...
    default: goto X1;
}

X1:
[Cleanup code corresponding to exit of scope]
```

```
[enclosing the try block]
_Unwind_Resume();

H1: // Handler label
adjusted_exception_ptr = __cxa_get_exception_ptr(exception);
[Initialize catch parameter]
__cxa_begin_catch(exception);
[User code]
goto R1;

H2:
adjusted_exception_ptr = __cxa_get_exception_ptr(exception);
[Initialize catch parameter]
__cxa_begin_catch(exception);
[User code]
buz(); // Call attributes: Landing pad L2, action record A2
goto R1;

R1: // Resume label:
__cxa_end_catch();
goto E1;

L2:
C2:
// Make sure we cleanup the current exception
__cxa_end_catch();

X2:
[Cleanup code corresponding to exit of scope]
[enclosing the try block]
_Unwind_Resume();
```

The various components of this example segment are discussed in Section 2.5 above (ABI requirements), and below.

3.5.2 The Personality Routine

Appendix R: Revisions

[050504] Add __cxa_get_exception_ptr, add return value to __cxa_begin_catch.

[011126] Add __cxa_current_exception_type.

[010731] Clarify linkage of constants.

[010315] Clarify exception object structure (2.2.1). Rename IA-64 to Itanium throughout.

[001110] Eliminate __cxa_throw_type_info in favor of storing its components separately in the exception header and passing them separately in __cxa_throw.

[001010] Correct phase2 error behavior of _Unwind_RaiseException and _Unwind_ForcedUnwind (1.3). Define __cxa_throw_type_info correctly (2.2.1). Todo: fix cross-references in moved material in Level III.

[000911] Add Jason's corrections. Remove section 2.4.3.

[000810] Add C++ exception data (2.2), begin initialization (2.3) and throwing exceptions (2.4).

[000707] Add auxiliary runtime APIs (2.6).

[000502] Begin integration of Level 2/3 specifications.

[000131] Integrate comments from 27 January meeting. Put literal names in reserved name space.

[000126] Integrate comments from 20 January meeting.

[000118] Integrate comments from 13 January meeting.

[000112] Integrate comments from 6 January meeting. See in particular **`_Unwind_ForcedUnwind`**.

[991230] Integrate HP specification, Chapter 8.

[990909] Original version.
