std::execution in Asio Codebases

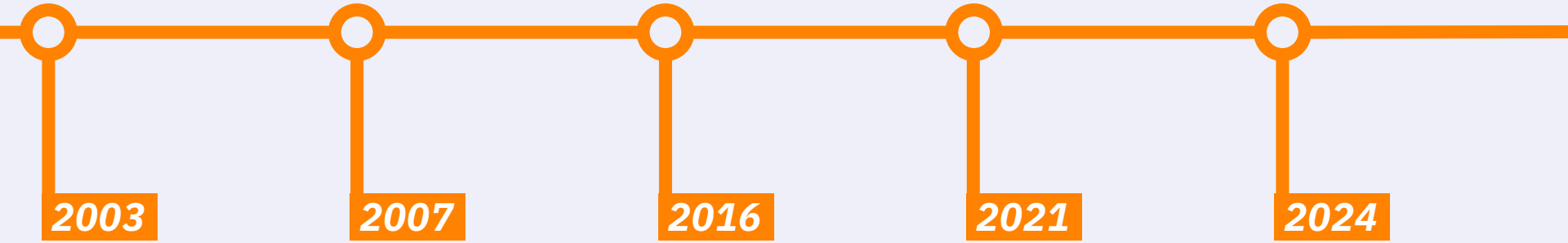
Adopting Senders Without a Rewrite

ROBERT LEAHY

C++26

Senders & receivers have arrived

C++ Asynchronous Programming Timeline



2003

Asio Released

Accepted into Boost in 2005.

Available in Boost and standalone flavors to this day.

2007

Networking Library Proposal for TR2

N2175.

Boost.Asio held up as reference implementation, evolved into *C++ Extensions for Networking* (AKA “The Networking TS”) (N4771) (2018).

2016

A Unified Executors Proposal for C++

P0443.

Revised 14 times with the final revision being published in 2020.

Senders & receivers included from revision 10 onwards having originated in P1055 (see also P1194, P1341, P1658, P1660).

2021

`std::execution`

P2300.

Executors are discarded completely.

2024

WG21 Meeting in St. Louis, MO

P2300 revision 10 voted into the C++26 working draft.

20+ years!

- That’s a long time, especially in software engineering.
- A lot of Asio-based code has been written.
- New method has advantages (as we’ll see), but we’re tethered to old code (which we can’t casually discard).
- Classic problem: How do we adopt newer, better patterns without a complete rewrite?

```
asio::io_context ctx;
asio::ip::tcp::resolver resolver(ctx);
asio::ip::tcp::socket socket(ctx);
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
auto sender = exec::sequence(

        );
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured)
```

```
      );
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
```

```
    );
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
```

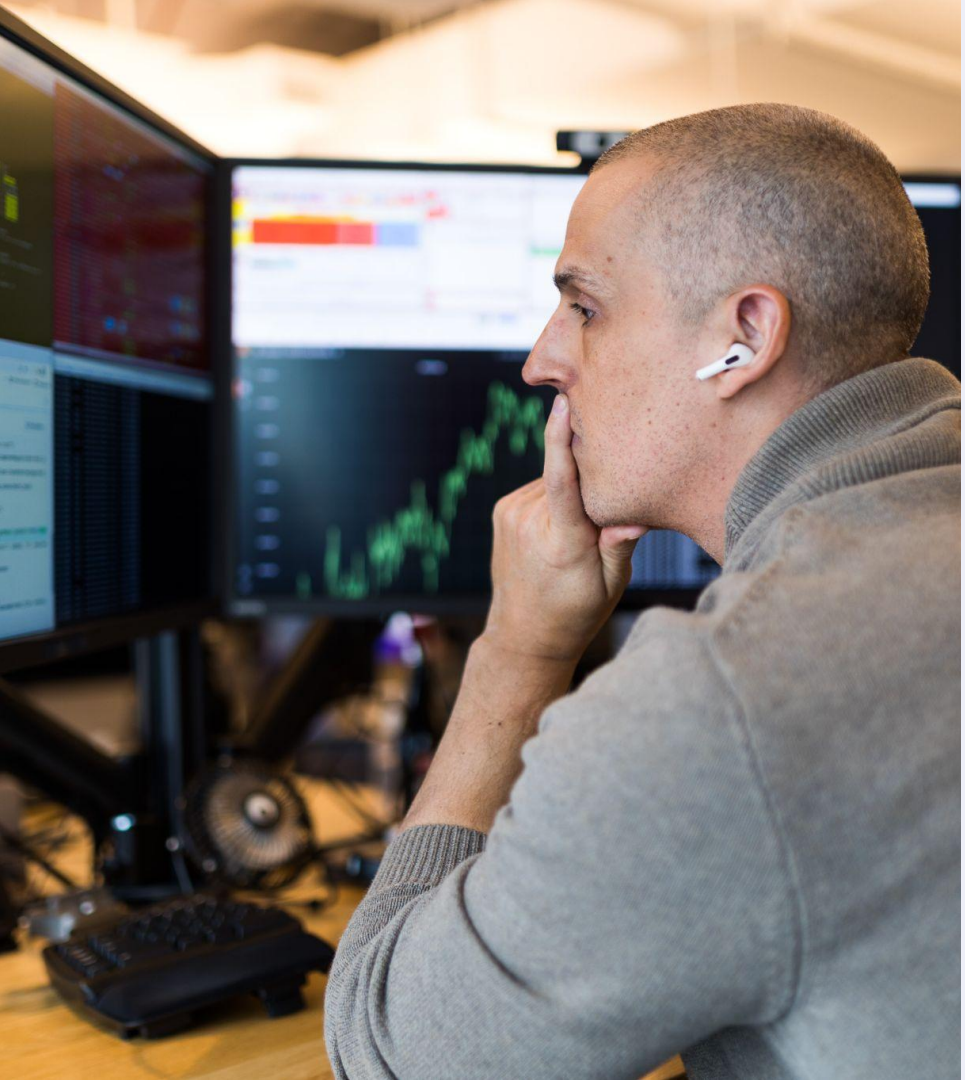
```
      );
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
  exec::repeat_effect_until(
```

```
    ));
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
  exec::repeat_effect_until(
    std::execution::just() |
    std::execution::let_value([&]() {
      const auto prev = read.size();
      read.resize(read.size() + 64);
      const asio::mutable_buffer buffer(
        read.data() + prev,
        read.size() - prev);
      return
        socket.async_read_some(buffer)




    })));
```

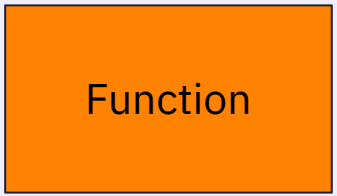
```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
  exec::repeat_effect_until(
    std::execution::just() |
    std::execution::let_value([&]() {
      const auto prev = read.size();
      read.resize(read.size() + 64);
      const asio::mutable_buffer buffer(
        read.data() + prev,
        read.size() - prev);
      return
        socket.async_read_some(buffer) |
        std::execution::then([&, prev](const auto bytes_transferred) {
          read.resize(prev + bytes_transferred);
          return bool(bytes_transferred);
        });
    })));
```



S&R vs. Asio

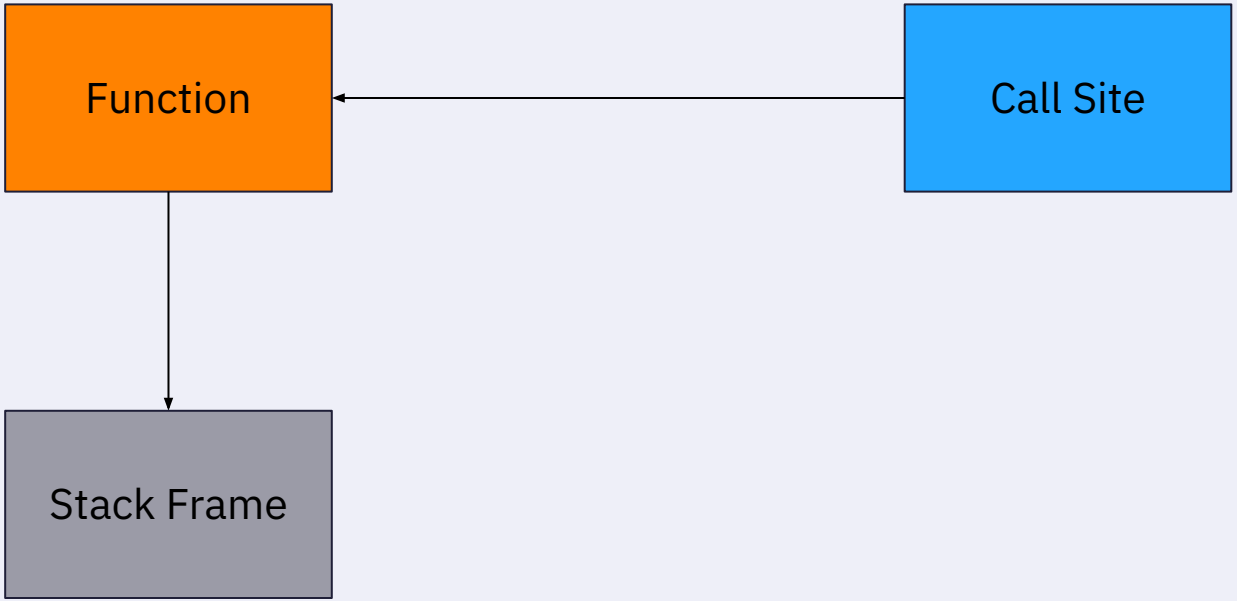
What’s the difference?

Function Call

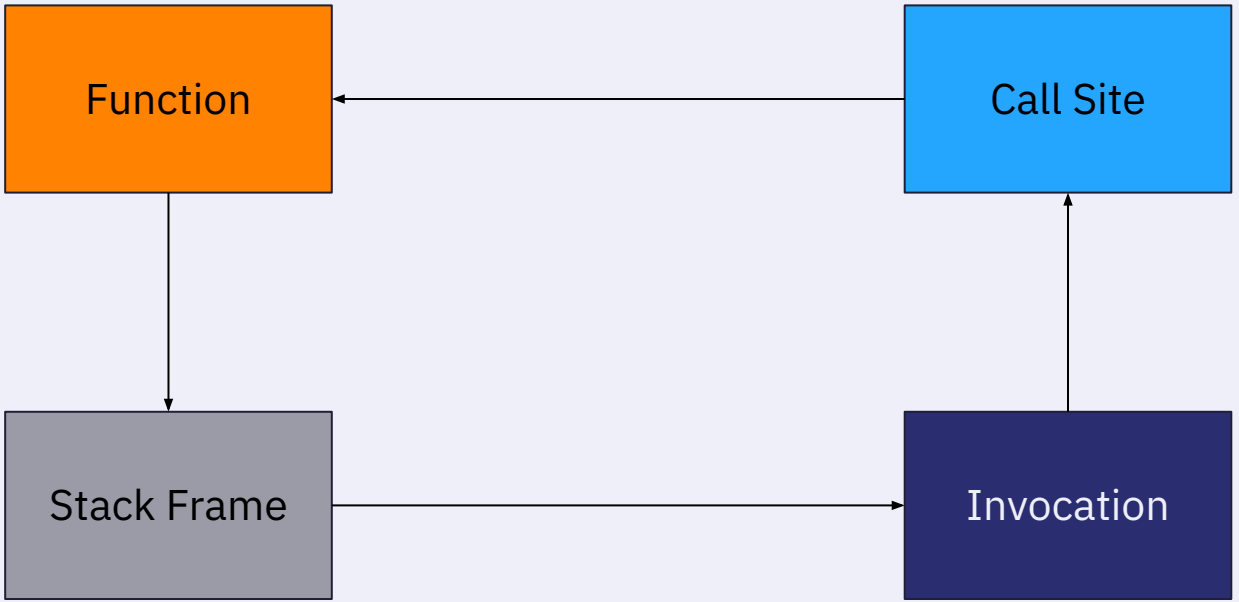


Function Call

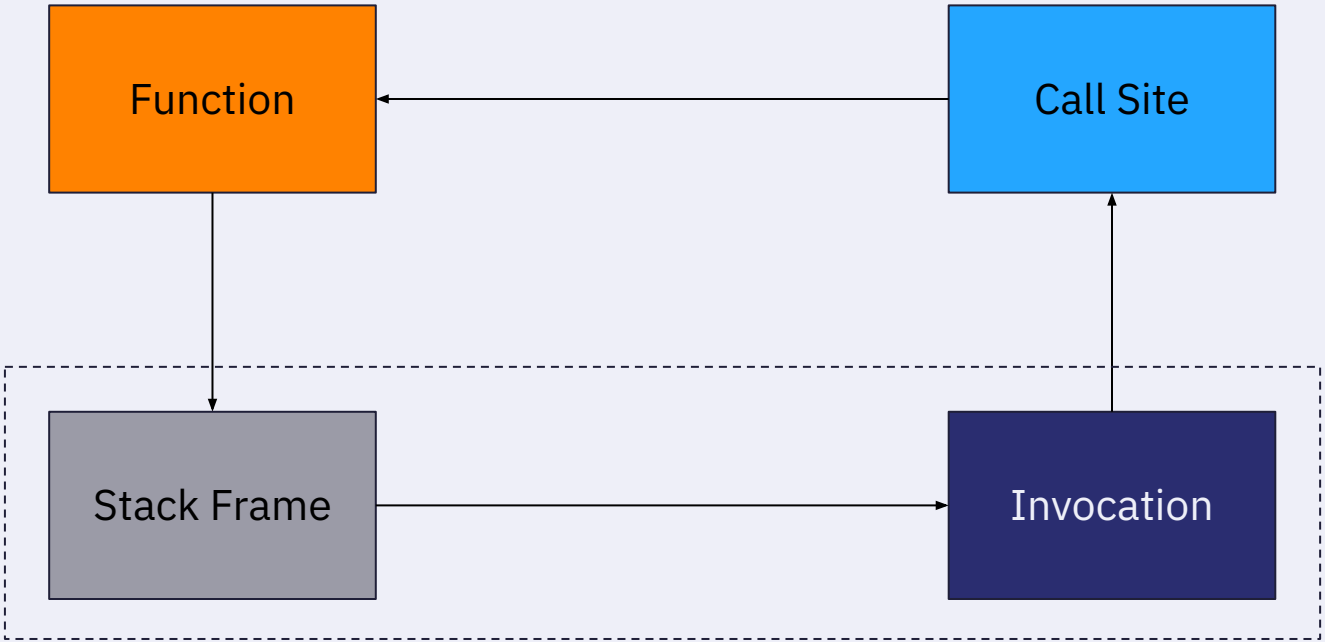
Function Call



Function Call



Function Call



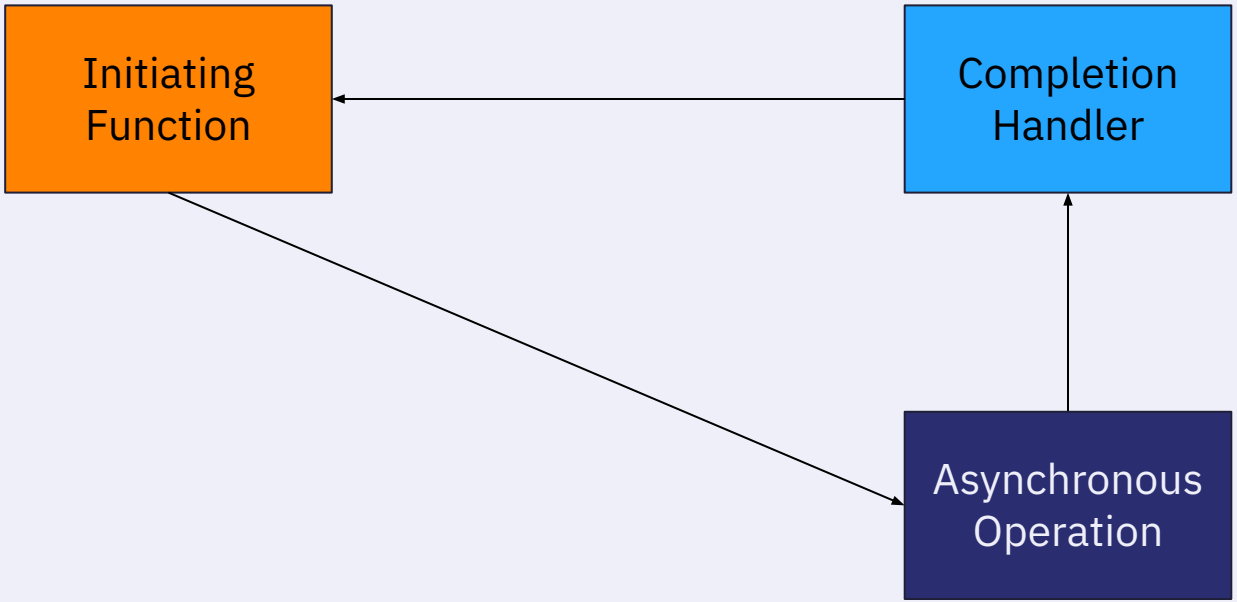
```
void read(const int fd, std::span<std::byte> span) {
  do {
    const auto read = ::read(fd, span.data(), span.size());
    if (!read) {
      throw eof_error{};
    }
    if (read == -1) {
      throw /* ... */;
    }
    span = span.subspan(read);
  } while (!span.empty());
}
```

```
void read(const int fd, std::span<std::byte> span) {
  do {
    const auto read = ::read(fd, span.data(), span.size());
    if (!read) {
      throw eof_error{};
    }
    if (read == -1) {
      throw /* ... */;
    }
    span = span.subspan(read);
  } while (!span.empty());
}
```

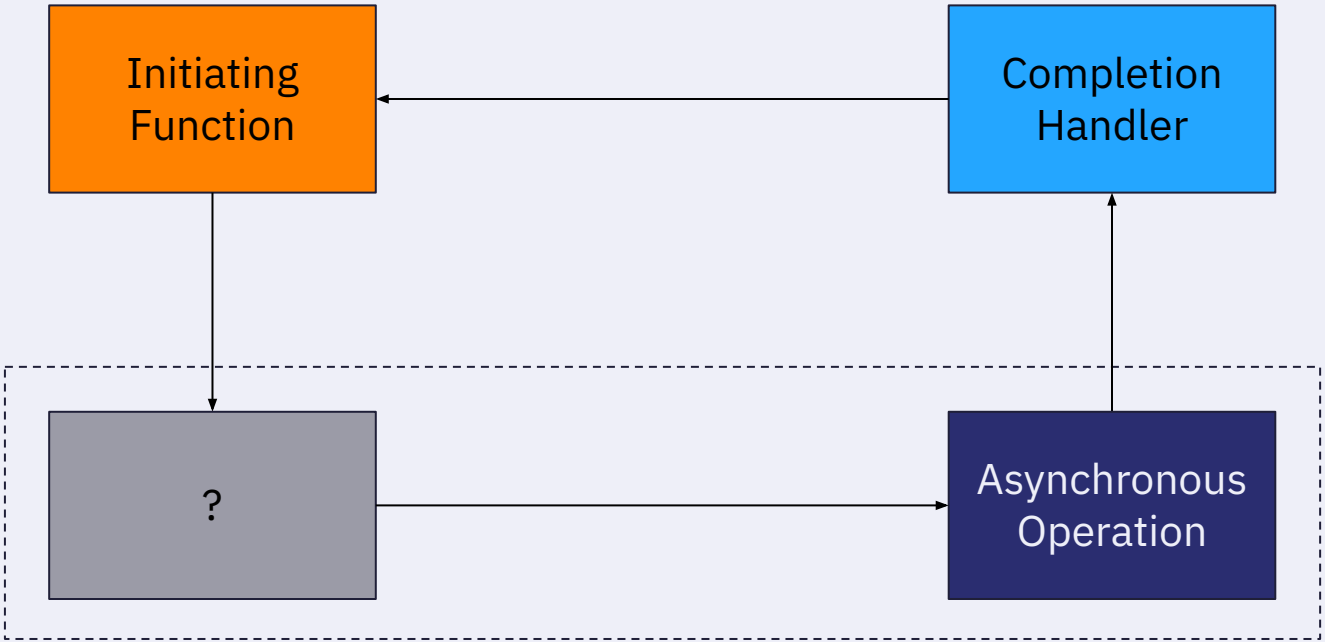
Asio

Asio

Asio


Initiating Function
Completion Handler
Asynchronous Operation

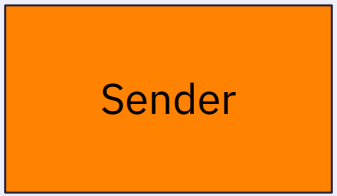
Asio, maybe?


Initiating Function
Completion Handler
?
Asynchronous Operation

```cpp
template<typename Readable, typename Handler>
void read(Readable& readable, const std::span<std::byte> span, Handler h) {
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    if (ec) {
      std::move(h)(ec);
      return;
    }
    if (!bytes) {
      std::move(h)(make_error_code(asio::errors::eof));
      return;
    }
    span = span.subspan(bytes);
    if (span.empty()) {
      std::move(h)(ec);
      return;
    }
    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, std::move(self));
  });
}
```

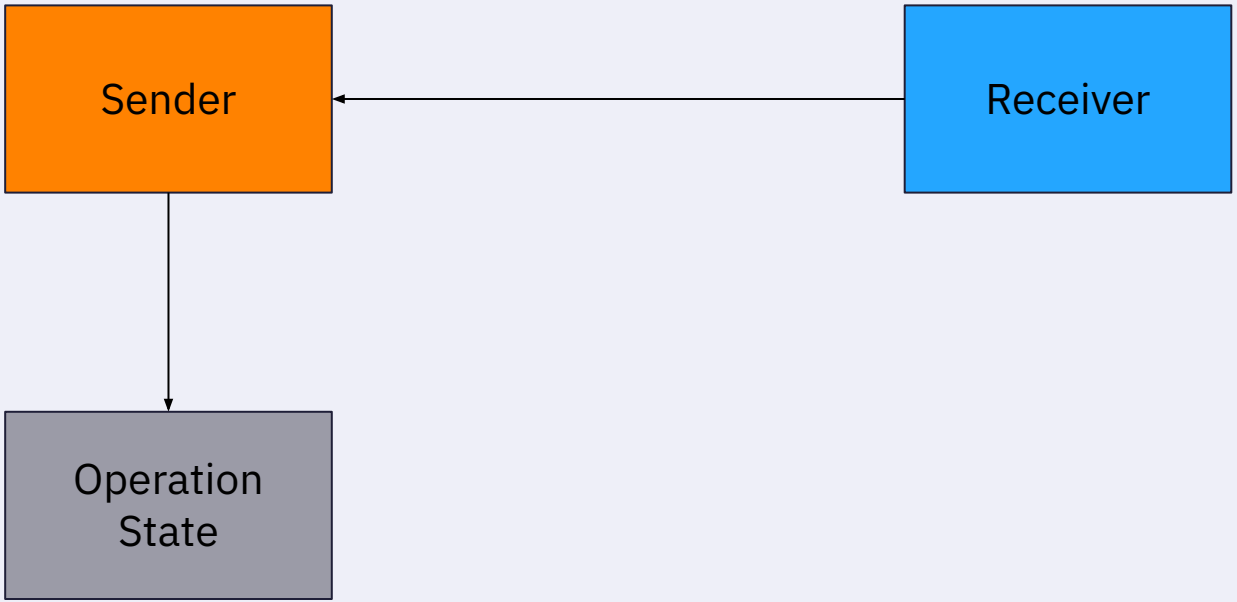
```
template<typename Readable, typename Handler>
void read(Readable& readable, const std::span<std::byte> span, Handler h) {
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    if (ec) {
      std::move(h)(ec);
      return;
    }
    if (!bytes) {
      std::move(h)(make_error_code(asio::errors::eof));
      return;
    }
    span = span.subspan(bytes);
    if (span.empty()) {
      std::move(h)(ec);
      return;
    }
    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, std::move(self));
  });
}
```

std::execution

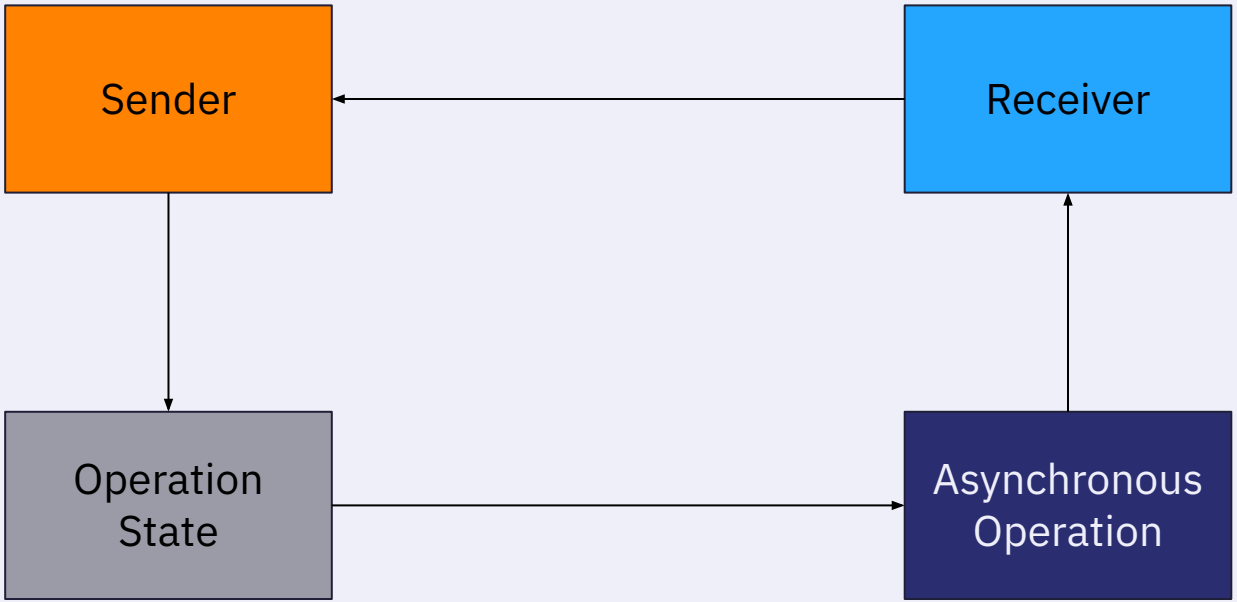


std::execution

std::execution



std::execution



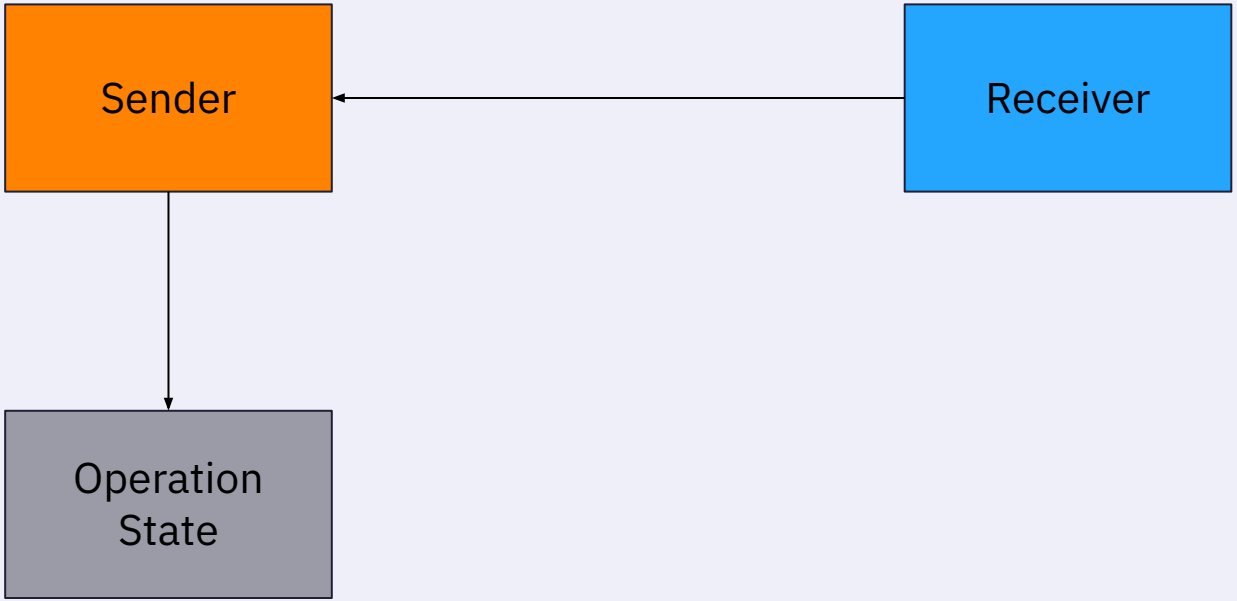
```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::execution::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        }) |
        std::execution::then([&span](std::size_t bytes) {
          if (!bytes) {
            throw eof_error{};
          }
          span = span.subspan(bytes);
          return span.empty();
        }));
    });
}
```

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::execution::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        }) |
        std::execution::then([&span](std::size_t bytes) {
          if (!bytes) {
            throw eof_error{};
          }
          span = span.subspan(bytes);
          return span.empty();
        }));
    });
}
```

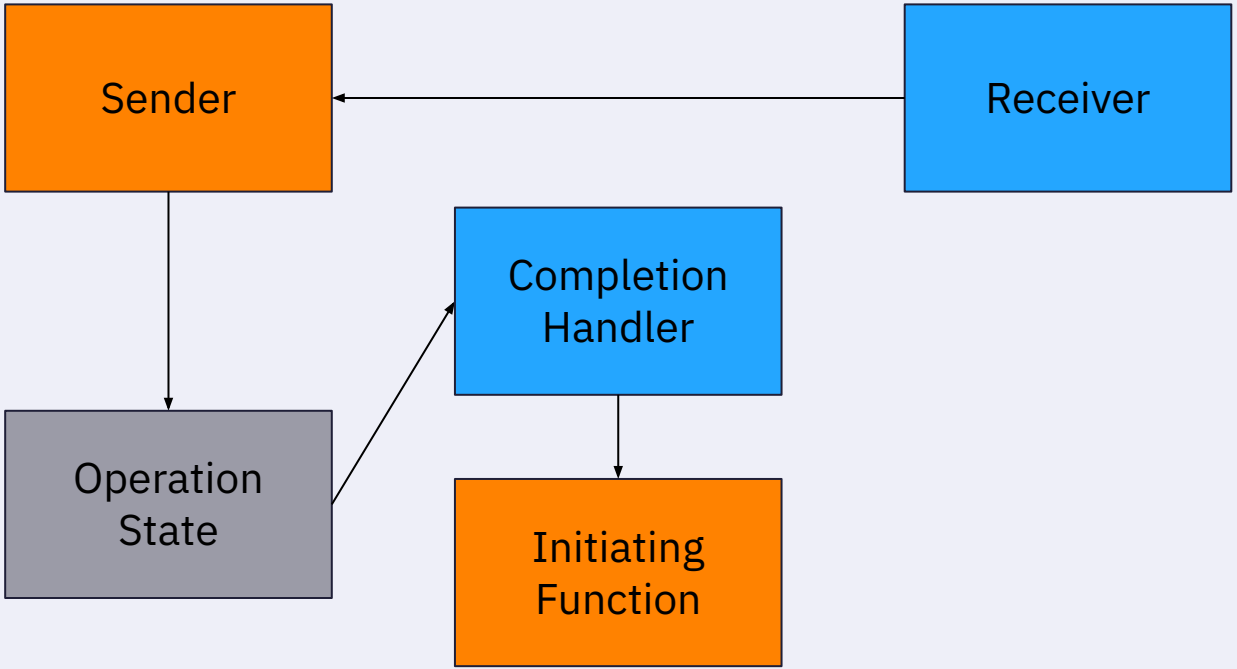
S&R & Asio

How can they work together?

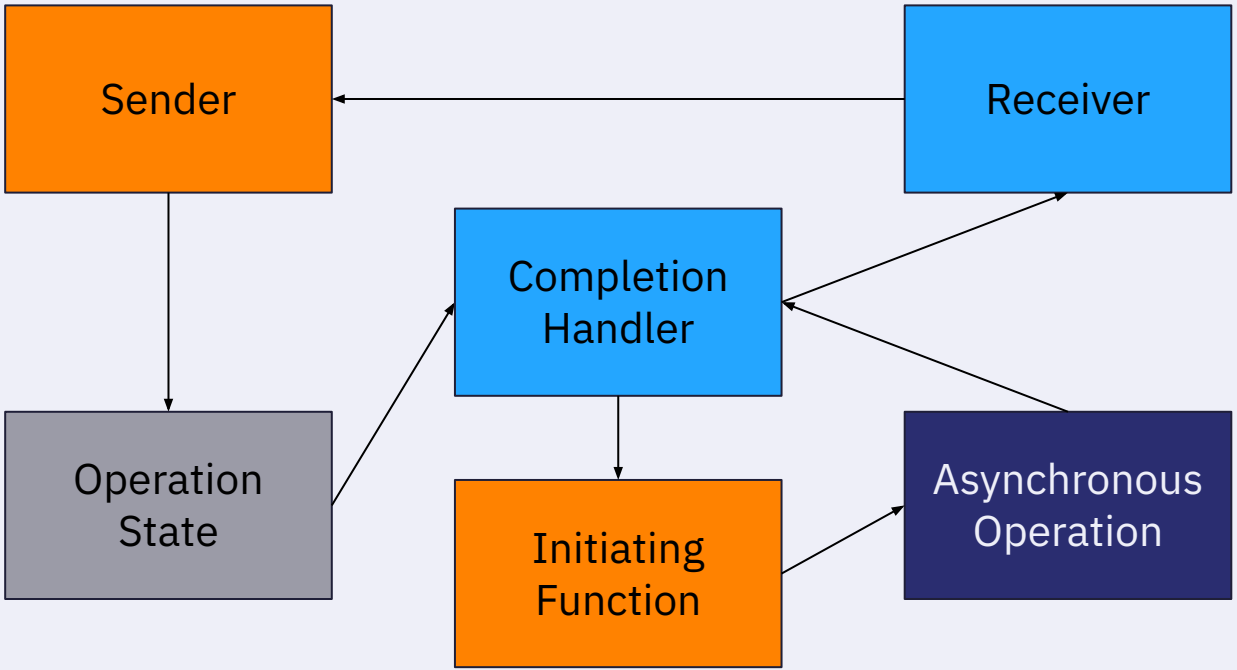
S&R & Asio



S&R & Asio


Sender
Receiver
Completion Handler
Operation State
Initiating Function

S&R & Asio


Sender
Receiver
Completion Handler
Operation State
Initiating Function
Asynchronous Operation

```cpp
template<typename Initiation>
constexpr auto make_asio_sender(Initiation init) {
  return sender<Initiation>{std::move(init)};
}
```

```cpp
template<typename Initiation>
struct sender {

  using sender_concept = std::execution::sender_t;

  Initiation init_;

  template<typename Self, typename Receiver>
  constexpr auto connect(
    this Self&& self,
    Receiver recv)
  {
    return operation<Receiver, Initiation>{
      std::move(recv),
      std::forward<Self>(self).init_};
  }
};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  using operation_state_concept = std::execution::operation_state_t;

  Receiver recv_;
  Initiation init_;

  constexpr void start() & noexcept {
    std::invoke(
      std::move(init_),
      [this](auto&&... args) noexcept {
        std::execution::set_value(
          std::move(recv_),
          std::forward<decltype(args)>(args)...);
      });
  }
};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  using operation_state_concept = std::execution::operation_state_t;

  Receiver recv_;
  Initiation init_;

  constexpr void start() & noexcept {
    std::invoke(
      std::move(init_),
      [this](auto&&... args) noexcept {
        std::execution::set_value(
          std::move(recv_),
          std::forward<decltype(args)>(args)...);
      });
  }
};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  using operation_state_concept = std::execution::operation_state_t;

  Receiver recv_;
  Initiation init_;

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        [this](auto&&... args) noexcept {
          std::execution::set_value(
            std::move(recv_),
            std::forward<decltype(args)>(args)...);
        });
    } catch (...) {
      std::execution::set_error(
        std::move(recv_),
        std::current_exception());
    }
  }
};
```

Return Values

Async function signatures

```
std::execution::completion_signatures<
  std::execution::set_value(int),
  std::execution::set_error(std::exception_ptr)>;
```

```
template<typename Initiation>
struct sender {

  template<typename Self, typename Env>
  static consteval
    std::execution::completion_signatures</* ???? */>
    get_completion_signatures()
  {
    return {};
  }

};
```

```cpp
template<typename Readable, typename Handler>
void read(
  Readable& readable,
  const std::span<std::byte> span,
  Handler h)
{



    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, [&readable, span, h = std::move(h)](
      this auto&& self,
      const std::error_code ec,
      const std::size_t bytes) mutable
    {
      /* ... */
    });



}
```

```cpp
template<typename Readable, typename Handler>
void read(
  Readable& readable,
  const std::span<std::byte> span,
  Handler h)
{

    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, [&readable, span, h = std::move(h)](
      this auto&& self,
      const std::error_code ec,
      const std::size_t bytes) mutable
    {
      /* ... */
    });

}
```

```cpp
template<typename Readable, typename CompletionToken>
void read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    /* ... */
  });
}
```

```cpp
template<typename Readable, typename CompletionToken>
void read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{



    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, [&readable, span, h = std::move(h)](
      this auto&& self,
      const std::error_code ec,
      const std::size_t bytes) mutable
    {
      /* ... */
    });



}
```

```cpp
template<typename Readable, typename CompletionToken>
void read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{



    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, [&readable, span, h = std::move(h)](
      this auto&& self,
      const std::error_code ec,
      const std::size_t bytes) mutable
    {
      /* ... */
    });



}
```

```cpp
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

```cpp
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

```cpp
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

```cpp
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

```cpp
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

```
template<typename Readable, typename CompletionToken>
decltype(auto) read(
  Readable& readable,
  const std::span<std::byte> span,
  CompletionToken&& token)
{

  return asio::async_initiate<CompletionToken, void(std::error_code)>(
    [&readable, span](auto h) {

      const asio::mutable_buffer cb(span.data(), span.size());
      readable.async_read_some(cb, [&readable, span, h = std::move(h)](
        this auto&& self,
        const std::error_code ec,
        const std::size_t bytes) mutable
      {
        /* ... */
      });
    },

    token);
  });
}
```

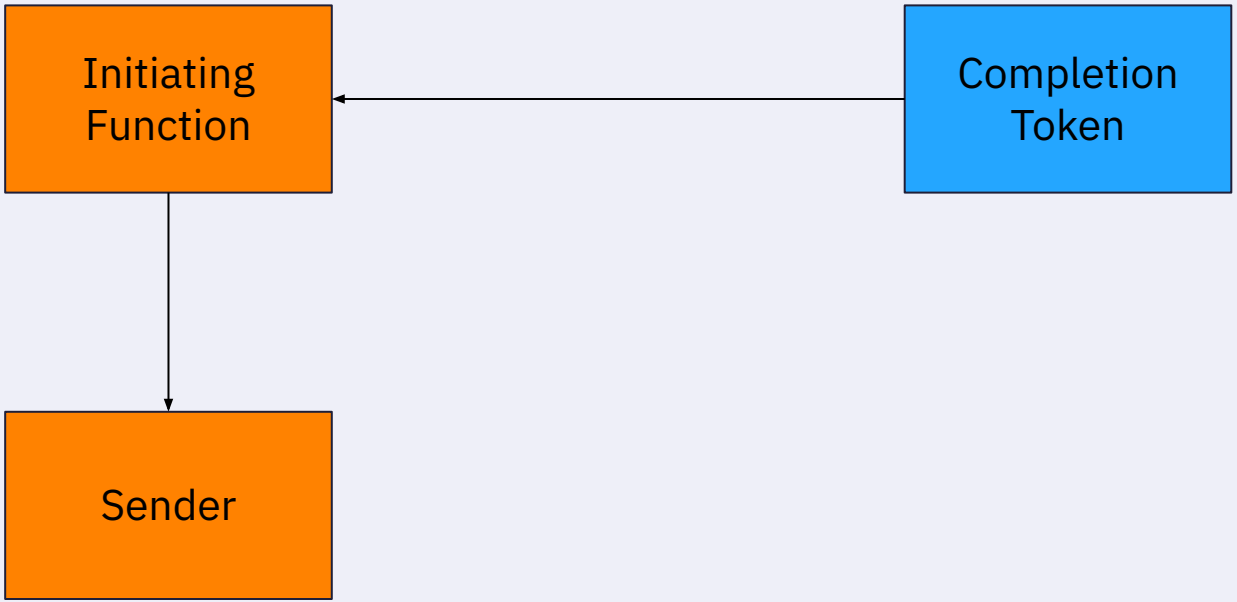
Completion Tokens

Transforming Asio operations

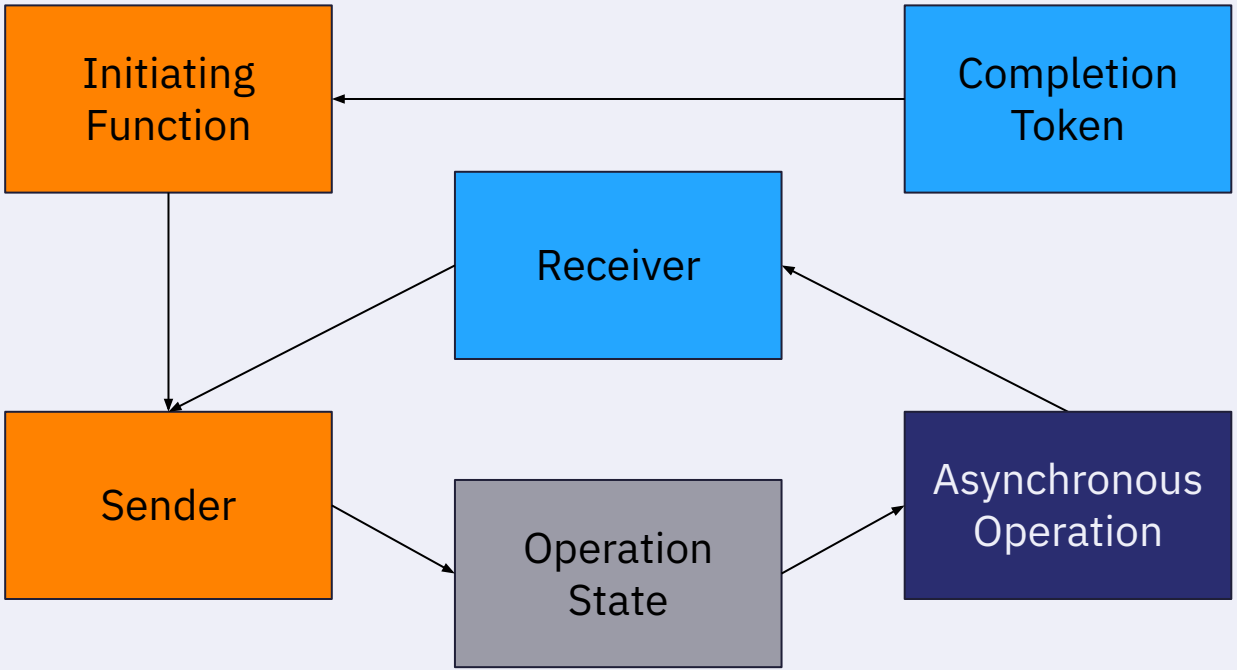
Completion Token

Completion Token

Completion Token



Completion Token



```
struct completion_token_t {};

inline constexpr completion_token_t completion_token;
```

```cpp
namespace asio {

template <typename... Signatures>
struct async_result<completion_token_t, Signatures...> {

  template <typename Initiation, typename... Args>
  static constexpr auto initiate(
    Initiation init,
    const completion_token_t&,
    Args... args)
  {
    auto f = [init = std::move(init), ...args = std::move(args)](
      this auto&& self, auto h)
    {
      std::invoke(
        std::forward_like<decltype(self)>(init),
        std::move(h),
        std::forward_like<decltype(self)>(args)...);
    };
    return sender<transform_signatures<Signatures...>, decltype(f)>{
      std::move(f)};
  }

};
```

```cpp
template<typename T>
struct transform_signature;

template<typename... Args>
struct transform_signature<void(Args...)> {
  using type = std::execution::set_value_t(Args...);
};

template<typename... Signatures>
using transform_signatures = std::execution::completion_signatures<
  typename transform_signature<Signatures>::type...,
  std::execution::set_error_t(std::exception_ptr)>;
```

```
template<typename Signatures, typename Initiation>
struct sender {



  template<typename Self, typename Env>
    requires std::is_constructible_v<
      Initiation,
      decltype(std::forward_like<Self>(std::declval<Initiation&>()))>
  static consteval Signatures get_completion_signatures() noexcept {
    return {};
  }



};
```

```
template<typename Signatures, typename Initiation>
struct sender {

  template<typename Self, typename Receiver>
    requires std::execution::receiver_of<
      Receiver,
      std::execution::completion_signatures_of_t<
        sender,
        std::execution::env_of_t<Receiver>>>
  constexpr auto connect(this Self&& self, Receiver r) {
    return operation<Receiver, Initiation>{
      std::move(r),
      std::forward<Self>(self).init_};
  }


};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        [this](auto&&... args) noexcept {
          std::execution::set_value(
            std::move(r_),
            std::forward<decltype(args)>(args)...);
        });
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        [this](auto&&... args) noexcept {
          std::execution::set_value(
            std::move(r_),
            std::forward<decltype(args)>(args)...);
        });
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

Exceptions

How and where does Asio throw?

```cpp
template<typename Readable, typename Handler>
void read(Readable& readable, const std::span<std::byte> span, Handler h) {
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    if (ec) {
      std::move(h)(ec);
      return;
    }
    if (!bytes) {
      std::move(h)(make_error_code(asio::errors::eof));
      return;
    }
    span = span.subspan(bytes);
    if (span.empty()) {
      std::move(h)(ec);
      return;
    }
    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, std::move(self));
  });
}
```

```cpp
template<typename Readable, typename Handler>
void read(Readable& readable, const std::span<std::byte> span, Handler h) {
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    if (ec) {
      std::move(h)(ec);
      return;
    }
    if (!bytes) {
      std::move(h)(make_error_code(asio::errors::eof));
      return;
    }
    span = span.subspan(bytes);
    if (span.empty()) {
      std::move(h)(ec);
      return;
    }
    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, std::move(self));
  });
}
```

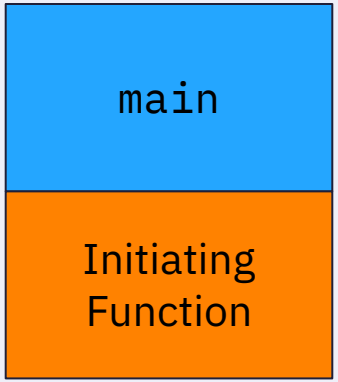
```cpp
template<typename Readable, typename Handler>
void read(Readable& readable, const std::span<std::byte> span, Handler h) {
  const asio::mutable_buffer cb(span.data(), span.size());
  readable.async_read_some(cb, [&readable, span, h = std::move(h)](
    this auto&& self,
    const std::error_code ec,
    const std::size_t bytes) mutable
  {
    if (ec) {
      throw std::system_error(ec);

    }
    if (!bytes) {
      throw eof_error{};

    }
    span = span.subspan(bytes);
    if (span.empty()) {
      std::move(h)();
      return;
    }
    const asio::mutable_buffer cb(span.data(), span.size());
    readable.async_read_some(cb, std::move(self));
  });
}
```

Executors

Where does the continuation run?

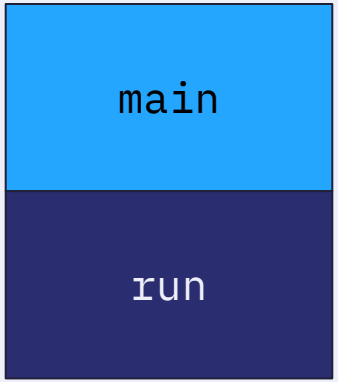
The Call Stack

The Call Stack

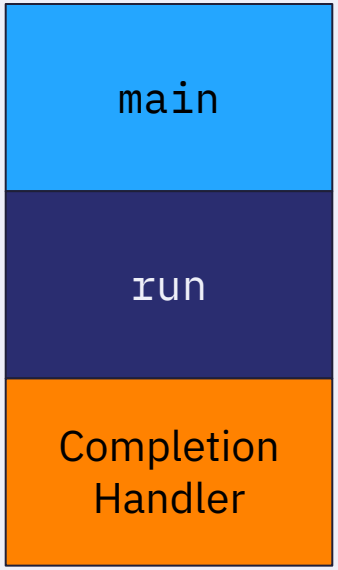


The Call Stack

The Call Stack



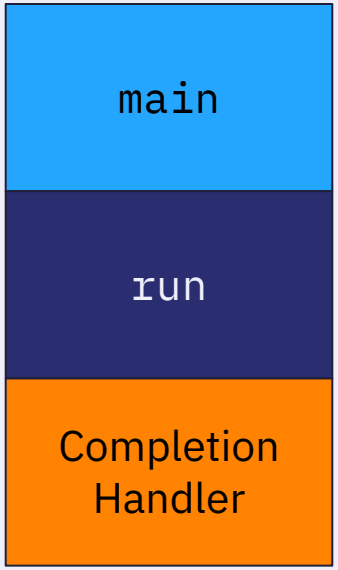
The Call Stack



Unstructured Concurrency

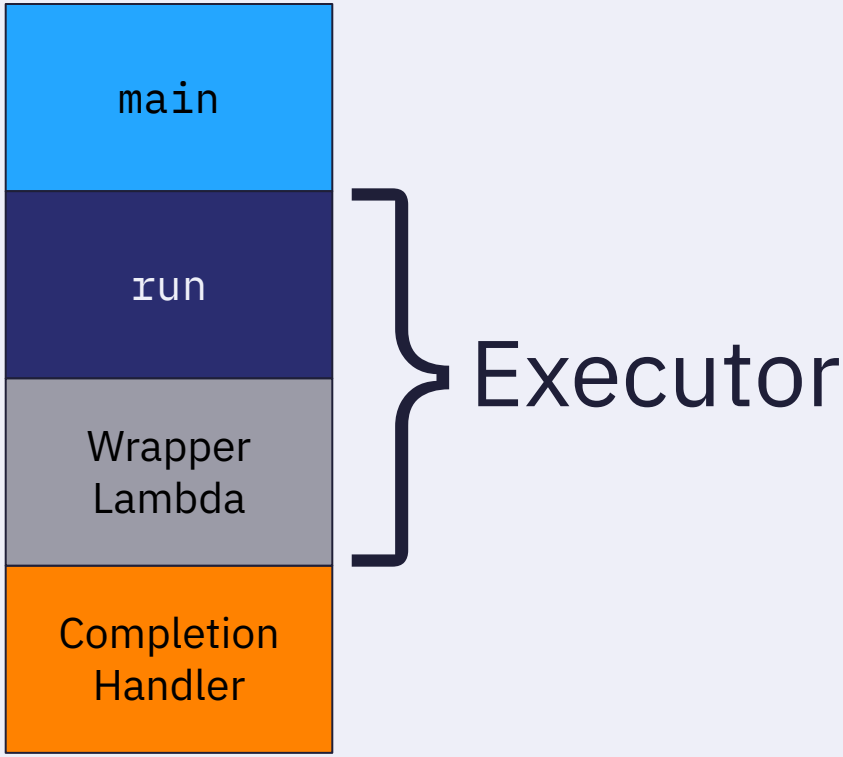
- Exceptions escaping and being handled in an executor-defined manner can result in the continuation never being run.
- Analogous to calling a function and it never returning, instead execution jumps somewhere else.
- By analogy to unstructured programming this is referred to as “unstructured concurrency.”
- **`std::execution`** operations must complete (the so-called “receiver contract”) so a solution is needed to bridge Asio and **`std::execution`**.

The Call Stack



The Call Stack

The Call Stack



```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  operation<Receiver, Initiation>& self_;
  Executor ex_;

  constexpr bool operator==(const executor& other) const noexcept {
    return (ex_ == other.ex_) && (&self_ == &other.self_);
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const {
    ex_.execute(std::forward<F>(f));
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute(std::forward<F>(f));
    } catch (...) {
      std::execution::set_error(
        std::move(self_.r_),
        std::current_exception());
    }
  }









};
```

```cpp
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          std::execution::set_error(
            std::move(self.r_),
            std::current_exception());
        }
      });
    } catch (...) {
      std::execution::set_error(
        std::move(self_.r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  /* ... */

};
```

Associators

Parameterizing Asio Operations

```
namespace asio {

template<typename Receiver, typename Initiation, typename Executor>
struct associated_executor<
  completion_handler<Receiver, Initiation>,
  Executor>
{

  using type = executor<Executor, Receiver, Initiation>;

  static constexpr type get(
    const completion_handler<Receiver, Initiation>& h,
    Executor ex = Executor()) noexcept
  {
    return type{*h.self_, std::move(ex)};
  }

};

}
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {

  template<typename... Ts>
  constexpr void operator()(Ts&&... ts) const noexcept {
    std::execution::set_value(
      std::move(self_->r_),
      std::forward<Ts>(ts)...);
  }

  operation<Receiver, Initiation>* self_;

};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        [this](auto&&... args) noexcept {
          std::execution::set_value(
            std::move(r_),
            std::forward<decltype(args)>(args)...);
        });
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        [this](auto&&... args) noexcept {
          std::execution::set_value(
            std::move(r_),
            std::forward<decltype(args)>(args)...);
        });
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  constexpr void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

Abandonment

- The Asio model does not provide guarantees that operations will end
- It’s fine for an execution context to simply stop running work and be destroyed, transitively destroying all completion handlers without invoking them
- We’ll call this “abandonment” and we need to figure out how to handle this in our mapping to **`std::execution`** as well
- Note that **`std::execution`** doesn’t allow for this behavior due to the “receiver contract”

Outcomes of Step

- Completion of the asynchronous operation
  - `std::execution::set_value`
- Deferral (i.e. asynchronous operation is composed and consists of further asynchronous parts)
  - No action (we'll complete somehow later)
- Exception (with or without abandonment)
  - `std::execution::set_error`
- Abandonment
  - `std::execution::set_stopped`

Outcomes of Step

- Completion of the asynchronous operation
  - `std::execution::set_value`
- Deferral (i.e. asynchronous operation is composed and consists of further asynchronous parts)
  - No action (we'll complete somehow later)
- Exception (with or without abandonment)
  - `std::execution::set_error`
- Abandonment
  - `std::execution::set_stopped`

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      std::execution::set_error(
        std::move(r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
  }


};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  std::exception_ptr ex_;

};
```

```cpp
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          std::execution::set_error(
            std::move(self.r_),
            std::current_exception());
        }
      });
    } catch (...) {
      std::execution::set_error(
        std::move(self_.r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          std::execution::set_error(
            std::move(self.r_),
            std::current_exception());
        }
      });
    } catch (...) {
      std::execution::set_error(
        std::move(self_.r_),
        std::current_exception());
    }
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          self.ex_ = std::current_exception();


        }
      });
    } catch (...) {
      self_.ex_ = std::current_exception();
    }
  }



};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  bool abandoned_{false};


};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  bool abandoned_{false};
  std::recursive_mutex m_;

};
```

Lifetime Issues

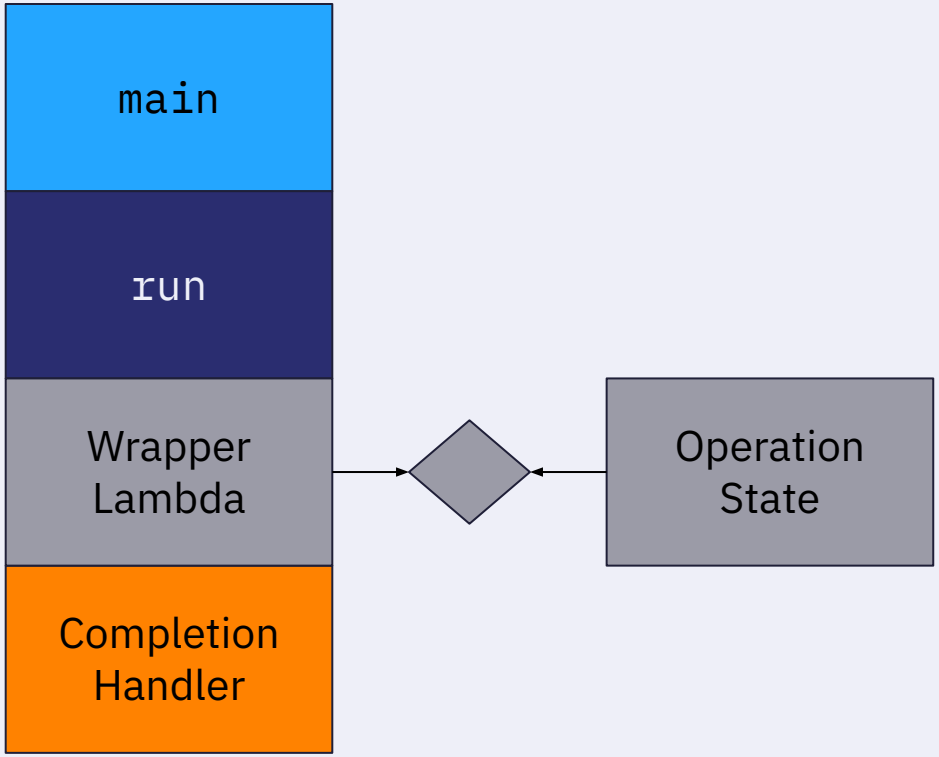
- Returning from a regular function destroys the stack frame (all variables with automatic storage duration)
- Operation state is the asynchronous analogue of the synchronous stack frame
- First completion signal conceptually ends the lifetime of the operation state (may not actually but that’s a non-generic property of the receiver)
- Can’t use the operation state for communication (or otherwise) after a potential completion signal

Outcomes of Step

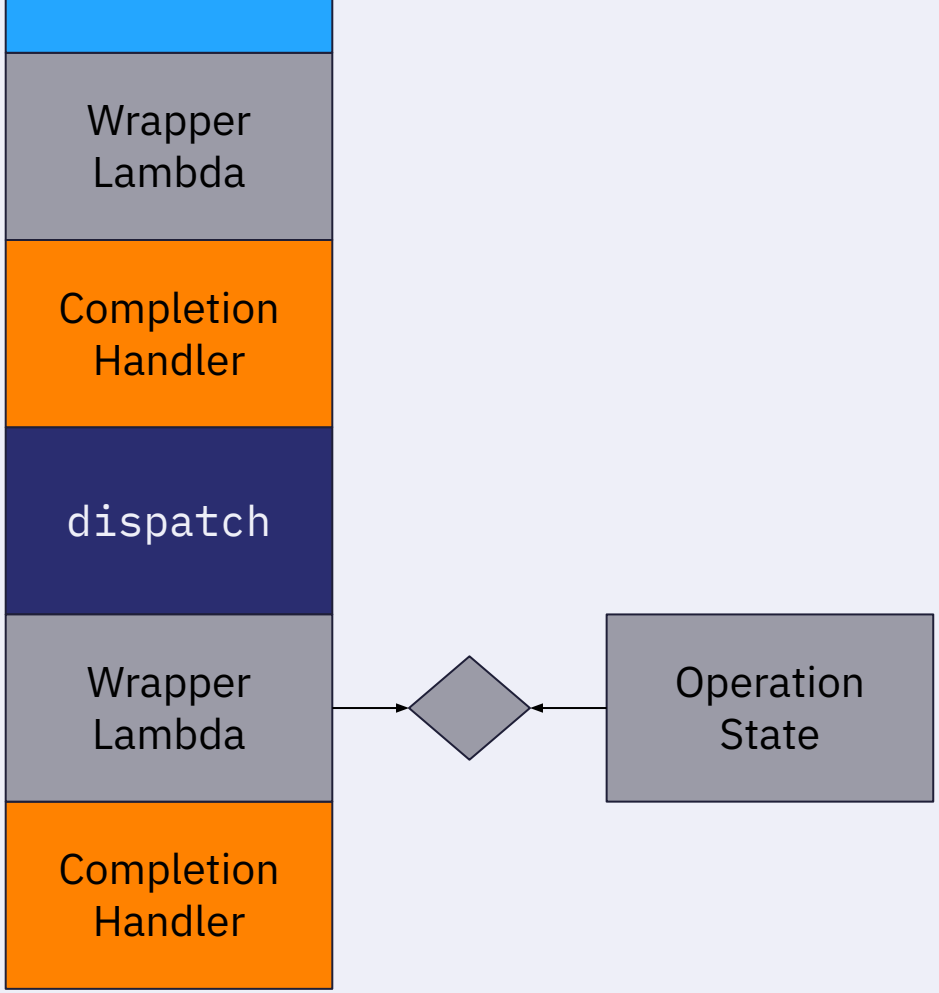
- Operation complete
  - **Operation state potentially outside lifetime!**
- Operation ongoing
  - Do nothing
- Operation abandoned
  - Finalize if we’re the last involved component

The Call Stack

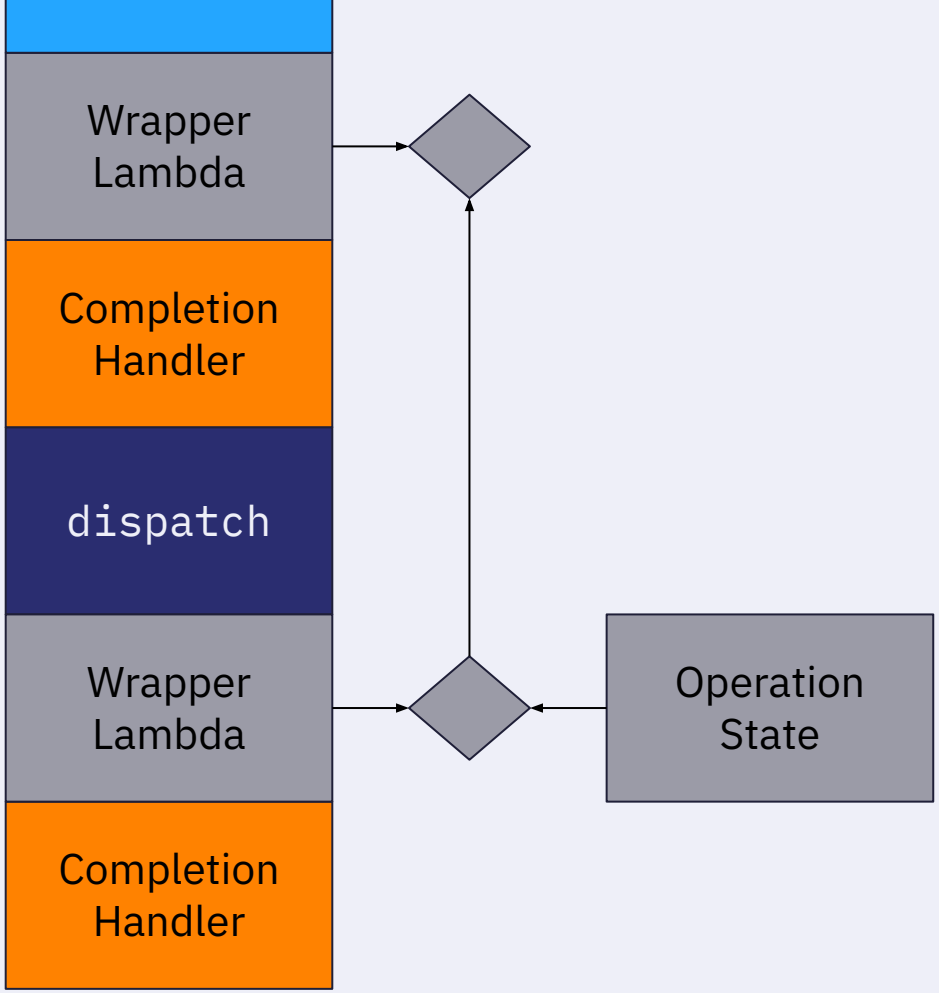
The Call Stack



Reentrant


Wrapper
Lambda
Completion
Handler
dispatch
Wrapper
Lambda
Operation
State
Completion
Handler

Reentrant


Wrapper
Lambda
Completion
Handler
dispatch
Wrapper
Lambda
Completion
Handler
Operation
State

```
template<typename Receiver, typename Initiation>
struct frame;
```

```cpp
template<typename Receiver, typename Initiation>
struct frame;

template<typename Receiver, typename Initiation>
struct operation {

  boost::intrusive::slist<frame<Receiver, Initiation>> frames_;

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  operation<Receiver, Initiation>* self_;

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

  constexpr explicit operator bool() const noexcept {
    return bool(self_);
  }

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  explicit frame(operation<Receiver, Initiation>& self) noexcept
    : self_(&self)
  {
    self_->m_.lock();
    self_->frames_.push_front(*this);
  }

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  void release() noexcept {
    self_->frames_.pop_front();
    self_->m_.unlock();
    self_ = nullptr;
  }

};
```

```cpp
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  ~frame() {
    if (self_) {
      self_->frames_.pop_front();
      const auto should_complete =
        self_->frames_.empty() && self_->abandoned_;
      self_->m_.unlock();
      if (should_complete) {
        if (self_->ex_) {
          std::execution::set_error(
            std::move(self_->r_),
            std::move(self_->ex_));
        } else {
          std::execution::set_stopped(std::move(self_->r_));
        }
      }
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  ~frame() {
    if (self_) {
      self_->frames_.pop_front();
      const auto should_complete =
        self_->frames_.empty() && self_->abandoned_;
      self_->m_.unlock();
      if (should_complete) {
        if (self_->ex_) {
          std::execution::set_error(
            std::move(self_->r_),
            std::move(self_->ex_));
        } else {
          std::execution::set_stopped(std::move(self_->r_));
        }
      }
    }
  }

};
```

```
template<typename... Signatures>
using transform_signatures = std::execution::completion_signatures<
  typename transform_signature<Signatures>::type...,
  std::execution::set_error_t(std::exception_ptr)>;
```

```
template<typename... Signatures>
using transform_signatures = std::execution::completion_signatures<
  typename transform_signature<Signatures>::type...,
  std::execution::set_error_t(std::exception_ptr),
  std::execution::set_stopped_t()>;
```

```
template<typename Receiver, typename Initiation>
struct operation {


  void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {


  void start() & noexcept {
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    const frame<Receiver, Initiation> f(*this);
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
  }

};
```

```cpp
template<typename Executor, typename Receiver, typename Initiation>
struct executor {


  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          self.ex_ = std::current_exception();
        }
      });
    } catch (...) {
      self_.ex_ = std::current_exception();
    }
  }

};
```

```
template<typename Executor, typename Receiver, typename Initiation>
struct executor {


  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        try {
          std::move(f)();
        } catch (...) {
          self.ex_ = std::current_exception();
        }
      });
    } catch (...) {
      self_.ex_ = std::current_exception();
    }
  }

};
```

```cpp
template<typename Executor, typename Receiver, typename Initiation>
struct executor {

  template<typename F>
  constexpr void execute(F&& f) const noexcept {
    const frame<Receiver, Initiation> _(self_);
    try {
      ex_.execute([&self = self_, f = std::forward<F>(f)]() mutable noexcept {
        const frame<Receiver, Initiation> _(self_);
        try {
          std::move(f)();
        } catch (...) {
          self.ex_ = std::current_exception();
        }
      });
    } catch (...) {
      self_.ex_ = std::current_exception();
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {


  constexpr completion_handler(operation<Receiver, Initiation>* self) noexcept
    : self_(self)
  {}

  constexpr completion_handler(completion_handler&& other) noexcept
    : self_(other.self_)
  {
    other.self_ = nullptr;
  }



};
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {




  ~completion_handler() {
    if (self_) {
      const frame<Receiver, Initiation> _(*self_);
      self_->abandoned_ = true;
    }
  }




};
```

```cpp
template<typename Receiver, typename Initiation>
struct completion_handler {

  template<typename... Ts>
  void operator()(Ts&&... ts) noexcept {
    {
      const std::lock_guard _(self_->m_);
      while (!self_->frames_.empty()) {
        self_->frames_.front().release();
      }
    }
    auto&& r = self_->r_;
    self_ = nullptr;
    std::execution::set_value(
      std::move(r),
      std::forward<Ts>(ts)...);
  }

};
```

Cancellation

Stopping In Flight Operations

Stop Tokens

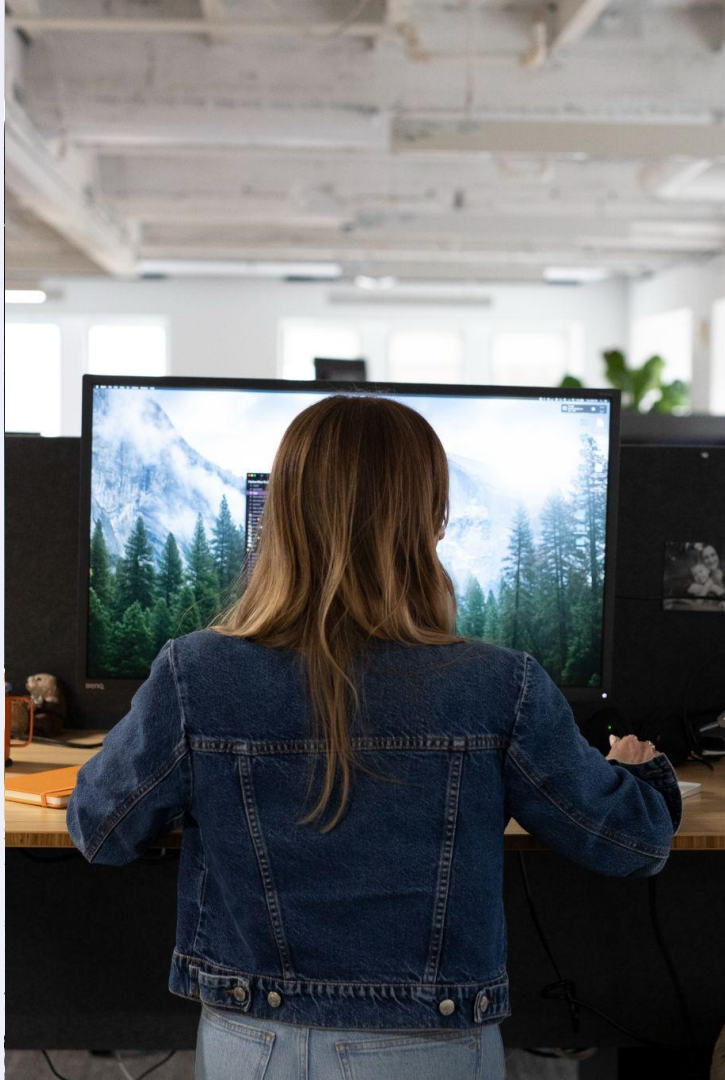
- **`std::execution`** uses a *stop token* from the receiver's *environment* to communicate *stop requests*

- The interface of a stop token includes:
  - **`stop_requested`**: Whether a stop has been requested
  - **`callback_type`**: Sets up an association which performs invocation when a stop has been requested

- Note that stop requests are "sticky"

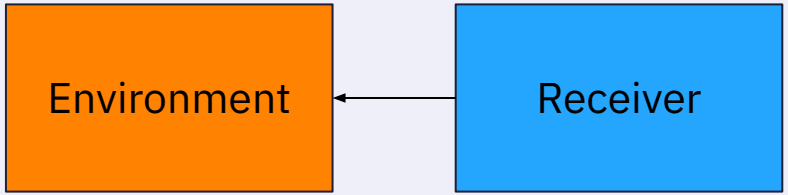
Cancellation Slots

- Asio uses a *cancellation slot* associated with the completion handler to transmit *cancellation signals*
- Cancellation slot can hold a single invocable which is invoked when a signal is emitted
- Unlike stop requests cancellation signals are not “sticky”



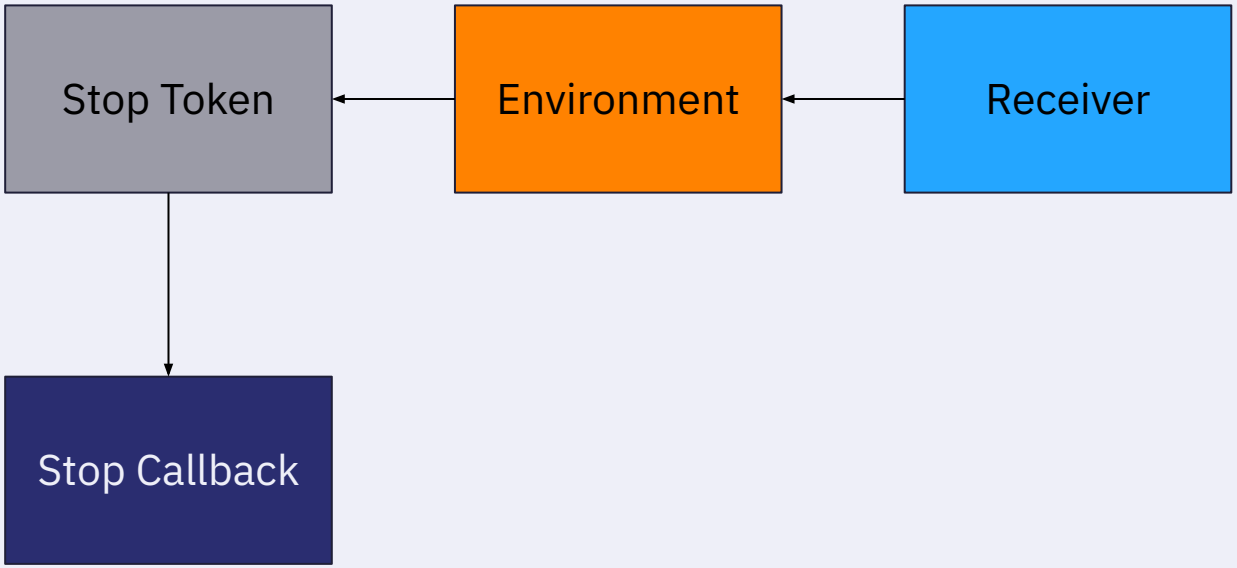
Cancellation

Cancellation

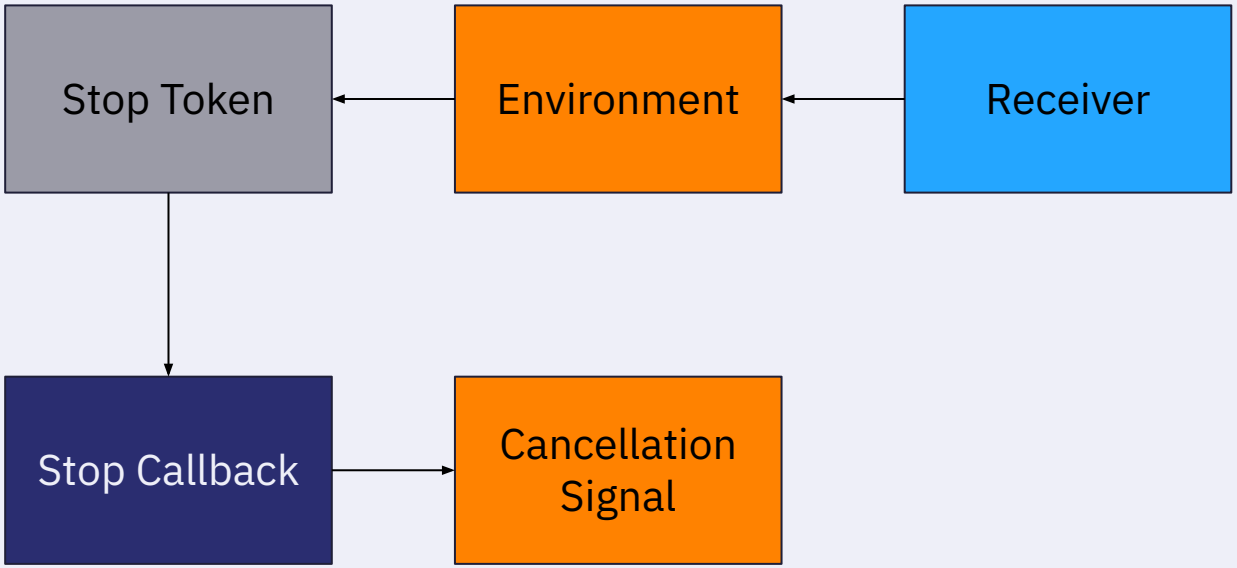


Cancellation

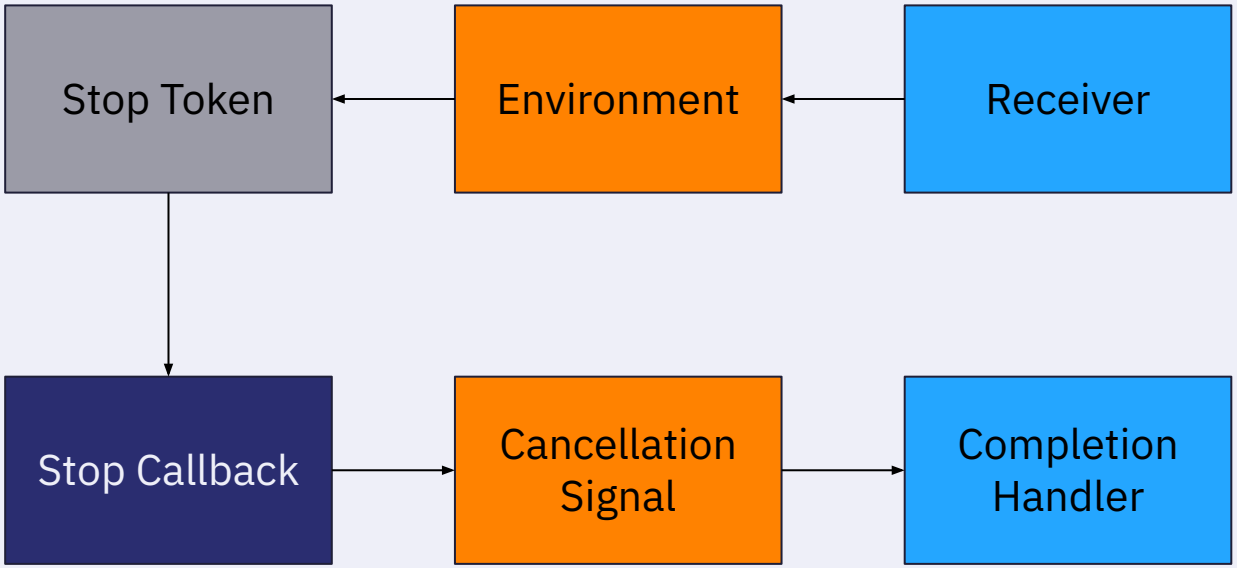
Cancellation



Cancellation



Cancellation



```cpp
template<typename Receiver, typename Initiation>
struct operation {

  struct on_stop_request_;

  asio::cancellation_signal signal_;
  std::optional<
    std::stop_callback_for_t<
      std::stop_token_of_t<
        std::execution::env_of_t<Receiver>>,
      on_stop_request_>> callback_;

};
```

```
template<typename Receiver, typename Initiation>
struct operation {




  struct on_stop_request_ {

    void operator()() && noexcept {
      const std::lock_guard _(self_.m_);
      self_.signal_.emit(asio::cancellation_type::all);
    }

    operation& self_;

  };




};
```

Asio Requirements

“When emitting a cancellation signal, the thread safety rules apply as if calling a member function on the target operation's I/O object. For non-composed operations, this means that it is safe to emit the cancellation signal from any thread provided there are no other concurrent calls to the I/O object, and no other concurrent cancellation signal requests. For composed operations, care must be taken to ensure the cancellation request does not occur concurrently with the operation's intermediate completion handlers.”

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    const frame<Receiver, Initiation> f(*this);
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
  }

};
```

```cpp
template<typename Receiver, typename Initiation>
struct operation {

  void start() & noexcept {
    const frame<Receiver, Initiation> f(*this);
    try {
      std::invoke(
        std::move(init_),
        completion_handler<Receiver, Initiation>{this});
    } catch (...) {
      ex_ = std::current_exception();
    }
    if (f) {
      callback_.emplace(
        std::get_stop_token(
          std::execution::get_env(r_)),
        on_stop_request_{*this});
    }
  }

};
```

Asio Requirements

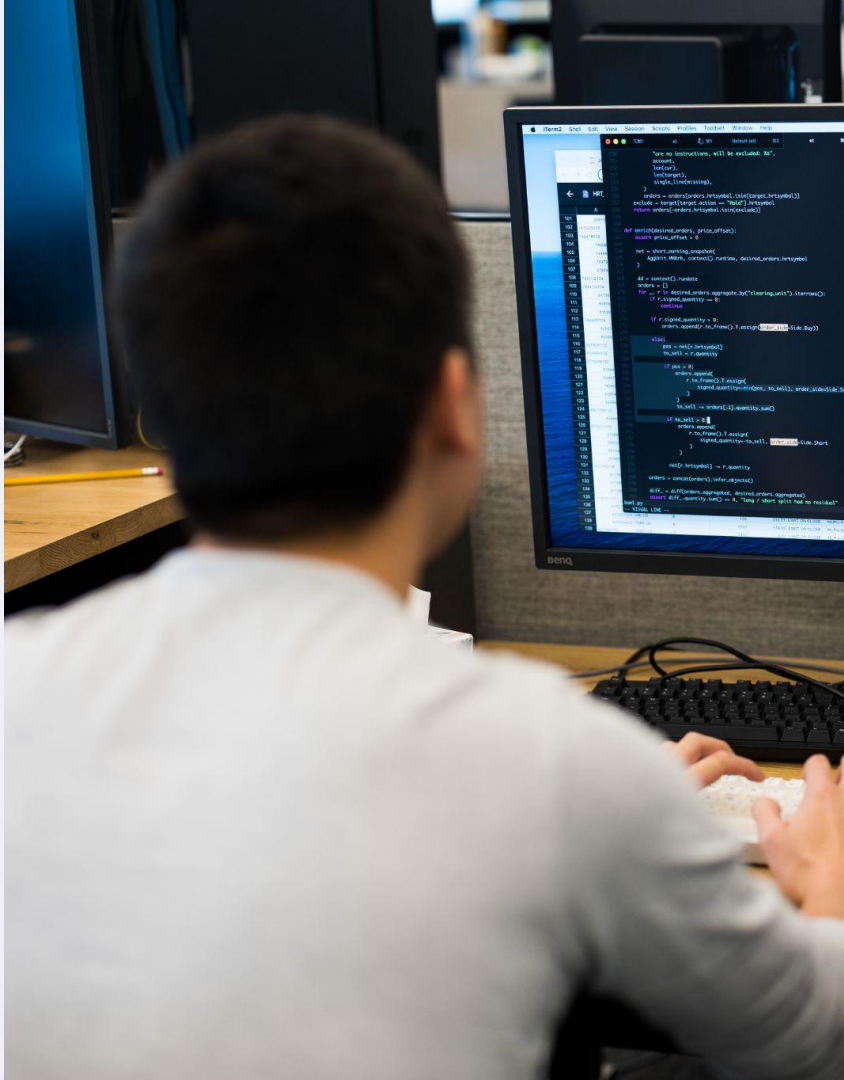
“Cancellation requests should not be emitted during an asynchronous operation's initiating function. Cancellation requests that are emitted before an operation starts have no effect. Similarly, cancellation requests made after completion have no effect.”

```
namespace asio {

template<typename Receiver, typename Initiation, typename CancellationSlot>
struct associated_cancellation_slot<
  completion_handler<Receiver, Initiation>,
  CancellationSlot>
{

  using type = asio::cancellation_slot;

  static constexpr type get(
    const completion_handler<Receiver, Initiation>& h,
    CancellationSlot slot = CancellationSlot()) noexcept
  {
    return h.self_->signal_.slot();
  }

};

}
```

From the Standard

“Let `rcvr` be a receiver and let `op_state` be an operation state associated with an asynchronous operation created by connecting `rcvr` with a sender. Let `token` be a stop token equal to `get_stop_token(get_env(rcvr))`. `token` shall remain valid for the duration of the asynchronous operation's lifetime.”

“The *lifetime of an asynchronous operation*, also known as the operation's *async lifetime*, begins when its start operation begins executing and ends when its completion operation begins executing.”



```cpp
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {


  ~frame() {
    if (self_) {
      self_->frames_.pop_front();
      const auto should_complete =
        self_->frames_.empty() && self_->abandoned_;
      self_->m_.unlock();
      if (should_complete) {
        if (self_->ex_) {
          std::execution::set_error(
            std::move(self_->r_),
            std::move(self_->ex_));
        } else {
          std::execution::set_stopped(std::move(self_->r_));
        }
      }
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {


  ~frame() {
    if (self_) {
      self_->frames_.pop_front();
      const auto should_complete =
        self_->frames_.empty() && self_->abandoned_;
      self_->m_.unlock();
      if (should_complete) {
        if (self_->ex_) {
          std::execution::set_error(
            std::move(self_->r_),
            std::move(self_->ex_));
        } else {
          std::execution::set_stopped(std::move(self_->r_));
        }
      }
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct frame : boost::intrusive::slist_base_hook<> {

  ~frame() {
    if (self_) {
      self_->frames_.pop_front();
      const auto should_complete =
        self_->frames_.empty() && self_->abandoned_;
      self_->m_.unlock();
      if (should_complete) {
        self_->callback_.reset();
        if (self_->ex_) {
          std::execution::set_error(
            std::move(self_->r_),
            std::move(self_->ex_));
        } else {
          std::execution::set_stopped(std::move(self_->r_));
        }
      }
    }
  }

};
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {

  template<typename... Ts>
  void operator()(Ts&&... ts) noexcept {
    {
      const std::lock_guard _(self_->m_);
      while (!self_->frames_.empty()) {
        self_->frames_.front().release();
      }
    }
    auto&& r = self_->r_;
    self_ = nullptr;
    std::execution::set_value(
      std::move(r),
      std::forward<Ts>(ts)...);
  }


};
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {

  template<typename... Ts>
  void operator()(Ts&&... ts) noexcept {
    {
      const std::lock_guard _(self_->m_);
      while (!self_->frames_.empty()) {
        self_->frames_.front().release();
      }
    }
    auto&& r = self_->r_;
    self_ = nullptr;
    std::execution::set_value(
      std::move(r),
      std::forward<Ts>(ts)...);
  }


};
```

```
template<typename Receiver, typename Initiation>
struct completion_handler {

  template<typename... Ts>
  void operator()(Ts&&... ts) noexcept {
    {
      const std::lock_guard _(self_->m_);
      while (!self_->frames_.empty()) {
        self_->frames_.front().release();
      }
    }
    self_->callback_.reset();
    auto&& r = self_->r_;
    self_ = nullptr;
    std::execution::set_value(
      std::move(r),
      std::forward<Ts>(ts)...);
  }

};
```

Complete

Now What?

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
  exec::repeat_effect_until(
    std::execution::just() |
    std::execution::let_value([&]() {
      const auto prev = read.size();
      read.resize(read.size() + 64);
      const asio::mutable_buffer buffer(
        read.data() + prev,
        read.size() - prev);
      return
        socket.async_read_some(buffer) |
        std::execution::then([&, prev](const auto bytes_transferred) {
          read.resize(prev + bytes_transferred);
          return bool(bytes_transferred);
        });
    })));
```

```
auto sender = exec::sequence(
  resolver.async_resolve(
    "google.com",
    "http",
    asio::ip::tcp::resolver::address_configured) |
    std::execution::let_value([&](const auto& results) {
      return socket.async_connect(results.begin()->endpoint());
    }),
  asio::async_write(socket, asio::buffer(to_write)),
  exec::repeat_effect_until(
    std::execution::just() |
    std::execution::let_value([&]() {
      const auto prev = read.size();
      read.resize(read.size() + 64);
      const asio::mutable_buffer buffer(
        read.data() + prev,
        read.size() - prev);
      return
        socket.async_read_some(buffer) |
        std::execution::then([&, prev](const auto bytes_transferred) {
          read.resize(prev + bytes_transferred);
          return bool(bytes_transferred);
        });
    })));
```

```
asio::io_context ctx;
asio::ip::tcp::resolver resolver(ctx);
asio::ip::tcp::socket socket(ctx);
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
asio::io_context ctx;
asio::ip::tcp::resolver resolver(ctx);
asio::ip::tcp::socket socket(ctx);
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
asio::io_context ctx;
auto resolver = completion_token.as_default_on(impl::ip::tcp::resolver(ctx));
auto socket = completion_token.as_default_on(impl::ip::tcp::socket(ctx));
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
void handler(
  const boost::system::error_code& error, // Result of operation.
  resolver::iterator iterator             // Forward-only iterator that can
                                          // be used to traverse the list
                                          // of endpoint entries.
);

void handler(
  const boost::system::error_code& error // Result of operation.
);

void handler(
  const boost::system::error_code& error, // Result of operation.
  std::size_t bytes_transferred           // Number of bytes written from the
                                          // buffers. If an error occurred,
                                          // this will be less than the sum
                                          // of the buffer sizes.
);

void handler(
  const boost::system::error_code& error, // Result of operation.
  std::size_t bytes_transferred           // Number of bytes read.
);
```

```
void handler(
  const boost::system::error_code& error, // Result of operation.
  resolver::iterator iterator             // Forward-only iterator that can
                                          // be used to traverse the list
                                          // of endpoint entries.
);

void handler(
  const boost::system::error_code& error // Result of operation.
);

void handler(
  const boost::system::error_code& error, // Result of operation.
  std::size_t bytes_transferred           // Number of bytes written from the
                                          // buffers. If an error occurred,
                                          // this will be less than the sum
                                          // of the buffer sizes.
);

void handler(
  const boost::system::error_code& error, // Result of operation.
  std::size_t bytes_transferred           // Number of bytes read.
);
```

```
asio::io_context ctx;
auto resolver = completion_token.as_default_on(impl::ip::tcp::resolver(ctx));
auto socket = completion_token.as_default_on(impl::ip::tcp::socket(ctx));
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
asio::io_context ctx;
auto resolver = use_sender.as_default_on(impl::ip::tcp::resolver(ctx));
auto socket = use_sender.as_default_on(impl::ip::tcp::socket(ctx));
const std::string_view to_write("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n");
std::vector<char> read;
```

```
Resolved: [2607:f8b0:4023:100f::64]:80
Connected
Wrote 36 bytes
Read 64 bytes:
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Read 64 bytes:

Content-Type: text/html; charset=UTF-8
Content-Security-Polic
Read 64 bytes:
y-Report-Only: object-src 'none';base-uri 'self';script-src 'non
Read 64 bytes:
ce-0JxFIQOB2IYwkwHjiQWWxg' 'strict-dynamic' 'report-sample' 'uns
Read 64 bytes:
afe-eval' 'unsafe-inline' https: http:;report-uri https://csp.wi
Read 64 bytes:
thgoogle.com/csp/gws/other-hp
Date: Sun, 24 Aug 2025 20:58:56 G
Read 64 bytes:
MT
Expires: Tue, 23 Sep 2025 20:58:56 GMT
Cache-Control: publi
Read 64 bytes:
c, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Prot
Read 64 bytes:
ection: 0
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http
Read 64 bytes:
-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>
Read 64 bytes:
301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document h
Read 64 bytes:
as moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HT
Read 5 bytes:
ML>
```

```cpp
auto timer = use_sender.as_default_on(asio::steady_timer(ctx));
```

```
auto timer = use_sender.as_default_on(asio::steady_timer(ctx));
timer.expires_from_now(std::chrono::seconds(1));
```

```
auto timer = use_sender.as_default_on(asio::steady_timer(ctx));
timer.expires_from_now(std::chrono::seconds(1));
auto timeout_sender = exec::when_any(
  std::move(sender),
  timer.async_wait());
```

How to Use

Copy the Slides?

Viewer Feedback

“Man I would really love to see the source for this talk. It's so hard to remember the overall context for all the code given even though Robert tries to elide out the (currently) irrelevant parts on each slide to emphasize. This stuff is really useful!”

commit 35a3e31590e736fbb7dd55324b3a7f991a059ce3
Author: Robert Leahy <rleahy@rleahy.ca>
Date: Sat May 10 14:03:59 2025 -0400

    asioexec::completion_token & ::use_sender (#1503)

    * asioexec::completion_token & ::use_sender

    Adds two completion tokens for interop with Asio (either Boost.Asio or
    standalone Asio).

    asioexec::completion_token performs the most basic transformations
    necessary to transform an Asio initiating function into a sender
    factory:

    - The initiating function returns a sender
    - Initiation is deferred until the above-mentioned sender is connected
      and the resulting operation state is started
    - The completion handler provided to the initiation (see asio::
      async_initiate) has the following properties:
      - Invocation results in the arguments thereto being sent via a value
        completion signal (this means that errors transmitted via a leading
        error_code parameter (i.e. in Asio style) are delivered via the
        value channel, see below)
      - Abandonment thereof (i.e. allowing the lifetime of the completion
        handler, and all objects transitively derived by moving therefrom,
        to end without invoking any of them) results in a stopped completion
        signal
      - Any exception thrown from any intermediate completion handler, or
        the final completion handler, is sent via an error completion signal
        with a std::exception_ptr representing that exception (this is
        accomplished by wrapping the associated executor)
      - The cancellation slot is connected to a cancellation signal which
        is sent when a stop request is received via the receiver's
        associated stop token

    The fact that invocations of the completion handler are passed to the
    value channel untouched reflects the design intent that the above-
    described completion token perform only "the most basic transformations
    necessary." This means that the full context of partial success must be
    made available and since the error channel is unary this must be
    transmitted in the value channel.

    For a more ergonomic experience than that described above asioexec::
    use_sender is also provided. This uses asioexec::completion_token to
    adapt an Asio initiating function into a sender factory and wraps the
    returned sender with an additional layer which performs the following
    transformations to value completion signals with a leading error_code
    parameter (note that when configured for standalone Asio std::error_code
    is matched whereas when configured for Boost.Asio both boost::system::
    error_code and std::error_code are matched):

    - If that argument compares equal to errc::operation_cancelled
      transforms the value completion signal into a stopped completion
      signal, otherwise
    - If that argument is truthy transforms the value completion signal into
      an error completion signal with an appropriate std::exception_ptr
      (i.e. one which points to a std::system_error for std::error_code,
      boost::system::system_error for boost::system::error_code), otherwise
    - Sends the remainder of the arguments (i.e. all but the error_code) as
      a value completion signal

    * add ASIO tests to the CI matrix

    Co-authored-by: Eric Niebler <eniebler@nvidia.com>

Conclusion

`std::execution` has arrived but that won't rewrite two decades (more!) of Asio-based code.

Thanks to the highly-generic nature of Asio, operations written with its guarantees and requirements in mind can be perfectly adapted to `std::execution`.

With some glue code (already written in stdexec!) the community can hit the ground running with `std::execution`, immediately leveraging more than two decades worth of asynchronous code.

Thank you!

Questions?