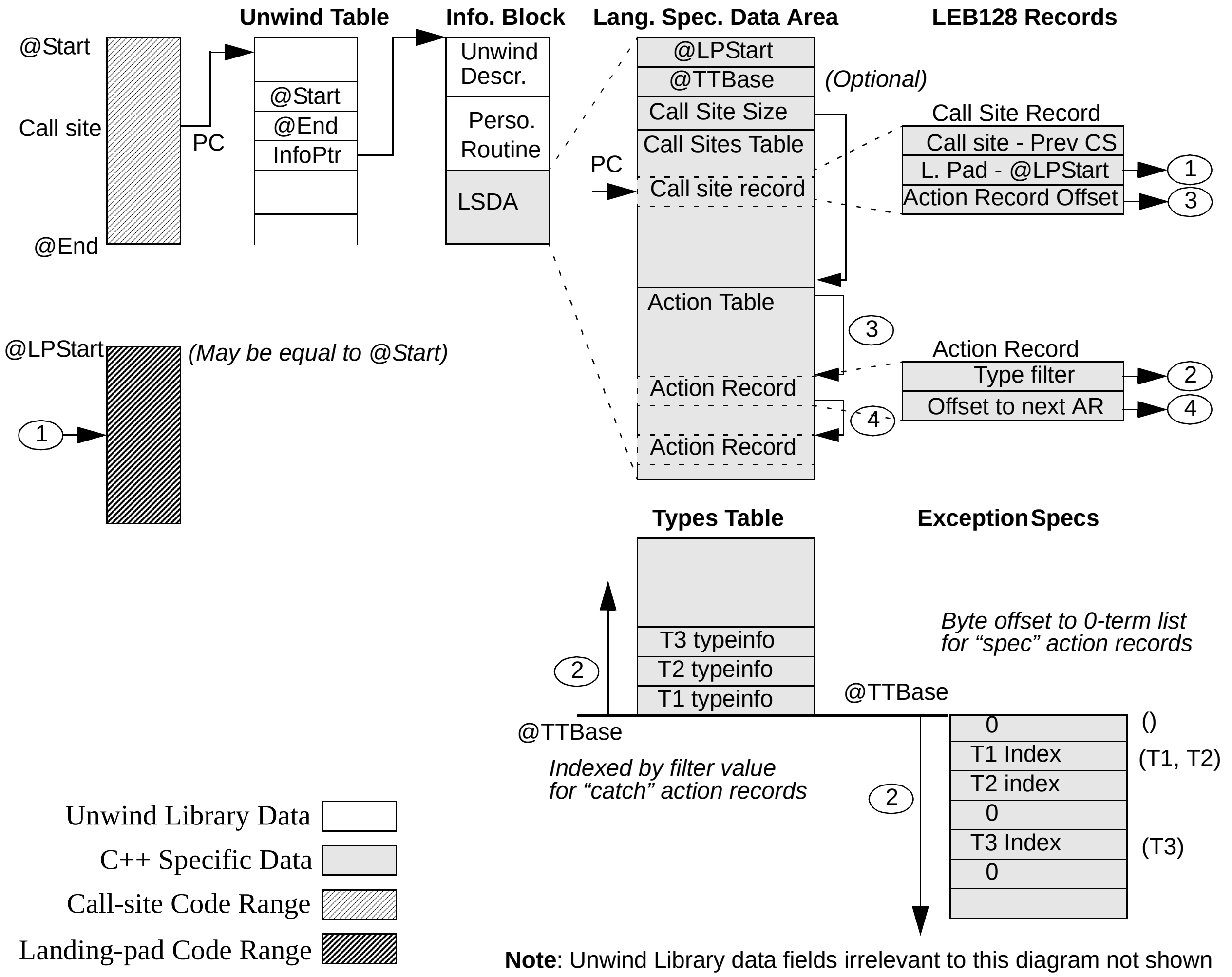
7. Exception Handling Tables

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.

7.1. Overview

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



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

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

7.2. 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.

7.3. 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 7.8 on page 33.

7.3.1. 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 7.4 on page 30. 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.

7.3.2. 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. The IA64 architecture specifies that a single call will execute in a given 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 7.8, "Decoding exception records" on page 33).

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 forbids 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().

7.3.3. 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 7.4 on page 30.
- Exception specification, described in Section 7.5 on page 31.

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 7.8, “Decoding exception records” on page 33). 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.

7.4. Catch Clause

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

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 7.8, “Decoding exception records” on page 33).

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.

7.5. 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 __cxa_unexpected 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 7.8, “Decoding exception records” on page 33).

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 an appropriate negative value. If not, the exception is propagated out.

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

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 "default" exit.

7.6. 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.

7.7. 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 7.6, 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 negative of 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).

7.8. 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

8. Unwind Library Interface

This section defines the Unwind Library interface as exposed for the common C++ ABI.

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.

Last, 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.

8.1. Design Discussion

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, stack is unwound while the exception propagates, but it is expected that each runtime knows if it wants to catch the exception or let it go through. This task is delegated to the personality routine, which is supposed to act properly for any type of exception, either "native" or "foreign". Some guidelines for "acting properly" are given below.
- During "forced unwinding", on the other hand, an external force is driving the unwinding. For instance, this can be the longjmp routine. This external force, not each personality routine, knows when to stop unwinding. The fact that a personality routine is not given a choice about whether unwinding will go further or not is indicated by the EH_FORCE_UNWIND flag.

To accomodate for these differences, two different routines are proposed. _Unwind_RaiseException performs unwinding under the personality routines control. _Unwind_ForcedUnwind, on the other hand, performs unwinding, but gives the "external force" 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 force supersede the defaults of the personality routine.