`std::execution` from the metal up

Paul M. Bendixen

Meeting C++
November 2022

Agenda

```
class Presentation
{
    public:
        Presentation();
        Insight whatIsSenderReceivers();
        Insight connecting();
        Program writingAnImplementation();
        ~Presentation();
    private:
        Insight whatIsAnISR(Microcontroller type);
        Insight whatIsFreestanding();
};
```

Introduction

```
/**
 * @author EE by training, member of SG14, father of two,
 *         Firmware pilot at Trifork, He/him
 *
 * @brief  An exploratory talk on senders and receivers
 *         aka P2300 execution
 *
 * @warn   Most of this is not production ready code
 */
```

Motivation

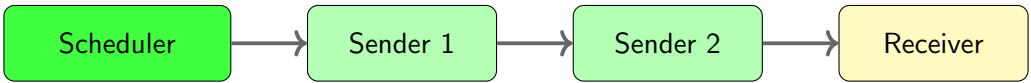
Working with Asynchrony Generically: A Tour of C++ Executors (part 1/2) - Eric Niebler - CppCon 21 at 5:55 "A callback, ... and we call that a receiver"

Motivation

Working with Asynchrony Generically: A Tour of C++ Executors (part 1/2) - Eric Niebler - CppCon 21 at 5:55 "A callback, ... and we call that a receiver"

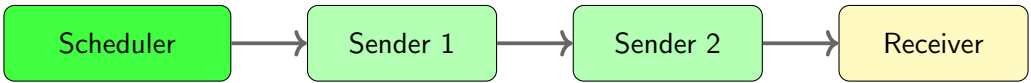
If only there was a way to not have to use exceptions...
Enter P2532: Removing exception_ptr from the Receiver Concepts

Chaining




```
AScheduler sch;
auto row = sch.schedule() | then([]{return 1;});
sink(row);
```

Chaining




```
AScheduler sch;
auto row = sch.schedule() | then([]{return 1;});
sink(row);
```

We can now reason about our asynchronous code

A Sender is lazy work

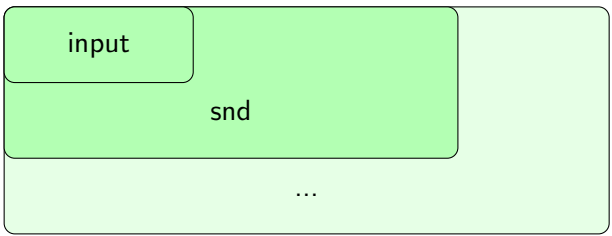
```
template<class S, class E = no_env>
concept sender =
  move_constructible<remove_cvref_t<S>> &&
  requires (S&& s, E&& e) {
    { get_completion_signatures( std::forward<S>(s),
                                 std::forward<E>(e)) }
        -> valid-completion-signatures<E>;
  };

completion_signatures< set_value(), set_error(), set_done()>
```

A Receiver is a callback

```
template<class T>
concept receiver =
  move_constructible<remove_cvref_t<T>> &&
  constructible_from<remove_cvref_t<T>, T> &&
  requires(const remove_cvref_t<T>& t) {
    execution::get_env(t);
  };
```

Chaining senders part II

```
sender auto input = get_input();
sender auto snd = then(input, [](auto... args) {
    std::print(args...);
});
```



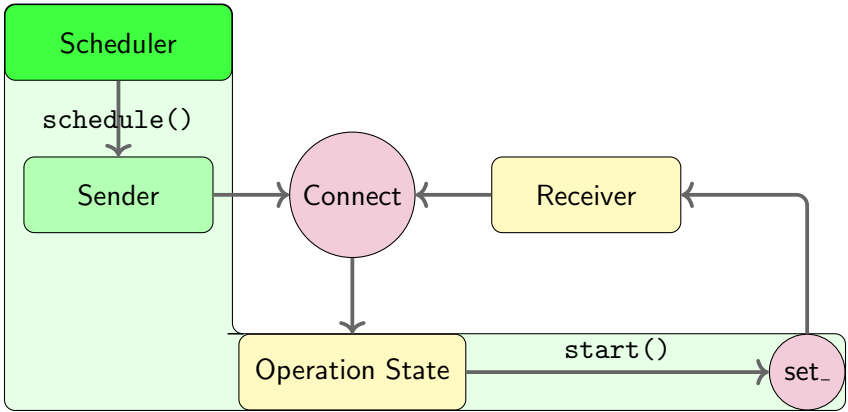
Connecting Senders and Receivers

```
template<class O>
concept operation_state =
  destructible<O> &&
  is_object_v<O> &&
  requires (O& o) {
    { execution::start(o) } noexcept;
  };

template<sender S, receiver R>
auto tag_invoke(connect_t, S&& s, R&& r) ->
    operation_state
```

Scheduling to get work done

```
template<class S>
concept scheduler =
  requires(S&& s)
  {
    { execution::schedule(std::forward<S>(s)) }
        -> sender;
  } &&
  equality_comparable<remove_cvref_t<S>> &&
  copy_constructible<remove_cvref_t<S>>;
```

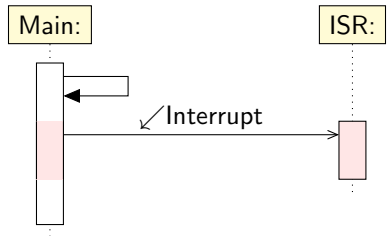
An overview


Scheduler
schedule()
Sender
Connect
Receiver
Operation State
start()
set_

An interrupt

Interrupts

Interrupts are like threads, except when they are not.



An asynchronous pipeline

What we need is:

- A scheduler - something to call schedule on
- A sender - something to use as a tag to get the correct operation state
- (algorithms) - Possibly, something to do something with the input
- A receiver - to get the value out

The implementaion

- Reference implementation by Bryce Adelsteing Lelbach, Eric Niebler et. al
  `https://github.com/NVIDIA/stdexec`
- Implementaion of the "Freestanding C++" library for AVR
  `https://gitlab.com/avr-libstdcxx`

Building an ISR Sender

Building a sender starts at the scheduler, it needs to return the sender

```
struct UART
{
    ...
    UARTSender void friend tag_invoke( stdexec::schedule_t,
            const UART& self ) noexcept
    {
        return {};
    }
};
```

The UART sender

```
struct UARTSender
{
    using completion_signatures =
        stdex::completion_signatures< stdex::set_value(char)>;

    friend UART tag_invoke(
        stdex::get_completion_scheduler_t< stdex::set_value>,
        const UART& ) noexcept
    {
        return {};
    }
}
```

Creating the Operation State

```
template<stdexec::receiver Rec>
struct OperationState
{
    ReceiverCaller f;
    explicit OperationState(Rec& rec)
        :f(rec)
    {}
    friend void tag_invoke(stdexec::start_t,
                           const OperationState& self) noexcept
    {
        // Enable interrupt for UART RX
    }
};
```

Creating the Operation State

```
template<stdexec::receiver Rec>
struct OperationState
{
    ReceiverCaller f;
    explicit OperationState(Rec& rec)
        :f(rec)
    {}
    friend void tag_invoke(stdexec::start_t,
                           const OperationState& self) noexcept
    {
        // Enable interrupt for UART RX
    }
};

static std::optional<OperationState<???>*> op;
```

Type Erasure

```
class ReceiverCaller
{
private:
    void *receiverObject = nullptr;
    void (*setValue)(void* receiver, char value);
public:
    void operator()(char value) {
        (*setValue)(receiverObject, value);
    }

    template<stdexec::receiver Rec>
    explicit ReceiverCaller(Rec& rec) {
        receiverObject = static_cast<void*>(&rec);
        setValue = [](void* receiver, char value) {
            if (receiver) {
                stdexec::set_value(
                        *static_cast<Rec *>(receiver),
                        (char &&) value);
            }
        };
    }
};
```

Type Erased Sender

```
struct OperationState
{
    Rec localReceiver;
    friend void tag_invoke(stdexec::start_t, const OperationState& self) noexcept
    {
        if (UartReceiverCallback) {
            stdexec::set_error(self.localReceiver, EBUSY);
        } else {
            UartReceiverCallback = ReceiverCaller(self.localReceiver);
            enableInterrupt();
        }
    }
};
```

Type Erased Sender

```
struct OperationState
{
    Rec localReceiver;
    friend void tag_invoke(stdexec::start_t, const OperationState& self) noexcept
    {
        if (UartReceiverCallback) {
            stdexec::set_error(self.localReceiver, EBUSY);
        } else {
            UartReceiverCallback = ReceiverCaller(self.localReceiver);
            enableInterrupt();
        }
    }
};

ISR(USART_RXC_vect) {
    if( UartReceiverCallback ) {
        (*UartReceiverCallback)( UDR );
    }
    UartSenderReceiver::UART::disableInterrupt();
}
```

sleep_wait

```
template<stdexec::sender Sender>
struct async_sleep {
    stdexec::connect_result_t<Sender, sleepReceiver> op;
    std::optional<char> value;
    volatile bool hasResult = false;
    struct sleepReceiver;
    template< stdexec::sender Snd>
            requires std::is_same_v<Snd,Sender>
    async_sleep(Snd&& snd)
            : op(stdexec::connect(std::forward<Sender>(snd),
    sleepReceiver{*this})) {
        stdexec::start(op);
        while (!hasResult) { // Go to sleep
        }
    }
};
template<typename S>
async_sleep(S) -> async_sleep<S>;
```

Creating the Receiver

```
struct sleepReceiver
{
    async_sleep& parent;
    struct sleepEnvironment{};
    sleepEnvironment friend tag_invoke(stdexec::get_env_t,
            const sleepReceiver& self) noexcept
        { return {}; }
    void friend tag_invoke(stdexec::set_value_t,
            sleepReceiver&& self, char&& val) noexcept {
        self.parent.value = val;
        self.parent.hasResult = true;
    }
    void friend tag_invoke(stdexec::set_error_t,
            sleepReceiver&& self, int&& error) noexcept {
        self.parent.hasResult = true;
    }
};
```

Running the code

```
int main()
{
    UartSenderReceiver::UART serial;
    for(;;)
    {
        auto result = async_sleep(stdexec::schedule(serial));
        if (result.value)
        {
                // SUCCESS
        }
    }
}
```

Testing as you go

```
using debugReceiver = stdexec::__debug::__debug_receiver<
    stdexec::__empty_env,
    stdexec::completion_signatures<stdexec::set_value_t(char),
    stdexec::set_error_t(int)>>;
static_assert(stdexec::operation_state<OperationState<debugReceiver>>);

static_assert(stdexec::sender<UartSender>);
static_assert(stdexec::sender_of<UartSender, stdexec::set_value_t(char)>);
static_assert(stdexec::scheduler<UartSenderReceiver::UART>);

static_assert(stdexec::receiver<sleepReceiver>);
static_assert(stdexec::receiver_of<sleepReceiver,
     stdexec::completion_signatures<stdexec::set_value_t(char)>>);
static_assert(stdexec::receiver_of<sleepReceiver,
     stdexec::completion_signatures<stdexec::set_error_t(int)>>);
```

Freestanding

Freestanding extensions are needed for optional, tuple and variant

Freestanding also gives embedded a lot of other niceties

Future work

Going forward in your journey take a look at:

- Using the available algorithms
- Creating algorithms with the `sender_adaptor` and `receiver_adaptor`
- Use the Environment and stop tokens to support cancellation

~Presentation

- `std::execution` is possible in a freestanding environment

~Presentation

- `std::execution` is possible in a freestanding environment
- Writing asynchronous code is enjoyable with sender receivers

~Presentation

- `std::execution` is possible in a freestanding environment
- Writing asynchronous code is enjoyable with sender receivers
- We need your expertise to figure out where the cracks are