Document Number: N4045
Revises: N3964
Date: 2014-05-24
Reply To Christopher Kohlhoff <chris@kohlhoff.com>

Library Foundations for Asynchronous Operations, Revision 2

1 Introduction

Networking has a long history of programs being written using event-driven and asynchronous programming designs. The use of asynchronous operations with continuations, in particular, offers a good model for abstraction and composition. Asynchronous operations may be chained, with each continuation initiating the next operation. The composed operations may then be abstracted away behind a single, higher level asynchronous operation with its own continuation.

N3857 proposes an evolution of `std::future` with the addition of a `then()` member function, amongst other things. This function allows one to attach a continuation to a future, and is intended for use with asynchronous operations. With these extensions, `std::future` is proposed as "a standard representation of asynchronous operations".

N3858 builds on these extensions to `std::future` with a new language facility, resumable functions. The new keywords, `async` and `await`, are intended to ease composition of asynchronous operations by enabling the use of imperative flow control primitives.

In this paper, we will first examine how futures can be a poor choice as a fundamental building block of asynchronous operations in C++. The extended `std::future` brings inherent costs that penalise programs, particularly in domains where C++ use is prevalent due to its low overheads. An asynchronous model based on a pure callback approach, on the other hand, allows efficient composition of asynchronous operations.

However, cognizant that some C++ programmers may have valid reasons for preferring a futures-based approach, this paper introduces library foundations for an extensible model of asynchronous operations. This model supports both lightweight callbacks and futures, allowing the application programmer to select an approach based on appropriate trade-offs.

Finally, we will see how these library foundations can be leveraged to support other models of composition. This paper presents implementation experience of this extensible model, which includes several pure library implementations of resumable functions, or coroutines. Programmers have the opportunity to express asynchronous control flow in an imperative manner, without requiring the addition of new keywords to the language.

1.1 Changes in this revision

This document supersedes N3964. In this revision, the `handler_type<>` trait has been modified to be SFINAE-friendly, and an extended example has been added to section 9.2 to illustrate how the traits may be specialised by a user to add a new completion token type.

2 Callbacks versus futures

This paper uses the terms *callbacks* and *futures* as shorthand for two asynchronous models.

These two models have several concepts in common:

- An *initiating function* that starts a given asynchronous operation. The arguments to this function, including an implicit `this` pointer, supply the information necessary to perform the asynchronous operation.
- A *continuation* that specifies code to be executed once the operation completes.

The *callbacks* model[1] refers to a design where the continuation is passed, in the form of a function object, as an argument to the initiating function:

```
socket.async_receive(args, continuation);
```

In this example, `async_receive()` is the initiating function. When the asynchronous receive operation completes, the result is passed as one or more arguments to the callback object `continuation`. In Boost.Asio, these continuations are called *handlers*.

With the *futures* model, the initiating function returns a future object. The caller of the initiating function may then attach a continuation to this future:

```
socket.async_receive(args).then(continuation);
```

Or, more explicitly:

```
std::future<size_t> fut = socket.async_receive(args);
...
fut.then(continuation);
```

Alternatively, the caller may choose to do something else with the future, such as perform a blocking wait or hand it off to another part of the program. This separation of the asynchronous operation initiation from the attachment of the continuation can sometimes be a benefit of the futures model. However, as we will see below, this separation is also the source of the runtime costs associated with the model.

To maximise the usefulness of a C++ asynchronous model, it is highly desirable that it support callbacks. Reasons for preferring a callbacks model include:

- Better performance and lower abstraction penalty.
- A fundamental building block. Futures, resumable functions, coroutines, etc. can be efficiently implemented in terms of callbacks.
- Not tied to threading facilities. It is possible to implement efficient callback-based network programs on platforms that have no thread support.

This paper aims to demonstrate that supporting callbacks need not be mutually exclusive to providing support for futures.

[1] An implementation of the callbacks model in C++ can be found in the Boost.Asio library, in Boost versions up to and including 1.53.

3 Performance matters

Long-time network programmers may have encountered the following misconception: that performance is not a primary concern, due to the high latencies involved in network I/O.

Latency may be an acceptable justification for a high abstraction penalty in the context of HTTP clients on desktop operating systems. However, higher overheads mean lower throughput. For programs, such as servers, that handle network I/O events from multiple sources, the latency to the peer is often irrelevant; the program needs to be ready to handle the next event immediately. Furthermore, the deleterious effects of queuing and congestion are felt well before a system reaches 100% utilisation.

And, while it is true that typical Internet latencies are high, often measured in tens or hundreds of milliseconds, high latency is not an inherent attribute of network I/O. By way of illustration, consider the following two data points:

⇒ Transmit a 64-byte UDP packet from a user-space application on one host to a user-space application on another host (i.e. RTT/2), across a 10GbE network.

2 microseconds

⇒ On the same hardware and operating system, wake up and switch to a thread that is blocked on a `std::future` object.

3 microseconds

There are many real world use cases where C++ is used because it allows for high-level abstractions with a low abstraction penalty. For example, the author is familiar with systems in the financial markets domain where performance differences measured in microseconds have a significant impact on an application's efficacy.

It is for these use cases that the choice of asynchronous model matters most. If C++ were to adopt a restricted asynchronous model based only on futures, potential C++ standard library components such as networking would have their usefulness limited. To meet their application's performance requirements, programmers of these systems would have to step outside the standard library. Put another way, C++ and its standard library would have less to differentiate it from other, higher-level languages.

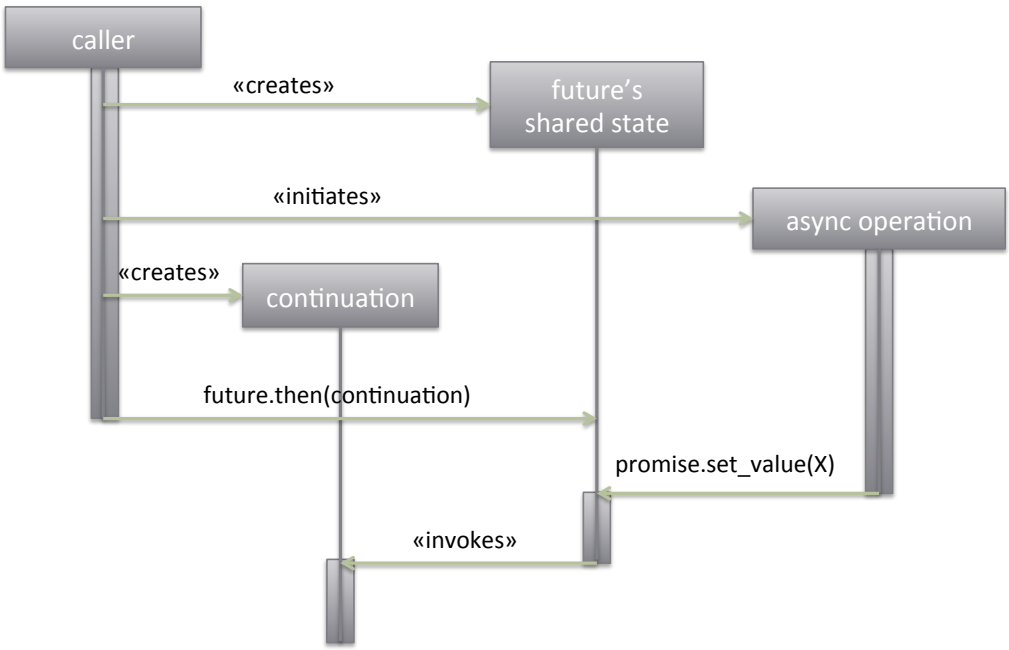
4 Anatomy of an asynchronous operation

To see the inherent costs of a futures model, let us take code of the form:

```
socket.async_receive(args).then(continuation);
```

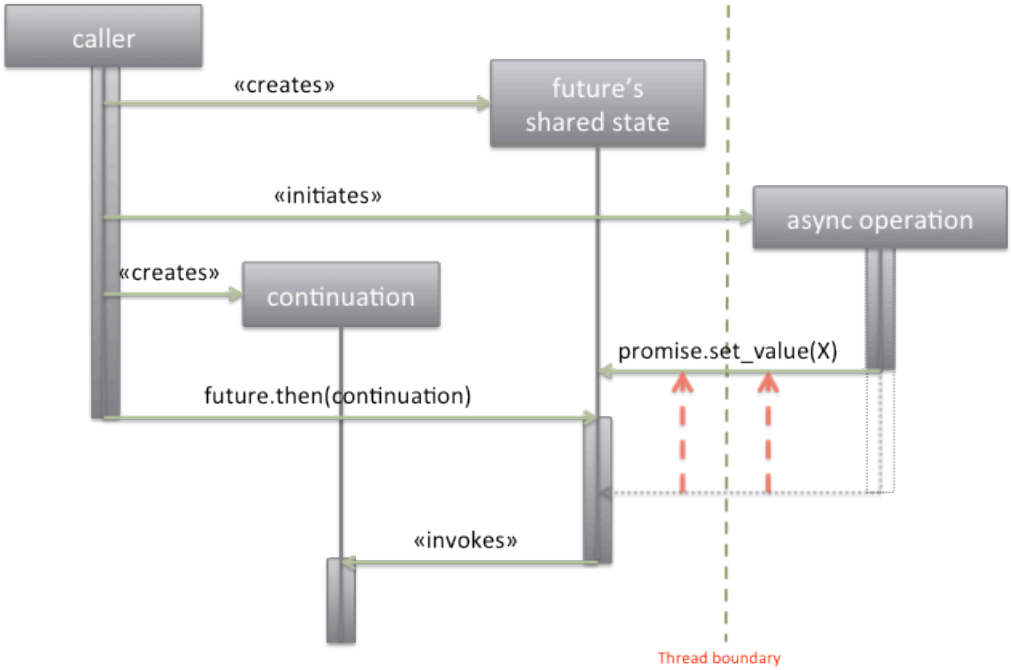
and consider the underlying sequence of events[2]:

[2] In reality, the implementation is more complicated than presented, as the `then()` member function itself returns an additional `std::future` object with another shared state. It is likely that this second shared state is where the continuation is ultimately stored.



The initiating function creates the future's shared state and launches the asynchronous operation. The caller then attaches the continuation to the future. Some time later, the asynchronous operation completes and the continuation is invoked.

However, after the asynchronous operation is initiated, it is logically executing in its own thread of control. It is executing in parallel to the caller, and so it is possible for the operation to complete before the continuation is attached, as shown in the following sequence of events:

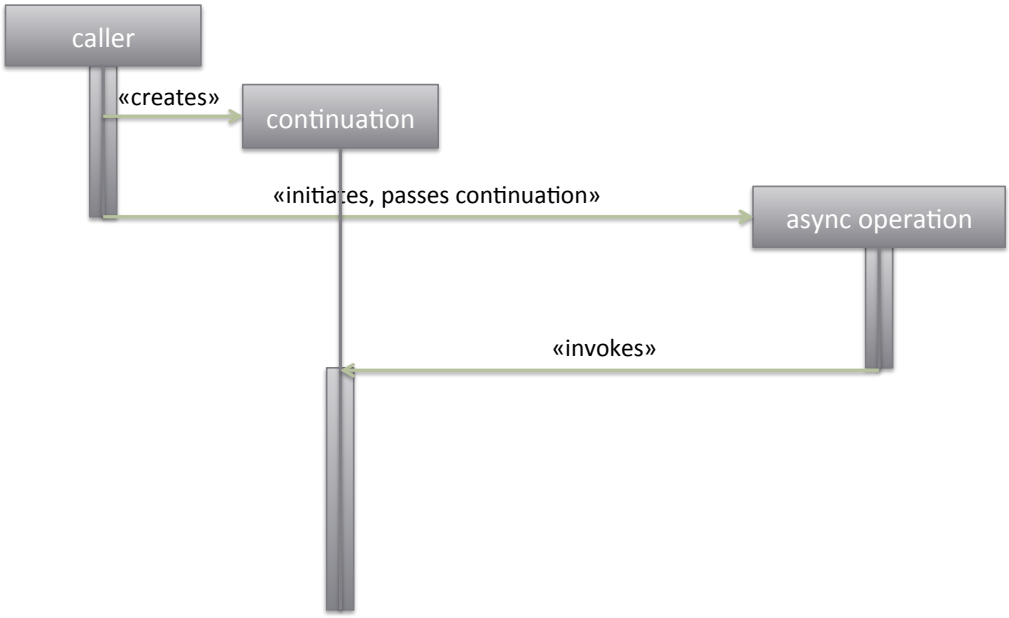


As a consequence, the shared state object has a non-deterministic lifetime, and requires some form of synchronisation to coordinate the attachment and invocation of the continuation.

In contrast, when the callbacks model has code of the form:

```
socket.async_receive(args, continuation);
```

we see the following, simpler sequence of events:



The initiating function accepts the continuation object and launches the asynchronous operation. The caller's flow of control ceases at this point[3], and the asynchronous operation is not executing in parallel with it. Unlike the futures model, there are no shared objects with non-deterministic lifetime, and no additional synchronisation is required.

5 Composition of operations

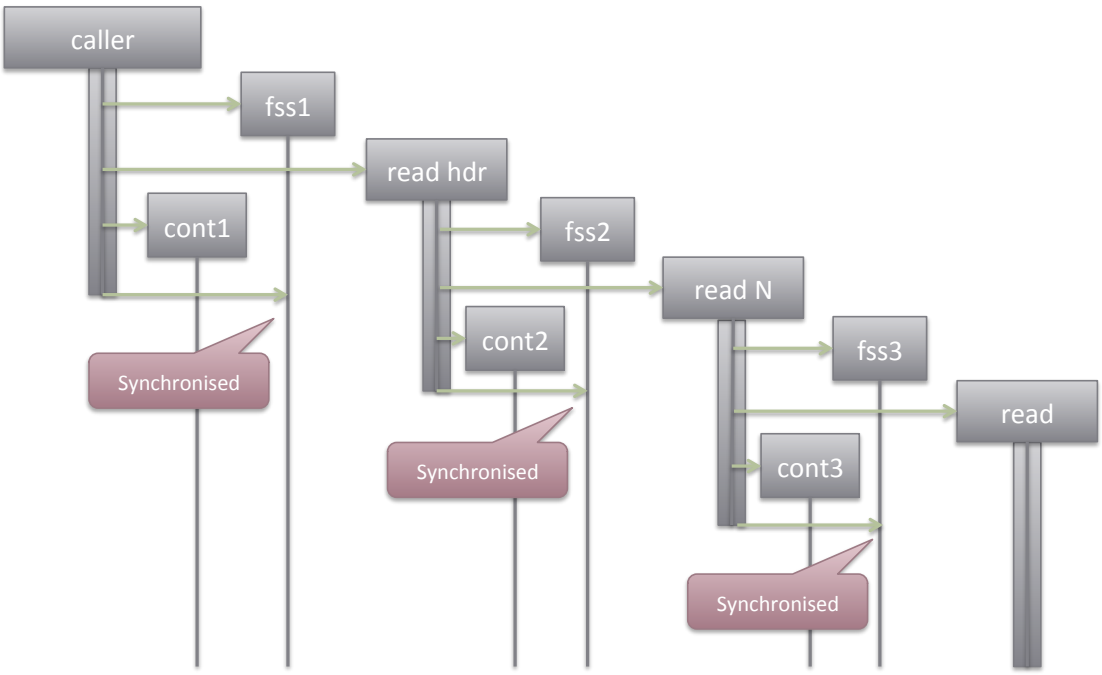
Let us now consider what happens when asynchronous operations are composed. A typical scenario involves a network protocol with a fixed-length header and variable-length body. For this example, the tree of operations might be as follows:

⇒ Read message
  ⇒ Read header
    ⇒ Read N bytes
      ⇒ Read data off socket
  ⇒ Read body
    ⇒ Read N bytes
      ⇒ Read data off socket

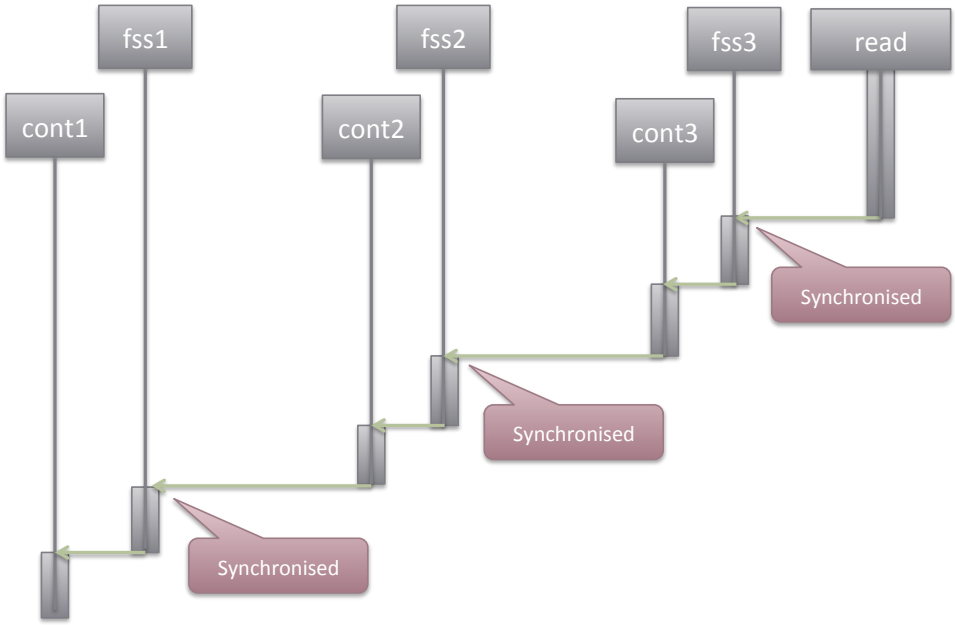
Each operation represents a level of abstraction, and has its own set of post-conditions that must be satisfied before its continuation can be invoked. For example, the "Read N bytes" operation exists to manage the problem of partial reads (where a socket returns fewer bytes than requested), and cannot call its continuation until the requested number of bytes is read or an error occurs.

With futures, as we go down the tree we see a sequence of events similar to the following:

[3] Technically, the caller's lifetime is not required to end at this time. It can continue to perform other computations or launch additional asynchronous operations. The important point is that it is not required to continue in parallel to the asynchronous operation in order to attach a continuation.

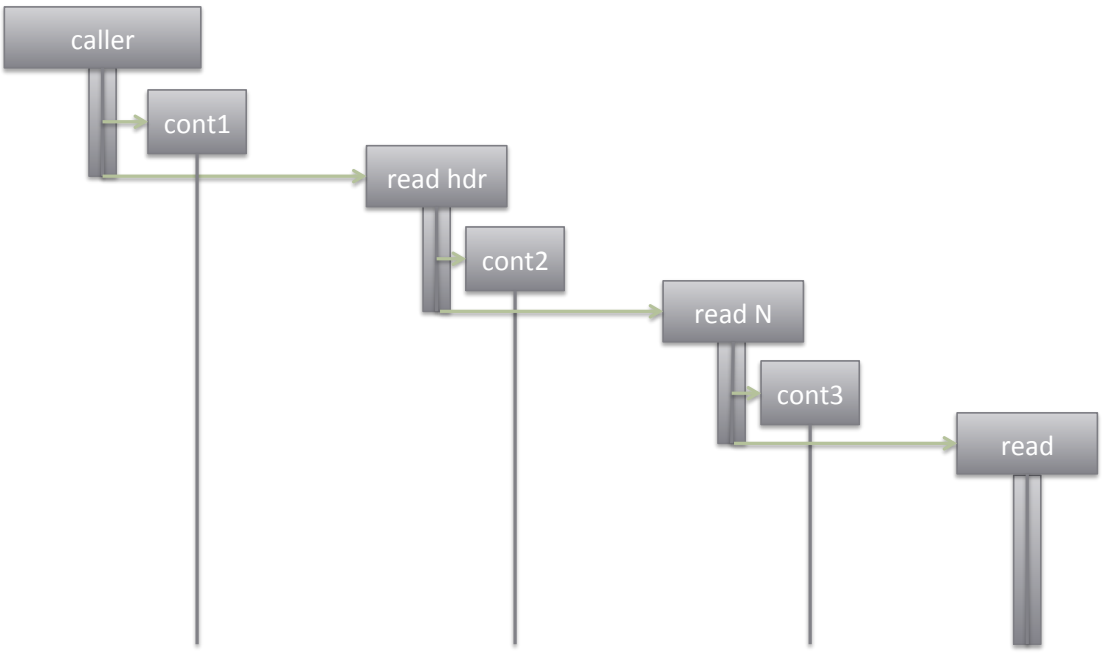


When the operations complete, and assuming each post-condition is immediately satisfied, the sequence of events is:

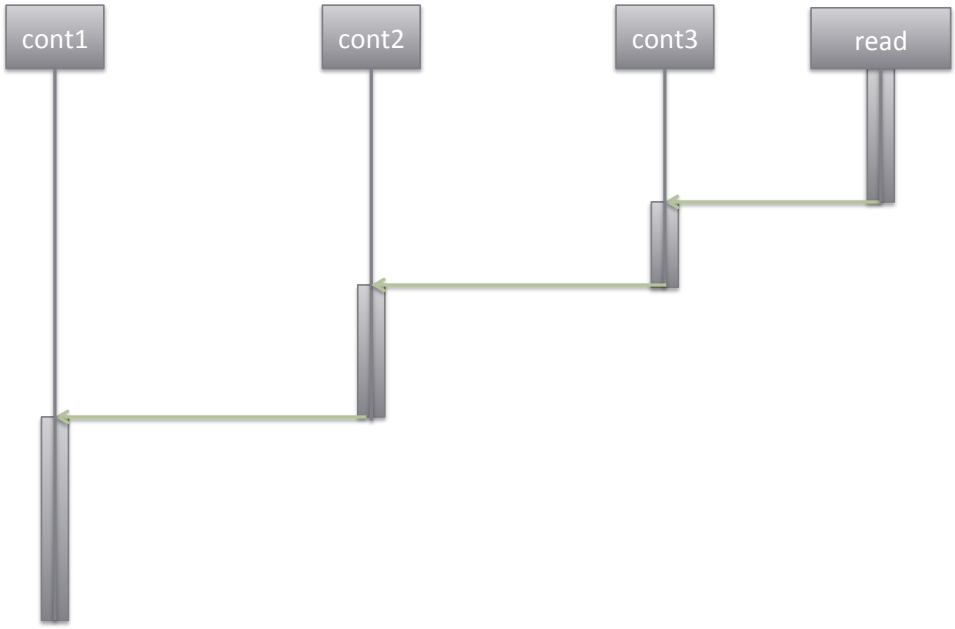


As you can see, each layer of the abstraction adds synchronisation overhead. On the systems available to the author, the cost of synchronisation mechanisms such as a `mutex` (when uncontended) or sequentially consistent atomic variable is some 10-15 nanoseconds per use.

When composing operations with callbacks, these costs are not incurred on the way down the tree of operations:



Nor are they present on the way back up:



In addition to avoiding synchronisation costs at each layer, when written as templates the compiler is given the opportunity to further optimise, such as inlining continuations. This means that it is possible to create layers of abstraction that have little or no runtime abstraction penalty.

6 A plethora of futures

Let us limit ourselves, for a moment, to considering only a futures model. All asynchronous operations would take the following form:

```
class socket {
  ...
  std::future<size_t> async_receive(buffer_type buffer);
  ...
};
```

This approach raises the following questions:

- How do we specify a custom allocator?
- How do we support alternate future types? For example:
  - A future that waits / does not wait on destruction.
  - A future with no blocking operations at all.
  - A future for use only in single-threaded programs.
  - A future that works with 3rd party coroutines.
  - A future that works with a 3rd party executor.

In acknowledging that there are use cases where there are valid reasons for preferring a futures model to a callbacks model, the question is: can we have a single model that supports callbacks *and* futures (in all their incarnations)? Furthermore, can a single model be defined in a way that supports extensibility to other models of composition?

7 An extensible model

In Boost 1.54, the Boost.Asio library introduced an extensible asynchronous model that supports callbacks, futures, and resumable functions or coroutines. The model is also user extensible to allow the inclusion of other facilities. Before we look at how the model works, in this section we will review some of the ways in which the model can be used.

7.1 Callbacks

Callbacks continue to work as before. As they are simply function objects, they can be specified using function pointers:

```
void handle_receive(error_code ec, size_t n) { ... }
...
socket.async_receive(buffer, handle_receive);
```

or lambdas:

```
socket.async_receive(buffer,
  [](error_code ec, size_t n)
  {
    ...
  });
```

or with function object binders:

```
socket.async_receive(buffer,
  std::bind(handle_receive, _1, _2));
```

With some suitable macro magic[4], we can even have callbacks that are implemented as "stackless" coroutines:

```
struct handler : coroutine
{
  ...
  void operator()(error_code ec, size_t n)
  {
    reenter (this)
    {
      ...
      while (!ec)
      {
        yield socket.async_receive(buffer, *this);
        ...
      }
      ...
    }
  }
  ...
};
```

Here we can make use of imperative control flow structures to implement complex asynchronous logic in a synchronous manner. Runtime overhead is minimal: the coroutine state is stored in an integer, and re-entering a coroutine is equivalent to resuming a `switch`-based state machine.

7.2 Futures

By passing a special value `use_future` (similar in concept to how the global object `std::nothrow` is used to tag overloaded functions), initiating functions return a `std::future` that can be used to wait for the asynchronous operation to complete:

```
try
{
  future<size_t> n =
    socket.async_receive(
      buffer, use_future);
  // Use n.get() to obtain result
}
catch (exception& e)
{
  ...
}
```

The correct return type is automatically deduced based on the asynchronous operation being used. For example, this code calls `async_receive`, and the result is a `future<size_t>` to represent the number of bytes transferred. In section 8 we will see how this type is obtained.

[4] Implemented in terms of a `switch` statement using a technique similar to Duff's Device.

If the asynchronous operation fails, the `error_code` is converted into a `system_error` exception and passed back to the caller through the future.

The `use_future` special value also allows a custom allocator to be specified:

```
try
{
  future<size_t> n =
    socket.async_receive(
      buffer, use_future[my_allocator]);
  // Use n.get() to obtain result
}
catch (exception& e)
{
  ...
}
```

The `use_future` object may likewise be passed to asynchronous operations that are themselves compositions. If these compositions are built using callbacks, the intermediate operations and their continuations are executed efficiently as in the callbacks model. Only at the final step is the future made ready with the result. However, should any intermediate step result in an exception, that exception is caught and stored on the future, where it will be re-thrown when the caller performs `get()`.

7.3 Coroutines / Resumable functions

Support for "stackful" coroutines has been implemented on top of the Boost.Coroutine and Boost.Context libraries. This is a pure library solution of resumable functions that does not require the addition of any new keywords.

A `yield_context` object is used to represent the current coroutine. By passing this object to an initiating function, we indicate that the caller should be suspended until the operation is complete:

```
void receive_message(yield_context yield)
{
  try
  {
    size_t n = socket.async_receive(buffer, yield);
    ...
  }
  catch (exception& e)
  {
    ...
  }
}
```

The return type of the initiating function is deduced based on the operation being called. If the operation fails, the `error_code` is converted into a `system_error` exception and passed back to the caller through the future. In many use cases, an error is not exceptional, and it is preferable that it be handled using other control flow mechanisms. With these coroutines, the error can be captured into a local variable:

```
void receive_message(yield_context yield)
{
  ...
  error_code ec;
  size_t n = socket.async_receive(buffer, yield[ec]);
  if (ec) ...
}
```

As each coroutine has its own stack, local variables and complex control flow structures are available, exactly as they would be in a synchronous implementation of the algorithm:

```
void do_echo(yield_context yield)
{
  try
  {
    char data[128];
    for (;;)
    {
      size_t n = socket.async_read_some(buffer(data), yield);
      async_write(socket, buffer(data, n), yield);
    }
  }
  catch (exception& e)
  {
    // ...
  }
}
```

A `yield_context` object may be passed to composed operations that are built only using callbacks. The coroutine functions themselves also compose easily through direct function calls. These functions share a stack, and the bottommost function suspends the coroutine until an asynchronous operation completes. Unlike futures, returning a result from a lower abstraction layer has minimal cost; it is the same as returning a result from a normal function.

Finally, as a comparison, here is an example of an `async/await`-based resumable function shown side-by-side with its equivalent using Boost.Asio's coroutines:

Example using Microsoft's PPL[5]

```
task<string>
read(string file, string suffix)
    __async {
  istream fi = __await open(file);
  string ret, chunk;
  while((chunk = __await fi.read()).size())
    ret += chunk + suffix;
  return ret;
}
```

Equivalent using library-based coroutines

```
string
read(string file, string suffix,
     yield_context yield) {
  istream fi = open(file, yield);
  string ret, chunk;
  while((chunk = fi.read(yield)).size())
    ret += chunk + suffix;
  return ret;
}
```

[5] Copied from `http://video.ch9.ms/sessions/build/2013/2-306.pptx`

7.4 Proposed Boost.Fiber library

Boost.Fiber[6], a library that is being developed and proposed for inclusion in Boost, provides a framework for cooperatively scheduled threads. As with "stackful" coroutines, each fiber has its own stack and is able to suspend its execution state. The Boost.Fiber library supplies many concurrency primitives that mirror those in the standard library, including mutexes, condition variables and futures.

The proposed Boost.Fiber library has been enhanced to support the extensible asynchronous model. This has been achieved without requiring any Boost.Fiber-specific extensions to Boost.Asio.

Firstly, we can use Boost.Fiber's future class in the same way as `std::future`, except that waiting on the future suspends the current fiber:

```
try
{
  boost::fibers::future<size_t> n =
    socket.async_receive(buffer,
      boost::fibers::asio::use_future);
  // Use n.get() to obtain result
}
catch (exception& e)
{
  ...
}
```

Secondly, we can suspend the current fiber automatically when performing an asynchronous operation, in a similar fashion to the integration with Boost.Coroutine shown above:

```
try
{
  size_t n = socket.async_receive(buffer,
      boost::fibers::asio::yield);
  ...
}
catch (exception& e)
{
  ...
}
```

In both cases, the `use_future` and `yield` names refer to special values, similar to `std::nothrow`. When passed, the appropriate return type is deduced based on the asynchronous operation that is being called.

7.5 Synchronous execution contexts

Some simple use cases sometimes require only synchronous operations. When developing a library of protocol abstractions, we may wish to avoid defining the composition twice; that is, once for synchronous operations and once for asynchronous operations. With the extensible model, it is feasible to implement just the asynchronous composition and then apply it in a

[6] `https://github.com/olk/boost-fiber`

synchronous context. This can be achieved without the overhead of futures or the additional background threads that run the event loop. For example:

```
try
{
  size_t n = async_read(
      socket, buffer, block);
  ...
}
catch (exception& e)
{
  ...
}
```

Here, `block` is an object that knows how to run the event loop. As with coroutines and futures, if the operation fails then the `error_code` is converted into a `system_error` exception and propagated to the caller.

This simulated synchronous model can also be enhanced to support timeouts:

```
try
{
  size_t n = async_read(
      socket, buffer,
      block.wait_for(seconds(10)));
  ...
}
catch (exception& e)
{
  ...
}
```

If preferred, the error can be captured into a `std::error_code` object rather than throwing an exception:

```
error_code ec;
size_t n = async_read(
    socket, buffer,
    block.wait_for(seconds(10))[ec]);
```

8 How the extensible model works

To understand the extensible asynchronous model, let us first consider the callbacks model from Boost.Asio (used in Boost version 1.53 and earlier). We will then examine the incremental changes that have been applied to create the extensible model.

In a callbacks model, a typical initiating function will look something like this:

```
template <class Buffers, class Handler>
void socket::async_receive(Buffers b, Handler&& handler)
{
  ...
}
```

To convert to the extensible asynchronous model, we need to determine the return type and how to obtain the return value:

```
template <class Buffers, class CompletionToken>
???? socket::async_receive(Buffers b, CompletionToken&& token)
{
  ...
  return ????;
}
```

This is achieved by introducing two customisation points into the implementation.

8.1 Customisation point 1: Convert completion token to handler

In the callbacks model, the type `Handler` was the function object type to be invoked on completion of the operation. In the extensible model, we want to use a placeholder type, such as `yield_context` or the type of the `use_future` object, in addition to function objects. This placeholder is called a *completion token*, and it identifies the completion handler that will process the result of the asynchronous operation.

Therefore it is first necessary to determine the type of the handler from this completion token. This is achieved by the following type trait:

```
template <typename CompletionToken, typename Signature>
struct handler_type {
  typedef ... type;
};
```

and its corresponding template alias:

```
template <typename CompletionToken, typename Signature>
  using handler_type_t =
    typename handler_type<CompletionToken, Signature>::type;
```

The `Signature` template parameter is based on the callback arguments for the given asynchronous operation. For a socket receive operation the `Signature` is `void(error_code, size_t)`, and the handler type may be deduced as follows:

```
handler_type_t<CompletionToken, void(error_code, size_t)>
```

The handler type must support construction from the completion token type, as the initiating function will attempt to construct a handler as follows:

```
handler_type_t<CompletionToken, void(error_code, size_t)>
  handler(std::forward<CompletionToken>(token));
```

The `handler_type` template would be specialised for any completion token type that must participate in the extensible asynchronous model. For example, when we write:

```
auto fut = socket.async_receive(buffers, use_future);
```

the initiating function performs the equivalent of:

```
handler_type_t<use_future_t, void(error_code, size_t)> handler(token);
```

which produces a `handler` object of type `promise_handler<size_t>`. The `promise_handler<>` template is an implementation detail of Boost.Asio, and it simply sets a `std::promise<>` object's value when an asynchronous operation completes[7].

8.2 Customisation point 2: Create the initiating function's result

With the callbacks model, initiating functions always have a `void` return type. In the extensible model, the return type must be deduced and the return value determined. This is performed through the `async_result` type:

```
template <typename Handler>
class async_result {
public:
  // The return type of the initiating function.
  typedef ... type;

  // Construct an async result from a given handler.
  explicit async_result(Handler& handler) { ... }

  // Obtain initiating function's return type.
  type get() { return ...; }
};
```

The `async_result` template is specialised for handler types, and acts as the link between the handler (i.e. the continuation) and the initiating function's return value. For example, to support `std::future` the template is specialised for the `promise_handler<>` template:

```
template <typename T>
class async_result<promise_handler<T>> {
public:
  // The return type of the initiating function.
  typedef future<T> type;

  // Construct an async result from a given handler.
  explicit async_result(promise_handler<T>& h) { f_ = h.p_.get_future(); }

  // Obtain initiating function's return value.
  type get() { return std::move(f_); }

private:
  future<T> f_;
};
```

[7] A complete, standalone implementation of `use_future_t` can also be found in section 14.4.

8.3 Putting it together

Thus, to implement an initiating function that supports the extensible asynchronous model, the following modifications are required:

```
template <class Buffers, class CompletionToken>
typename async_result<
  handler_type_t<CompletionToken,
    void(error_code, size_t)>>::type
socket::async_receive(Buffers b, CompletionToken&& token)
{
  handler_type_t<CompletionToken, void(error_code, size_t)>
    handler(std::forward<CompletionToken>(token));

  async_result<decltype(handler)> result(handler);

  ...

  return result.get();
}
```

Deduce the initiating function's return type

Construct handler object

Link initiating function's return value to handler

Return the initiating function's result

To illustrate how this operates in practice, let us manually work through the steps that the compiler performs for us when we write:

```
auto fut = socket.async_receive(buffers, use_future);
```

First, after expanding uses of the `handler_type` trait to be the handler type, we get:

```
template <class Buffers>
typename async_result<promise_handler<size_t>>::type
socket::async_receive(Buffers b, use_future_t&& token)
{
  promise_handler<size_t> handler(std::forward<CompletionToken>(token));

  async_result<decltype(handler)> result(handler);

  ...

  return result.get();
}
```

Second, we expand the uses of the `async_result` template to get:

```
template <class Buffers>
std::future<size_t>
socket::async_receive(Buffers b, use_future_t&& handler)
{
  promise_handler<size_t> handler(std::forward<CompletionToken>(token));

  future<size_t> f = handler.p_.get_future();

  ...

  return std::move(f);
}
```

8.4 What happens with plain ol' callbacks

The default implementations of the `handler_type` and `async_result` templates behave as follows:

```
template <typename CompletionToken, typename Signature>
struct handler_type {
  typedef CompletionToken type;
};

template <typename Handler>
class async_result {
public:
  typedef void type;
  explicit async_result(Handler& h) { /* No-op */ }
  type get() { /* No-op */ }
};
```

These defaults are used when passing a simple callback (i.e. function object) as a completion token. In this case the compiler expands the templates such that the code is effectively:

```
template <class Buffers, typename CompletionToken>
void
socket::async_receive(Buffers b, CompletionToken&& token)
{
  Handler handler(std::forward<CompletionToken>(token));

  /* No-op */

  ...

  /* No-op */
}
```

This is equivalent to the code in Boost versions 1.53 and earlier. This means that the extensible model does not introduce any additional runtime overhead for programs that use callbacks.

9 Participating in the extensible model

9.1 Writing a participating asynchronous operation

Let us now summarise how to write an asynchronous operation that participates in the extensible model. Consider a hypothetical asynchronous operation with an initiating function named `async_foo`:

```
template <class Buffers, class CompletionToken>                          #1
auto async_foo(socket& s, Buffers b, CompletionToken&& token)
{                                                                        #2
  handler_type_t<CompletionToken, void(error_code, size_t)>              #3
    handler(std::forward<CompletionToken>(token));

  async_result<decltype(handler)> result(handler);                       #4

  ...                                                                    #5

  return result.get();                                                   #6
}
```

Participation in the model involves the following steps:

1. The initiating function accepts a completion token argument. The completion token specifies how the caller should be notified when the asynchronous operation completes.

2. The return type of the initiating function is automatically deduced.

3. The completion token is converted into a handler, i.e. a function object to be called when the asynchronous operation completes. The signature specifies the arguments that will be passed to the handler.

4. The handler is linked to the initiating function's result.

5. The asynchronous operation itself is performed. The implementation must call the handler only once, after the operation completes.

6. The linked result is returned.

9.1.1 Simplifying participation in the extensible model

Up to this point, this paper has focused on defining the customisation points required for an extensible asynchronous model. However, there is still some verbosity when instrumenting an asynchronous operation to participate in the extensible model. To address this, let us now introduce a new `async_completion` template, which is responsible for reifying a completion token into a handler and its linked asynchronous result.

```
template <class CompletionToken, class Signature>
struct async_completion
{
  typedef handler_type_t<CompletionToken, Signature> handler_type;

  async_completion(remove_reference_t<CompletionToken>& token)
    : handler(std::forward<CompletionToken>(token)),
      result(handler) {}

  handler_type handler;
  async_result<handler_type> result;
};
```

The hypothetical `async_foo` function above can then be simplified to:

```
template <class Buffers, class CompletionToken>
auto async_foo(socket& s, Buffers b, CompletionToken&& token)
{
  async_completion<CompletionToken,
    void(error_code, size_t)> completion(token);

  ...

  return completion.result.get();
}
```

This approach is currently used as an implementation detail of Boost.Asio.

In addition to simplifying code, use of the `async_completion` template enables further optimisation. For example, a copy/move may be elided if `token` is already an rvalue reference to a handler function object. The proposed wording and sample implementation below includes this optimisation.

For a complete, standalone example of a participating asynchronous operation, see section 12.2.

9.2 Writing a participating completion token

Next we will examine how to develop a custom completion token type. As a simple example, let us define a new special value named `block`:

```
constexpr struct block_t { constexpr block_t() {} } block;
```

The purpose of `block` is to make an asynchronous operation appear synchronous:

```
size_t length = socket.async_receive(buffers, block);
```

To do this, when passed to an asynchronous operation's initiating function, `block` will prevent the function from returning until the operation is complete. Once complete, the function will return the result of the asynchronous operation by mapping the completion handler signature to the function's return type as follows:

- If the signature is `void()` then the initiating function returns `void`.
- If the signature is `void(T)` then the function returns `T`.
- If the signature is `void(T...)` then the function returns `tuple<T...>`.

- If the signature is `void(error_code)` then the function returns void and any failure error code is thrown as a `system_error`.
- If the signature is `void(error_code, T)` then the function returns `T` and `system_error` is thrown on failure.
- If the signature is `void(error_code, T...)` then the function returns `tuple<T...>` and `system_error` is thrown on failure.

The `block` special value's synchronous emulation will be implemented using a `std::promise` object and a corresponding `std::future`. To integrate `block` into the extensible model we perform the following steps:

1. A handler type, `block_handler`, is defined with the responsibility to make a `std::promise` object ready when called.

```
template <typename... T>
struct block_handler
{
  typedef tuple<T...> value_type;
  promise<value_type> p;

  explicit block_handler(block_t) {}

  template <class... Args>
  void operator()(Args&&... args)
  {
    p.set_value(value_type(forward<Args>(args)...));
  }
};
```

The handler must be constructible from an rvalue of the completion token type, i.e. `block_t`

2. To map error codes into exceptions, we will also add a template specialisation for handlers that accept an `error_code` as their first argument.

```
template <class... T>
struct block_handler<error_code, T...>
{
  typedef tuple<T...> value_type;
  promise<value_type> p;

  explicit block_handler(block_t) {}

  template <class... Args>
  void operator()(error_code e, Args&&... args)
  {
    if (e)
      p.set_exception(make_exception_ptr(system_error(e)));
    else
      p.set_value(value_type(forward<Args>(args)...));
  }
};
```

3. The relationship between the completion token type, `block_t`, and the handler type, `block_handler`, is registered with the extensible model by specialising the `handler_type` trait.

```
namespace std {
  template <class R, class... Args>
  struct handler_type<block_t, R(Args...)>
  {
    typedef block_handler<decay_t<Args>...> type;
  };
} // namespace std
```

4. Finally, a specialisation of `async_result` completes the integration.

```
namespace std {
  template <class... T>
  class async_result<block_handler<T...>>
  {
    typedef typename block_handler<T...>::value_type value_type;
    future<value_type> f;

    static void unpack(tuple<>) {}

    template <typename U>
    static U unpack(tuple<U> t)
    {
      return move(std::get<0>(t));
    }

    template <typename... U>
    static tuple<U...> unpack(tuple<U...> u)
    {
      return u;
    }

  public:
    typedef decltype((unpack)(declval<value_type>())) type;

    explicit async_result(block_handler<T...>& h)
      : f(h.p.get_future()) {}

    type get() { return (unpack)(f.get()); }
  };
} // namespace std
```

Helper functions to convert a tuple into the correct return type and value, according to the mapping rules listed above

10 Further work

Polymorphic lambdas offer another approach to simplifying participation in the extensible model. In this approach, the `async_foo` function in section 9 could use a hypothetical `async_impl` helper function to implement the necessary boilerplate:

```
template <class Buffers, class CompletionToken>
auto async_foo(socket& s, Buffers b, CompletionToken&& token)
{
  return async_impl<CompletionToken, void(error_code, size_t)>(
    token,
    [&s, b](auto handler)
    {
      ...
    });
}
```

This approach may also be combined with specific models of composition. Here is a possible utility function that simplifies the creation of asynchronous operations using the stackless coroutines described in section 7.1:

```
template <class Buffers, class CompletionToken>
auto async_foo(socket& s, Buffers b, CompletionToken&& token)
{
  return go<void(error_code, size_t)>(
    std::forward<CompletionToken>(token),
    [&s, b](auto context)
    {
      reenter (context)
      {
        ...
        yield async_write(s, b, context);
        ...
      }
    });
}
```

It is worth noting that these approaches are not mutually exclusive. They may be added as future extensions without impacting existing code that is based on the extensible model.

11 Impact on the standard

Support for an extensible model of asynchronous operations is based entirely on library additions and does not require any language features beyond those that are already available in C++11.

Defining an extensible model would involve the addition of two new type traits, `handler_type` and `async_result`, and the class template `async_completion`. Furthermore, the standard may provide guidance on the use of these type traits in the implementation of asynchronous operations.

The standard library may also be extended to provide seamless support for `std::future` under the extensible model. This support is orthogonal to other proposed modifications to `std::future`, such as the addition of the `then()` member function.

If the extensible model is adopted then other facilities, such as library-based coroutines, may be considered as separate proposals.

12 Relationship to other proposals

12.1 Improvements to futures

As noted above in section 11, the standard library may be extended to provide the necessary traits specialisations for `std::future` support. However, this is orthogonal to the proposed modifications to `std::future` and related APIs.

12.2 Schedulers and executors

The extensible model proposed here is independent of proposals related to schedulers and executors. All that is required of an asynchronous operation is that it has a callback that is invoked after the operation completes. The delivery mechanism is not important.

By way of illustration, consider a simple wrapper around the Windows API function `RegisterWaitForSingleObject`. This function registers a callback to be invoked once a kernel object is in a signalled state, or if a timeout occurs. The callback is invoked from the system thread pool.

Original callback-based wrapper

```
template <class Handler> struct wait_op {
  atomic<HANDLE> wait_handle_;
  Handler handler_;
  explicit wait_op(Handler handler)
    : wait_handle_(0),
      handler_(handler) {}
};

template <class Handler>
void CALLBACK wait_callback(
    void* param, BOOLEAN timed_out)
{
  unique_ptr<wait_op<Handler>> op(
    static_cast<wait_op<Handler>*>(param));

  while (op->wait_handle_ == 0)
    SwitchToThread();

  const error_code ec = timed_out
    ? make_error_code(errc::timed_out)
    : error_code();
  op->handler_(ec);
}

template <class Handler>
void wait_for_object(
    HANDLE object, DWORD timeout,
    DWORD flags, Handler handler)
{









  unique_ptr<wait_op<Handler>>
    op(new wait_op<Handler>(handler));


  HANDLE wait_handle;
  if (RegisterWaitForSingleObject(
        &wait_handle, object,
        &wait_callback<Handler>,
        op.get(), timeout,
        flags | WT_EXECUTEONLYONCE))
  {
    op->wait_handle_ = wait_handle;
    op.release();
  }
  else
  {
    DWORD last_error = GetLastError();
    const error_code ec(
        last_error, system_category());
    op->handler_(ec);
  }


}
```

Traits-enabled wrapper

```
template <class Handler> struct wait_op {
  atomic<HANDLE> wait_handle_;
  Handler handler_;
  explicit wait_op(Handler handler)
    : wait_handle_(0),
      handler_(move(handler)) {}
};

template <class Handler>
void CALLBACK wait_callback(
    void* param, BOOLEAN timed_out)
{
  unique_ptr<wait_op<Handler>> op(
    static_cast<wait_op<Handler>*>(param));

  while (op->wait_handle_ == 0)
    SwitchToThread();

  const error_code ec = timed_out
    ? make_error_code(errc::timed_out)
    : error_code();
  op->handler_(ec);
}

template <class CompletionToken>
auto wait_for_object(
    HANDLE object, DWORD timeout,
    DWORD flags, CompletionToken&& token)
{
  async_completion<CompletionToken,
    void(error_code)> completion(token);

  typedef handler_type_t<CompletionToken,
    void(error_code)> Handler;

  unique_ptr<wait_op<Handler>>
    op(new wait_op<Handler>(
      move(completion.handler)));

  HANDLE wait_handle;
  if (RegisterWaitForSingleObject(
        &wait_handle, object,
        &wait_callback<Handler>,
        op.get(), timeout,
        flags | WT_EXECUTEONLYONCE))
  {
    op->wait_handle_ = wait_handle;
    op.release();
  }
  else
  {
    DWORD last_error = GetLastError();
    const error_code ec(
        last_error, system_category());
    op->handler_(ec);
  }

  return completion.result.get();
}
```

Once we have added support for the extensible asynchronous model, the `RegisterWaitForSingleObject` wrapper can be used with any model of composition presented in this paper. For example, here is the wrapper function when used with `std::future`:

```
try
{
  HANDLE in = GetStdHandle(STD_INPUT_HANDLE);
  future<void> fut = wait_for_object(
      in, 5000, WT_EXECUTEDEFAULT, use_future);
  ...
  fut.get();
}
catch (exception& e)
{
  ...
}
```

13 Proposed wording

Change the end of §20.10.2 [meta.type.synop] to add:

```
    // 20.10.8 traits for asynchronous operations
    template <class CompletionToken, class Signature, class = void>
      struct handler_type;
    template <class Handler> class async_result;
    template <class CompletionToken, class Signature>
      struct async_completion;

    template <class CompletionToken, class Signature>
      using handler_type_t =
        typename handler_type<CompletionToken, Signature>::type;
  } // namespace std
```

At the end of §20.10 [meta], add a new section 20.10.8 [meta.async]:

20.10.8 Traits for asynchronous operations [meta.async]

20.10.8.1 General [meta.async.general]

TODO – Add context, definitions, guidance and examples here.

The `handler_type` trait is used to determine the handler type for an asynchronous operation. A handler is a function object (20.9 [function.objects]) that is invoked on completion of the operation.

The `async_result` trait enables customization of the return type and return value of an asynchronous operation's initiating function.

The `async_completion` template may be used within an initiating function to reify a completion token into a handler and its linked asynchronous result.

20.10.8.2 Class template handler_type [meta.async.handler_type]

```
namespace std {
  template <class CompletionToken, class Signature, class = void>
  struct handler_type {
    typedef see below type;
  };
} // namespace std
```

Template parameter `CompletionToken` specifies the model used to obtain the result of the asynchronous operation. Template parameter `Signature` is the call signature (20.9.1 [func.def]) for the handler type invoked on completion of the asynchronous operation.

A program may specialize this trait if the `CompletionToken` template parameter in the specialization is a user-defined type.

Specializations of `handler_type` shall define a nested handler type `type` that satisfies the `MoveConstructible` requirements, and objects of type `type` shall be constructible from an lvalue or rvalue of the type specified by the `CompletionToken` template parameter.

20.10.8.2.1 handler_type members [meta.async.handler_type.members]

`typedef see below type;`

> *Type:* `CompletionToken` if `CompletionToken` and `decay_t<CompletionToken>` are the same type; otherwise, `handler_type_t<decay_t<CompletionToken>, Signature>`.

20.10.8.3 Class template async_result [meta.async.async_result]

```
namespace std {
  template <class Handler> class async_result {
  public:
    typedef void type;
    explicit async_result(Handler&);
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;
    type get();
  };
} // namespace std
```

Template argument `Handler` is a handler type produced by `handler_type_t<T, S>` for some completion token type `T` and call signature `S`.

A program may specialize this template if the `Handler` template parameter in the specialization is a user-defined type.

Specializations of `async_result` shall satisfy the `Destructible` requirements in addition to the requirements in the table below. In this table, `R` is a specialization of `async_result` for the template parameter `Handler`; `r` is a modifiable lvalue of type `R`; and `h` is a modifiable lvalue of type `Handler`.

Expression	Return type	Note
`R::type`		`void`; or a type satisfying `MoveConstructible` requirements.

`R r(h);`		
`r.get()`	`R::type`	The `get()` member function shall be used only as a return expression.

20.10.8.3.1 `async_result` members [meta.async.async_result.members]

```
explicit async_result(Handler&);
```

Effects: Does nothing.

```
type get();
```

Effects: Does nothing.

20.10.8.3 Class template `async_completion` [meta.async.async_completion]

```
  namespace std {
    template <class CompletionToken, class Signature>
    struct async_completion {
      typedef handler_type_t<CompletionToken, Signature> handler_type;

      explicit async_completion(remove_reference_t<CompletionToken>& t);
      async_completion(const async_completion&) = delete;
      async_completion& operator=(const async_completion&) = delete;

      see below handler;
      async_result<handler_type> result;
    };
  } // namespace std
```

Template parameter `CompletionToken` specifies the model used to obtain the result of the asynchronous operation. Template parameter `Signature` is the call signature (20.9.1 [func.def]) for the handler type invoked on completion of the asynchronous operation.

20.10.8.2.1 `async_completion` members [meta.async.async_completion.members]

```
explicit async_completion(remove_reference_t<CompletionToken>& t);
```

Effects: If `CompletionToken` and `handler_type` are the same type, binds `handler` to `t`; otherwise, initializes `handler` with the result of `forward<CompletionToken>(t)`. Initializes `result` with `handler`.

```
see below handler;
```

Type: `handler_type&` if `CompletionToken` and `handler_type` are the same type; otherwise, `handler_type`.

Change the end of §30.6.1 [futures.overview] to add:

```
    // 30.6.10 class template use_future_t
    template <class Allocator = allocator<void>> class use_future_t;
    constexpr use_future_t<> use_future;

    // 30.6.10.2 handler_type specialization
    template <class Allocator, class Ret, class... Args>
      struct handler_type<use_future_t<Allocator>, Ret(Args...)>;
  } // namespace std
```

At the end of §30.6 [futures], add a new section 30.6.10 [futures.use_future]:

30.6.10 Class template `use_future_t` [futures.use_future]

```
  namespace std {
    template <class Allocator = allocator<void>> class use_future_t {
    public:
      // member types
      typedef Allocator allocator_type;

      // 30.6.10.1 members
      constexpr use_future_t() noexcept;
      explicit use_future_t(const Allocator& a) noexcept;
      template <class OtherAllocator> use_future_t<OtherAllocator>
        operator[](const OtherAllocator& a) const noexcept;
      allocator_type get_allocator() const noexcept;
    };
  } // namespace std
```

The class template `use_future_t` defines a set of completion token types (20.10.8.2 [meta.async.handler_type]) for use with asynchronous operations.

30.6.10.1 `use_future_t` members [futures.use_future.members]

```
constexpr use_future_t() noexcept;
```

Effects: Constructs a `use_future_t` with default-constructed allocator.

```
explicit use_future_t(const Allocator& a) noexcept;
```

Effects: Constructs an object of type `use_future_t` with post-condition `get_allocator() == a`.

```
template <class OtherAllocator> use_future_t<OtherAllocator>
  operator[](const OtherAllocator& a) const noexcept;
```

Returns: A `use_future_t` object where `get_allocator() == a`.

```
allocator_type get_allocator() const noexcept;
```

Returns: The associated allocator object.

30.6.10.2 use_future_t traits [futures.use_future.traits]

```
template <class Allocator, class Ret, class... Args>
  struct handler_type<use_future_t<Allocator>, Ret(Args...)> {
    typedef see below type;
};
```

An object `t1` of the nested function object type `type` is an asynchronous provider with an associated shared state (30.6.4 [futures.state]). The type `type` provides `type::operator()` such that the expression `t1(declval<Args>()...)` is well formed. The implementation shall specialize `async_result` for `type` such that, when an `async_result` object `r1` is constructed from `t1`, the expression `r1.get()` returns a future with the same shared state as `t1`.

The semantics of `async_result::type` and `type::operator()` are defined in the table below. In this table, N is the value of `sizeof...(Args)`; let i be in the range `[0,`N`)` and let T_i be the i^{th} type in `Args`; let U_i be `decay<`T_i`>::type` for each type T_i in `Args`; and let a_i be the i^{th} argument to `type::operator()`.

N	U_0	`async_result::type`	`type::operator()` effects
0		`future<void>`	Makes the shared state ready.
1	`error_code`	`future<void>`	If a_0 evaluates to `true`, atomically stores the exception pointer produced by `make_exception_ptr(system_error(`a_0`))` in the shared state. The shared state is made ready.
1	`exception_ptr`	`future<void>`	If a_0 is non-null, atomically stores the exception pointer a_0 in the shared state. The shared state is made ready.
1	all other types	`future<`U_0`>`	Atomically stores a_0 in the shared state and makes that state ready.
2	`error_code`	`future<`U_1`>`	If a_0 evaluates to `true`, atomically stores the exception pointer produced by `make_exception_ptr(system_error(`a_0`))` in the shared state; otherwise, atomically stores a_1 in the shared state. The shared state is made ready.
2	`exception_ptr`	`future<`U_1`>`	If a_0 is non-null, atomically stores the exception pointer a_0 in the shared state; otherwise, atomically stores a_1 in the shared state. The shared state is made ready.
2	all other types	`future<` `  tuple<`U_0`,`U_1`>>`	Atomically stores the result of `make_tuple(`a_0`,` a_1`)` in the shared state and makes that state ready.

>2	`error_code`	`future<`
		`  tuple<`
		`  `U_1`,…,`U_{N-1}`>>`
			If a_0 evaluates to `true`, atomically stores the exception pointer produced by `make_exception_ptr(system_error(`a_0`))`

			in the shared state; otherwise, atomically stores the result of `make_tuple(`a_1`,…,`a_{N-1}`)` in the shared state. The shared state is made ready.

>2	`exception_ptr`	`future<`
		`  tuple<`
		`  `U_1`,…,`U_{N-1}`>>`
			If a_0 is non-null, atomically stores the exception pointer a_0 in the shared state; otherwise, atomically stores the result of `make_tuple(`a_1`,…,`a_{N-1}`)` in the shared state. The shared state is made ready.

>2	all other types	`future<`
		`  tuple<`
		`  `U_0`,…,`U_{N-1}`>>`
			Atomically stores the result of `make_tuple(`a_0`,…,`a_{N-1}`)` in the shared state and makes that state ready.

14 Sample implementation

14.1 Class template `handler_type`

```
namespace std {

template <class _CompletionToken, class _Signature, class = void>
struct handler_type
{
  typedef typename conditional<
    is_same<_CompletionToken, typename decay<_CompletionToken>::type>::value,
    decay<_CompletionToken>,
    handler_type<typename decay<_CompletionToken>::type, _Signature>
  >::type::type type;
};

template <class _CompletionToken, class _Signature>
using handler_type_t = typename handler_type<_CompletionToken, _Signature>::type;

} // namespace std
```

14.2 Class template `async_result`

```
namespace std {

template <class _Handler>
class async_result
{
public:
  typedef void type;
  explicit async_result(_Handler&) {}
  async_result(const async_result&) = delete;
  async_result& operator=(const async_result&) = delete;
  void get() {}
};

} // namespace std
```

14.3 Class template `async_completion`

```
namespace std {

template <class _CompletionToken, class _Signature>
struct async_completion
{
  typedef handler_type_t<_CompletionToken, _Signature> handler_type;

  explicit async_completion(
    typename remove_reference<_CompletionToken>::type& __token)
    : handler(static_cast<typename conditional<
        is_same<_CompletionToken, handler_type>::value,
        handler_type&, _CompletionToken&&>::type>(__token)),
      result(handler) {}

  async_completion(const async_completion&) = delete;
  async_completion& operator=(const async_completion&) = delete;

  typename conditional<
    is_same<_CompletionToken, handler_type>::value,
    handler_type&, handler_type>::type handler;
  async_result<handler_type> result;
};

} // namespace std
```

14.4 Class template `use_future_t`

```
namespace std {

template <class _Alloc = allocator<void>>
class use_future_t
{
public:
  typedef _Alloc allocator_type;

  constexpr use_future_t() noexcept {}
  explicit use_future_t(const _Alloc& __a) noexcept : _M_allocator(__a) {}

  template <class _OtherAlloc>
  use_future_t<_OtherAlloc> operator[](const _OtherAlloc& __a) const noexcept
  {
    return use_future_t<_OtherAlloc>(__a);
  }

  allocator_type get_allocator() const noexcept { return _M_allocator; }

private:
  allocator_type _M_allocator;
};

constexpr use_future_t<> use_future;

template <class... _Values>
struct __value_pack
{
  typedef tuple<_Values...> _Type;

  template <class... _Args>
  static void _Apply(promise<_Type>& p, _Args&&... __args)
  {
    p.set_value(std::make_tuple(forward<_Args>(__args)...));
  }
};

template <class _Value>
struct __value_pack<_Value>
{
  typedef _Value _Type;
  template <class _Arg>
  static void _Apply(promise<_Type>& p, _Arg&& __arg) { p.set_value(forward<_Arg>(__arg)); }
};

template <>
struct __value_pack<>
{
  typedef void _Type;
  static void _Apply(promise<_Type>& p) { p.set_value(); }
};

template <class... _Values>
struct __promise_handler
{
  promise<typename __value_pack<_Values...>::_Type> _M_promise;

  template <class _Alloc>
  __promise_handler(use_future_t<_Alloc> __u)
    : _M_promise(allocator_arg, __u.get_allocator()) {}

  template <class... _Args>
  void operator()(_Args&&... __args)
  {
```

```
    __value_pack<_Values...>::_Apply(_M_promise, forward<_Args>(__args)...);
  }
};

template <class... _Values>
struct __promise_handler<error_code, _Values...>
{
  promise<typename __value_pack<_Values...>::_Type> _M_promise;

  template <class _Alloc>
  __promise_handler(use_future_t<_Alloc> __u)
    : _M_promise(allocator_arg, __u.get_allocator()) {}

  template <class... _Args>
  void operator()(const error_code& __e, _Args&&... __args)
  {
    if (__e)
      _M_promise.set_exception(make_exception_ptr(system_error(__e)));
    else
      __value_pack<_Values...>::_Apply(_M_promise, forward<_Args>(__args)...);
  }
};

template <class... _Values>
struct __promise_handler<exception_ptr, _Values...>
{
  promise<typename __value_pack<_Values...>::_Type> _M_promise;

  template <class _Alloc>
  __promise_handler(use_future_t<_Alloc> __u)
    : _M_promise(allocator_arg, __u.get_allocator()) {}

  template <class... _Args>
  void operator()(const exception_ptr& __e, _Args&&... __args)
  {
    if (__e)
      _M_promise.set_exception(__e);
    else
      __value_pack<_Values...>::_Apply(_M_promise, forward<_Args>(__args)...);
  }
};

template <class... _Values>
class async_result<__promise_handler<_Values...>>
{
public:
  typedef __promise_handler<_Values...> _Handler;
  typedef decltype(declval<_Handler>()._M_promise) _Promise;
  typedef decltype(declval<_Promise>().get_future()) type;

  explicit async_result(_Handler& __h) : _M_future(__h._M_promise.get_future()) {}
  async_result(const async_result&) = delete;
  async_result& operator=(const async_result&) = delete;
  type get() { return std::move(_M_future); }

private:
  type _M_future;
};

template <class _Alloc, class _R, class... _Args>
struct handler_type<use_future_t<_Alloc>, _R(_Args...)>
{
  typedef __promise_handler<typename decay<_Args>::type...> type;
};

} // namespace std
```

15 Conclusion

Asynchronous operations have gained widespread use and acceptance in domains such as network programming. In many of these use cases, performance is important and the inherent runtime penalties of `std::future` make it an inappropriate choice for a fundamental building block.

With the extensible asynchronous model presented in this paper, we have the ability to select an asynchronous approach that is appropriate to each use case. With appropriate library foundations, a single extensible asynchronous model can support:

- Callbacks, where minimal runtime penalty is desirable.
- Futures, and not just `std::future` but also future classes supplied by other libraries.
- Coroutines or resumable functions, without adding new keywords to the language.

Perhaps most importantly, with the customisation points that the extensible model provides, it can support other tools for managing asynchronous operations, including ones that we have not thought of yet. An extensible asynchronous model broadens the vocabulary that is available for managing asynchronous control flow, but with a relatively small impact on the standard.

16 Acknowledgements

The author would like to thank Jamie Allsop for providing extensive feedback, suggestions and corrections. The author also gratefully acknowledges the assistance of Oliver Kowalke in allowing extensible model support to be added to his proposed Boost.Fiber library.