Proposal for a new calling convention within the C language

Jacob Navia

WG14 document number n2285

August 13, 2018

1 Introduction

In this document a new calling convention for specially marked functions is proposed. This calling convention standardizes error handling, also called ”out of band” processing.

1.1 Motivation

Error handling within the current C language framework is done in many cases like this:

```
int result;
int status = doSum(int *data,&result);
if (status) { // Zero is OK, non-zero is an error
    // Error handling
}
// work with "result"
```

Problems arise when in the same API or even in the same library, another function requires that you write:

```
int result;
int status = doSum(int *data,&result);
if (! status) { // 1 is OK, zero an error
    // Error handling
}
// work with "result"
```

This is not an exaggeration in any way. In their paper, the researchers Kang, Ray, and Jana [1] write:

> For example, in OpenSSL, for some API functions 0 indicates errors, while for others 0 is used to indicate success.

There are other problems, subtle and not so subtle, like writing: `if (status != OK)` with a wrong value of `OK`, etc.

A standard way for testing for errors would eliminate these problems, providing a simple, standard way of writing error handling.

1.2 Objectives

1. Clarity. Standardization makes it easier to understand at a glance what the code is doing. There is a single way of doing out of band processing, independent of what library you are using, eliminating the need to learn for each function you call that could return an error, how that function returns the error information.

2. Efficiency. The proposed solution needs at most a few machine instructions (testing a bit and a conditional jump) to implement. Since there is no need to pass a reference to the result, we can write again:
   `int result = doSum(data);`
   instead of
   `int status = doSum(data,&result);`

   The normal case is simplified and faster, no need to prepare the "result" variable to receive the result. The compiler can store that variable in a register without extensive use analysis: its address isn't taken anywhere.

3. Compatibility with existing binaries. Code compiled without this feature will continue to work and interact with new code without any modifications. Old code can go on calling new code, this feature is completely transparent to it since it just doesn't test the "exception" bit.

4. No special hardware requirements. This feature can be implemented in any machine with more than 1 register. Alignment requirements are minimal and at most a single RAM cell is needed. No wasted space.

5. Simplifying automatic analysis. Automatic tools have an almost impossible job when trying to determine the failure set of a function. It is possible to develop with a lot of effort an algorithm that tries to infer the failure set of a function from the usages overall in the code, but even with the best efforts of [1] the success rate is well below 100%.

   It is interesting to note that in that paper, the authors write:

   > Manually generating error specifications is error-prone and tedious. It is particularly hard for low-level languages such as C that do not provide any specialized exception handling

mechanisms and contain only generic channels of communication (e.g., return value, arguments passed by reference) between the caller and callee functions. Therefore, the developers of each API must decide individually which values indicate failure. Consequently, the error specifications are function-specific and may vary widely across functions. While there are values that intuitively indicate errors, such as negative integers or NULL pointers, such conventions do not always hold and thus will result in highly inaccurate specifications.

The C standards committee is in an unique position for solving this problem.

2 Conceptual description

In this section, the concepts needed for this feature to work are explained without any new syntax. The purpose here is to convey a clear view of *what* is being done, without bothering at first about *how* this is done.

2.1 The exception return type

Following the proposal of Herb Sutter [2], we introduce the concept of a function that returns a union of its result type, and an error return. This conceptual union could be defined in standard C as follows for a function that returns values of type T:

```
struct result {
    volatile _Bool failed:1;
    union {
        uintptr_t error;
        T result;
    };
};
```

The single bit `failed` is set when the function returns an error, unset otherwise. If the bit is set, only the ”error” member of the union is valid. The ”result” member is invalid, and any usage of it invokes undefined behavior. If the bit is unset, any usage of the error member is equally invalid.

In case of an error return, the called function can return either a pointer to an error description or error-frame, or simply an integer error code.

To discriminate between integers and pointers we require that all pointers must point to data aligned at an even address, and that all error codes should be odd. We use then, the least significant bit of the error to discriminate between a pointer to some data containing further information about the error (`error` is even), and an integer that directly encodes information about the error (`error` is odd) [1].

It is expected that the `failed` bit is stored in an unused CPU flag, and the union itself can be returned in the normal register used for returning pointers or integers [2].

2.2 Accessing the members of the union

The programmer can access the members of this conceptual union in read/write mode within the called function, and in read only mode (as if declared `const`) in the calling function. The name of the union is the same as the name of the function and it behaves as if a declaration of a union with the same name as the function would have been seen in the program text.

Example: For a function called "process":

```
_Exception double process(double *data);
```

The associated return value could be defined as follows:

```
struct process {
    volatile _Bool failed:1;
    union {
         uintptr_t error;
         double result;
    };
};
```

[1]Note that a return of a zero error code is interpreted as a NULL pointer since zero is even. This can be used to convey a general, not further specified error condition.

[2]Most computers nowadays have a carry flag that can be used for this. Otherwise, an unused scratch register can be used to hold the exception bit. The carry flag is preferred so that code compiled with different compilers remains compatible.

This definition is in scope within the "process" function[3]. For functions that call "process", it is defined in the same way, but prefixed with the `const` qualifier. There is no sense modifying the error code you receive, so any assignments to the values of this union outside "process" are a constraint violation [4].

2.3 The return exception statement

When a function detects an error that makes continuation impossible, it returns with a special return statement that sets the `failed` bit of the union. It is assumed that before this statement, the program sets at least the `error` member to some descriptive value or that builds an error-frame containing more detailed information about the error. Since error returns are now explicit, the set of failure modes of a function is easy to determine from the program text and simplifies the work of automatic tools.

If no statement directtly or indirectly sets the field `error` of the union, its value defaults to zero.

2.4 Testing for error returns

The calling function must test right after the call the state of the `error` bit. This bit can be destroyed by any statement that happens to use the CPU flag or the register used for storing it, so it is necessary to test it right after the call. The consequence of this is that functions that return exceptions can't be used within complicated expressions where several function calls exist. They can't be used either within complicated expressions since their usual return value could be invalid. For instance:

```
if (doSum(data)/23 < threshold) { } //WRONG
```

If `doSum` can return an exception, the value returned could be invalid (or even worst, it could be an error code!) and should never be used in further calculations. But this is already the case in current code that tests for a status return, so this is not a real problem.

[3]It is legal to define a structure with the same name as the function in C.

[4]Of course the programmer can assign the errorcode to a suitable variable and use it further with no restrictions. Like all const qualified data

2.5 Controlling the behavior of this feature

1. The user can request that this feature should be turned off. Many reasons come to mind why this could be necessary in some circumstances:

   - Processing speed needs to be maintained at all costs, and the programmer is confident that errors can't occur.
   - The priority of the calculations/processing being done is low, and it is not worth to follow any errors. If we are calculating the color of a pixel in a full HD image at 32fps, if one pixel is wrong hasn't a big importance and error handling would slow down the display.

   Obviously, disabling error handling is a radical measure that should be used with utmost care. It is up to the programmer writing the code to decide if this is useful or not, depending on the context.

2. A function can pass an exception return to higher level functions with the following code:

   ```
   // calling function is called "process"
   // called function is called "doSum"
   if (doSum.failed) {
       process.error = doSum.error;
       return _Exception;
   }
   ```

   The directive `_Pragma(_Exception,pass,on)` will provoke making this code the default, i.e. the above code will be automatically added after all function calls that use this calling convention unless the programmer explicitely writes exception handling code.

   The directive `_Pragma(_Exception, pass,off)` will suppress the effects of a previos `on` directive. Other variants may exist, like

   ```
   // Save the current state and set it to "on"
   _Pragma(_Exception, pass, push on)
   // Pop the previous state
   _Pragma(_Exception, pass,pop)
   ```

   if the compiler supports storing these directives in a stack-like fashion.

3. If the `main` function executes a `return _Exception`, the program behaves as if the function `abort()` would have been called. It is expected that the runtime system shows a message to the user through standard error or other means, to inform him/her of an abnormal termination of the program.

4. The programmer can request that within a scope, all exception returns are logged into a text file that contains the error return, and the coordinates of the error (the contents of `__FILE__` and `__LINE__` macros). This is an optional feature.

   Any error condition that happens during the writing of the log file disables this feature permanently during the rest of the program.

3 Syntax

A single keyword is introduced: `_Exception` that has different meanings, depending on context.

3.1 Declaration/Definition of functions that return exceptions

A new type qualifier is introduced: `_Exception` that specifies that the type that follows is the `"result"` member of the return union.

```
_Exception double fn(double arg1); // Declaration
_Exception double fn(double arg1) { } // Definition
```

As always, a function declared with the `_Exception` type qualifier should be also defined with the same characteristics. If any mismatch occurs, the program is ill formed.

3.2 Accessing the fields within the called function

The programmer has access to the fields of the union with the syntax:

```
function_name . field
```

This has the advantage of eliminating any need for new keywords. For instance:

```
_Exception double fn(double *data,int size)
{
    // An error condition is detected
    buffer = malloc(size * sizeof(double));
    if (buffer == NULL) {
        fn.error = ERROR_NOMEMORY; // Set the error member
        return _Exception; // Set the "failed" bit
    }
}
```

If the function executes a `return _Exception` without setting the `error` field, the returned value defaults to zero.

3.3 Accessing the fields in the calling function

In the calling function, the fields of the union are accessible right after the call statement. The volatile `failed` bit is tested immediately after the call. The other fields are longer lived and can be tested or assigned to a variable in principle until the end of the current scope. There is however only one place reserved for this set of data, that will be overwritten at each call.

In the example below, a function calls a procedure to add some numbers, and if there is an overflow, it tries adding up the numbers in smaller chunks. This example is obviously for explaining purposes only, and doesn't show any real life situation. One of the practical uses of out of band processing is to try to change the algorithm to avoid the error, i.e. adaptive code.

Note too that after the first failure, the code doesn't analyze the (possible) second error code and just aborts if any failure occurs.

The sequence for testing errors should look like this:

```
_Exception double doSum(double *data,size_t n);

_Exception int processData(double *data)
{
    // ...
    long double sum = doSum(data,CHUNKSIZE);
    if (doSum.failed) { // Test if "doSum" failed
        if (doSum.error == ERROR_OVERFLOW) {
            sum = doSum(data,CHUNKSIE/2);
            if (doSum.failed) {
                abort();
            }
            sum += doSum(data,CHUNKSIZE/2);
            if (doSum.failed) {
                abort();
            }
        }
        else return _Exception;
    }
    // If we arrive here "sum" contains the correct value
}
```

3.4 Controlling exceptions

- Turning this feature on/off. The pragma `_Exception(on/off)` will turn on or off this feature.

  ```
  #pragma STDC _Exception(on)
  ```

  This pragma will turn on this feature, and the keyword `Exception` will be recognized as such. This pragma is valid only at the global scope [5]. In a similar vein, the pragma

  ```
  #pragma STDC _Exception(off)
  ```

  will turn off this feature. Any code that uses it will be still recognized but:

  – Any expression (including assignment expressions) where this syntax appears will be understood as the empty statement.

  – Any `return _Exception` statement will be understood as the empty statement.

  – All tests for the `failed` bit will return zero.

[5]It would be very complicated for the compiler to figure out what will be generated if that pragma appears in the middle of an if (fn.failed) statement, for instance. To avoid unnecessary complications, it is wiser to limit this possibility to the global scope, i.e. outside any function definition.

4 Impact to the standard library

In principle, all functions of the standard library should use this way of returning errors. This would make it much easier to use them, instead of learning for each function how errors are returned.

Compatibility with older code is retained if the existing error result convention is not changed: all functions of the library would retain their current method for returning errors but in addition to that, they would setup the error code and the `_Exception` bit for use with the syntax proposed here. If they are called from old code, the `_Exception` bit will be ignored and since the result is the same, they would continue to work with old code *and* they would allow this calling convention to be used in new code.

Obviously, since the result of an exception return is a *union* of the result type and the error code, for functions that have already a *fixed* error return value, for instance NULL, or `-1` the only change in the functions is to execute a `return _Exception` instead of the normal return. Anything else would break binary compatibility with older code.

For functions that return a non-zero value in case of error (for instance `fflush`, `remove`, `setvbuf`, `rename` and many others) they could be freely modified to return more detailed information about the failure in the error code using this feature. Also, functions like `fprintf`, that return a negative value for error could use a range of negative error codes to remain binary compatible with old code and at the same time use this feature.

5 What to do with errno.h?

The whole errno system should stay as it is, and this proposal would require a new header file with the new (odd) error codes using new names[6]. It is also possible to develop a "category" classification for errors: I/O, hardware, ressource exhaustion, network, software, thread/multi-tasking, for instance. This would simplify error handling.

The standard committee would reserve several bits for error codes, and the rest would be free for the programmer to use. Having portable errors would allow for more portable code.

[6] Today, we aren't constrained by scarce storage and could start using a bit longer and more descriptive error names.

References

[1] Yuan Kang, Baishakhi Ray, Suman Jana, *APEx: Automated Inference of Error Specifications for C APIs*. 31st IEEE/ACM International Conference on Automated Software Engineering, 2016.

[2] Herb Sutter *Zero-overhead deterministic exceptions: Throwing values* Document Number: P0709 R0 SG14 2016