Exception Handling for aC++ on Tahoe

This document describes the C++ specific portion of exception handling for aC++ on Tahoe. Its purpose is to gather all C++ information in one place.

1. History

Please feel free to add your own comments and ideas. For this, please add your comments prefixing them with your initials in the ***Author*** character style.

Revision	Date	Initials	Name
0.1	08/27/98	ddd1	Christophe de Dinechin
0.2	08/28/98	ddd2	Christophe de Dinechin
0.3	9/10/1998	bradd2	Bradd W. Szonye
0.4	9/25/1998	bradd3	Bradd W. Szonye
0.5	9/29/1998	ddd3	Christophe de Dinechin
1.0	10/05/1998	bradd4	Bradd W. Szonye
1.1	10/29/1998	ddd4	Christophe de Dinechin
1.2	12/15/1998	ddd5	Christophe de Dinechin
1.3	01/11/1999	ddd6	Christophe de Dinechin

Note Please number your comments (ddd1, ddd2, ddd3). This may help tracking at what time a given comment was made, and checking whether it is still valid.

2. References

[1] *Exception Handling on HP-UX* (Cary Coutant), Version 2.3 Draft 3
http://cllweb.cup.hp.com/runtime/tahoe/docs/supplement/eh.pdf

[2] *64-Bit Runtime Architecture and Software Conventions for IA-64* (Tahoe Runtime Architecture Task Force), Version 2.4 Draft 1.
http://cllweb.cup.hp.com/runtime/tahoe/docs/ia64rt.pdf

[3] *iA-64 Stack Unwind Library for HP-UX* (Chip Chapin, Cary Coutant), Version 2.5b.
http://cllweb.cup.hp.com/runtime/tahoe/docs/supplement/unwind-api.pdf

[4] *ISO C++ Final Draft International Standard* (Now an ISO standard)
http://cllweb.cup.hp.com/acxx/aCC/drafs/FDIS)

[5] *Ucode-2 proposal to support C++ EH on Tahoe* E-mail message from John Kwan, dated Mon, 30 Nov 1998 14:04:36 -0800

[6] *Ucode-2 External Specification*

3. General Description

3.1. Architecture

The general framework of exception processing on Tahoe is described in "*Exception Handling on HP-UX*" [1] for HP-UX specific details. A more general description of stack unwinding for IA-64 systems can be found in "*64-Bit Runtime Architecture and Software Conventions for IA-64*" [2].

Exceptions are processed using two separate components:

- A system-wide *unwind library* handles stack unwinding, as described in "*64-Bit Runtime Architecture and Software Conventions for IA-64*", Chapter 11.
- A set of language-specific *personality routines* perform language-specific cleanup actions. For aC++, this involves in particular calling the destructors as the stack is being unwound.

The intent is to allow exceptions to be processed in a somewhat language-independent manner, and also to flow accross architecture boundaries (IA-64 or PA-RISC).

Note As of this version of the document, *none of these two features is planned for aC++*. More specifically, it is not known whether C++ could catch exceptions raised from another language such as Java, even in a catch-all (`catch (...)`) clause. And being able to throw exceptions from PA-RISC code to IA-64 code or conversely is currently not requested.

3.2. Exception Propagation

When a C++ exception occurs, control is transferred to the next enclosing `catch()` block that matches the thrown exception type. The process of cleaning up the call stack to restore the context of the exception handler is called *stack unwinding* in each of "*Exception Handling on HP-UX*", "*64-Bit Runtime Architecture and Software Conventions for IA-64*" and the "*ISO C++ Final Draft International Standard*".

This corresponds however to slightly different actions:

- For the unwind library, stack unwinding as defined in "*64-Bit Runtime Architecture and Software Conventions for IA-64*" is essentially restoring the register state, the stack pointers and the instruction pointer to the context where the handler will execute.
- For C++, the unwind process described in "*ISO C++ Final Draft International Standard*" assumes that the previous exists, and then adds the requirement that destructors be properly called for all C++ functions which are exited during this process. It also describes more precisely what it means to throw, catch, handle, and finish an exception in the context of a C++ program.

3.3. Unwind Phases

The stack unwinding process described in "*Exception Handling on HP-UX*" uses two phases:

- *Phase 1* corresponds to a search for an appropriate handler, without actually unwinding the stack.
- *Phase 2* corresponds to actual stack unwinding.

Each of these two phases combines the unwind library and the personality routines, since the validity of a given handler and the way to transfer control to it are language-dependent, but the means of locating and restoring previous stack frames are language independent.

A two-pass exception-handling model is not strictly necessary to implement C++ language semantics, but it does provide some benefits. For example, the first pass 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 original point). While C++ does not support resumptive exception handling, other languages do, and the two-pass model allows C++ to coexist with those languages on the stack.

Note that even with a two pass model, we may execute each of the two passes 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 phase1. 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 instance, if the first two frames to unwind contain only cleanup code, and the third frame contains a catch clause, the personality routine in pass 1 does not need to return `EH_HANDLER_FOUND` for the first two frames. It has to for the third frame, because it is unknown how the exception will propagate out of this third frame.

3.4. Landing Pads and Alternate Return

C++ exceptions are synchronous. Specifically, they can propagate into a function in *only* two ways:

- From a `throw` expression.
- From a call site which itself (indirectly) throws an exception.

C++ does not guarantee that any form of asynchronous exception would work.[1] We can thererfore associate all exception-handling behavior with a call site, since even throw expressions are transformed into calls by the front end. The exception path from a call site is its *alternate return* path, and the code executed during alternate (exceptional) return begins with a *landing pad*, which is a special kind of label.

The alternate return path performs three major tasks, each of which is optional:

- Execute any *compensation code* (see "*Exception Handling on HP-UX*" Appendix A). This code is required to preserve language semantics when optimizations moved code after the call in the normal flow of execution.
- Invoke destructors for all automatic objects constructed in the scope of the exception path, according to **[except.ctor]**, “Constructors and Destructors,” 15.2.
- Transfers control to user-defined catch clause code (for catch clauses) or C++ runtime code (for exception specification violations).

3.5. Exception Object

Supporting exceptions involves passing C++-specific information needed by the C++ runtime support library through the unwind library. For that purpose, the unwind library receives from the C++ runtime support library a pointer to an *exception object*. This exception object is opaque to the unwind library.

1. For instance, throwing an exception from a signal handler is explicitly forbidden to user code (see "*ISO C++ Final Draft International Standard*", **[lib.support.runtime]**, 18.7.5, and in particular note 34), although the compiler and runtime system is not required to diagnose a violation. Similarly, an illegal memory access or an illegal floating point operation is not supposed to trigger a C++ exception. See also Section 11.1, “Exceptions and Signal Handlers” on page 46.

Caution

In the PA-RISC runtime library, the *exception object* is the exception currently being thrown. This document gives an extended meaning to this exception object, which not only contains the exception being thrown, but also some additional bookkeeping information which is needed by the runtime to propagate the exception.

This whole information used to be passed as separate parameters in the PA-RISC model. However, the unwind library now only gives us a single parameter, which is a pointer to the language-dependent exception object.

Note

This documents therefore uses the following terms: the *exception object* is the complete information that the runtime uses for exception processing; the *thrown exception* is the temporary copy of the exception to throw; the *throw argument* is the actual argument of `throw`; the *exception object* is made of the thrown exception and an additional *exception header* used for internal bookkeeping.

For C++, this exception object shall contain at least the following:

- Access to the thrown exception,
- Access to the exception type information. This cannot be retrieved from the thrown exception because non-polymorphic types can be thrown (including, for instance, `int`). This information will be used to find the appropriate `catch` handler,
- A copy of the `unexpected` and `terminate` handlers. The reason for keeping a copy is that "*ISO C++ Final Draft International Standard*" **[lib.unexpected]** (18.6.2.4) states that you should use the handlers which are active immediately after evaluating the throw argument. If destructors change these handlers during unwinding, you are not supposed to use the new value.
- A pointer to the next (most recently caught) exception object, forming a stack of currently-caught exceptions for `throw;` and helping to determine exception lifetime (see Section 10.12 on page 45).
- A count of how many handlers have caught this exception object, also used to determine exception lifetime (see Section 10.12 on page 45).
- The handler switch value found in the search phase of unwinding. By storing this in the exception object, the cleanup phase will not need to re-examine action records.

Note

The dynamic type of the thrown exception is the *static* type of the throw argument. The thrown exception and throw argument may have different dynamic types when throwing a reference or dereferenced pointer.

We do not need version information, since the general unwind library framework specifies an *exception class identifier*, which we may change should the layout of the exception object change significantly.

The thrown exception and the additional information will be kept contiguous. The layout of the exception object will therefore be the following `__exception` header, followed by the thrown exception itself:

```
struct __exception
{
      __type_info *          exceptionType;
      unexpected_handler     unexpectedHandler;
      terminate_handler      terminateHandler;
      void *                 nextException;
      int                    handlerCount;
      int                    handlerSwitchValue;
};
```

This structure must be padded to guarantee proper alignment of the thrown exception whatever its own alignment requirements. The interface between the runtime and the generated code will be the address of the thrown exception, the `__exception` object being accessed at a negative offset.

Note

The reason for using the address of the thrown exception rather than the address of the header is to prevent the generated code from having to know anything about the header. If the following conventions are used, the generated code does not need to know anything about the `__exception` header, allowing later runtime libraries to change the layout or even size of that header.

3.6. Unwind Tables

Stack unwinding is controlled by *unwind tables*, as described in Chapter 11 of "*64-Bit Runtime Architecture and Software Conventions for IA-64*" (See in particular Figure 11-2). These unwind tables contain a language-specific portion.

The C++-specific data area of the unwind table contains (see figure in Section 6.1 on page 22):

- a *call site table* that describes call sites within the function and their associated landing pads and action records
- a list of *action records* that describe what specific actions need to take place during stack unwind and exception propagation.

4. Throwing an Exception

This section describes what the C++ generated code and runtime library need to do to throw an exception.

4.1. Outline

The generated code will do the following to throw an exception:

- Call `__allocate_exception` to create an exception object (see Section 4.2 on page 7).
- Evaluate the thrown expression, and copy it into the buffer returned by `__allocate_exception`, possibly using a copy constructor (see Section 4.3 on page 9). If evaluation of the thrown expression or the copy constructor itself exits by throwing an exception, that exception will propagate *instead of* the expression itself. Cleanup code should be added to ensure that we call `__free_exception` on the just allocated exception object.
- Call `__throw` to handle the exception to the runtime library (see Section 4.4 on page 10). `__throw` never returns.

Front-End Generated Code: Throwing an object as in:

```
throw X;
```

will be transformed into:

```
temp1 = __allocate_exception(sizeof(X));
// Attributes: Never throws (no EH info)

#if COPY_ELISION
[evaluate X in temp1]
#else
[evaluate X in temp2]
__copy_ctor(temp1, temp2)
// Attribute: Landing Pad L1
#endif

__throw(temp1, typeid(X));
// Attribute: Never returns

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

Back-End Transformations: Back-end transformations will be detailed on a case-by-case basis in the next sections.

4.2. Allocating the Exception Object

4.2.1. Requirements

Storage is needed for the exceptions being thrown. This storage must persists while stack is being unwound, since it will be used by the handler.

- PA-RISC used space in the throwing routine's stack space, in effect extending the stack using an `alloca`-like stack frame. This solution is not retained for the current Tahoe runtime architecture[1].
- Allocating heap space causes problems under low-memory conditions. If heap space is used, a backup solution must be found, at least for throwing `bad_alloc` when out of memory. This is currently the proposed solution.

1. One problem that was found on PA-RISC was the impossibility for the routine extending the stack to correctly allocate space for input arguments that don't fit into registers. A function call using many arguments and placed in a catch() clause might not get the stack space it requires for the arguments.

- Static storage cannot be used alone, because the same `throw` statement may be executing more than once at a given time (for instance in recursions initiated from the `catch` block).
- The selected storage must also be thread safe. However, we cannot simply use thread-local storage (at least in its hardware-based implementation), because of its specific limitations[1].

Note

The semantic of throwing an object always implies a copy (see "*ISO C++ Final Draft International Standard*" **[except.throw]** (15.1.3)). Specifically, a throw-expression uses its operand to initialize a temporary object with the *static* type of that operand. The type of the temporary object is the static type of the thrown exception, and the size of the temporary can be determined at compile time

4.2.2. Low-Memory Emergency Buffer

The major issue with exception object allocation is how to deal with a shortage of memory. From the point of view of the user, the following is a reasonable set of expectations regarding low-memory conditions:

- Throwing `bad_alloc` when out of memory should be permitted (the out-of-memory condition may have been detected through another mean, such as a `new(nothrow)` returning 0).
- Ideally, any other exception should be raisable at that time (within a given reasonable size constraint). One reason is that the "*ISO C++ Final Draft International Standard*" suggests throwing classes *derived* from `bad_alloc` to report out-of-memory conditions from a library.
- An out-of-memory condition typically occurs simultaneously for all threads of a same process. The mechanism to use in that case should not break if several threads encounter an out-of-memory condition at the same time.
- Under low memory conditions, nested exceptions and exceedingly large exception object could fail to allocate the necessary memory. In that case, `terminate` should still be called properly.

Under these assumptions, the C++ runtime library `__allocate_exception` function will allocate exception objects as follows:

- A static *emergency buffer* of a fixed size (64K) is allocated for exceptions under low-memory conditions.
- Exception objects are normally allocated on the heap. However, if the heap allocation fails, the emergency buffer is used.
- Allocation in the emergency buffer is made using a bitmap, in 1K chunks. Any single exception thrown using the emergency buffer is not allowed to exceed 1K minus the header size, or `terminate()` is called.

1. When using hardware TLS, the resulting code cannot be dynamically loaded as part of a DLL.

- At most 16 threads are allowed to use the emergency buffer at any given time. Following threads are blocked.

This mechanism offers the following guarantees:

- Any kind of exception of up to 1K in size can be thrown even under low-memory conditions.
- A deadlock generated by this mechanism would involve at least 16 user locks. This is assumed to be unlikely during a low-memory cleanup processing.
- Each thread is either blocked, or starts a cleanup in which it can have at least 4 nested exceptions of 1K each.

4.2.3. Exception Allocation Routines

Memory will be allocated by the `__allocate_exception` runtime library routine. This routine will receive the size of thrown exception (*not* including the size of the `__exception` header), and will return a pointer to the temporary space for the exception.

The temporary space will be freed by `__free_exception` (see Section 5.8, “Finishing and Destroying the Exception” on page 21), taking the address returned by a previous `__allocate_exception`.

These routines are thread-safe, and may block threads after the maximum number of threads allowed to use the emergency buffer has been reached. This would typically be done using a semaphore.

Prototypes:

```
void *__allocate_exception(size_t thrown_size);
void __free_exception(void *thrown_exception);
```

4.3. Creating the Exception Object

4.3.1. Normal Case: Avoid Temporary Copy

In most cases, we can directly use the buffer returned by `__allocate_exception` as a target for evaluating the expression. In that case, we avoid an unnecessary copy. This method is allowed by the "*ISO C++ Final Draft International Standard*" **[except.throw]** (15.1.5), and will be preferred when possible for efficiency reasons.

4.3.2. Complex Case: Temporary Copy

A copy is still necessary in several cases:

- If the throw argument is a static object,

- If the throw argument has a dynamic type which may be different from the static type (dereferenced pointer or reference),
- If the thrown argument would not be evaluated in memory normally (for instance if the thrown object is a constant scalar value or a local variable.

In all these cases, the copy will be made as usual, including calling the copy constructor if necessary, before calling `__throw`. Note that the copy of the `unexpected` and `terminate` handlers occurs as part of the execution of `__throw`, immediately after the copy constructor.

If the copy constructor itself throws an exception, that exception will propogate *instead of* the operand of throw, just as if `foo()` throws an exception in `throw (foo() + bar()).`

4.4. Calling `__throw`

The `__throw` runtime library routine is given control by the generated code at the end of the `throw` statement. This routine never returns. The arguments are:

- The address of the thrown exception (which is in the middle of the full exception object, see Section 3.5, “Exception Object” on page 4).
- A `__type_info` pointer, used for matching the potential catch sites to the thrown exception. Note that this is the static type of the thrown argument, not its dynamic type. Also, the `__type_info` object is related to but not necessarily the same as `std::type_info`.

4.5. Implementation of `__throw`

The `__throw` library routine will perform the following:

- Get the `__exception` header from the thrown exception object address, which can be computed as (see Section 3.5, “Exception Object” on page 4):
```
    __exception *header = ((__exception *) thrown_exception - 1);
```
- Save the current `unexpected_handler` and `terminate_handler` in the `__exception` header.
- Save the `__type_info` argument to `__throw` in the `__exception` header.
-
- Increment the `uncaught_exception` flag.

It then calls `_RaiseException` in the system unwind library, with the following arguments:

- The first argument of `_RaiseException` is the exception class. It will be set to a 64-bit value representing the ASCII string “`HPaC++01`”, as suggested in "*Exception Handling on HP-UX*".
- The second argument to `_RaiseException` will be a pointer to the thrown exception, which `__throw` itself received as an argument.

`__RaiseException` begins the process of stack unwinding, described in Section 5.1 on page 12.

In special cases, such as an inability to find a handler, `_RaiseException` may return. In that case, `__throw` will call `terminate`, assuming that there was no handler for the exception. It is reasonable to assume that this would be the “best possible one action” if other reasons caused the stack unwinding to fail.

Prototype:

```
int _RaiseException(
      UInt64      exception_class,
      void *      exception_object
);
```

4.6. Caught Exceptions Stack (Current Exception)

For rethrowing an exception and for managing the lifetime of exception objects, we need to keep track of caught exceptions. A stack of most-recently caught exceptions, implemented as a linked list, seems appropriate for this use.

In C++, an exception is considered *caught*:

- When initialization is complete for the formal parameter of the corresponding catch clause (or, upon entry to a catch-all clause, which has no parameter).
- Upon entry to `unexpected()` because of a violated exception-specification.
- Upon entry to `terminate()` when called because of a throw.

Therefore, catch clauses, unexpected, and terminate are all exception *handlers*. Consider the execution of a catch clause, a call to unexpected, or a call to terminate to be surrounded by a handler prologue and epilogue.

The prologue saves a pointer to the handled exception and calls `__begin_catch(exception)`, which:

- Increments the exception’s handler count.
- Places the exception on a stack of currently-caught exceptions if it is not already on the stack, linking the exception to the previous top of the stack.
- Decrements the `uncaught_exception` count.

Upon exit from the handler by any means, the epilogue calls `__end_catch()`, which:

- Locates the most recently caught exception and decrements its handler count.
- Removes the exception from the “caught” 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;`.

The runtime stores the top of the current exception stack as a pointer in thread-local storage. When user code requests re-throw, the mechanism checks the current exception pointer. If the stack is empty, the runtime mechanism calls `terminate()`; otherwise, it reactivates the most recently caught exception (the top of the stack).

See also Section 5.6, “Rethrowing” on page 20.

5. Catching Exceptions

5.1. Unwinding the Stack

In the current runtime architecture, stack unwinding itself is begun by calling `__RaiseException()` and performed by the unwind library. The stack unwind library runs two passes on the stack.

The unwind process is 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 "*64-Bit Runtime Architecture and Software Conventions for IA-64*" [2].
- Call the personality routine 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 5.3 on page 16). 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 5.3)[1] 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, a local variable

1. Landing pads are special kinds of labels, declared with the Ucode2 PAD operator.

whose lifetime is limited by the try block enclosing the scope 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 is similar to a switch statement[1], where the switch value is determined by the runtime based on the exception type (see Section 5.4), and passed in a landing pad argument.
- As soon as the runtime determines that the execution will go to a catch clause, the unwinding process is considered complete for the unwind library. A catch clause may still rethrow the current exception (see Section 5.6.) or a different exception, but a new unwind process will occur in both cases. Otherwise, after the code in a catch clause has executed, execution resumes at the end of the try block which defines this catch clause.
- If none of the possible catch clauses 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 some additional cleanup code, which will call all destructors that need to be called for the current stack frame (see Section 5.5.). For instance, a local variable whose lifetime is limited by the function containing the try block, but not by the try block itself would be destroyed here.
- At the end of this current cleanup code, control is transferred back to the unwind library, to unwind one more stack frame (see Section 5.7.)

Overview of Generated Code: A call site that may throw in a try-catch block, as in:

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

will be transformed as follows:

```
// In "Normal" section
foo();
// Call Attributes: Landing Pad L1, Action Record A1
goto E1;

E1: // End Label
bar()

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

1. There is a special Ucode2 operator for this kind of switch statement, which is called XHJP (Exception Handling Jump)

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

S1: // Switch label: (Ucode2 XHJP operator)
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]
[RESX]

H1: // Handler label
[Initialize catch parameter]
__begin_catch(exception);
[User code]
goto R1;

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

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

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

Note

The [RESX] statement is a special Ucode2 operator. The [RESX] statement would normally be a call to `_ResumeUnwind`, but would be transformed into a branch in case the above code is inlined. The target of the branch would be the enclosing landing pad label (L) for the try block in which the function is inlined.

Conventions for label naming: In the rest of this document, labels will be numbered as follows (these conventions exist only for clarity):

- **L** labels will be used for landing pads. Landing pad labels are generated empty by the front-end, and will be used for compensation code generated by the back-end. Landing pad labels are special because the runtime jumps to the label after setting some special arguments in dedicated registers (see Section 5.3 on page 16). They can also be connected to the `[RESX]` statement in the case of inline functions.
- **C** labels correspond to front-end generated cleanup code, such as the call to destructors.
- **S** labels correspond to a “XHJP” Ucode2 operator. The back-end generates a switch statement, selecting the various possible `catch` clauses. The corresponding switch values, are also generated by the back-end. The default of this “switch” statement correspond to the `[RESX]` statement.

- **H** labels correspond to the handler code in the catch clause. This front-end generated code corresponds to the user code actually written in the `catch` clause.
- **R** labels are used to resume to normal execution. Their only purpose is to share the call to `__end_catch()` that is shared at the exit of all catch clauses.
- **E** labels are the end of the try/catch block. E labels are in the normal flow of execution (they are not “in” the landing pad”.)
- **X** labels corresponds to an exit through the RESX operator, and the corresponding cleanup code if any.

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 the unwind library (see "*Exception Handling on HP-UX*"):

Prototype:

```
int __cxxUnwindPersonality(
      int          version,
      int          phase,
      UInt64       exception_class,
      void *       exception_object,
      UnwindState *unwind_state
);
```

The arguments are the following:

`version`	Version number of the unwind library
`phase`	One of the two constants: EH_SEARCH_PHASE EH_CLEANUP_PHASE
`exception_class`	The 64-bit integer value corresponding to ‘HPaC++01’ for aC++-generated exception. If other values received, see Section 11.2, “Exception Accross Different Languages” on page 47.
`exception_object`	The pointer to the thrown exception, from which the `__exception` header can be found (see Section 4.2 on page 7).
`unwind_state`	The internal state of the unwind library. This semi-opaque object provides access to the UnwindContext, UnwindInfo, and other important information about the function.

The personality routine works only on the current frame. It does the following:

- Use the UnwindState to obtain the UnwindContext and UnwindInfo.
- Use the UnwindContext to obtain the IP.

- Use the UnwindInfo to obtain the language-specific data area and the base pointers for function-relative and other offsets. The language-specific data area is a set of C++-specific table (see Section 6, “Exception Handling Tables” on page 22).
- Lookup the current PC into the call-sites table stored in the language-specific data area. See Section 6.3.2, “Call-site table” on page 24 for a description of the call site table.
- From the call-site table, find the start of the possible actions for that call-site. Follow the actions list until an action is found that matches the thrown exception type. See Section 6.4 on page 26 for a list of possible actions.
- If the current stack frame has a landing pad, request the unwind library to branch to this landing pad by setting up a context and returning it to the unwind library (see Section 5.3 on page 16.) This also involves setting the proper landing pad parameters.

5.3. Branching to the Landing Pad

To branch to the landing pad, the C++ personality routine will execute the following (during phase 2 only):

- Compute the start address of the landing pad (see Section 6 on page 22).
- Set the PC of the current stack frame to this computed address, using the `UnwindContext_modifyPC` function.
- Set additional parameters to the landing pad by placing them in the appropriate registers, using the `UnwindContext_putGR` function.
- Return the `EH_INSTALL_CONTEXT` value to the unwind library, causing the context to be restored and the landing pad to be called.

Note

The Unwind library owns the UnwindState object, not the C++ personality routine or landing pad. This document does not specify the lifetime of the UnwindState object, but there are three likely possibilities. (1) The UnwindState is a “simple” object that does not require memory management. (2) The UnwindState is a “complex” object deleted before branching to the landing pad each time. (3) The UnwindState is a “complex” object saved before installing a cleanup context but deleted before installing a handler context. This document assumes the last case in places where it makes a difference.

On entry to a landing pad, the context will be set with the following parameters:

Name	Type	Description
`exception_class`	uint64	Exception class value, normally set to ‘HPaC++01’ (see Section 5.2 on page 15)

Name	Type	Description
`exception_object`	pointer	Pointer to the thrown exception, allocated (see Section 3.5 on page 4)
`unwind_state`	pointer	Pointer to the unwind library's internal state, saved between landing pad calls.
switch_value	int64	Switch value used for determining the proper catch clause or other C++ handler (see Section 6.4 and Section 6.5)

Argument Passing Conventions: Arguments are passed in registers, and set in the unwind context using special routines from the unwind library. Registers have to be scratch registers, since they are not expected to be kept accross the call. The actual register names are unknown to C++; the runtime library accesses them through symbolic names provided in the UnwindContext interface.

However, as far as Ucode2 is concerned, these values are moved into temporaries immediately. Therefore, the runtime cannot rely on any of these registers holding the proper value past the landing pad "label".

Front-End Generated Code: The front-end will access the landing pad arguments through temporaries ("tags"). The tags are set using the U2_Set_Routine_Landing_Pad_Bindings() function.

```
void U2_Set_Routine_Landing_Pad_Bindings (
      U2_TAG            routine,
      U2_TAG            exception_class,
      U2_TAG            exception_object,
      U2_TAG            unwind_state,
      U2_TAG            selector_value
);
```

5.4. Handling the Exception

There are three ways to handle an exception in C++: with a catch clause, with a violated exception specification (which results in a call to unexpected), or by detecting an error in unwinding itself (which results in a call to terminate).

In any of the three handlers, it is possible to rethrow the handled exception (see Section 5.6 on page 20), although it is really only useful to rethrow in a catch clause. Upon entering a handler, the language-independent unwind process is "finished," although the exception is not "finished" in the C++ sense until all handlers for that exception have ended without a rethrow.

5.4.1. Catch Clauses

If the current stack frame contains a `try` statement, its `catch` parts may be able to catch the thrown exception. Several conversions are acceptable when matching a thrown exception to the `catch` argument. These conversions are listed in **[except.handle]**, 15.3.3, in the "*ISO C++ Final Draft International Standard*" [4].

During unwinding, the personality routine will macth the type of the thrown exception with the type of the catch arguments in the current frame, which are described by a specific action record (see Section 6.4, “Catch Clause” on page 26), and transmit a “switch selector” to the landing pad as an additional argument. If no match is found, a default switch selector (0) will be used, which will cause all “switch” statements to exit through their “default” exit.

For instance, for the following code:

```
try { }
catch (A) { }
catch (B) { }
```

the generated code will be equivalent to:

```
S1:
switch (switchSelector) // XHJP operator
{
      case A: goto Handle_A;
      case B: goto Handle_B;
      default: goto X1;
}
X1:
[RESX]
```

5.4.2. Catch-All Clauses

A catch-all clause exists, it will be used for any exception being thrown. This is implemented by using the “default” exit of the XHJP operator.

For instance, for the following code:

```
try { }
catch (A) { }
catch (...) { }
```

the generated code will be equivalent to:

```
S1:
switch (switchSelector) // XHJP operator
{
      case A: goto Handle_A;
      default: goto Handle_DotDotDot;
}
```

Note that the runtime does not have to specifically use switch selector value 0 to go to this catch-all clause. Any value that does not correspond to a previous catch clause will work. However, value 0 will be reserved for that use.

A call-site may be generated, which has an associated landing pad, but no action record. In that case, the call-site entry has a “0” value in its action record field. (see Section 6.3.2).

5.4.3. Unexpected Exceptions

If a thrown exception violates an exception specification, the runtime passes a (negative) switch selector value indicated in the action record. Any other value (including 0) is the value of an exception that may go through this exception specification.. Zero would be used if the exception can pass through the exception specification, but there is no following catch clause that can catch this exception.

This mechanism permits inlining of a function with exception specfications inside a catch clause. If more than one function is inlined, different negative values will be used to indicate a violation exiting any particular function.

If the switch selector is the negative switch selector corresponding to that particular exception specification violation, then control goes to a function called `__CallUnexpected`. This function itselfs calls `unexpected()`, which may only exit by throwing another exception or aborting the program. If `unexpected()` throws an exception, `__CallUnexpected` checks the new exception type against the original exception specification. If the new exception is allowed, `__CallUnexpected` exits and propagates the new exception. If it is still a violation, but the exception type `std::bad_exception` is allowed, `__CallUnexpected` propagates a `std::bad_exception`. Otherwise, it calls `terminate()`.

For instance, the code for a function like:

```
void f() throw(A, B) { g(); }
```

will look like:

```
g(); // Landing pad L1
return;

L1:
C1:
S1:
// XHJP statement for exception specification violation
switch(switchSelector)
{
      case NEGATIVE_FILTER_VALUE: goto H1;
}
X1:
[RESX]
H1:
__CallUnexpected();
```

The main differences between a catch clause and an exception specification check are in the type matching and the type of handler used; the personality behavior is otherwise very similar.

5.4.4. Termination Conditions

Several errors in unwinding (such as having no handler for an exception or throwing an exception from a destructor) lead to termination of the program. The `terminate()` function may only exit by aborting the program (or some other irrevocable action such as reinitializing memory and restarting).

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

5.5. Cleanup

All object cleanup can be done by generated code on the alternate return path, either by calling a destructor, or by calling a runtime routine to handle more complex cases such as array destruction. Object cleanup thus does not require a specific action record type or types(s), as there is no additional runtime intervention required beyond that for any landing pad.

The cleanup code execution corresponding to the try block is independent of the existence of a suitable `catch` clause in the current frame. It will always be executed.

Cleanup code corresponding to the function exit is executed immediately prior to resuming unwinding with the [RESX] operator. This code is executed only if none of the catch clauses matches the exception being thrown.

5.6. Rethrowing

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.

To re-throw, the runtime library checks the top of the caught exceptions stack (using the "current exception" pointer in thread-local storage). If that stack is not empty, the runtime sets the `uncaught_exception` flag and starts the unwinding process anew. Rethrow is very similar to ordinary exception throwing, except that the exception object (and all of its bookkeeping information) already exists.

Note that rethrow does not attempt to manage object lifetime; instead, `__begin_catch` and `__end_catch` (both on the normal and exceptional exit paths from a handler) take care of exception destruction.

As defined in the "*ISO C++ Final Draft International Standard*" **[except.throw]**, 15.1.8, rethrowing without a current exception immediately calls `terminate()`.

5.7. Resuming Unwinding

Unwinding is resumed at the end of the landing pad when:

- The landing pad performs only cleanup tasks, but does not contain any user-defined `catch` clause.
- There are `catch` clauses, but none matches the thrown exception type. In that case, the switch selector value is set to a value that does not correspond to any switch statement (0).

In that case, a [RESX] operator is emitted by the front-end.

- If the function was not inlined, a call to `_ResumeUnwind` is generated `_ResumeUnwind` takes its arguments from the landing pad arguments (see Section 5.3.)
- If the function was inlined, a branch to the call-site landing pad in the calling function is generated instead. Note that the landing pad is then turned into a "regular" label, and no longer has landing pad parameters.

5.8. 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 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 `__begin_catch` and `__end_catch` runtime functions, which keep track of what handlers exist for which exceptions. When `__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 exception lifetime occurs at runtime, because it is impossible to determine statically. 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 lifespan 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.

- 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 `__free_exception`, as described in Section 4.2, “Allocating the Exception Object” on page 7. This is not normally called directly by the generated code, but rather by the `__end_catch` routine, which also removes the exception from the caught-exceptions stack.

6. Exception Handling Tables

This section describes the data that the compiler generates to enable the runtime to find appropriate information on the actions to take in case of exception.

6.1. Overview

The process of finding exception handling information from the current PC is summarized in the diagram below:

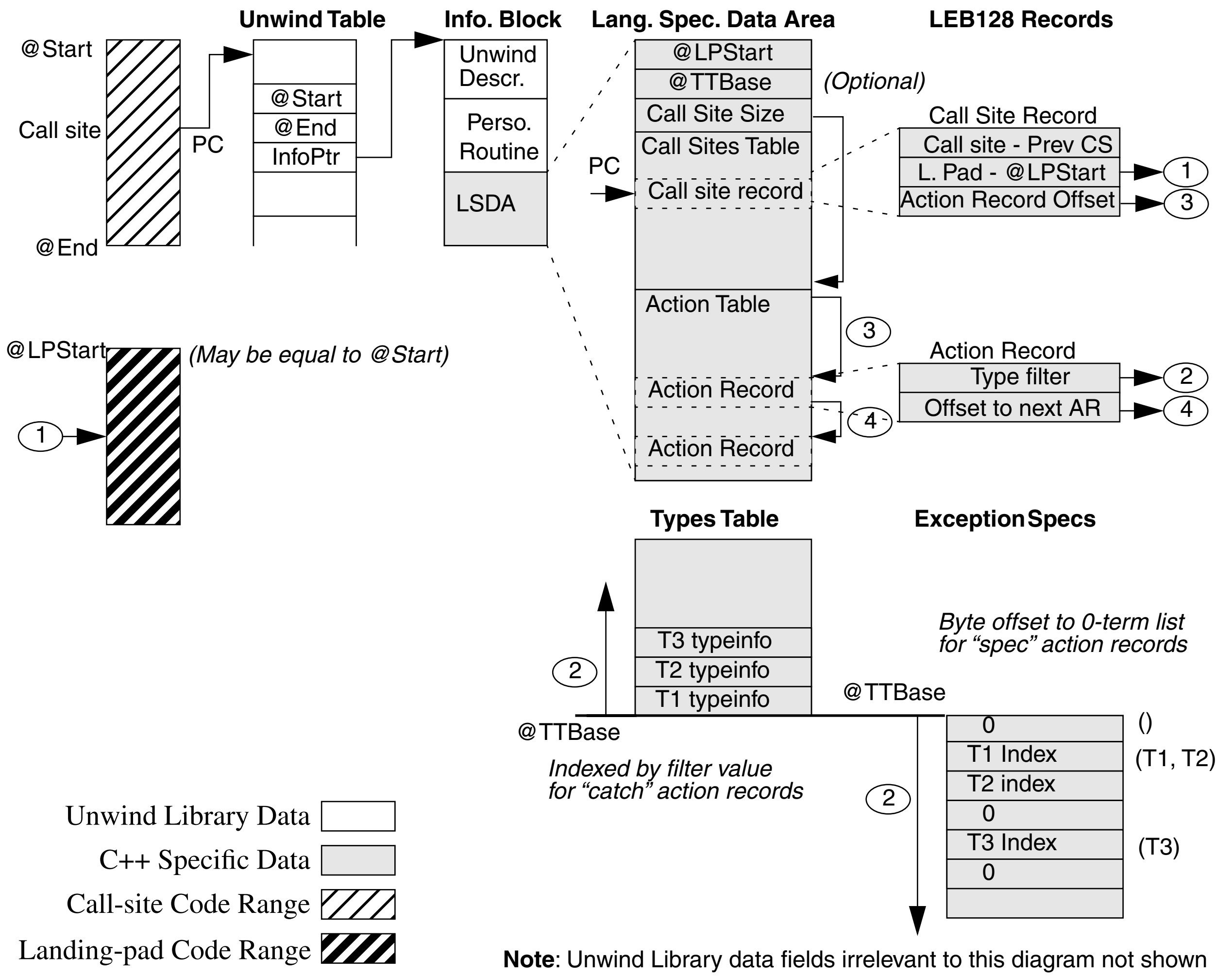
All tables are in “text” space. The types pointed by the typeinfo pointers are identified by a GP-relative offset.

6.2. System unwind tables

These are described in "*64-Bit Runtime Architecture and Software Conventions for IA-64*" [2]. The most important field for C++ exception handling is the “start” field of the unwind table entries. Call sites are stored as offsets relative to the procedure fragment start. Note that a single procedure may be split into more than one procedure fragment.

If a procedure is being split and causes more than one procedure fragment to exist, landing pads can reside in any of the possible fragments. There may even be a fragment specifically for the landing pads, which typically correspond to infrequently executed code. However, there is currently no special provision for more than one landing pad fragment per procedure fragment (“hot” and “cold” landing pads, for instance).

This can only be achieved by duplicating the unwind table entries and LSDA for each such fragment. An alternative was considered, where a bit would indicate whether a landing pad was relative to the procedure fragment or landing-pad fragment, but the benefit was considered insufficient compared to the space loss.



Note: Unwind Library data fields irrelevant to this diagram not shown

6.3. The language-specific data area

The language-specific data area (LSDA) contains pointers to related data, a list of call sites, and a list of action records. Each procedure fragment coming from C++ code (nominally a function) has its own LSDA. Several parts of the LSDA use the LEB128 compression scheme, which is described in Section 6.8 on page 28.

6.3.1. LSDA header

The LSDA header contains fields which apply to a procedure fragment. Currently, there are two fields defined:

- The landing pad start pointer. This is a self-relative offset to the start of the landing-pad code for the procedure fragment. Landing pad fields in the call-site table are relative to this pointer. A value of 0 means that the LSDA is otherwise empty. The low four bits are reserved. A value of 0000 means that there is a type table pointer. A value of 0001 means that there is no type table pointer. In the rest of this document, this address is called `LPStart`.

- The types table pointer. This is a self-relative offset to the types table (catch clause and exception-specification types), described in Section 6.4 on page 26. This word does not exist if the value of the low four bits of the landing pad offset have value 0001. In the rest of this document, this address is called `TTBase`.

6.3.2. Call-site table

The call-site table is a list of all call sites that may throw an exception (including C++ 'throw' statements) in the procedure fragment. It immediately follows the LSDA header. Each entry indicates, for a given call, the first corresponding action record and the corresponding landing pad.

The table begins with the number of bytes, stored as a LEB128 compressed, unsigned integer. The records immediately follow the record count. They are sorted in increasing call-site address. Each record indicates:

- The position of the call-site,
- The position of the landing-pad,
- The first action record for that call-site.

Call-site record fields:

call site	Offset of the call site relative to the previous call site, counted in number of 16-byte bundles[a]. The first call site is counted relative to the start of the procedure fragement.
landing pad	Offset of the landing pad, counted in 16-byte bundles relative to the LPStart address.
action record	Offset of the first associated action record, relative to the start of the actions table. This value is biased by 1 (1 indicates the start of the actions table), and 0 indicates that there are no actions.

a. There can be a single call that throws per bundle. Multiple calls may be placed in a single bundle (for instance, in an expression like `a ? b() : c()`) only if they can share the same landing pad.

All fields of the landing pad table are compressed using the LEB128 encoding (described in Section 6.8, "Decoding exception records" on page 28).

A missing entry in the call-site table indicates that a call is not supposed to throw. Such calls include:

- Calls to destructors within cleanup code. C++ semantics forbits these calls to throw.
- Calls to intrinsic routines in the standard library which are known not to throw (`sin`, `memcpy`).

If the runtime does not find the call-site entry for a given call, it will call `terminate()`.

Front-end generated code: The C++ front end does not actually generate the landing pad tables, but instead emits Ucode to associate a "head" action record and a landing pad with a call site. It generates something like:

```
// This is the landing pad for the call site.
Declare_Landing_Pad tag: L0
PAD tag: L0
      // landing pad code here

// This is another action linked to A0
// Needs to be declared first in Ucode2
Declare_Catch_Action_Record
      tag: A1
      type_info: __typeinfo_tag1
      catch_label: L1
      profile_count:
      next_action: NULL_TAG

// This is the "initial" action record for the call site
Declare_Catch_Action_Record
      tag: A0
      type_info: __typeinfo_tag0
      catch_label: L2
      profile_count:
      next_action: A1

// The call info ties the landing pad
// and actions to the call site
Declare_Call_Info tag: C0
      Set_Call_Info_Actions call_info: C0 actions: A0
      Set_Call_Info_Alt_Return call_info: C0 landing_pad: L0
End_Call_Info tag: C0
```

6.3.3. Action table

The action table follows the call-site table in the LSDA. The individual records are one of two types:

- Catch clause, described in Section 6.4 on page 26.
- Exception specification, described in Section 6.5 on page 27.

The two record kinds have the same format, with only small differences. They are distinguished by the "switch value" field: Catch clauses have strictly positive switch values, and exception specifications have strictly negative switch values. Value 0 indicates a catch-all clause.

Action record fields:

type filter	Used by the runtime to match the type of the thrown exception to the type of the catch clauses or the types in the exception specification.
action record	Self-relative signed displacement in bytes to the next action record, or 0 if there is no next action record.

All fields are compressed using the LEB128 encoding (described in Section 6.8, "Decoding exception records" on page 28). The structure of the action table is determined by the C++ front-end but subject to modification by inlining and other optimizations. Code generation is responsible for assigning actual switch values and "next record" offsets.

6.4. Catch Clause

The code for the catch clauses following a same try is similar to a `switch` statement. The catch clause action record is informs the runtime about the type of a catch clause and about the associated `switch` value. The Ucode2 operator generating the switch statement is called XHJP.

Note

Note that the runtime may apply some conversions when an exception is thrown with a different type (see acceptable conversion in **[except.handle]**, 15.3.3, in the "*ISO C++ Final Draft International Standard*".) So the pointer to the type information cannot directly be used as a switch value, for instance.

Action Record Fields:

filter value	Positive value, starting at 1. Index in the types table of the `__typeinfo` for the catch-clause type. 1 is the first word preceding TTBase, 2 is the second word, and so on. Used by the runtime to check if the thrown exception type matches the catch-clause type. Back-end generated switch statements check against this value.
next	Signed offset, in bytes from the start of this field, to the next chained action record, or zero if none.

All fields are compressed using the LEB128 encoding (described in Section 6.8, "Decoding exception records" on page 28).

The order of the action records determined by the `next` field is the order of the catch clauses as they appear in the source code, and must be kept in the same order. The C++ language allow two catch clauses in a same procedure to cover related types (such as base and derived). As a result, changing the order of the catch clause would change the semantics of the program.

Runtime Action: If the thrown exception type matches the catch clause type, the switch value of the action record will be passed by the runtime to the landing pad in the "switch selector" argument.

Front-end: The front-end generates an XHJP operator that references this action record.

Back-end: The back-end will assign switch values. If two XHJP operators may be reached from a same landing pad, then they cannot share any switch value, except to represent the exact same type. The XHJP operators will be transformed into switch statements, which branch to the catch clause code if the switch selector value matches the switch value of the action record.

6.5. Exception Specification

An exception specification violation is indicated by the runtime by setting the "switch selector" value to a negative value. Code in the landing pad checks if the switch selector value is that negative value, and if so, calls the `__CallUnexpected` routine. Otherwise, the exception is propagated out.

Action Record Fields:

List of C++ types	Negative value, starting at -1, which is the byte offset in the types table of a null-terminated list of type indexes. The list will be at TTBase+1 for -1, at TTBase+2 for -2, and so on. Used by the runtime to match the type of the thrown exception with the types specified in the "throw" list. Back-end generates a switch statement that checks for that particular value.
next	Signed offset, in bytes from the start of this field, to the next chained action record, or zero if none.

All fields are compressed using the LEB128 encoding (described in Section 6.8, "Decoding exception records" on page 28).

The exception specification acts very much like a catch clause: when the thrown exception violates the exception, unwind pass 1 indicates that a handler was found, and pass 2 transfers control to a handler in the generated code.

Runtime Action: The exception handling library will check if the thrown exception is within the list of possible exception types. If not, it will set the landing pad "switch selector" argument to the indicated negative value.

Front-end Generated Code: The generated code for an exception-specification handler will check if the switch selector value is zero. If not, exception is propagated out.

```
S1:
      // Corresponding to an XHJP statement
      switch(switchSelector)
      {
            case NEGATIVE_SWITCH_VALUE: goto H1
      }
X1:
      [RESX]
H1:
      __CallUnexpected();
```

Note that after inlining of a function with an exception specification, some code may actually use the switch selector value in the calling function, if it does not match the negative value specified in the action record and control falls through the `XHJP` statement.. In other words, the [RESX] operator above may actually branch to an enclosing XHJP statement.

6.6. Type table

The type table is an array of uncompressed GP-relative displacements to `__type_info` objects describing C++ types. Filter values in a catch-clause record are indexes into this array. This type is generated by the back-end.

For instance, in a code containing catch(A), catch(B) and catch(C), the table may contain:

- The `__typeinfo` for A in the first word before TTBase, corresponding to filter value 1.
- The `__typeinfo` for B in the second word before TTBase, corresponding to filter value 2.
- The `__typeinfo` for C in the third word before TTBase, corresponding to filter value 3.

6.7. Exception Specification Table

This table contains lists of types used in exception specifications. The lists are null-terminated sequential runs of compressed indices into the type table. The table is generated by the back-end.

For example, given the type descriptions of Section 6.6, we may encode two functions with throw(A, B) and throw (C) using the following bytes:

```
0,
1, 2, 0     // throw(A,B)
3,0         // throw(C)
```

In an action record, the exception specification is encoded as the offset from the beginning of the table. `throw(A, B)` would have a filter value of -1, and `throw(C)` would have a filter value of -4.

Note that the type indices may be longer than one byte (they are LEB128 encoded).

6.8. Decoding exception records

As noted in the sections on action records and the unwind tables, almost all fields in the exception tables are stored in compressed format to save space. The format used is Little-Endian Base 128 (LEB128). This is the same compression scheme as that used in the DWARF object module format.

To decode LEB128:

- Collect a run of bytes with the high bit set followed by a single byte with the high bit clear. (A most-significant bit of 0 is a sentinel that indicates the end of a LEB128 value).
- Discard the high bit of each byte. Now *N* 7-bit bytes remain.
- Form a 7*N*-bit binary number from the bytes in little-endian order (the last byte is most significant).

- If the value is signed, interpret it as a twos-complement number with the most significant bit as sign.

To encode LEB128:

- Divide the value into groups of 7 bits, beginning with the least significant bits (little-endian order).
- If the value is unsigned, zero-extend the final group to 7 full bits. If the value is signed, sign-extend the final group to 7 full bits.
- Discard all groups of "leading" zeroes, but keep at least the first (least significant) group if the number is 0. If the value is signed, discard all groups of redundant "leading" ones (sign extension), but be sure to keep at least one set sign bit (see the example for -128 below).
- Mark all but the last group with a most-significant bit of 1; mark the last group with a most-significant bit of 0.

Examples (sentinel bits bold, sign bits underlined):

LEB128-encoded bytes	binary value	value (signed)
00000000	0000000	0
00111111	0111111	63
01111111	1111111	127 (-1)
10000000 **0**0000001	00000010000000	128
10000001 **0**0000001	00000010000001	129
10000000 **0**1111111	11111110000000	16256 (-128)
10001000 **0**0001100	00011000001000	1544
10000000 **0**1000000	10000000000000	8192 (-4096)
10001010 **1**0000101 **0**0000011	000001100001010001010	49802

7. Ucode-2 Interface

Ucode2 defines the following specific functions and operators for use in exception handling, as defined in "*Ucode-2 proposal to support C++ EH on Tahoe*" [5]. Section numbers refer to the .

7.1. Section 3.1.2.1: Link Unit Minor Attributes

The U2_Set_Link_Unit_Cplusplus_EH operation specifies whether this link unit is compiled with C++ exception handling enabled; and, if so, which exception handling model will be used. If exception handling is enabled the code generator will emit exception handling tables into the object file for use by the C++ runtime system.

The cplusplus_eh argument specifies the exception handling model to be used. If the value is *no* (the default), exception handling is not enabled and the other arguments are ignored. If the value is *interpretive*, then interpretive exception handling is enabled. If the value is *compiled*, then compiled exception handling is enabled. Most targets will support one exception handling model or the other, but not both.

The flagbits and version arguments specify values defined and used by the interpretive C++ Runtime System. For the compiled model their values must be 0.

Operation:

```
Set_Link_Unit_Cplusplus_EH
      link_unit:<tag>
      cplusplus_eh:<no/interpretive/compiled>
      flagbits:<>
      version:<>
```

Concrete Interface:

```
typedef enum {
      U2_CPLUSPLUS_EH_no_k,
      U2_CPLUSPLUS_EH_interpretive_k,
      U2_CPLUSPLUS_EH_compiled_k,
} U2_CPLUSPLUS_EH;

void U2_Set_Link_Unit_Cplusplus_EH(
      U2_TAG            link_unit,
      U2_CPLUSPLUS_EH   cplusplus_eh,
      U2_INTEGER        flagbits,
      U2_INTEGER        version
);
```

7.2. Section 3.8.5: Landing Pads

A landing pad is defined by a `PAD` instruction and represents a gateway to a block of code generated by a front end to support exception handling. For each declared landing pad, there must be exactly one associated `PAD` instruction. The runtime system may transfer control to a landing pad as part of the exception handling process. This is the only legal way to reach a landing pad. If control falls through to a landing pad, the runtime behavior is undefined. Code following the landing pad is responsible for such activities as data cleanup (e.g., invoking destructors) and/or transferring control to code that handles the exception. Every call that may throw an exception must have exactly one landing pad.

A single landing pad can be specified as the alternate return of a call by using the `Set_Call_Info_Alternate_Return` operator. This indicates that during the execution of a called routine, it is possible to return to the specified landing pad instead of the instruction immediately following the call. This will happen if an exception propagates from the callee to the caller.

A landing pad must be declared between the `Begin_Code` and `End_Code` operators of the routine from which it will be referenced, and it must be declared prior to the `PAD` instruction that defines it.

A landing pad is analogous to an entry point, except that while the latter is used when calling a new routine, the former is used by the runtime system to return control to a point in a routine that has already been called. `Declare_Landing_Pad` is analogous to `Declare_Entry`; the `PAD` instruction is analogous to the `ENTRY` instruction.