Out-of-band bit for exceptional return and errno replacement

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 N 2361

March 30, 2019 v 1

Out-of-band bit for exceptional return and errno replacement
interface proposal for future C

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose a mechanism for out-of-band return of exceptional conditions that are designed, among other things, to replace **errno** and similar thread-local mechanisms for error return, and to allow C libraries and other programming languages to implement value based exception return on top of it. Our method is designed to be conservative: there is no need of ABI replacement for platforms, only amendment, and existing executables and compiled libraries remain valid.

1. INTRODUCTION

Recently, WG14 and WG21 have seen several papers that propose improvements to C's and C++'s error handling capacities. The main motivations for these seem twofold:

Value based exceptions. After the failure of the type based dynamic exception model [Gregor(2010)], modern programming languages seek means to express simple value based propagation of error (or exceptional) information and conditions [Sutter(2018)]. A new mechanism is sought that should be efficient, independent of dynamic choices and in line with modern optimization techniques [Douglas(2018a)] [Renwick et al.(2019)Renwick, Spink, and Franke].

C library error handling. The C error handling mechanisms through **errno** and the floating point state is at a dead end. First, for a typical function in the <math.h> header the correct handling of possible error conditions via thread local state may have an overhead that is of the same order of magnitude as the "useful" part of the function itself. Second, since the error state is handled globally, calls to such functions cannot be well integrated into their caller.

For C, we build on two preceding papers, [Navia(2018)] and [Douglas(2018b)], by complementing their ideas to provide a "C only" backwards compatible interface. Partly, their proposed new error condition handling that extends existing ABI, require the rewrite of parts of the C library, change error code conventions, or introduce additions to C's type system. In particular, both do not seem directly suitable for a seamless migration, because they either require to implement a whole new series of library functions, or because they cannot necessarily combine legacy user code or legacy C libraries with code and libraries that would adhere to the new ABI.
Our intent is provide an interface that is

— backward compatible,
— efficient,
— minimal,
— straight forward to implement, *and*
— extensible.

The interface is constructed such that combinations of previously compiled user code and libraries should work as shown in Table I. Note that the "*link error*" case only occurs if the caller uses an inconsistent declaration of the callee that includes an annotation with an out-of-band attribute, and that the caller has not been compiled with the same attributes as the callee.

The low-level techniques that have been proposed in previous papers for such new error strategies are based on transferring *out-of-band* information from the called function to the

Table I. Combinations of binaries with and without out-of-band support

	callee without oob	callee uses oob
caller without oob	unchanged	callee optimized
caller uses oob	*link error*	both optimized

caller. This is information that complements the "normal" return value of a function, such that the caller may detect an exceptional condition, easily, but that on the other hand does not change the global state, such that it can easily be ignored and such that it does not inhibit optimization.

Techniques to do so are usually based on amending a platform ABI, such that a previously unused hardware register or flag is used for the transfer of information from callee to caller. The idea of the present paper is to use the same type of mechanism, so we don't think that is necessary to describe it in all detail, here. In Section 4 we will though present some ideas for implementations and also about a generic extension strategy for platform ABI. We only use this mechanism *ad minima*:

Note. *Out-of-band error handling transfers* one *additional bit, the out-of-band bit, from the callee to the caller. A function call encountered an exceptional condition, if and only if this bit is* 1 *when returning from the call.*

In fact, the use of a reserved hardware flag for this interface isn't even necessary. The whole approach can also be easily described via a hidden extra parameter of a special "version" of a function with out-of-band bit support. The specification is willingly lose to allow enough slack for efficient implementations and platform ABI.

This is achieved by describing the out-of-band bit extension with four different calling conventions and four different return conventions. The former can be used by callers of a function to indicate the level of details that it needs from the function and permits implementations more (or less) agressive optimizations of the calling context. The calling conventions are **oob_direct**, **oob_plain**, **oob_ignore** and **oob_capture**. They are designed such that they can easily be integrated into typical C library implementations. For example a typical occurrence of an error handling such as

```
errno = EOVERFLOW;
return -1;
```

may simply be replaced by

```
oob_return_minus1(EOVERFLOW);
```

and an annotation of the function with the attribute `[[oob_return_error]]`.

The specific return convention allows the implementation of the function to give details of the channels that it intends to use for the return of out-of-band information. The return conventions are **oob_default**, **oob_return**, **oob_return_never** and **oob_return_error**. In addition, there are some specialized conventions that are derived from **oob_return_error**, such as **oob_return_minus1** and **oob_return_neg**.

Table II shows the effects that the different combinations of conventions may have. Observe that *side effects* only occur for combinations of **oob_direct**, **oob_plain** and **oob_return_error**.

Our idea is to specify the behavior of out-of-band augmented functions by the new *attributes* feature that C most likely will adopt, similar to the ones in C++. Standard attributes must be such that compiling a code that is conforming in presence of the attribute remains conforming when removing the attribute. Our intent with this proposal is to respect that requirement.

Table II. Behavior and side effects of combinations of calling conventions and return conventions. Here, **return** indicates conventional return, *bit* additional return of the out-of-band bit, *undefined* that a code path has undefined behavior, **errno** and *fp state* indicate that these may be modified.

	oob_default	**oob_return**	**oob_return_never**	**oob_return_error**
oob_direct	**return**	**return**	*undefined*	**return**, **errno**, fp state
oob_plain	**return**	**return**	**return**	**return**, **errno**, fp state
oob_ignore	**return**	**return**	**return**	**return**
oob_capture	**return**	**return**, bit	**return**, bit	**return**, bit

Examples:

```
[[oob_return_error]] long strtol(char const[static 1], char**, int);
[[oob_return_error]] int snprintf(char*, size_t, char const[static 1], ...);
[[oob_return]] void tss_delete(tss_t);
```

The attribute(s) should correspond to the special return convention that the function uses. *E.g*, the first two are specified to return errors in **errno**, so the **oob_return_error** convention is chosen. For both functions a return of an out-of-band bit can be sufficient to reconstruct all necessary failure information in the caller: **strtol** has only one failure mode (so it must be occurrence of a domain error) and **snprintf** returns a negative value on failure, so that value can be chosen to be the negative of the error code that otherwise would be returned in **errno**.

The function **tss_delete** is a special case for which the C standard does not foresee failure, but corresponding OS features such as `pthread_key_delete` do indeed have failure conditions. Here, an annotation with **oob_return** could indicate that errors can be captured, nonetheless.

Optimization opportunities

We distinguish several closely related properties of functions (or generally code snippets) that have a direct influence on the optimization opportunities at their call site:

— *unsequenced:* A function is *unsequenced* if a call can be effected as soon as all the arguments are known. As a consequence, such a function can only read unmutable state such as **const** objects. There can only be very limited side effects, *e.g* if stores in different calls are idempotent (writing the same value in an exclusive object) or interchangeable (printing the same string or performing a read-modify-write operation that discards the value that is read). Such side effects must not depend on the input and must not disclose information (such as addresses) to unexpected parts of the program execution. Such a call can be subject to common subexpression elimination, local memoization, lazy evaluation or parallelization.

— *side effect free:* The function only uses its parameters to determine its result and reads other objects only because they are global or because pointer arguments point to them, but it will never change them. It cannot do IO and cannot create or delete objects via **malloc**, **free** and similar functions.

— *state independent:* The result of such function can be deduced from its parameters alone. Besides its return value, a state independent function could possibly write state without reading it, for example write a value to a stream or store an error code in **errno**.

— *referentially transparent:* A function has this property if it is side effect free and state independent. Such a function reads no state other than its parameters and leaves no trace in the program state other than its return value, thus as a consequence it is also unsequenced.

— A call of such a function that receives only compile time constant arguments can be evaluated at compile time.
— Such a call can be subject to common subexpression elimination, global memoization, lazy evaluation, parallelization and vectorization.

Note. *In some programming communities the term* pure *is used for either* side effect free *(e.g by Fortran, or a* gcc *attribute) or for* referentially transparent *(functional programming, Keil compiler), and so it does not seem appropriate to use this term, here.*

Since the property of being unsequenced is relatively difficult to assert, many frameworks only deal with the easier to handle properties of being side effect free (gcc attribute `[[gnu::pure]]`) or referentially transparent (gcc attribute `[[gnu::const]]`). To determine if a particular combination of our conventions has one of these two properties it is sufficent to see if the function executed with the **`oob_ignore`** convention has the sought property.

For example, most C library functions specified by `<math.h>` are state independent, but because of their possible use of **`errno`** or the floating-point state many of them have non-commuting side effects and thus are also not unsequenced. When implemented with out-of-band return and executed with **`oob_ignore`** or **`oob_capture`** they become in fact side effect free and thus also referentially transparent.

In most cases an error convention can be applied such that when used with **`oob_capture`** still all necessary failure information is available to the caller.

2. CALLING CONVENTIONS

In the following let `RETURN` be one of the four return conventions and let us suppose that we have a declaration:

```
[[__RETURN__]] R toto(A a, B b);
```

where `R`, `A` and `B` are arbitrary complete types, and a corresponding definition

```
[[__RETURN__]] R toto(A a, B b){
    ...
}
```

This here is just meant as an example to illustrate the concepts, this whole mechanism should also work if `R` or the parameter list (or both) are **`void`**.

Such a declaration is semantically equivalent to the declaration of several functions with names that are not accessibly to the user code:

```
  extern R toto::direct(A a, B b);
  extern R toto::plain(A a, B b);
  extern R toto::ignore(A a, B b);
  extern R toto::capture(bool flap[restrict static 1], A a, B b);
```

These four functions, or equivalent constructs are instantiated in the TU that sees the definition. To ease the implementation of the features, the corresponding `[[__RETURN__]]` attribute must be applied to all declarations of the function and may not be combined with the storage class specifiers **`static`**, nor with a function specifier **`noreturn`**.

Here, `toto::`**`direct`** is a function that will be called when `toto` or its address is used in a conventional function call, whereas `toto::`**`plain`** is a function that has all the effects that a normal definition of `toto` would have.

The **`plain`**, **`ignore`** and **`capture`** functions are not callable directly, only through corresponding constructs, see below. As said, they don't have names that would be visible to the user code, and an implementation can in effect decide to realize these features by any means they see fit, as long as it preserves the required side effects.

In particular, the additional `flap` parameter for `toto::capture` only serves as a description of semantic effects of the feature. The `flap[0]` object is supposed to be a "write-only" parameter that is used to transfer the one and single out-of-band bit, and who's address is never considered to have escaped the calling context. Implementations are strongly encouraged to find other means to transfer that bit from the call to the callee, implementing it as additional parameter should only be a last resort on platforms that cannot use any hardware register or flag for this purpose.

2.1. direct and plain call

The first mode for calling an out-of-band function is a **direct** call, that is the function is called as usual. The call

```
toto(a0, b0)                // equivalent
oob_direct(toto, a0, b0)
```

by using the function name `toto` or a function pointer to `toto`, is sematically equivalent to

```
toto::direct(a0, b0)
```

During such an execution the special out-of-band return constructs, see below, may kick in and perform certain tasks, see Table III. The **direct** calling convention should also apply to calls that are inlined, either because they where declared with **inline** and the definition is visible in a header, or because the compiler chooses so in the compilation unit where the function is defined.

Table III. Return effects for direct and plain calls with return value of type `R`. The analogous constructs for **void** functions omit the return value. For a description of `oob_error_set` see Section 3.3.

out-of-band construct	conventional C construct
return val	**return** val
oob_return(val)	**return** val
oob_return_never(val)	**direct**: undefined **plain**: **return** val

oob_return_error(err, ...)	**do** {
	  oob_error_set(err);
	  return (__VA_ARGS__);
	} **while**(**false**)

Observe that the first three conventions have no additional side effects.
Similarly, the **plain** calling convention

```
oob_plain(toto, a0, b0)
```

is sematically equivalent to

```
toto::plain(a0, b0)
```

Compared to **direct**, **plain** only changes the behavior of **oob_return_never**, which then is defined to return the value.

For both modes, **direct** and **plain**, **oob_return_error** may have the effect of setting **errno** or to raise the floating point exception flag. For a discussion about the values and the case analysis that is performed by **oob_error_set** see the discussion for **oob_return_error**, below. This can be used for two purposes by the code of `toto`: to return an error condition or raise a floating point exception flag (if `err` is non-zero) or by clearing **errno** and possibly the floating-point environment from all accidental values that may have occurred during its execution.

Optimization opportunities

For the function `toto::plain` to be referentially transparent, is necessary that the function definition only uses the first three return conventions, since otherwise it could have an effect on `errno` or the floating point exception flag. If that is the case, it follows that `toto::direct` is referentially transparent, too.

2.2. Special calling conventions

The remaining calling conventions change the semantics of the called function such that they avoid the side effects of changing `errno` or the floating point exception flag.

2.2.1. ignoring the out-of-band bit and error conditions. If a user of a function has enough knowledge about a call they might deduce that the error path will never be visited, that the error conditions can safely be ignored, or that the usual return value provides enough information to deal with errors. In such a situation a call as in

```
oob_ignore(toto, a0, b0);
```

is equivalent to

```
toto::ignore(a0, b0);
```

The effect is that all occurrences of special out-of-band return constructs simply result in a conventional return of the value. Only the derived constructs `oob_return_minus1` and `oob_return_neg` still allow the propagation of error information, a `-1` for the former and the negated error code for the latter.

Optimization opportunities

The function `toto::ignore` is referentially transparent (or unsequenced), if the only possible side effects of `toto::plain` are via an out-of-band return that changes `errno` or the floating point state.

This can be the case for most functions in `<math.h>`, so calls to these functions with `oob_ignore` could be optimized by common subexpression evaluation.

2.2.2. capturing the out-of-band bit. Calling a function in `capture` mode is semantically the most complex. In the following we use two objects, `f` and `ret`, to receive the out-of-band bit and the normal return value, respectively. These two objects only have to meet minimal requirements; they must be assignable *lvalues* and of a compatible type. The latter means that `f` has to be assignable from integers, that is any integer or floating type `IT` would do.[1] For `ret` this means that it must have a type `RR` that is assignment compatible from `R`.

The code snippet

```
register IT f;    // Need not to have an address
register RR ret;  // Need not to have an address
ret = oob_capture(f, toto, a0, b0);
```

is equivalent to the following:

```
register IT f;    // Need not to have an address
register RR ret;  // Need not to have an address
{
```

[1]Pointers are not allowed because they would need conversion.

```
    bool flag;      // Cannot alias, cannot escape
    ret = toto::capture(&flag, a, b);
    f = flag;
  }
```

If `R` is **`void`**, the obvious simplifications apply, that is there is no need for `ret` and thus no assignment to it.

Table IV. Return effects for capture calls.

out-of-band construct	conventional C construct
`return val`	`do { flap[0] = 0; return val; } while(false)`
`oob_return(val)`	`do { flap[0] = 1; return val; } while(false)`
`oob_return_never(val)`	`do { flap[0] = 1; return val; } while(false)`
`oob_return_error(err, ...)`	`do { flap[0] = err; return (__VA_ARGS__); } while(false)`

Optimization opportunities

The function `toto::capture` itself is not referentially transparent, because it uses the parameter `flap` to store the out-of-band bit, and thus it is not unsequenced, either. But the usage as described above (within the replacement block) is in fact referentially transparent, for the same cases as for `toto::ignore`. If the only possible side effects of `toto::plain` are via `oob_errror`, the only values that are propagated outside the block are `ret` and `f`.

This is the case for most functions in `<math.h>`, so calls to these functions with `oob_capture` could be optimized by common subexpression evaluation. In addition to the usual return value, such common subexpression evaluation may also take a possible out-of-band bit into account.

This mechanism also allows to build more complex error or exception handling on top, but which remains efficient and leads to referentially transparent pseudo-functions. For example, the `oob_return_error` feature can be used to reinterpret the normal return value of type `R` as an error code or value exception if the out-of-band bit is set. Other efficient strategies could be to have a combined return type

```
  typedef union { rettype retval; errtype errval; } R;
```

that holds a "normal" return value `retval` if the out-of-band is not set, and and error val `errval` otherwise.

2.3. Call convention dependent switching

Functions that implement the out-of-band calling conventions might themselves distinguish their action according to the calling convention they were called themselves. There is an enumeration type

```
  enum { oob_direct, oob_plain, oob_ignore, oob_capture, };
```

and a variable as if defined at the beginning of each out-of-band function

```
  register int const oob_mode = value;
```

In the following example we assume that we have another out-of-band function `my_other_func` that should be called with the same convention as the current function.

In the case that an out-of-band bit is captured, the error and the return code are just passed through to the caller.

```
register R ret;
register bool flag = false;
switch (oob_mode) {
  default:           ret = oob_direct(my_other_func, abc); break;
  case oob_plain:    ret = oob_plain(my_other_func, abc);  break;
  case oob_ignore:   ret = oob_ignore(my_other_func, abc); break;
  case oob_capture:  ret = oob_capture(flag, my_other_func, abc);
}
if (flag) oob_return(ret);  // pass through
else return 5 + ret;
```

Depending on the implementation, in the compilation unit of a particular calling convention such a **switch** can then compile to just the same code as the corresponding call to my_other_func and an addition of 5.

3. SPECIAL RETURN CONVENTIONS

The usual return from an out-of-band function should always be a normal **return** as from any other function with the given prototype. Out-of-band special return constructs should only be used in code path that are "rare", "special", "exceptional", "erroneous" or similar. The prototypes of functions that use these special return conventions must be annotated with a corresponding attribute such that callers may know what error convention to expect.

If several return conventions can be used, all of them must appear in an attribute. *E.g*, a function that may have undefined behavior for some error paths, and set **errno** or the floating-point state in others could be annotated as:

```
[[oob_return_never, oob_return_error]] double log(double);
```

— For functions that are not declared with **oob_return_never**, **direct** and **plain** mode are equivalent. Implementations may chose to realize them as one function, or to simplify the **oob_direct** and **oob_plain** features at the calling side.
— Functions that only use any combination of [[**oob_return**]] and [[**oob_return_never**]] but not [[**oob_return_error**]] (or one of the derived return conventions) must neither touch **errno** nor the floating-point state. Thereby a caller may deduce that none of these have changed across calls to the function.
— The [[**oob_return_error**]] convention implies [[**oob_return**]].

All the following special features may set the out-of-band bit when the function is executed under **capture** conditions and ignore the bit when under **ignore**. Behavior of these constructs when they are encountered during **direct** execution varies according to the construct: it may just set the out-of-band bit, set **errno** to some value or be undefined.

As above, R represents the return type of the function in which the construct is found. Some of the constructs work as if they were declared as generic functions with the given prototype. In particular, these are as if declared **void** and **noreturn**. Other than a normal **return** statement, they always need parenthesis, even if R is **void**.

3.1. oob_return: return value and eventually set the out-of-band bit

This construct suggests to the compiler that the path to the place it is used is a rarely taken and that optimization should prefer other paths.

The only observable change from normal **return** is the out-of-band bit that is set under **capture**, but not when called via any of the other modes.

Synopsis:.

```
    noreturn void oob_return(R val);
```

`val` is returned in all modes.

Synopsis:.

```
    noreturn void oob_return(void);
```

This variant is for **`void`** functions and returns to the caller as a usual **`return`**.

3.2. `oob_return_never`: optimize out or return value and set the out-of-band bit

This construct is intended for the implementation of functions that impose consistency requirements to the caller of a function. Many functions in the C library in fact impose that certain argument values are not permitted (*e.g* null pointers, or `0` for **`log`**), or have to be in a certain range. This return convention allows to implement checks for such conditions, but that can be effectively eliminated for "normal" execution. For example, a test as the following

```
 // Check eliminated in direct mode.
 if (!x) oob_return_never(42);
```

can be completely skipped in **`direct`** mode. Still it is there to document the requirement and to help to debug erroneous calls through **`capture`** mode.

For **`plain`**, **`ignore`** and **`capture`** execution, this construct is equivalent to a simple **`return`** statement with the corresponding value, with the addition for **`capture`** that the out-of-band bit is set. The latter is intended to facilitate debugging of exceptional conditions.

For **`direct`** execution, this construct suggest that the path leading to it is never taken. If taken anyhow, the behavior is undefined.

Synopsis:.

```
    noreturn void oob_return_never(R val);
```

`val` is returned in all modes that are not **`direct`**. When **`direct`** mode reaches this construct the behavior is undefined.

Synopsis:.

```
    noreturn void oob_return_never(void);
```

This variant is for **`void`** functions and for **`plain`**, **`ignore`** and **`capture`** returns to the caller as a usual **`return`**. When **`direct`** mode reaches this construct the behavior is undefined.

Optimization opportunities

If excuted directly, code paths that end in **`oob_return_never`** are undefined. Therefore, optimization can completely eliminate such a path.

In the latter case, an optimizer can be much more agressive than for **`plain`** mode and cut branches in the function that are never visited. Thereby, it may produce binary code that is more efficient than without that knowledge. Table V lists the functions in the C library that could profit from this feature.

Table V. C library functions that could use oob_return_never

atof	cnd_destroy	qsort	thrd_yield
atoi	free	rewind	tss_delete
call_once	mtx_destroy	setbuf	
clearerr	perror	srand	

3.3. oob_return_error: set errno or raise a floating point exception

This construct is intended for the implementation of functions that follow the error convention of the C library. Depending on the argument and possibly on the value of **math_errhandling**, these functions may set **errno** or raise a floating point exception to return error conditions.

The functionality in **direct** or **plain** mode is as if there is a macro

```
#define oob_error_set(...)                              \
do {                                                    \
  int err = (__VA_ARGS__);                              \
  if (err < 0) {                                        \
    if (math_errhandling & MATH_ERRNO)                  \
       oob_code_errno(err);                             \
    if (math_errhandling & MATH_ERREXCEPT)              \
       oob_code_fexcept(err);                           \
  } else {                                              \
    errno = err;                                        \
  }                                                     \
} while(false)
```

If R is **void**, in **direct** or **plain** mode **oob_return_error**(ERR) is as if replaced by a call to:

```
#define oob_return_error_void(...)                      \
do {                                                    \
  oob_error_set(__VA_ARGS__);                           \
  return;                                               \
} while(false)
```

Otherwise, in **direct** or **plain** mode **oob_return_error**(ERR, ...) is as if replaced by a call to:

```
#define oob_return_error_value(ERR, ...)                \
do {                                                    \
  oob_error_set(ERR);                                   \
  return (__VA_ARGS__);                                 \
} while(false)
```

All these constructs are required to be implemented as if they were macros. This ensures that the local state of the floating-point environment of the function in which these constructs occur is used. The (__VA_ARGS__) conventions ensure that compound literals including commas can be passed as arguments.

The enumeration **oob_code** can be used to deal with floating-point errors. It has has negative symbolic values and functions **oob_code_fexcept** and **oob_code_errno** with side effects as given in Table VI.

For **direct** and **plain** execution, side effects are performed according to ERR. The return value then is (__VA_ARGS__) or nothing if R is **void**.

A caller can avoid the side effects, by calling a function that follows this return convention under **ignore** or **capture**. Here, **errno** and the floating-point exceptions remain untouched.

Table VI. translation of floating-point exception modes and effects

fp exception	oob_code err	oob_code_fexcept(err)	oob_code_errno(err)
	err ≥ 0	untouched	errno=err
none	oob_code_clear	feclearexcept(FE_ALL_EXCEPT)	errno=0
	oob_code_transmit	untouched	untouched
inexact	oob_code_inexact	feraiseexcept(FE_INEXACT)	untouched
divide-by-zero	oob_code_divbyzero	feraiseexcept(FE_DIVBYZERO)	errno=ERANGE
invalid	oob_code_invalid	feraiseexcept(FE_INVALID)	errno=EDOM
overflow	oob_code_overflow	feraiseexcept(FE_OVERFLOW)	errno=ERANGE
underflow	oob_code_underflow	feraiseexcept(FE_UNDERFLOW)	errno=ERANGE

In addition, under **capture** the out-of-band bit is assigned with ERR, such that the caller may check if an error occurred and interpret the return value accordingly.
If R is **void**, in **capture** mode **oob_return_error**(ERR) is as if replaced by a call to:

```
#define oob_return_bit_void(...)                    \
do {                                                \
  if (__VA_ARGS__) oob_return();                    \
  else return;                                      \
} while(false)
```

Otherwise, in **capture** mode **oob_return_error**(ERR, ...) is as if replaced by a call to:

```
#define oob_return_bit_value(ERR, ...)              \
do {                                                \
  R ret = (__VA_ARGS__);                            \
  if (ERR) oob_return(ret);                         \
  else return ret;                                  \
} while(false)
```

Under **ignore**, ERR is dropped. No error condition other than possibly encoded in the regular return value is propagated to the caller.
Most of the functions in <math.h> could use this out-of-band return convention. Table VII lists other functions in the C library that could profit from this feature and that are not suited for one of the derived return modes that follow.

Table VII. C library functions that could use oob_return_error directly

fgetwc	strtoimax	wcstod	wcstoll
fputwc	strtold	wcstof	signal
strtod	strtoll	wcstoimax	
strtof	strtol	wcstold	

3.4. oob_return_minus1 (derived): set errno and return -1

This construct is intended for the implementation of functions that follow the legacy **errno** convention to return error codes, but have the additional convention to return -1.
A caller can avoid to trigger the use of **errno**, by calling this function under capture. The specific error code is then lost.

Synopsis:.

```
#define oob_return_minus1(...) oob_return_error((__VA_ARGS__), -1)
```

This requires that R must be assignment compatible with **int**.

Table VIII lists the functions in the C library that could profit from this feature.

Table VIII. C library functions that could use oob_return_minus1

c16rtomb	mbrtoc32	wcsrtombs	strtoull
c32rtomb	mbrtowc	wcstoull	strtoul
ftell	mbsrtowcs	wcstoul	strtoumax
mbrtoc16	wcrtomb	wcstoumax	

3.5. oob_return_neg (derived): set errno and return its negative

This construct is intended for the implementation of functions that follow the legacy **errno** convention to return error codes, but have the additional convention to return a negative value on error.

A caller can avoid to trigger the use of **errno**, by calling this function under **capture** or **ignore**.

If you must use an **errno** convention, use this one, because it can effectively avoid using **errno** at all.

— If otherwise all valid return values of the functions are non-negative, users can call such a function through **oob_ignore** and still receive all error information through the return value.

— If valid returns of the functions may be negative, (e.g as for **printf**) users can call such a function through **oob_ignore** and still receive all error information through the return value.

Synopsis:.

```
#define oob_return_neg(...)                     \
do {                                            \
  int code = (__VA_ARGS__);                     \
  oob_return_error(code, -code);                \
} while(false)
```

This requires that R must be assignment compatible with **int**.

Table IX lists the functions in the C library that could profit from this feature.

Table IX. C library functions that could use oob_return_neg

fgetpos	sprintf	vsnprintf	vwscanf
fprintf	sscanf	vsprintf	wprintf
fscanf	swprintf	vsscanf	wscanf
fsetpos printf	swscanf	vswprintf	
scanf	vprintf	vswscanf	
snprintf	vscanf	vwprintf	

3.6. oob_return_zero (derived): set errno and return (R){0}

This construct is intended for the implementation of functions that follow the legacy errno convention to return error codes, but have the additional convention to return 0 (if R is a real or complex type) or a null pointer (if R is a pointer type) or, more generally, a default-initialized rvalue.

A caller can avoid to trigger the use of errno, by calling this function under capture or ignore, but the additional information in the error code will be lost.

Synopsis:.

```
    #define oob_return_zero(...) oob_return_error((__VA_ARGS__), (R){ 0 })
```

This requires R to be a complete type.

4. POSSIBLE IMPLEMENTATIONS

The proposed API for out-of-band return can be implemented in very different ways, and it would be up to platform ABI designers to chose the appropriate conventions depending on the characteristics of their platform and on the objectives and constraints that have to be fulfilled.

Because the specification is voluntarily wage, the mechanisms to choose from for implementations are various:

(1) The four calling conventions can be implemented by
   (a) four different functions, one for each convention;
   (b) five different functions, one for each convention and and additional one to dispatch between them;
   (c) a single function that dispatches between the conventions according to **oob_mode**. This could be provided an additional function parameter, or a hidden register state;
   (d) one function for **direct** and **plain** mode and one for **ignore** and **capture**.
(2) The macros or functions **oob_plain**, ..., that are used to issue the function call according to the calling convention
   (a) may be based on pointer arithmetic for the function pointer (caller dispatch), or
   (b) on passing information through function parameter or a spare hardware register or flags into a function that serves as the discriminator **oob_mode** (callee dispatch).
(3) The out-of-band return convention can be implemented
   (a) verbatim, by providing a pointer to flap for the return of the out-of-band bit,
   (b) implicitly, by returning the out-of-band bit through a spare hardware register or flag.

In the following we will briefly expand different scenarios that show how to adapt the implementation to different needs, and finally propose a generic ABI extension that can guarantee that different implementation choices remain interoperable.

4.1. Implementation of the out-of-band bit itself

The easiest way to implement the out-of-band bit feature is certainly not to use the extra flap parameter and to keep all functions that are produced with the same prototype and ABI. To return the bit information for **capture**, we than have to find another out-of-band channel to transfer the bit.

There are several possibilities for such channels. Using thread local storage, would be one possibility, but it would reintroduce the same cost as for **errno** that we are trying to avoid in the first place. Another possibility would be to use some processor flag, such as a carry or overflow bit. Setting such a bit could be triggered immediately before the return from the callee, and checked immediately after the return by the caller. Implementing such an approach needs some good knowledge of the architecture in question, in particular it must be clear that setting such a bit cannot have influence on code that doesn't expect it.

A more direct and portable (to some extend) way to implement the out-of-band bit is to use a hardware register as the channel. Such a register would have to be such that the caller would not otherwise rely on the value of it. That is it must be either a so-called *scratch* register (a register that is completely at the disposal to functions without restrictions) or a *caller save* register (a register that is spilled before calling a function and restored afterwards). A list of such hardware registers for different common architectures is presented in Table X.

Table X. Common architectures and available registers

CPU architecture	scratch or caller save register
x86	ecx
x86-64	r10, r11
arm32	r12
arm64	x9 to x15
mips	$t4 to $t8
68k	d0, d1, a0, a1

After chosing an appropriate register, an implementation can then implement **oob_return** with something similar as the following code. It and the following code are examples for gcc and the x86 architecture. We use `__typeof__`, `__asm__` and `{( ... )}` extensions of gcc, the ecx hardware register for this particular architecture and we assume that data and function pointers have the same representation. Obviously, other compilers would have to use their own magic for that.

```
#define oob_return(...)                                          \
({                                                               \
  /* Compute the return value first, so there are  */            \
  /* no side effects on hardware registers, later. */            \
  __typeof__(__VA_ARGS__) __oob_ret = (__VA_ARGS__);             \
  /* now set the bit, mark the register as clobbered, */         \
  /* and ensure that this will not be reordered      */          \
  __asm__ volatile("movl␣$1, %ecx" : : : "c");                   \
  return __oob_ret;                                              \
})
```

Usually **oob_return** should result in just the addition of the movl instruction to the function code.

Capturing this bit is a bit more complicated, because we have also have to somehow get our hands on the address of the **capture** variant of the called function. Here, `__oob_align` is a fixed alignment boundary, where the compiler places the direct function itself and a table of function pointers to the different out-of-band functions, and where the **capture** function is placed in slot 3, `__oob_capture` represents the modified calling sequence itself.

```
#define oob_capture(FL, FU, ...)                                                        \
({                                                                                      \
  /* Compute the function address first, see below. */                                  \
  unsigned char const* __oob_base = (void const*)(FU);                                 \
  /* This a pointer to a table of functions.          */                                \
  __typeof__ (*(FU))*const*__oob_tab = (void const*)(__oob_ base-__oob_align);\
                                                                                        \
  /* Do the capture function call itself and retain the return values. */               \
  uint32_t __oob_flag;                                                                  \
  __typeof__((FU)(__VA_ARGS__)) __oob_ret;                                              \
  __oob_capture(__oob_ret, __oob_flag, __oob_tab[3], __VA_ARGS__);                      \
                                                                                        \
    /* FL just has to be an assignable lvalue, maybe without address */         \
    (FL) = __oob_flag;                                                            \
    /* last expression is the "return" of the block     */                        \
    __oob_ret;                                                                    \
  })
```

`__oob_capture` should produce exactly the same sequence of instructions as a normal function call with return value stored in `__oob_ret`, only that immediately after the `call` instruction itself an instruction equivalent to the following `movl` into the memory ("m" constraint of the instruction) of `__oob_flag` should be issued.

```
    __asm__ volatile("movl␣%ecx,␣$0" : "m" (__oob_flag) : "c");
```

Thus, the overhead for the out-of-band bit itself are just moves to and from memory. Also observe that for this example on `gcc` it would be simple to adapt to another architecture by parametrizing the macros with the hardware register that is to be used.

With this strategy an implementation could use the **capture** function also for **ignore**, here placed in slot `__oob_tab[2]`:

```
  #define oob_ignore(FL, FU, ...)                                                 \
  ({                                                                              \
    /* Compute the function address first, see below. */                         \
    unsigned char const* __oob_base = (void const*)(FU);                          \
    /* This a pointer to a table of functions.          */                        \
    __typeof__ (*(FU))*const*__oob_tab = (void const*)(__oob_ base-__oob_align);\
                                                                                  \
    /* Do the capture function call itself and return the value. */              \
    __oob_tab[2](__VA_ARGS__);                                                    \
  })
```

With such an implementation, the only overhead that would occur for such an **ignore** call is one useless `movl` instruction for the out-of-band bit into the register before the return from the function.

4.2. Implementation for time efficiency

For an implementation that wants to take advantage of all the optimization opportunities and that is not too much constrained by the binary code size that would be produced, the natural approach is to compile four different functions for the four calling conventions. Thereby, the different return conventions can be integrated well into the function code and unfold their full optimization potential.

In particular, a function that uses **oob_return_never** should be able to eliminate all such code paths in **direct** mode, and the direct call to such a function should not incur any overhead. This is why **direct** mode here is best served by resolving the function symbol itself (*e.g* `toto`) to the function that implements the **direct** mode (*e.g* `toto`::**direct**).

To call the other three modes (**plain**, **ignore** and **capture**) the corresponding call macros must implement some “trick” that allows them to deduce the address of the corresponding derived function. Probably the simplest way to achieve that on most architectures will be to place the necessary information at a known offset from the start address of the direct function, namely a **const**-qualified table that contains the four different function addresses, see the above macro for an example. The overhead for determining the function pointer is then one subtraction (with an immediate value) and one indirection, and can completely be optimized by the compiler if the function itself is specified directly, and not via a function pointer.

```
[[__RETURN__]] R (*totoPointer)(A, B) = toto;
...
R ret = oob_capture(totoPointer, a, b); // overhead for finding toto::capture
...
R rot = oob_capture(toto, a, b);        // toto::capture known at compilation
```

4.3. Implementation under memory constraints

When implementing out-of-band under strong memory constraints, in particular for the code size, we have to ensure that only one complete function is generated for all calling conventions. To distinguish the different calling conventions we then have to put the discriminant value **oob_mode** in the calling environment. When called, a function can then distinguish with which convention it was called.

As a way to provide that information we can again chose to use a caller saved hardware register or a scratch register. This hardware register could be the same as for the out-of-band bit, but it must not necessarily, see Table X above.

If we don't want to be incompatible to an already existing ABI, we have to ensure that an caller that has been compiled without out-of-band support will not accidentally trigger a special calling convention of a function with out-of-band support. The assembly for such an amended function prolog can be seen in the following:

```
.globl         toto
.type          toto, @function
.globl         toto::xtra
.type          toto::xtra, @function
toto:          mov ecx, 0
toto::xtra:    push ecx
               mov ecx, 0
               /* Continue with the usual function prolog */
               ...
               /* recover the value from the stack position into oob_mode */
```

Here, we provide two different entry points to the function, toto and toto::xtra. The first for direct calls ensures that the hardware register is set to 0. The second ensures that it isn't and that the value of the register can be stored on the stack. At the end of the prolog, the value can then be retrieved into **oob_mode**.

The individual return conventions can then rely on **oob_mode** to switch between the different return strategies.

Overall, this strategy trades a little bit of computing against code size. Some instructions are added to each function prolog and out-of-band return conventions have additional branches according to the calling convention with which the function has been called.

4.4. A generic ABI extension strategy

The strategies presented above will not only depend on a particular architecture, but also on other parameters such as the optimization level or the specific needs of applications. Therefore, it seems important to provide a generic ABI extension that can be used for any of these strategies and that guarantees that code compiled for different requirements remains interoperable. We think that this can be established by the following choices:

(1) Designate a spare hardware register from Table X, OOB_REG, say, that will serve as **oob_mode** on entry and as out-of-band bit on return.
    (a) To dispatch to the convenient calling convention on entry, we must guarantee that OOB_REG can be arbitrary for **direct** mode, and that it is set to values 1, 2 and 3 for modes **plain**, **ignore** and **capture**, respectively.
    (b) In **capture** mode, return the out-of-band bit in OOB_REG on return.

(2) Chose an alignment value `__oob_align`. For most architectures this should probably be the size of four function pointers (`sizeof(void(*[4])(void))`), but larger values could be suitable, either for the original calling convention, or because the implementation wants to add some more information.
(3) Prefix the **`direct`** function with a readonly table for the four function pointers and align both (table and function) at `__oob_align` bytes.

An amended function prolog (still for `gcc` and `x86`) could for example look as follows:

```
        .globl  toto::table
        .align  16
        .type   toto::table, @object
toto::table:
        .quad   toto
        .quad   toto::plain   /* could be the same as toto         */
        .quad   toto::ignore  /* could be the same as toto::capture */
        .quad   toto::capture
        .size   toto::table, .-toto::table
        .globl  toto
        .align  16
        .type   toto, @function
toto:   /* Continue with the usual function prolog */
```

Such a general strategy has only a small overhead for any of the variants that we have introduced above:

— The memory overhead of the table is `__oob_align` bytes. In addition there can be some overhead for the alignment of maximally another `__oob_align` bytes. Usually this means a maximal overhead of $16 + 15 = 31$ bytes on 32 bit machines and $32 + 31 = 63$ bytes for 64 bit machines.
— Depending on the strategy a superfluous `movl` can be issued to initialize `OOB_REG` to the required constant.

5. CONCLUSION

We presented an ABI and generic API for an out-of-band error convention for C. It splits possible calls of functions into four different modes, and adds a distinction of four different return conventions. It has the advantage of being backwards compatible as much as possible. Nobody is forced to do anything:

— Applications can completely ignore this mechanism and find themselves with basically the same binary code as before.
— Compiler implementors, as soon as they implement C2x' attributes, may ignore all function attributes and only provide minimal syntactical support for the presented macros.
— Platform ABI need only a minimal specification to allow compatible implementations. Basically they have to decide on a function table offset and a spare hardware register that will be used to propagate the calling mode **`oob_mode`** into a function call and the out-of-band bit on return.
— C library implementors may start integrating these features step by step, function by function, as long as they keep their header files in line with their implementation.

The opportunities of such an interface are multiple:

— C's math functions can be annotated with attributes that makes calls to them suitable for constant propagation.
— Generally legacy interfaces can be used more efficiently at points where it is known that error paths will not be taken.

— New C interfaces can take advantage of these conventions and avoid to introduce error return via `errno`.
— C or other programming languages can build on top of such a mechanism to specify more sophisticated error handling such as try/catch schemes.
— This provides a migration path, such that, on the long run, C could get rid of the `errno` convention all together.

References

Niall Douglas. SG14 `status_code` and standard error object for P0709 Zero-overhead deterministic exceptions. Technical report, ISO/IEC JTC 1/SC 22/WG21, 2018a. URL https://wg21.link/P1028.

Niall Douglas. Zero overhead deterministic failure — A unified mechanism for `C` and `C++`. Technical Report N2289, ISO/IEC JTC 1/SC 22/WG14, 2018b. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2289.pdf.

Doug Gregor. Deprecating exception specifications. Technical report, ISO/IEC JTC 1/SC 22/WG21, 2010. URL https://wg21.link/N3051.

Jacob Navia. Proposal for a new calling convention within the C language. Technical Report N2285, ISO/IEC JTC 1/SC 22/WG14, 2018. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2285.pdf.

James Renwick, Tom Spink, and Björn Franke. Low-cost deterministic `C++` exceptions for embedded systems. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 76–86, 2019. . URL https://www.research.ed.ac.uk/portal/en/publications/lowcost-deterministic-c-exceptions-for-embedded-systems(2cfc59d5-fa95-45e0-83b2-46e51098cf1f).html.

Herb Sutter. Zero-overhead deterministic exceptions: Throwing values. Technical report, ISO/IEC JTC 1/SC 22/WG21, 2018. URL https://wg21.link/P0709.