Exception Handling on HP-UX

Runtime Architecture Supplement

Version 2.5 Draft
September 30, 1999

This document describes the API-level architecture of the exception handling mechanism for the HP-UX operating system on the iA-64 architecture. The high-level design of the exception handling mechanism is independent of any particular operating environment or programming language, and is described in the common Runtime Architecture document.

1. The Exception Handling Process

When an application encounters an exception, it makes that exception known to the system by calling an entry point in a language-specific runtime library, which then calls the exception dispatcher in the system library. The exception dispatcher first searches for an exception handler, then prepares to transfer control to the handler by removing stack frames that are above it and executing any cleanup actions necessary for each frame. Finally, the dispatcher transfers control to the handler. These phases are illustrated in Figure 1.

The system libraries are insulated from the language-specific details of exception handling through the use of personality routines, which are supplied by each language's runtime support library. The personality routines are responsible for interpreting the exception handling information generated by the compiler for each procedure, for controlling the operation of the exception filters ("exception declarations" in C++), and for performing cleanup operations as stack frames are removed from the stack.

For the purposes of this document, system libraries are defined to include the C library, the unwind library, and the language-independent part of the exception handling library. These latter two are components of the compiler library, libcl.

1.1 Raising an Exception

To raise an exception, the application calls an entry point in the runtime support library for the appropriate language, passing that entry point some indication of the type of exception and any details. The format and contents of

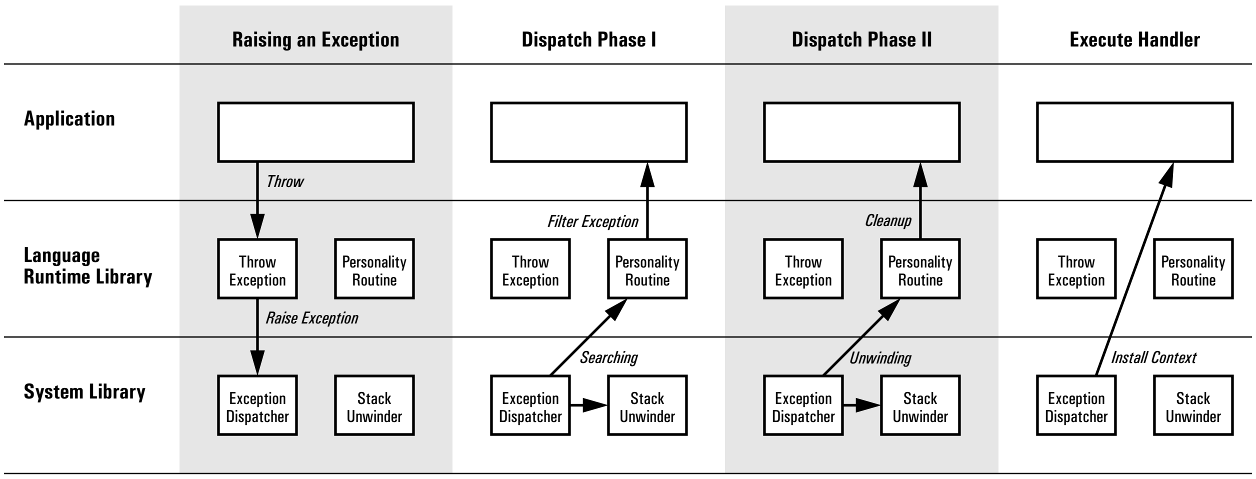

Figure 1. Exception Handling Phases

this information are language specific, and are therefore not specified by this document, except that it must pass sufficient information for the runtime support library (or compiler-generated code) to identify an exception handler capable of processing the exception.

The runtime support library then must encapsulate the exception information into a single object, and call the exception dispatcher entry point, `_RaiseException`, in the system library, passing a pointer to this object and an identifier that uniquely classifies the exception as belonging to a particular language or subsystem. These two values are not used directly by the exception dispatcher, but are passed to the runtime support library's personality routine during the search for an exception handler.

The exception class identifier is a single 64-bit value, designed to be chosen by each language or subsystem in such a way that does not introduce the need for any centralized administration of tokens. Each language or subsystem that implements an exception handling mechanism should choose a 64-bit value that is derived from its vendor and product name. This identifier is used so that an application can mix code from different compilers without conflict among the different exception handling mechanisms. For example, the HP ANSI C++ compiler might use the eight-byte string "`HPaC++01`".

The `_RaiseException` routine may return if no handler is found or if a problem was encountered unwinding the stack. The runtime support library must be prepared to handle these conditions.

1.2 Searching for an Exception Handler

The exception dispatcher begins searching for an exception handler to process the exception by obtaining an unwind context record for its caller (which is the runtime support library routine called by the application). An unwind context record contains the machine state as it existed at the last point in time within any particular stack frame. Unwind context records are constructed as required for each frame on the stack, starting from this first context record, proceeding one at a time down the stack towards the bottom. Each step down the stack is performed by the entry point unwind_step in the system's unwind library.

As the exception dispatcher visits each stack frame, it determines the procedure active for that frame, and checks the unwind table entry for that procedure. If the unwind table entry indicates that the frame requires any Phase I exception handling processing (i.e., the "UNW_FLAG_EHANDLER" flag is set for that procedure), the dispatcher invokes the personality routine indicated by the unwind table.

The personality routine is handed several parameters:

- A version parameter. This parameter identifies the version number of the personality routine interface, so that future extensions may be made to this interface without breaking compatibility.
- A phase parameter. This is the constant EH_SEARCH_PHASE, to indicate that dispatch phase I is in progress.
- The exception class identifier and the pointer to the encapsulated exception object. These are the two values that were passed to _RaiseException. If the exception class identifier does not match a language or subsystem that the personality routine is designed to work with, the personality routine is expected to ignore the exception.
- The pointer to the context record. This is passed so that the personality routine can obtain needed values from the stack frame. This is passed as an opaque object, and APIs are defined to access the information contained in the record.

The personality routine can also obtain the following information from the context record:

- The address of the language-specific data area. This value is obtained from the unwind table, and is passed to the personality routine so it can search the compiler-generated information for guarded regions in the procedure, and determine whether there are any exception filters that need to be executed to check the exception object.
- The pointer to the unwind information record. This is passed so that the personality routine can obtain information about the procedure (e.g., the address of the first instruction in the code may be required for translating PC offsets in the exception handling tables). This is passed as an opaque object, and APIs are defined to access the information contained in the record. Note that, by definition, each unwind table entry corresponds, one-to-one, to a procedure. If a compiler generates disjoint code regions for a procedure, it must generate separate exception handling information for each region.

Exception filters may be compiled code or they may take the form of an interpreted code in the exception handling tables. In the former case, the code must be generated so that the filter will execute correctly outside its normal stack frame, since the dispatcher is stepping down the stack in search of an exception handler, without actually removing any stack frames from the stack. The interface between the personality routine and the filter is private to the language runtime, and is not architected here. Nevertheless, it is suggested that the personality routine invoke the filter as a nested procedure, with at least two parameters: a pointer to the normal stack frame, and a pointer to the exception object.

The personality routine may return any of the following status codes to the exception dispatcher.

- If it does not find an exception handler to process the exception, and searching should continue, it should return the value EH_CONTINUE_SEARCH.
- If it finds an exception handler that can process the exception, it should return the value EH_HANDLER_FOUND.
- If it determined that no exception handler is necessary, and execution should continue from the point of exception, it should return the value EH_DISMISS_EXECUTION.
- If an error occurs, it may return an error status code. **Error status codes are not yet defined.**

Either the personality routine or filters may encounter an exception, and raise a nested exception. **The behavior in this case is not yet specified.**

1.3 Executing Cleanup Code

If a handler is found during dispatch phase I, the dispatcher will immediately begin phase II. It starts from the top of the stack again, and steps through the stack frames as in phase I. As it visits each frame, it checks the unwind table for the "UNW_FLAG_UHANDLER" flag that indicates that phase II processing is necessary. This bit should be set for any procedure that has cleanup actions that need to be performed before a stack frame is destroyed.

During phase II, the personality routine is handed the same parameters as during phase I, with the following exceptions:

- The phase parameter is set to the constant EH_CLEANUP_PHASE, to indicate that dispatch phase II is in progress.
- An additional flag, EH_HANDLER_FRAME, is also passed when a personality routine is called to visit the frame in which a handler was found during phase I. This flag is or'ed in to the phase parameter.
- An extra parameter, frame_handle, is passed. This handle is a reference to the handler frame found in the phase I. The usage of this parameter is discussed below.

Cleanup actions may be compiled code, or they may take the form of an interpreted code in the exception handling tables. The interface between the personality routine and the cleanup code is private to the language runtime, and is not architected here.

A recommended mechanism for compiled cleanup code is to generate out-of-line cleanup code that executes in the frame of the procedure with which it is associated. The personality routine can arrange to install the frame's context to execute the code in its proper stack frame. This code will directly execute the required cleanup actions, then exit via either of two paths. If the exception handler is in the same stack frame, it can transfer directly to the exception handler, and exception processing is complete. To continue the exception dispatch, it must return control to the exception dispatcher via the _ResumeUnwind entry point.

The compiler can generate cleanup code in this scheme so that it is statically determined whether or not the cleanup code will fall directly through to an exception handler, possibly by cloning cleanup code. By recording suitable information in the exception handling tables, the personality routine can then determine that cleanup and transfer of control to the exception handler can be done in a single step. For a more detailed discussion of this scheme, see Appendix A, "Using Landing Pads to Improve Optimization."

Because the dispatcher's local state is lost when the cleanup code is entered, the cleanup code must provide a pass-through mechanism for three 64-bit values: the exception class, the pointer to the exception object, and the reference to the handler frame. These values are passed according to a special "landing-pad" calling convention defined in Appendix B, "Conventions for Passing Landing Pad Parameters."

An alternate mechanism for compiled cleanup code is to generate the cleanup code as a nested procedure. The personality routine can then simply call the cleanup code, passing it the address of the procedure's stack frame as a form of static link. In this scheme, the cleanup code can simply return to the personality routine, which can then return to the dispatcher to continue the dispatch process.

The personality routine may return any of the following status codes to the exception dispatcher.

- If the frame containing the exception handler has not been reached, and unwinding should continue, it should return the value EH_CONTINUE_UNWIND.
- If the frame contains cleanup code that has been compiled to execute in the frame, it should modify the context so that the cleanup code will execute, and return the value EH_INSTALL_CONTEXT. The cleanup code may transfer control directly to the exception handler, if the handler is in the current frame, or it will resume the unwind process by calling the _ResumeUnwind entry point.
- If the frame containing the exception handler has been reached, (i.e., the flag EH_HANDLER_FRAME was set) it should modify the context as required for running the handler, and return the value EH_INSTALL_CONTEXT.
- If an error occurs, it may return an error status code. **Error status codes are not yet defined.**

Either the personality routine or cleanup code may encounter an exception, and raise a nested exception. **The behavior in this case is not yet specified.**

1.4 Invoking the Exception Handler

When a personality routine returns the code EH_INSTALL_CONTEXT during phase II, the exception dispatcher uses the context for that frame to restore the correct machine state and transfer control to the exception handler. The interface to the exception handler is private to the language runtime, and is controlled by the personality routine through the context record and the communication buffer.

1.5 Single-Phase Dispatching

Some subsystems may prefer to search for exception handlers and execute cleanup code in a single pass. This is possible if the exception handling semantics do not provide for the possibility of resuming execution from the point of the exception.

These interfaces are not yet specified.

2. ABI-Level Interfaces

The interfaces between the application and the language runtime support library are language-specific and are not specified in this document.

The interfaces between the language runtime support library and the system libraries are for raising an exception and invoking the personality routine.

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

int (*personality_routine)(
        int                 version,
        int                 phase,
        UInt64              exception_class,
        void *              exception_object,
        UnwindContext *     context,
        void *              frame_handle
);

int _ResumeUnwind(
        UInt64              exception_class,
        void *              exception_object,
        void *              frame_handle
);
```

The following constants are used for the phase parameter to the personality routine:

```
#define EH_SEARCH_PHASE             0x01
#define EH_CLEANUP_PHASE            0x02
#define EH_HANDLER_FRAME            0x10
```

The following constants enumerate the return values from the personality routine:

```
#define EH_CONTINUE_SEARCH          1
#define EH_HANDLER_FOUND            2
#define EH_DISMISS_EXCEPTION        3
#define EH_CONTINUE_UNWIND          4
#define EH_INSTALL_CONTEXT          5
```

The interfaces between the exception dispatcher and the stack unwind component of the system libraries are a combination of ABI-level interfaces and system-level interfaces. The ABI-level interfaces for obtaining a context and stepping through stack frames, and for accessing and modifying context records, are specified separately in the document “IA-64 Stack Unwind Library for HP-UX.”

Appendix A. Using Landing Pads to Improve Optimization

To do its job effectively, an optimizer needs considerable freedom to rearrange code within a procedure. Since any arbitrary procedure call may trigger an exception, however, the optimizer must be careful about how code is moved within a guarded region. If, for example, a store instruction is moved from above a procedure call to below the procedure call, the optimizer must ensure that the store is executed even if an exception is raised during that call. One technique that allows the optimizer to perform such code movement correctly involves the use of “landing pads.”

A landing pad is an out-of-line sequence of code that the compiler generates for each call site within a guarded region. The landing pad contains any cleanup code necessary if an exception is raised during the associated call, and, after cleanup has been performed, transfers control either to an exception handler or back to the exception dispatcher. The landing pad also provides a place for the optimizer to place “compensation code”—clones of any code sequences that were moved below the call, but which must be executed in the event of an exception. Even if the compiler front end can determine that the cleanup actions at two call sites are identical, it must provide two separate landing pads, because the optimizer may need to place different compensation code in each. The common cleanup code, however, does not need to be generated twice, as one landing pad may jump to the cleanup code in the other.

As an example, consider the nested C++ try/catch blocks shown below:

```
try {
        try {
                proca();            /* to Landing Pad A */
                procb();            /* to Landing Pad B */
        }
        catch ( A ) { }
        catch ( B ) { }
        procc();                    /* to Landing Pad C */
}
catch ( C ) { }
catch ( D ) { }
```

The compiler would generate a separate landing pad for each of the three call sites in this example, as shown in Figure 2. Each landing pad contains a placeholder for compensation code to be inserted by the optimizer, any cleanup code necessary for an exception raised at that point in the try block, and logic for transferring control after the cleanup.

Note *If the compiler puts common cleanup code in one landing pad, and jumps to that code from another landing pad, it must be careful to place separate*

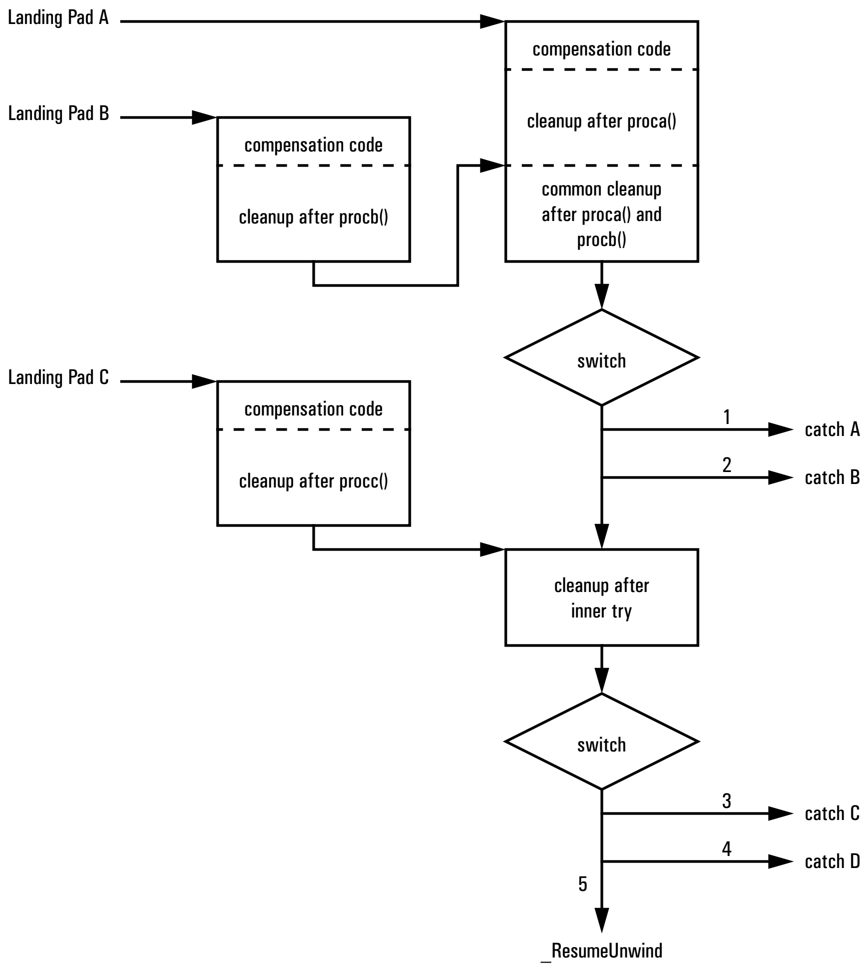

Figure 2. Landing pads for example with nested try blocks

labels for the compensation code and the common cleanup code. If, for example, the region labelled "cleanup after proca()" in landing pad A is empty, the compensation label and the common cleanup label will actually refer to the same location before the optimizer runs. When the optimizer places compensation code into the landing pad, it must move the common cleanup label down, so that the compensation code for landing pad A does not run when landing pad B jumps to the common cleanup code.

The example also shows how control is transferred after the cleanup code executes. The personality routine can determine, prior to jumping to the cleanup code, where control should go after the cleanup code. If for example, and exception is raised during a call to proca(), the cleanup code may transfer control directly to one of the two catch clauses in the inner try block, or it may propagate the exception to the outer try block, from which control may go to either of the catch clauses in the outer try block, or back to the exception dispatcher via _ResumeUnwind. Based on the exception being thrown, and the language-specific action tables generated, the personality routine can determine which of these five paths should be taken, and can pass an integer parameter between 1 and 5 to the landing pad. When control reaches the switch at the end of the landing pad, that parameter can direct the flow of control appropriately.

Appendix B. Conventions for Passing Landing Pad Parameters

The interface between a personality routine and a landing pad uses a special software convention for passing the parameters needed by the landing pad. In general, the convention allows for up to eight 64-bit integer parameters. The first three parameters are specified by the convention; the number of additional parameters and their usage are specific to the particular personality routine (and to the landing pads supported by that personality routine).

The first three parameters are those that must be passed from the personality routine, through the landing pad, to the __ResumeUnwind entry point. These are the exception class, a pointer to the exception object, and the frame handle, respectively passed in general registers r15, r16, and r17.

Additional parameters are passed in successive general registers, beginning with r18, continuing through r22 if necessary. In the example scheme shown in Appendix A, a fourth parameter would be used for the value used to direct the flow of control after the cleanup code executes.