Embedded Async Abstraction

Implementing Senders & Receivers without an OS

README Code of conduct BSL-1.0 license Security

C++ Bare Metal Senders and Receivers

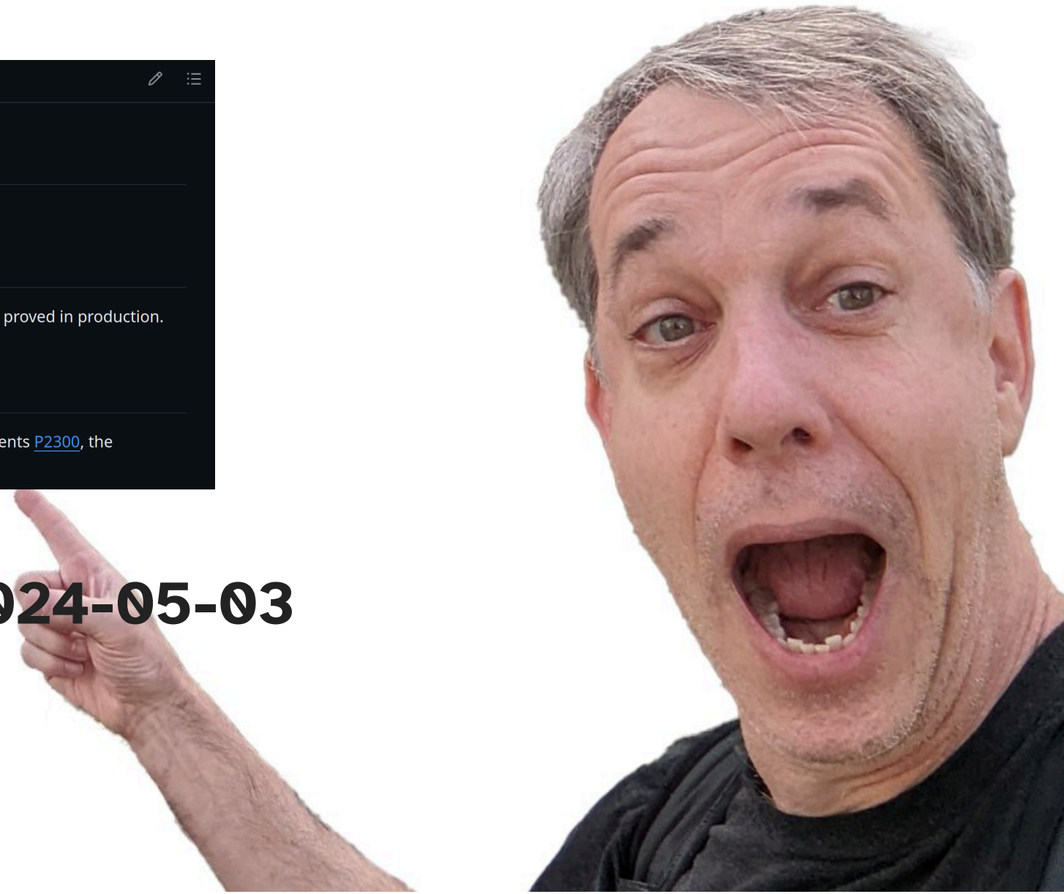
Unit Tests passing

WARNING: Work in Progress!

This library is under active development. It is a work in progress and has not yet been proved in production. Proceed at your own risk.

Abstract

C++ Bare Metal Senders and Receivers is a C++ header-only library that partially implements P2300, the proposed C++ standard design for asynchronous computation.



Ben Deane / C++Now / 2024-05-03

or... "How to draw the rest of the %@#?! owl"

Frontmatter

No AI/LLM was used in the creation of this talk.

Code is simplified for slides.
(I know this is C++Now, but it really has to be simplified. Assume all the `[[nodiscard]]`, `constexpr`, forwarding, etc exists.)

P2300 moves fast. This talk doesn't match all of it.
But it is an implementation in spirit.

Please ask questions as we go!

What's in this talk

A lot.

(Not even the whole owl. But maybe something that can fly and catch a lame mouse.)

1. How embedded development differs
2. Basics: composition and execution
3. A few schedulers
4. A tour of sender adaptors
5. Errors and cancellation
6. Allocation
7. Closing thoughts

Part 1

How is embedded development different from "normal" development?

To Begin With

"Embedded Development" means what?

Maybe you think you know.

Maybe you *do* know.

```
clang++ ... -fno-exceptions ... main.cpp
```

Sure, some mechanical stuff

- No runtime allocation? Sure.
- No exceptions? Whatever.
- Limited STL? Maybe.

These are relatively minor concerns in the grand scheme of things.

In many ways these are symptoms, not causes of differences.

More importantly, 1

Non-Embedded Dev assumes (mostly) that *programs end*.
Therefore it assumes that *resource lifetimes end*.

Even servers frequently deal with resources that have finite lifetimes.

Embedded programs frequently run forever.

More importantly, 2

Corollary: Non-Embedded Dev usually assumes the existence of a "main" thread.

You start a program on main thread; it does some stuff;
when the main thread ends, the program ends.

But at a low enough level, everything is event-driven, and the
"event loop" is hardware interrupts. There need be no main thread.

Your job in an interrupt-driven program

Get woken up by hardware with a brand new callstack.

Do what you need to do as efficiently as you can.

Return and go back to sleep.

Racing to sleep is the key to low power usage.

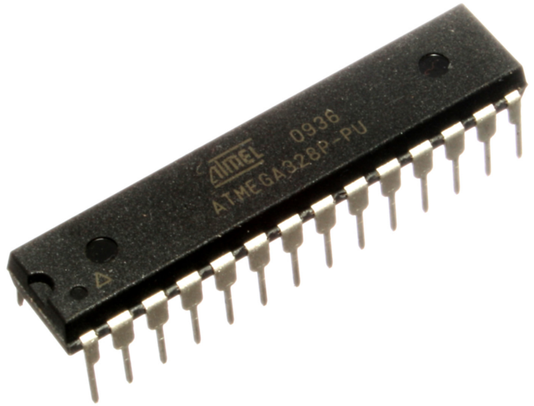
More importantly, 3

Non-embedded programs are almost all about computation.
The program *calculates* something useful.

Embedded programs are almost all about I/O.
The program *controls* something in a useful way.

(They're called micro*controllers* for a reason.)

More importantly, 3



- 8 bits
- 32KB flash
- 2KB SRAM
- 20 MHz

And 23 out of 28 pins for I/O.

Embedded is very heterogeneous

Program one desktop CPU, you've learned to program most desktop CPUs.

Program one microcontroller, and you've learned to program one microcontroller.

No OS, limited STL

Library	Component	Headers	Freestanding
Language support	Common definitions	<cstddef>	All
	C standard library	<cstdlib>	Partial
	Implementation properties	<cfloat> <climits> (since C++11) <limits> <version> (since C++20)	All
	Integer types	<cstdint> (since C++11)	All
	Dynamic memory management	<new>	All
	Type identification	<typeinfo>	All
	Source location	<source_location> (since C++20)	All
	Exception handling	<exception>	All
	Initializer lists	<initializer_list> (since C++11)	All
	Comparisons	<compare> (since C++20)	All
	Coroutines support	<coroutine> (since C++20)	All
	Other runtime support	<cstdarg>	All
	Debugging support	<debugging> (since C++26)	All
Concepts		<concepts> (since C++20)	All
Diagnostics	Error numbers	<cerrno> (since C++26)	Partial
	System error support	<system_error> (since C++26)	Partial
Memory management	Memory	<memory> (since C++23)	Partial
Metaprogramming	Type traits	<type_traits> (since C++11)	All
	Compile-time rational arithmetic	<ratio> (since C++23)	All
General utilities	Utility components	<utility> (since C++23)	All
	Tuples	<tuple> (since C++23)	All
	Function objects	<functional> (since C++20)	Partial
	Primitive numeric conversions	<charconv> (since C++26)	Partial
	Bit manipulation	<bit> (since C++20)	All
Strings	String classes	<string> (since C++26)	Partial
	Null-terminated sequence utilities	<cstring> (since C++26) <cwchar> (since C++26)	Partial
Iterators		<iterator> (since C++23)	Partial
Ranges		<ranges> (since C++23)	Partial
Numerics	Mathematical functions for floating-point types	<cmath> (since C++26)	Partial
Concurrency support	Atomics	<atomic> (since C++11)	All[1]
Deprecated headers		<ciso646> (until C++20) <cstdalign> (since C++11)(until C++20) <cstdbool> (since C++11)(until C++20)	All

No OS, limited STL

Freestanding is overly restrictive at the moment.
We are not sticking to the most restrictive interpretation of freestanding.
(More is on the way: thanks Ben Craig et al!)

This library targets C++20 and assumes the existence of parts of the library which are commonly supported and/or really have no reason not to exist.

Embedded execution contexts

1. The main thread (might as well end it?)
2. Interrupts
    1. Prioritized interrupts
    2. Timer interrupts
    3. Arbitrary I/O interrupts

"Interrupts are like threads..."

"...except when they are not."

Context switches are hardware driven.
But that doesn't mean they are unreasonable.

We should still expect correctly-implemented,
correctly-used atomics to do the right thing.
(Yes even in the presence of interrupts)

So, why P2300?

Much executor work so far has been concentrating on computation.

Embedded problems are more like:

- priority inversion
- concurrent hardware interaction
- race conditions
- lack of concurrency abstractions

Less "how to compute efficiently" and more "how to manage concurrent I/O".

The promises of senders & receivers

Declarative style that makes async code reason-able.

Execution abstraction that enables low power usage.

Part 2

Basics: composition and execution

The simplest scheduler

```
1: struct inline_scheduler {
2:   struct sender { ... };
3:
4:   auto schedule() {
5:     return sender{};
6:   }
7: };
```

Schedulers represent execution contexts. The simplest context is the current one. i.e. this scheduler does work as if by just calling a function.

A couple more schedulers

```
1: template <priority_t P>
2: struct priority_scheduler {
3:   auto schedule() -> sender auto;
4: };
5:
6: struct time_scheduler {
7:   auto scheduler() -> sender auto;
8:   duration_t dur;
9: };
```

This doesn't say much about how these work; we'll get to that.

But: senders from schedulers always send `void`.

Declaring senders

```
1: auto sndr = time_scheduler{1s}.schedule()
2:             | then([] { /* turn LED on */ })
3:             | continue_on(time_scheduler{1s})
4:             | then([] { /* turn LED off */ })
5:             | repeat();
```

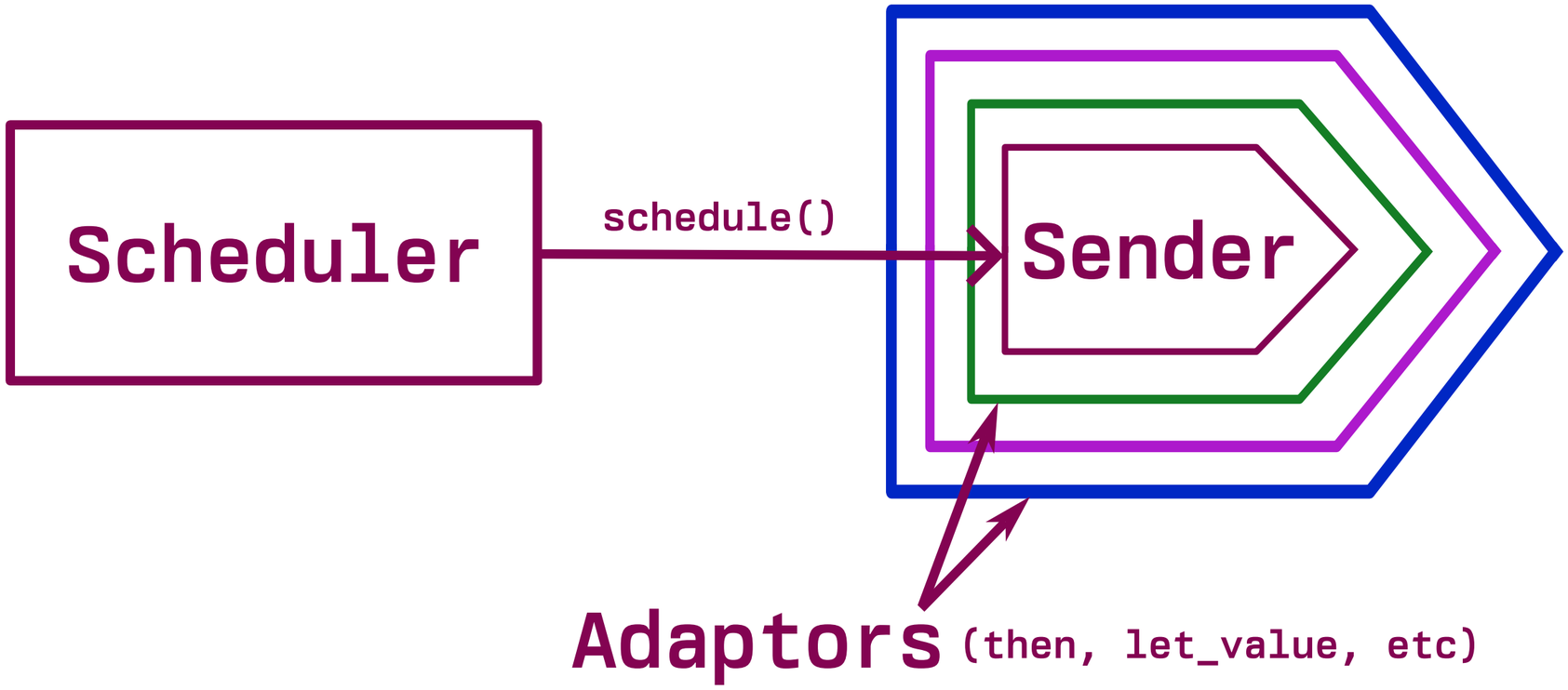
This is just a declaration. Nothing happened *yet*!

Declaring senders

```
1: auto sndr =
2:   repeat(
3:     then(
4:       continue_on(
5:         then(
6:           time_scheduler{1s}.schedule(),
7:           [] { /* turn LED on */ }),
8:         time_scheduler{1s}),
9:       [] { /* turn LED off */ }));
```

This is the same declaration.

Declaring senders

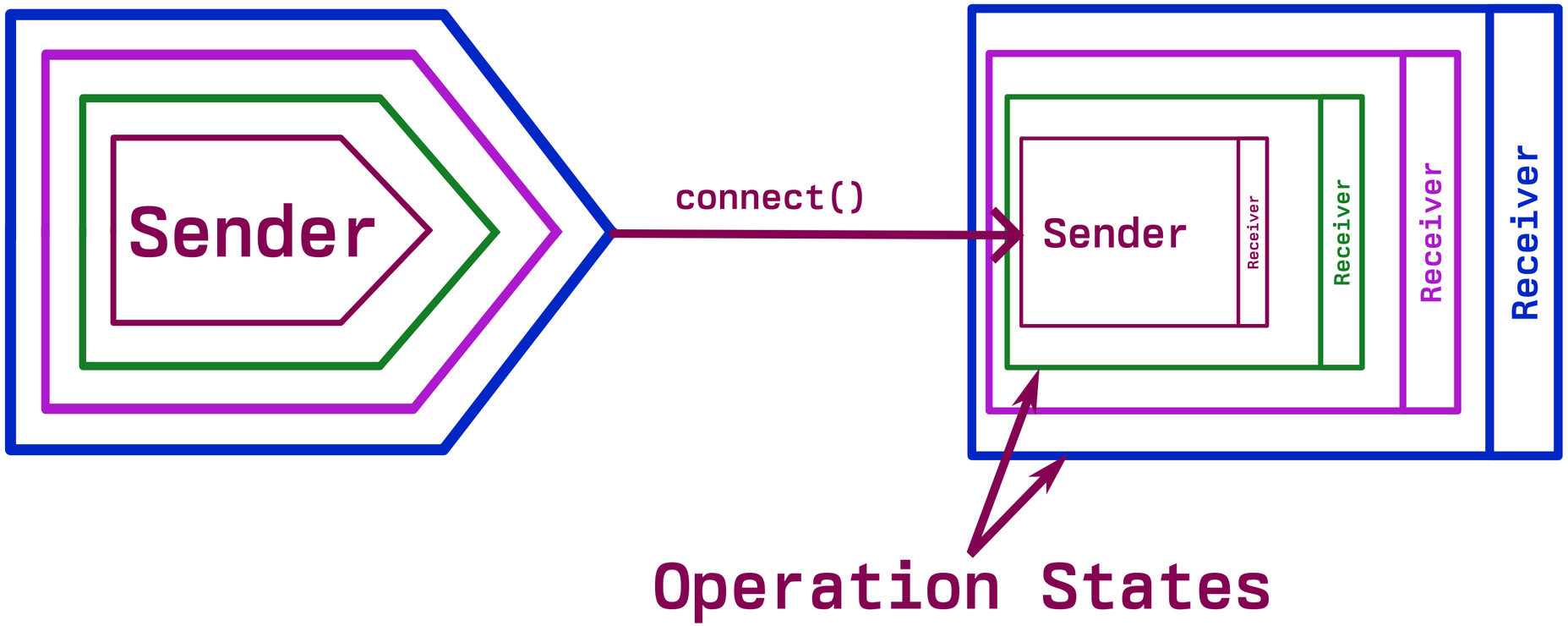


Executing senders

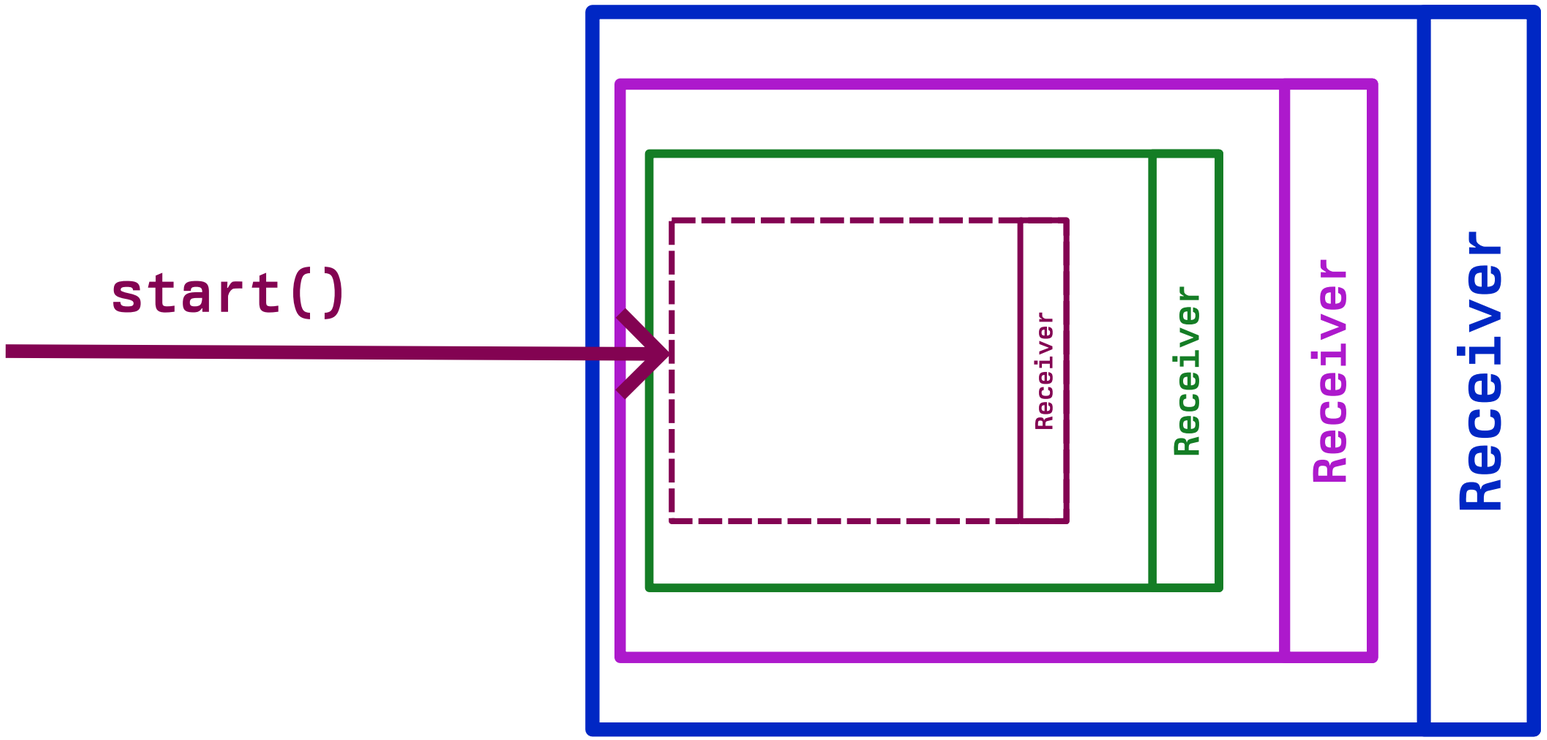
```
1: start_detached(sndr);
2:
3: // under the hood...
4: auto op = connect(sndr, rcvr);
5: start(op);
```

Now something kicks off.

Connecting

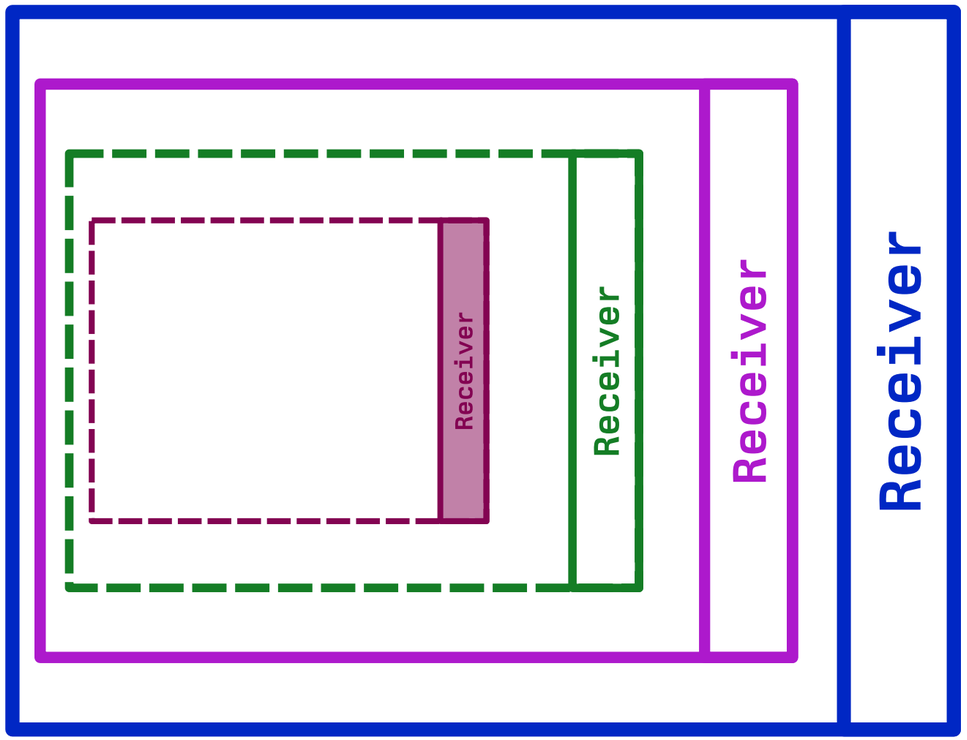

Sender
connect()
Sender
Receiver
Receiver
Receiver
Receiver
Operation States

Running



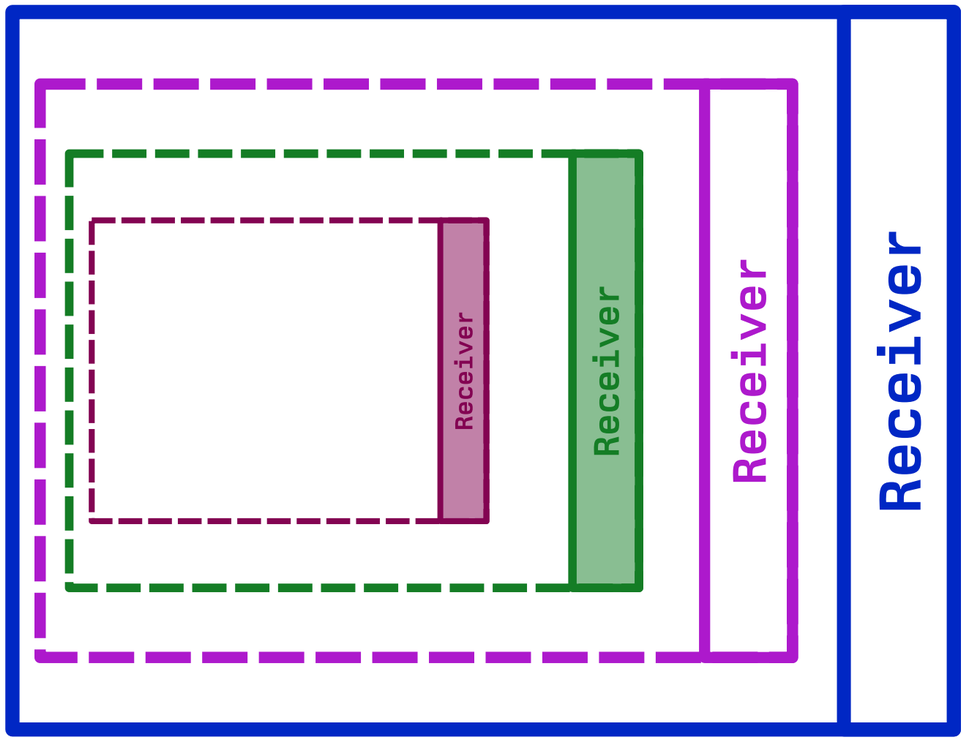
Running

```
running...
```



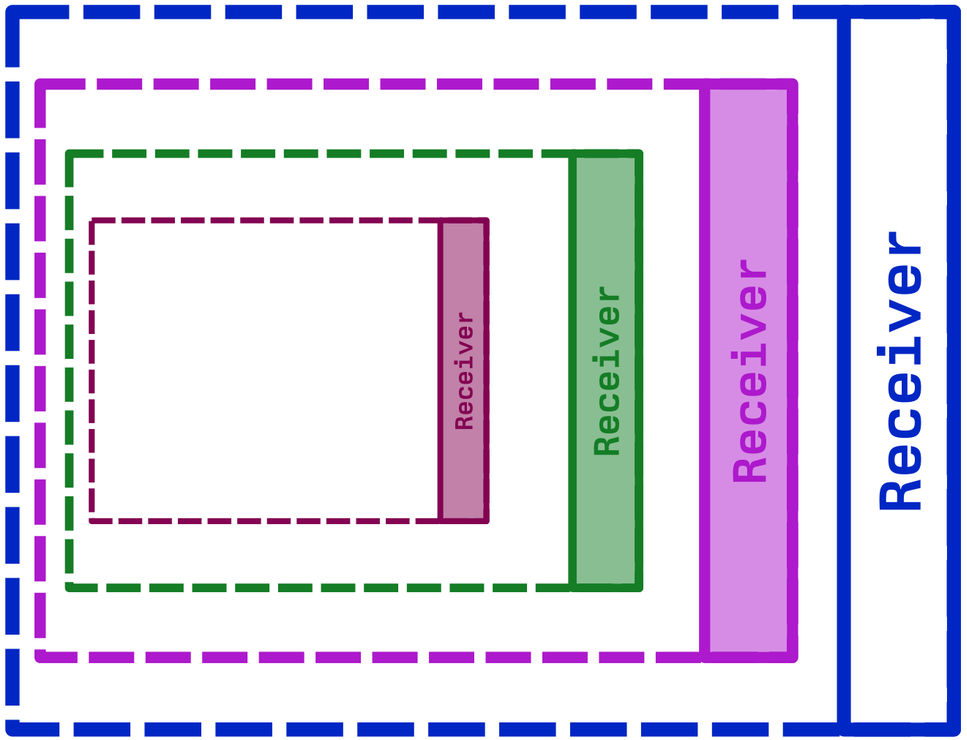
Running

```
running...
```



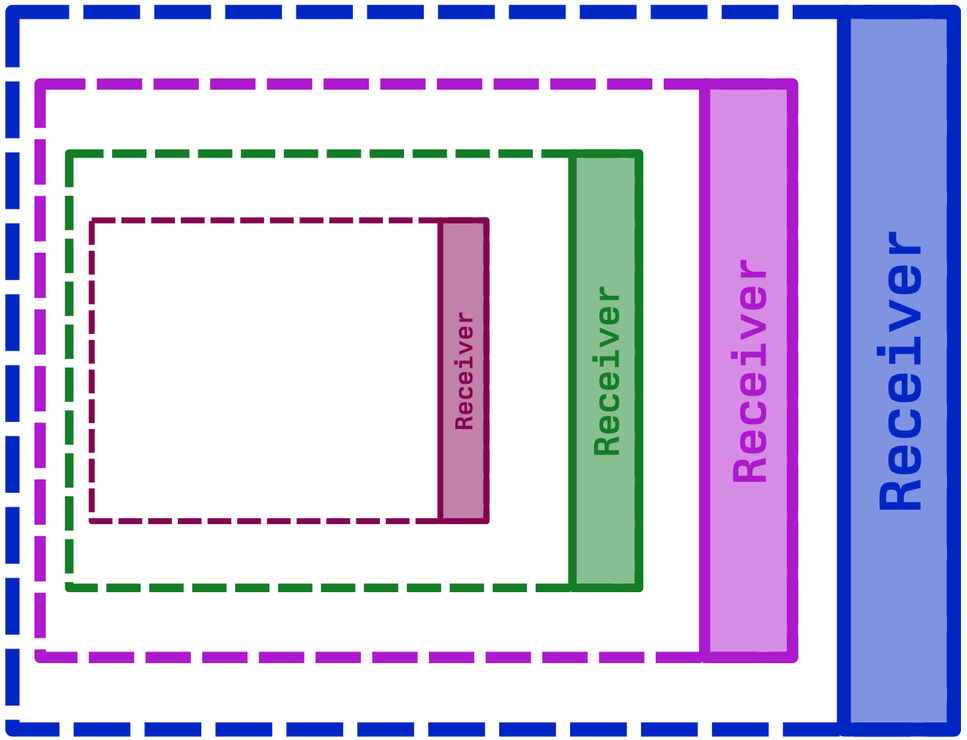
Running

```
running...
```



Running

`complete`



Composition

A sender chain starts with a sender *factory*...

```
1: auto result = time_scheduler{1s}.schedule()   // <- factory!
2:               | then([] { /* turn LED on */ })
3:               | continue_on(time_scheduler{1s})
4:               | then([] { /* turn LED off */ })
5:               | repeat()
6:               | start_detached();
```

Composition

...continues with as many sender *adaptors* as needed...

```
1: auto result = time_scheduler{1s}.schedule()
2:              | then([] { /* turn LED on */ })  // <- adaptor!
3:              | continue_on(time_scheduler{1s}) // <- adaptor!
4:              | then([] { /* turn LED off */ }) // <- adaptor!
5:              | repeat()                        // <- adaptor!
6:              | start_detached();
```

Composition

...and ends with a sender *consumer*.

```
1: auto result = time_scheduler{1s}.schedule()
2:              | then([] { /* turn LED on */ })
3:              | continue_on(time_scheduler{1s})
4:              | then([] { /* turn LED off */ })
5:              | repeat()
6:              | start_detached();                  // <- consumer!
```

Key concepts

Schedulers: represent execution contexts, produce senders

Senders: describe work, compose with adaptors

Receivers: behind the scenes, mostly small

Operation states: non-movable, state/lifetime objects

Part 3

A few schedulers

Time scheduler

Let's dive into how this actually works.

```
1: struct time_scheduler {
2:   auto schedule() -> _time::sender;
3:   duration_t dur;
4: };
```

This doesn't tell us much; we need to know: what is the operation state that results when this sender is connected?

(We know the sender itself doesn't do much.)

Time scheduler

```
1: template <same_as_unqual<_time::sender> S, receiver R>
2: auto tag_invoke(connect_t, S &&s, R &&r) {
3:     check_connect<S, R>();
4:     return _time::op_state<std::remove_cvref_t<R>>{
5:       std::forward<R>(r), s.dur};
6: }
```

The sender's `connect` call looks like this. This is fairly typical. Let's take a look at a few of the implementation patterns here.

connect

```
template <same_as_unqual<_time::sender> S, receiver R>
```

`same_as_unqual` is like `std::same_as` but without CV-ref qualifiers.

```
1: template <typename T, typename U>
2: concept same_as_unqual = std::same_as<std::remove_cvref_t<T>,
3:                                       std::remove_cvref_t<U>>;
```

(Also: `sender` here is the time scheduler's sender type.)

connect

```
template <same_as_unqual<_time::sender> S, receiver R>
```

`receiver` is a lightweight concept.

```
1: template <typename T>
2: concept receiver = /* movable and has an environment */;
```

A receiver is movable and has an environment. More on that later.

connect

```
1: template <same_as_unqual<_time::sender> S, receiver R>
2: auto tag_invoke(connect_t, S &&s, R &&r) {
```

`tag_invoke` has been removed from P2300... but I'm on C++20...
no explicit object parameters...

`tag_invoke`

I'm not really going to explain `tag_invoke`, but whenever you see:

```
auto tag_invoke(tag_t, obj, args...);
```

It's conceptually the same as:

```
obj.tag(args...);
```

Where `tag_t` might be `connect_t`, `start_t`, `set_value_t`, etc.

connect

```
1: template <same_as_unqual<_time::sender> S, receiver R>
2: auto tag_invoke(connect_t, S &&s, R &&r) {
3:   check_connect<S, R>();
```

`check_connect` is a `consteval` function that does a bit of metaprogramming: can R receive what S might send?

(This is also a good place to provide a decent diagnostic?)

connect

```
1: template <same_as_unqual<_time::sender> S, receiver R>
2: auto tag_invoke(connect_t, S &&s, R &&r) {
3:   check_connect<S, R>();
4:   return _time::op_state<std::remove_cvref_t<R>>{
5:     std::forward<R>(r), s.dur};
6: }
```

Now we return the operation state, passing it the receiver and the `duration_t`.

The operation state

```
 1: template <typename R>
 2: struct _time::op_state : timer_task {
 3:   [[no_unique_address]] R rcvr; // receivers are often empty
 4:   duration_t dur;
 5:
 6:   template <same_as_unqual<op_state> O>
 7:   friend auto tag_invoke(start_t, O &&o) -> void {
 8:     run_after(o, o.dur);
 9:   }
10:
11:   auto run() -> void final { set_value(rcvr); }
12: };
```

Timer management

```
1: struct timer_task {
2:   bool pending{};
3:   time_point_t tp{};
4:   timer_task *prev, *next;
5:
6:   constexpr timer_task(timer_task&&) = delete;
7:   virtual auto run() -> void = 0;
8: };
```

`timer_task`s are non-movable, have a `run` function,
and intrusive-list machinery.

They also have reference equality, and `operator<` that works on their expiry time.

Timer management

```
 1: struct timer_manager {
 2:   intrusive_list<timer_task> task_queue{};
 3:
 4:   auto run_after(timer_task &t, duration_t dur) {
 5:     // insert task in list at the right place
 6:     // set next expiry time to head-of-list time
 7:     // enable interrupt
 8:   }
 9:
10:   auto isr() {
11:     // dequeue next task and run it
12:     // reset interrupt for next (if any)
13:   }
14: };
```

The `timer_manager` handles the queue of tasks, and the interrupts.

The operation state

```
1: template <typename R>
2: struct _time::op_state : timer_task {
3:   template <same_as_unqual<op_state> O>
4:   friend auto tag_invoke(start_t, O &&o) -> void {
5:     run_after(o, o.dur);
6:   }
7:
8:   auto run() -> void final { set_value(rcvr); }
9: };
```

When we `start` the operation state, it queues the task (itself).

When the interrupt fires, the timer manager's ISR `run`s the task.

Which immediately calls `set_value` on the receiver.

Time scheduler: summary

- `schedule` produces a `sender`
- when the `sender` is `connect`ed, an `op_state` results
- when the `op_state` is `start`ed, it enables the timer interrupt
- when the timer interrupt fires, it runs the `op_state`
- and the downstream sender chain is now running on the timer interrupt

Priority scheduler

This works very similarly to the time scheduler.

```
1: template <priority_t P>
2: struct priority_scheduler {
3:   auto schedule() -> sender auto;
4: };
```

Priority scheduler

```
1: template <same_as_unqual<_priority::sender> S, receiver R>
2: auto tag_invoke(connect_t, S &&s, R &&r) {
3:     check_connect<S, R>();
4:     return _priority::op_state<P, std::remove_cvref_t<R>>{
5:         std::forward<R>(r)};
6: }
```

Looks very similar to the timer sender's connect call.

This time, the static priority is passed as a template argument.

The operation state

```
 1: template <priority_t P, typename R>
 2: struct _priority::op_state : priority_task {
 3:   [[no_unique_address]] R rcvr;
 4:
 5:   template <same_as_unqual<op_state> O>
 6:   friend auto tag_invoke(start_t, O &&o) -> void {
 7:     enqueue_task<P>(o);
 8:   }
 9:
10:   auto run() -> void final { set_value(rcvr); }
11: };
```

The priority scheduler's `op_state` is basically the same as the time scheduler's `op_state`. But instead of queueing the task for a certain time, it queues the task at the given priority.

Priority task management

```
1: struct priority_task {
2:   bool pending{};
3:   priority_task *prev, *next;
4:
5:   constexpr priority_task(priority_task&&) = delete;
6:   virtual auto run() -> void = 0;
7: };
```

`priority_task`s are again non-movable, have a `run` function, and intrusive-list machinery.

Again, they have reference equality.

Priority task management

```
 1: template <size_t NumPriorities>
 2: struct priority_task_manager {
 3:   array<intrusive_list<priority_task>, NumPriorities> queues{};
 4:
 5:   template <priority_t P>
 6:   auto enqueue_task(priority_task& t) {
 7:     // push task on to the appropriate queue
 8:     // enable the priority interrupt
 9:   }
10:
11:   template <priority_t P>
12:   auto isr() {
13:     // execute each task on the appropriate queue
14:   }
15: };
```

The `task_manager` handles the queue of tasks for each priority, and the interrupts.

Priority scheduler: summary

- `schedule` produces a `sender`
- when the `sender` is `connect`ed, an `op_state` results
- when the `op_state` is `start`ed, it enables the priority interrupt
- when the priority interrupt fires, it runs the `op_state`
- and the downstream sender chain is now running on that priority interrupt

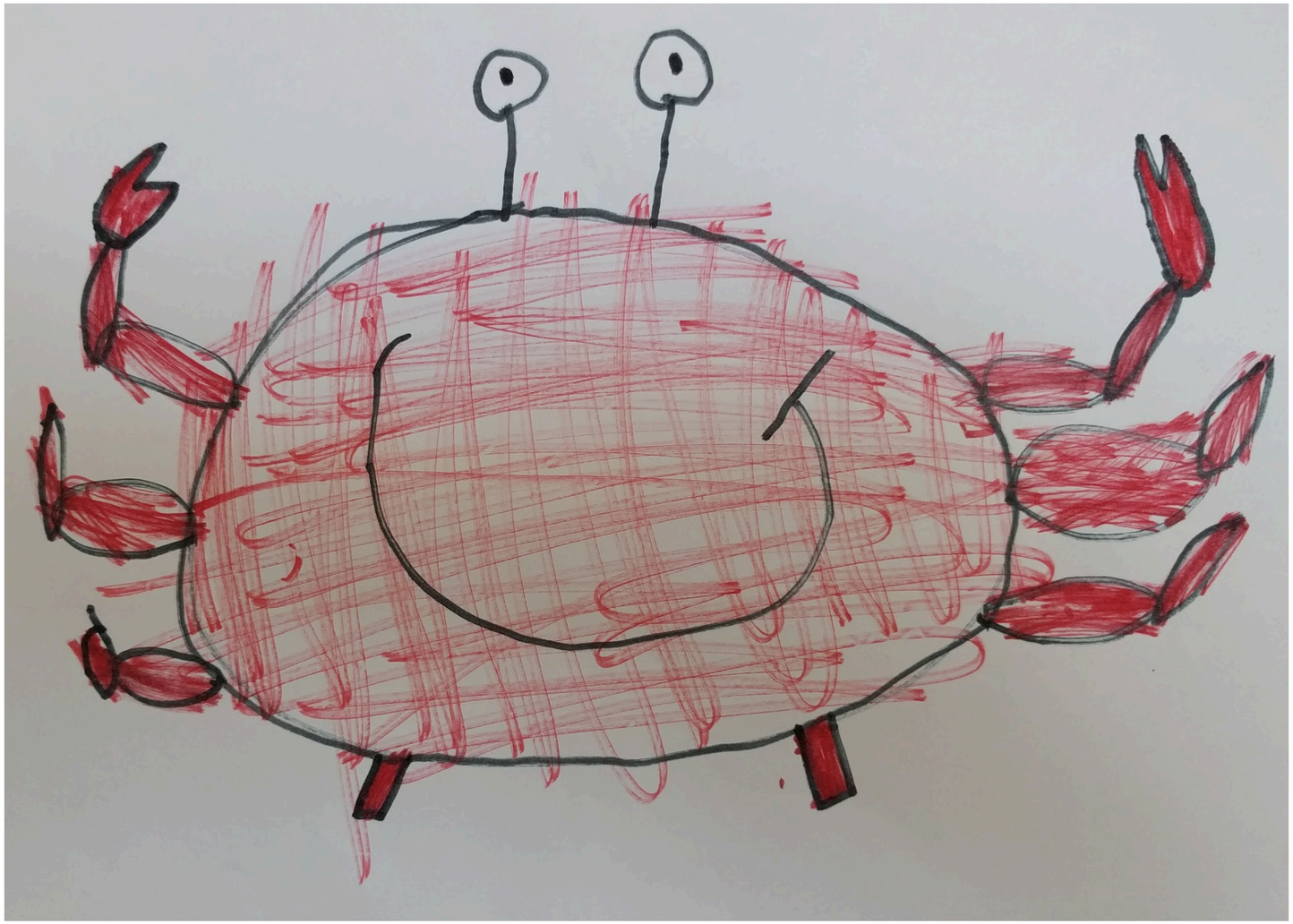
Priority scheduler execution

```
1: template <priority_t P, typename RequeuePolicy>
2: auto isr() {
3:   // execute each task on the appropriate queue
4:   // take care if tasks requeue themselves!
5: }
```

The priority scheduler should be parameterized on how to deal with tasks that will requeue themselves. (Because this could result in an infinite loop.)

Part 4

A tour of sender adaptors



First, receivers and completion channels

A receiver handles what a sender sends. A sender may complete in three ways.

```
1: struct universal_receiver { ... };
2: auto tag_invoke(
3:   channel_tag auto, universal_receiver const &, auto &&...) -> void {}
4:
5: template <typename T>
6: concept channel_tag =
7:     std::same_as<set_value_t, T>
8:  or std::same_as<set_error_t, T>
9:  or std::same_as<set_stopped_t, T>;
```

Here is just about the simplest receiver.

It handles anything (`set_value`, `set_error`, `set_stopped`), and does nothing.

Senders advertise what they send...

...AKA the ways they may complete.

```
1: struct a_sender {
2:   using completion_signatures =
3:     ::completion_signatures<set_value_t(int), set_value_t(int, float),
4:                             set_error_t(int),
5:                             set_stopped_t()>
6: };
```

Note: a sender may complete on a channel in more than one way.

Also, pipe syntax

Whenever you see:

```
a | b(args);
```

It's syntactic sugar for:

```
b(a, args...);
```

`then`

We've seen `then` in action already.

```
1: auto sndr = time_scheduler{1s}.schedule()
2:           | then([] { /* turn LED on */ });
```

In this case, the scheduler sender sent `void` -
so the function passed to `then` takes nothing.

`then` is `fmap` (the characteristic function of a functor).
It's not called `transform`, don't blame me.

then

Adaptor functions tend to be quite boring...

```
1: template <sender S, typename F>
2: auto then(S&& s, F&& f) -> sender auto {
3:   return _then::sender<remove_cvref_t<S>, remove_cvref_t<F>>{
4:     std::forward<S>(s), std::forward<F>(f)};
5: }
```

We're going to see that most of these adaptors tend to follow the same patterns.

`then` returns a sender formed by wrapping up the original sender with the function.

The `then` sender

So what does the `then` sender do when connected?

```
1: template <same_as_unqual<_then::sender> Self, receiver R>
2: auto tag_invoke(connect_t, Self &&self, R &&r) {
3:     check_connect<Self, R>();
4:     return connect(std::forward<Self>(self).s,
5:                    _then::receiver<std::remove_cvref_t<R>, F>{
6:                      std::forward<R>(r), std::forward<Self>(self).f});
7: }
```

Again, mostly boring.

`then` is simple enough that it doesn't need an operation state... just a receiver.

We'll put the function in that receiver, that also wraps the passed-in receiver.

The `then` receiver

```
 1: template <typename R, typename F>
 2: struct _then::receiver {
 3:   [[no_unique_address]] R r; // the original receiver
 4:   [[no_unique_address]] F f;
 5:
 6:   template <same_as_unqual<receiver> Self, typename... Args>
 7:   friend auto tag_invoke(set_value_t, Self &&self, Args &&...args) {
 8:     set_value(self.r, self.f(args...));
 9:   }
10: };
```

`set_value` on the `then` receiver does what we expect: calls `set_value` on the wrapped receiver, with the result of calling the function on the args.

That's pretty much it for `then`

Those are the basics of a workable `then` implementation.

But if we want to make it ~~fancier~~ more ergonomic...

```
1: auto sndr =
2:     just(42, 3.14f, true)  // -> set_value(rcvr, 42, 3.14f, true)
3:   | then([] (int) -> A { ... },
4:          [] (float, bool) -> void { ... },
5:          [] () -> B { ... });
6: // sndr now sends (A, B)
```

We can build in some "multiple value" machinery. This could be useful.

Otherwise `then` is asymmetric: it can take multiple values, but can't return multiple values. Users would end up messing about with tuples.

`upon_error` and `upon_stopped`

`upon_error` and `upon_stopped` behave the same as `then`,
but they transform the other completion channels.

```
1: auto e_sndr =
2:     just_error(1729)    // -> set_error(rcvr, 1729)
3:   | upon_error([] (int i) { println("Actually, {} IS interesting", i) });
4:
5: auto c_sndr =
6:     just_stopped()      // -> set_stopped(rcvr)
7:   | upon_stopped([] { ... });
```

In case we need to transform the result of an error case, or a cancellation.

Note: `e_sndr` and `c_sndr` still complete with `set_value`!

`let_value`

"`let_value` is like `then`, but the function itself returns a sender."

```
1: auto sndr = ...
2:           | let_value([] (int x) { return just(x + 1); });
```

...OK, that's technically true.

`let_value` is bind (the characteristic function of a monad).

So it's *for* a runtime choice.

So what does `let_value`'s function return?

Uh-oh.

We have a problem before we even think about implementing `let_value`.

```
1: auto sndr = ...
2:   | let_value([] (int n) {
3:       if (n % 2 == 0) {
4:         return /* one type of sender */;
5:       } else {
6:         return /* another type of sender? */;
7:       }
8:     });
```

We can't return two (or N!) different types from one function.

But this is what we want to do; this is the point of `let_value`...
we're going to need type erasure, or a variant.

variant_sender

```
1: auto sndr = ...
2:   | let_value([] (int n) {
3:       return make_variant_sender(
4:         n % 2 == 0,
5:         [] { return /* one type of sender */; },
6:         [] { return /* another type of sender */; });
7:     });
```

Now it works.

`let_value`

Again, adaptor functions tend to be quite boring...

```
1: template <sender S, typename F>
2: auto let_value(S&& s, F&& f) -> sender auto {
3:   return _let_value::sender<remove_cvref_t<S>, remove_cvref_t<F>>{
4:     std::forward<S>(s), std::forward<F>(f)};
5: }
```

Look familiar?

Good. It's about to get *much* more complex.

The `let_value` sender

So what does the `let_value` sender do when connected?

It returns an operation state. But this time it's more interesting.
Because we need to:

1. connect the first sender to an internal receiver, start the op state
2. receive the value and call the function with it, getting a new sender
3. then connect that sender to the passed-in receiver, start that op state

Computing `let_value` completions

We need to do some metaprogramming
to compute the `let_value` sender's possible completions.

```
 1: // all possible completions
 2: using raw_completions = mp_partition_q<completion_signatures_of_t<Sender>,
 3:                                        with_tag<set_value_t>>;
 4:
 5: // the error and stopped completions
 6: using unchanged_completions = mp_second<raw_completions>;
 7:
 8: using dependent_senders = /* all results of calling the function
 9:                              with each possible value completion */;
10:
11: using dependent_completions = /* all completions of dependent senders */;
12:
13: using completion_signatures = /* unchanged + dependent completions */;
```

The `connect` function

```
1: template <same_as_unqual<_let_value::sender> Self, receiver R>
2: auto tag_invoke(connect_t, Self &&self, R &&r)
3:     -> _let_value::op_state<S, std::remove_cvref_t<R>, F,
4:                             dependent_senders> {
5:   return {self.s,   // first sender
6:           r,        // passed-in receiver
7:           self.f};  // function that returns a sender
8: }
```

The operation state is going to connect the first sender to the "internal receiver" on construction.

The `let_value` receiver

The "internal" receiver that receives the result of the first sender.

```
 1: template <typename F, typename Ops>
 2: struct receiver {
 3:   [[no_unique_address]] F f;
 4:   Ops *ops;
 5:
 6:   template <typename... Args>
 7:   friend auto tag_invoke(set_value_t, receiver const& r, Args &&...args) {
 8:     r.ops->complete_first(r.f(std::forward<Args>(args)...));
 9:   }
10: };
```

When the first sender completes, we'll call the function getting the next sender. Then tell our operation state to deal with the completion.

The operation state is where the tricky part lives.

The `let_value` operation state

```
 1: template <typename Sndr, typename Rcvr, typename Func,
 2:           typename DependentSenders>
 3: struct op_state {
 4:   using first_rcvr = receiver<Func, op_state>;
 5:   using first_ops = connect_result_t<Sndr, first_rcvr>;
 6:
 7:   template <typename S>
 8:   using dependent_connect_result_t = connect_result_t<S, Rcvr>;
 9:   using dependent_ops = mp_apply<
10:       std::variant, mp_transform<dependent_connect_result_t,
11:                                  DependentSenders>>;
12:
13:   using state_t = mp_push_front<dependent_ops, first_ops>;
```

The `let_value` op state constructor

```
1: constexpr op_state(Sndr s, Rcvr r, Func f)
2:           : rcvr{r},
3:             state{std::in_place_index<0>,
4:                   with_result_of{[&] {
5:                     return connect(s, first_rcvr{f, this});
6:                   }}} {}
```

This is also a typical pattern.

Remember: operation states in general are *not movable*.
So we do a lot of this in-place construction.

with_result_of

Here's the well-known utility for in-place construction using implicit conversion.

```
1: template <typename F>
2: struct with_result_of : F {
3:   using R = std::invoke_result_t<F>;
4:
5:   constexpr operator R() { // "deducing this" please...
6:     return static_cast<F &>(*this)();
7:   }
8: };
```

Back to `let_value`

We're almost there! All we need is to handle the sender
that comes from the function, passed by the first receiver.

```
1: template <typename S> auto complete_first(S &&s) {
2:   using index = mp_find<DependentSenders, std::remove_cvref_t<S>>;
3:   auto &op = state.template emplace<index::value + 1>(
4:     with_result_of{[&] {
5:       return connect(std::forward<S>(s), rcvr); }});
6:   start(op);
7: }
```

And connect it to the originally passed-in receiver,
storing the operation state in the right place in the variant.

Recap

`then` connects the upstream sender with its own receiver.
When the upstream sender completes, we call a function with the sent value(s).
Then we call `set_value` on the downstream receiver with the result.

`let_value` also connects the upstream sender with its own receiver.
When the upstream sender completes, we call a function with the sent value(s).
Then we connect the result of that function with the downstream receiver.
And start the resulting operation state.

`let_error` and `let_stopped`

As for `let_value`, we can handle signals on the error and cancellation channels.

`let_error` and `let_stopped` have the same relationship to `let_value` as `upon_error` and `upon_stopped` have to `then`.

`let_error` and `let_stopped`

```
1: auto e_sndr =
2:     just_error(1729)    // -> set_error(rcvr, 1729)
3:   | let_error([] (int) { ... });
4:
5: auto c_sndr =
6:     just_stopped()      // -> set_stopped(rcvr)
7:   | let_stopped([] { ... });
```

Maybe it's easier to see here that `e_sndr` and `c_sndr` don't have to complete on the error and cancellation channels respectively.

We can do a lot already

Given `then` and `let_value`, we can already put together some useful senders.
(As we might expect, given `fmap` and `bind`.)

However, as we might also expect, `let_value` is often *more power* than we need.

```
1: auto s1 = /* some sender */;
2: auto s2 = /* another sender */;
3:
4: // adapting a sender is all very well,
5: // but how do we sequence two senders?
6: auto s = s1 | let_value([] { return s2; });
```

`sequence` is what we want?

```
1: // how about this?
2: auto s = s1 | sequence([] { return s2; }); // lazy s2
3: // or this?
4: auto s = s1 | seq(s2); // eager s2
```

The problem with `let_value` is that it blocks the
"forward transmission" certain of compile-time properties.

If we have the sender we want to sequence, there is no runtime choice,
and it's not dependent on the previous sender.

Put it another way

Monadic bind is a very powerful tool. But with that power comes constraints.

Because it's powerful enough to express runtime choice,
it must lose compile-time information.

There is a difference between needing an actual choice based on runtime values, and merely needing to sequence two independent senders.

Put it another other way

Maybe what we want to write are not *sender-based* interfaces,
but *sender-adaptor-based* interfaces.

Or... we don't compose values; we compose functions.

OK, back to the adaptors

If we have `then`, `let_value`, and `seq`, we can put together some useful things.

`start_on`

Here's an implementation of `start_on`.

```
1: template <scheduler Sched, sender S>
2: auto start_on(Sched &&sched, S &&s) {
3:   return sched.schedule() | seq(s);
4: }
```

A convenient way to run a sender on a scheduler.

`continue_on`

Here's an implementation of `continue_on`.

```
1: template <sender S, scheduler Sched>
2: auto continue_on(S &&s, Sched &&sched) {
3:   return s |
4:          let_value([sched] (auto &&...args) mutable {
5:            return start_on(sched, just(args...));
6:          });
7: }
```

A convenient way to switch where a computation is running.

Two more important adaptors

`when_all` and `when_any` are two remaining important adaptors.

```
1: auto all = when_all(s1, s2, s3);
2: auto any = when_any(s1, s2, s3);
```

If `let_value` was complex, these two are... more so.

I'm not going to overwhelm you with a(nother) wall of code.

`when_all`

What we need to do here:

- connect N senders to N receivers, producing N op states
- start all the op states
- catch all the sent results
- count down each finished sender; when all are done we are done
- if any error occurs, cancel the remaining senders

The receiver/op state machinery is similar to before, just tracking more things.

Reasonable limitations on `when_all`

There are some reasonable limitations we can apply to `when_all`:

- all the senders must have singular `set_value` completions
- all the senders must have the same `set_error` completion (if any)
- `when_all` is not pipeable (in the general case it is not binary)
- should we allow `when_all` with zero arguments? with one?

`when_all`… `then_all`?

`when_all` is functorial `apply`.
(The characteristic operation of an applicative functor.)

As usual in C++, the function is implicit tupling/packing.
(like `std::ranges::cartesian_product_view`)

```
1: auto when_all(sender auto...);              // identity-transform results
2:
3: auto then(sender auto, function);           // unary func transform
4:
5: auto then_all(function, sender auto...); // n-ary func transform
```

Remember: in C++ we have variadic templates… the "function-in-a-functor" interpretation of applicative is less useful than the "variadic fmap" interpretation.

`when_any`

`when_any` isn't really just one function... it has a few possible variations:

- complete with first successful completion
- complete with first success or error completion
- complete with first completion of any kind

Also some useful binary (pipeable!) variations:

- if second sender completes first, send error (`error_when`?)
- if second sender completes first, send stopped (`stop_when`?)

`when_any` is really useful

A binary operation racing two senders is useful as
a building block for timeout/cancellation scenarios.

```
1: auto timeout_after(sender auto&& s, duration_t d) {
2:   return s | error_when(time_scheduler{d}.schedule(), timeout_error{});
3: }
4:
5: auto cancel_on_user_click(sender auto&& s, button b) {
6:   return s | stop_when(gui_scheduler{b}.schedule());
7: }
```

`when_any` design decisions

Much the same as `when_all`:

- should all the senders have the same `set_error` completion?
- `when_any` is not pipeable in the general case
  - but binary pipeable versions are useful!
- we should find suitable names for variations
  - `first_successful`, `stop_when`, etc
- should we allow `when_any` with zero arguments? with one?

Part 5

Errors and cancellation

Error cases

We've already seen some of this.

Sender adaptors pass through the channels they don't handle, unaltered.

```
1: auto s = ...
2:   | then([] { /* handle success case */ })
3:   | upon_error([] { /* handle error case */ })
4:   | ...;
```

An error will bypass the `then` adaptor and be handled by `upon_error`.

Likewise `upon_stopped`. Also `let_error` and `let_stopped`.

Cooperative cancellation

Q. How does a sender know if it's cancelled?
A. By checking a `stop_token` (just like a `jthread`).

Q. Where does it get a `stop_token` from?
A. A receiver's environment.

It's up to the receiver to allow cancellation.
And it's up to the work to check for a cancellation request.

Environments

Receivers have *environments* that are queried with `get_env`.

The simplest environment is empty. This is what you get if you do nothing. There's no `stop_token` or anything else here.

```
struct empty_env {};
```

A richer environment may support various queries - like `get_stop_token`.

A receiver that exposes a `stop_token`

First, we're going to need a `stop_source`.

```
template <typename T> optional<inplace_stop_source> source{};
```

This can't live inside a receiver because it's not movable.
(It contains atomics and stuff.)

A receiver that exposes a `stop_token`

Now we need an environment that allows us to access a `stop_token` associated with a `stop_source`.

```
1: struct env {
2:   inplace_stop_token stop_token;
3:
4:   friend constexpr auto tag_invoke(get_stop_token_t, env const &self) {
5:     return self.stop_token;
6:   }
7: };
```

A receiver that exposes a `stop_token`

And we need a receiver that sets up the source, and gives us that environment.

```
 1: template <typename>
 2: struct stoppable_receiver {
 3:   stoppable_receiver() { source<stoppable_receiver>.emplace(); }
 4:
 5:   friend auto tag_invoke(get_env_t, stoppable_receiver const &) {
 6:     return env{stop_source<stoppable_receiver>->get_token()};
 7:   }
 8:
 9:   // plus set_value etc
10: };
```

Sources, Tokens, Callbacks

They're all platonically the same as the ones already in the standard.

An `inplace_stop_source` is non-movable: it's where the atomic state lives.
It has `request_stop` and `get_token` functions.

An `inplace_stop_token` is a lightweight handle obtained from a source.
It has a `stop_requested` predicate.

An `inplace_stop_callback` is registered to be called when stop is requested.
In this implementation, it's also non-movable (intrusive list)!

Naming note

Which should it be?

`in_place_*` or `inplace_*`?

Well, this is C++... we're already inconsistent.

- `std::inplace_merge`
- `std::in_place_{type, index}`
- `std::inplace_vector` (proposed)

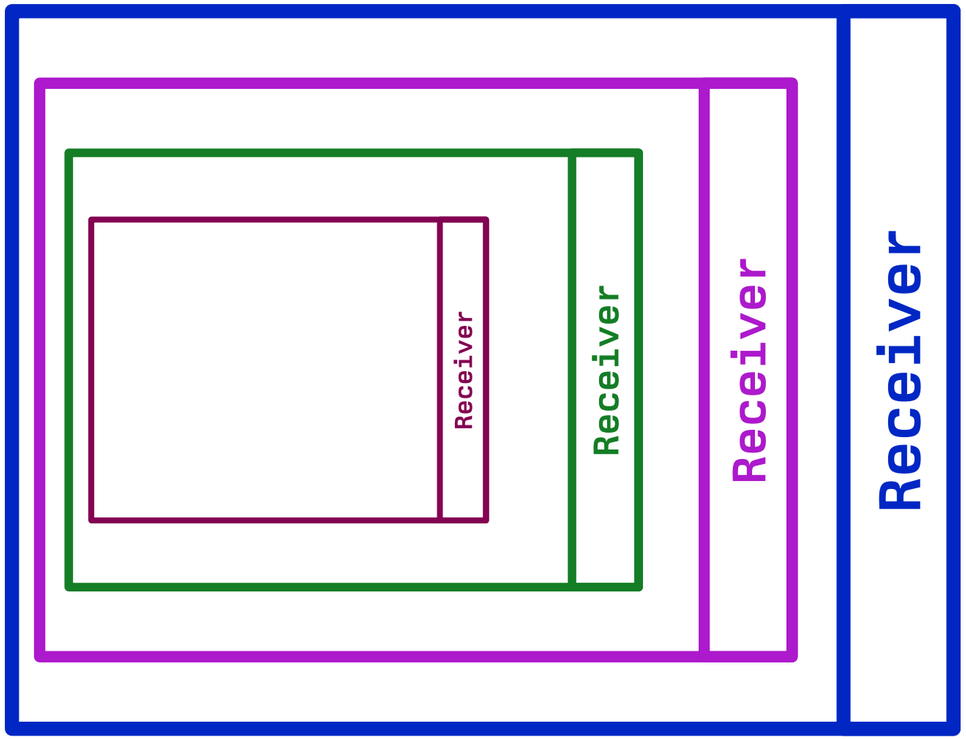
¯_(ツ)_/¯

OK, but how does cancellation work?

```
 1: auto sndr = ...
 2:            | some work
 3:            | more work
 4:            | etc
 5:            | ...;
 6:
 7: auto op = connect(sndr, stoppable_receiver<name_tag>{});
 8: start(op);
 9:
10: source<stoppable_receiver<name_tag>>->request_stop();
```

Something like this.

OK, but for real?



Only the receiver on the outside is stoppable, right?
The rest are all those "internal" receivers from the adaptors...

Forwarding queries

Some queries are designated *forwarding queries*.

```
1: template <typename E> struct forwarding_env {
2:     template <is_forwarding_query Q>
3:     friend auto tag_invoke(Q query, forwarding_env const &self) {
4:         return query(self.env);
5:     }
6:
7:     [[no_unique_address]] E env;
8: };
```

`get_stop_token` is a forwarding query.

The environments of "inner receivers" forward the environments of "outer receivers".

Example

Remember the `let_value` receiver?

```
 1: template <typename F, typename Ops>
 2: struct receiver {
 3:   [[no_unique_address]] F f;
 4:   Ops *ops;
 5:
 6:   friend auto tag_invoke(get_env_t, receiver const &r)
 7:       -> forwarding_env<env_of_t<Rcvr>> {
 8:     return forward_env_of(r.ops->rcvr);
 9:   }
10: };
```

Its receiver forwards the environment of the passed-in receiver.

Some adaptors explicitly handle cancellation

For example, `when_any` is one such adaptor.

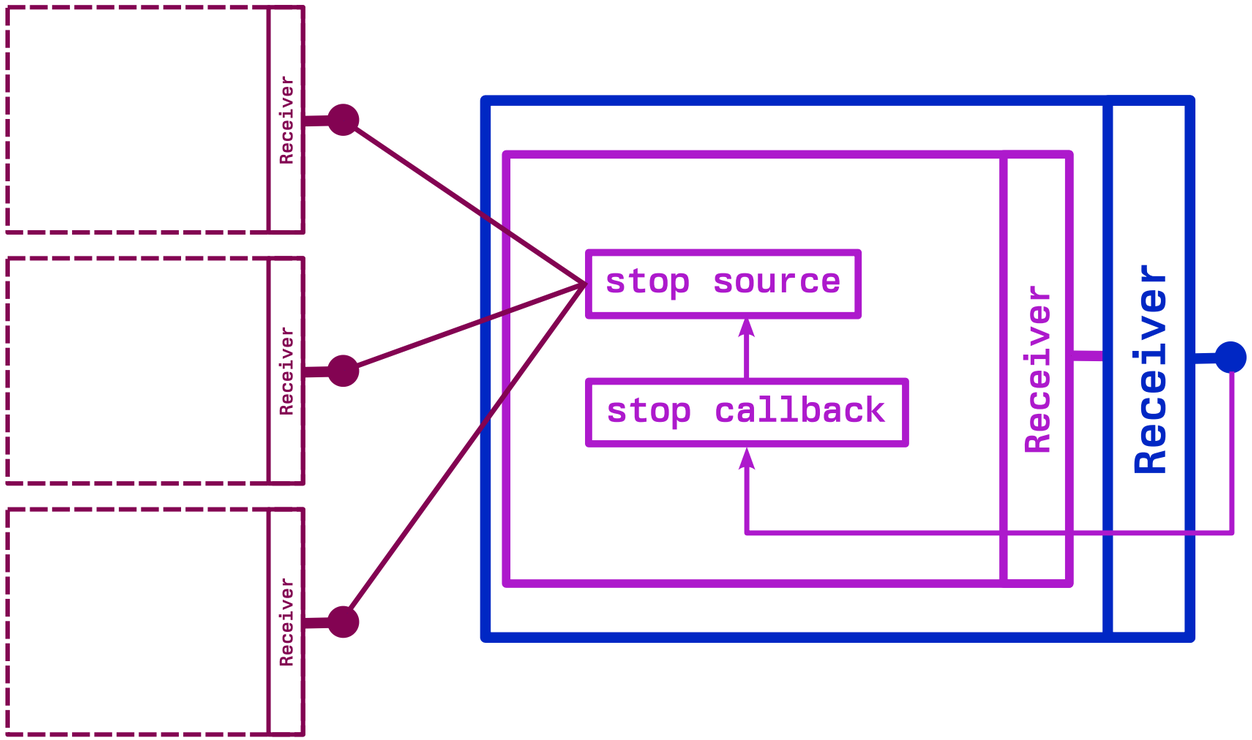
Its operation state contains *its own* `inplace_stop_source`.

Each sub-sender is connected to a receiver whose environment exposes the stop token from the internal stop source.

As soon as one sender completes successfully, calling `request_stop` on the stop source notifies the rest that they've been cancelled.

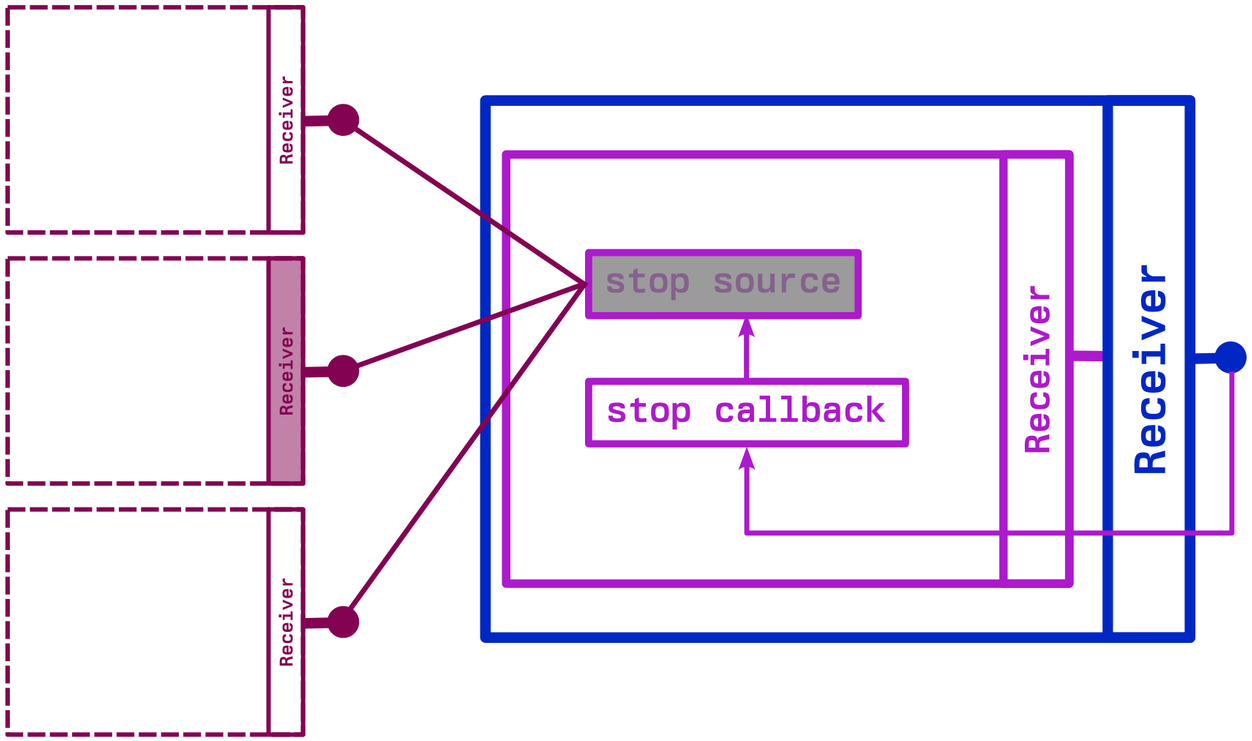
Lastly, if the passed-in receiver has a stop token, `when_any` hooks up an `inplace_stop_callback` to "forward" any external cancellation to its own internal stop source.

`when_any`: normal completion



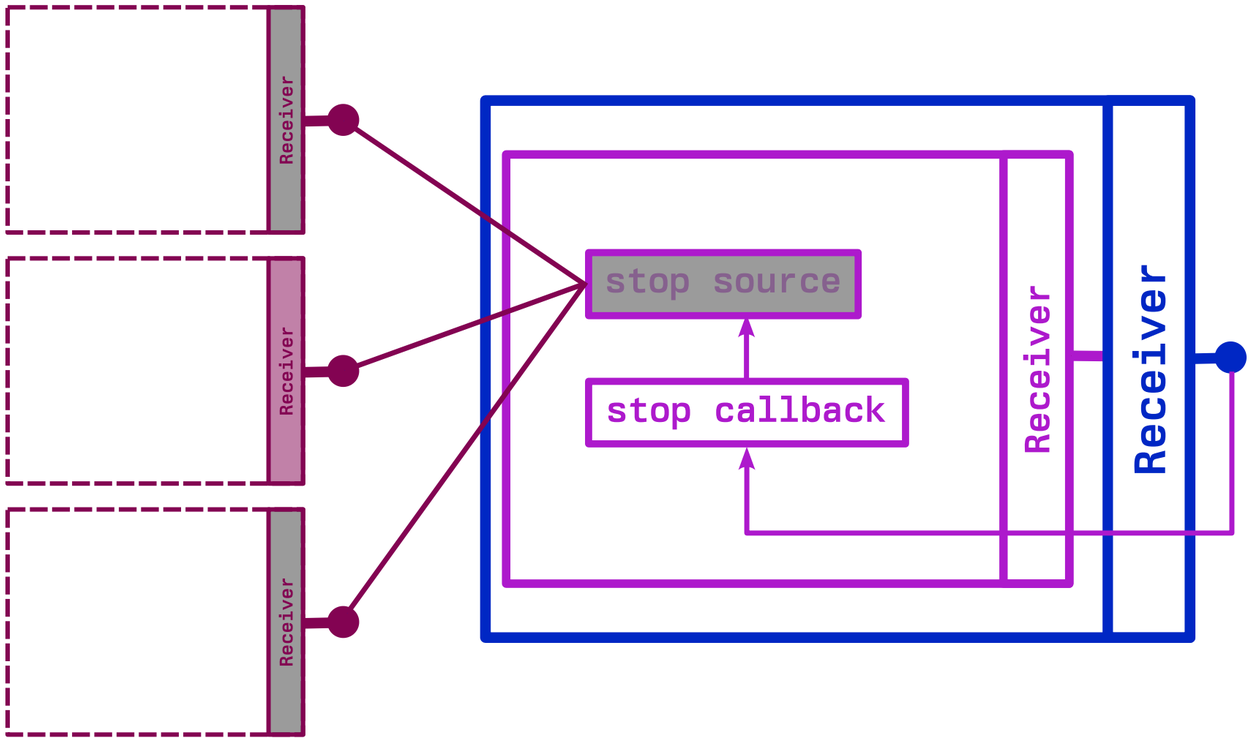
Running...

`when_any`: normal completion



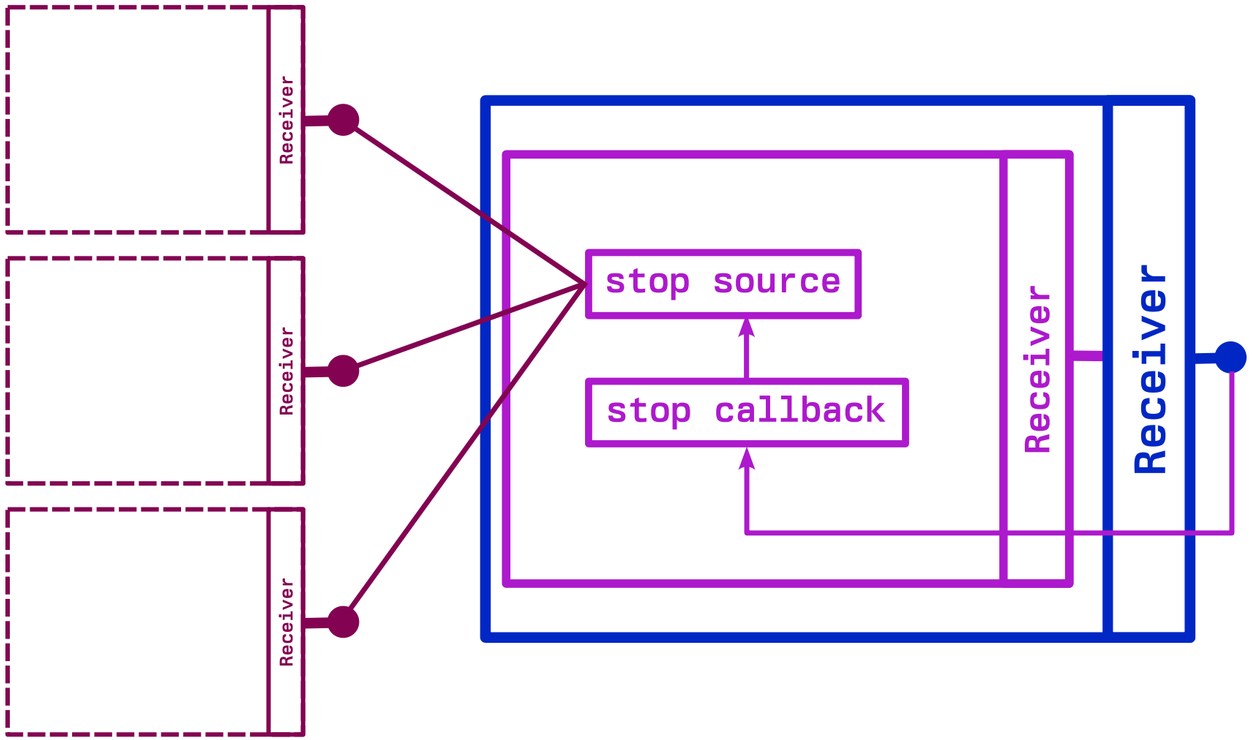
One sender finishes, `request_stop` on stop source

`when_any`: normal completion



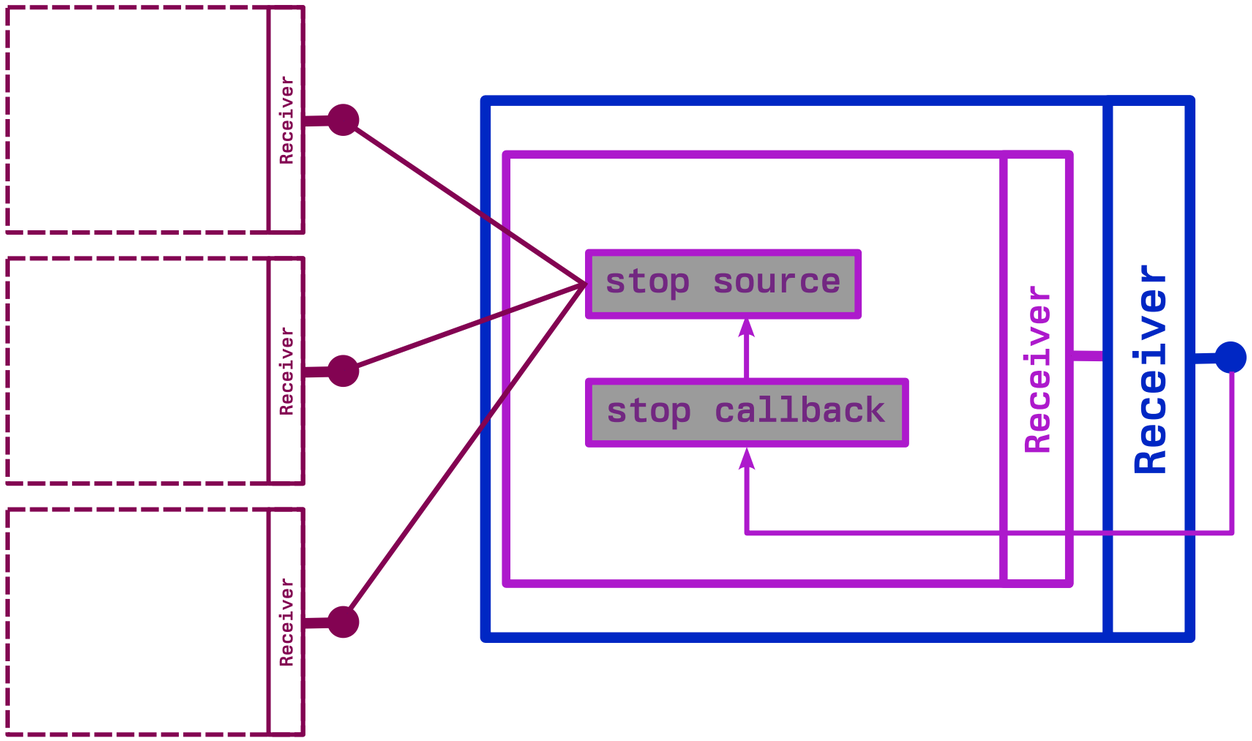
Other senders are cancelled, `when_any` finishes with `set_value`

`when_any`: cancellation



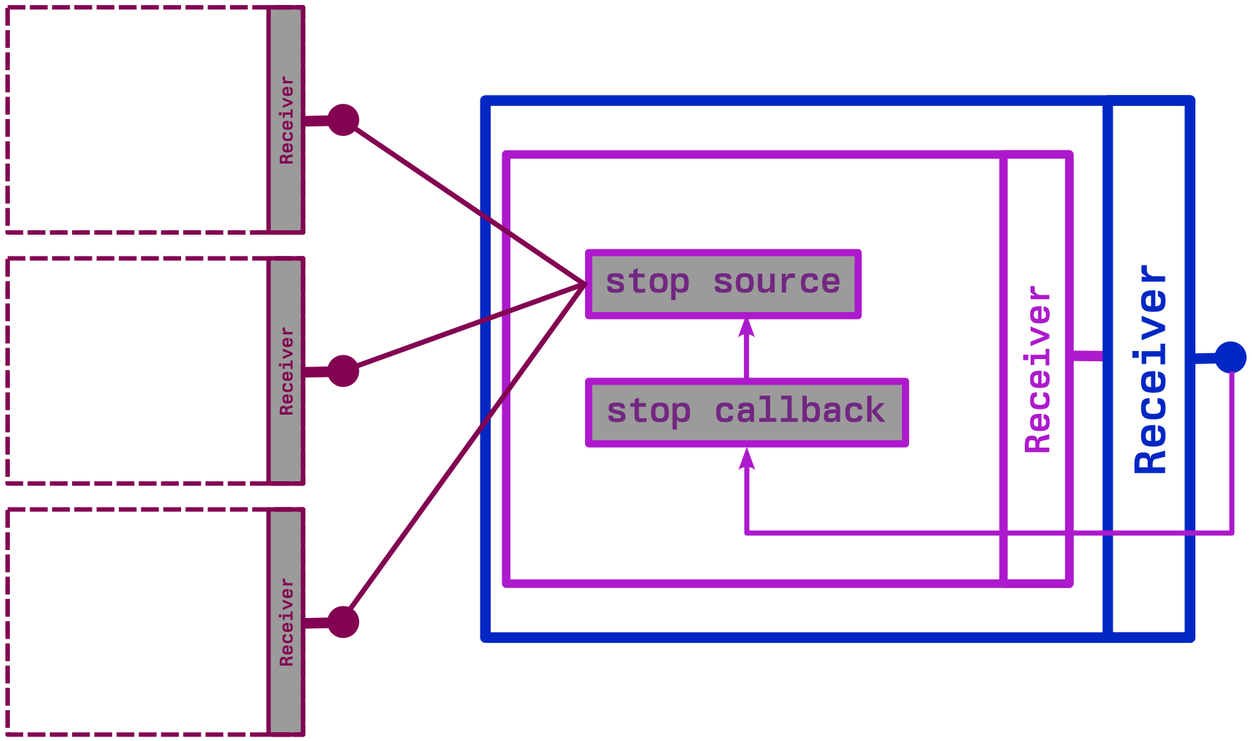
Running...

`when_any`: cancellation



External cancellation calls stop callback, `request_stop` on stop source

`when_any`: cancellation



Other senders are cancelled, `when_any` is cancelled

Receiver environments

Receiver environments hold queryable data.

Environments may nest together, mirroring operation structure:
some queries are forwarded.

This is a key to allowing cooperative cancellation from the outside.

Senders have a similar thing, but they call it attributes.

Part 6

Allocation

OK, I see examples like this one

(kind-of from Eric Niebler, edited for slideware)

```
1: sender auto hello = just("hello world"s);
2: sender auto print = hello
3:    | then([] (auto msg) { std::puts(msg.c_str()); return 0; });
4: sender auto work = start_on(io_thread, print);
5: auto [result] = sync_wait(work).value();
```

And I see how this is able to use no heap allocations.

But in embedded, I never want to use `sync_wait`.

My use cases are shaped like this

```
1: sender auto sndr = start_on(one_priority_scheduler)
2:    | read_a_register
3:    | then([] (auto reg) { return reg | 0x1; })
4:    | continue_on(another_priority_scheduler)
5:    | write_the_register;
6:
7: start_detached(sndr);
8: return; // from the current interrupt context
```

I'm living in a lovely sea of asynchronous work, controlled by the hardware.

I *never* want to block! But where does the operation state live? I can't heap allocate it.

(Also, how do I actually cancel this?)

Aside: `start_detached`

Like I said earlier, P2300 moves fast.
P3149 (`async_scope`) might supplant sender consumers in general.

The basic point: "structured concurrency" in embedded is not like "regular" lifetime-bound concurrency, because objects in embedded are not generally lifetime-bound.

So fire-and-forget (-ish) behaviour is just fine.

Well, memory has to live somewhere

That somewhere is: as part of a static allocation.
Which for my purposes, means it's a variable template.

```
 1: template <typename Uniq = decltype([]{}), sender S>
 2: auto start_detached(S &&s) -> optional<inplace_stop_source *> {
 3:   using A = allocator_of_t<env_of_t<S>>;
 4:   using O = op_state<Uniq, S, A>;
 5:   optional<inplace_stop_source *> stop_src{};
 6:   A::template construct<Uniq, O>(
 7:         [&](O &ops) {
 8:             stop_src = std::addressof(ops.stop_src);
 9:             start(ops);
10:         },
11:         std::forward<S>(s));
12:   return stop_src;
13: }
```

The `start_detached` operation state

The operation state is fairly pedestrian.
It connects the sender on construction, as we've seen before.

And on completion, the receiver causes the operation state to destroy itself.

```
1: auto tag_invoke(channel_tag auto, receiver const &r, auto &&...) {
2:   r.ops->die();
3: }
4:
5: template <typename Uniq, typename Sndr, typename Alloc>
6: struct op_state {
7:   auto die() { Alloc::template destruct<Uniq>(this); }
8: };
```

The allocator "driver"

```
1: template <typename Name, typename T, typename F, typename... Args>
2: static auto construct(F &&f, Args &&...args) -> bool {
3:   auto &a =
4:     allocator_v<Name, T, allocation_limit<Name>>;
5:   if (auto t = a.construct(std::forward<Args>(args)...); t != nullptr) {
6:     std::forward<F>(f)(*t);
7:     return true;
8:   }
9: return false;
```

The `construct` method uses the `Name` (AKA `Uniq` from before) to select a variable template specialization, which hold the actual static storage.

The allocator itself

```
 1: template <typename Name, typename T, std::size_t N>
 2: inline auto allocator_v = allocator_t<Name, T, N>{};
 3: template <typename Name>
 4: constexpr inline auto allocation_limit = std::size_t{1};
 5:
 6: template <typename Name, typename T, std::size_t N> struct allocator_t {
 7:   using storage_t = std::array<std::byte, aligned_size * N>;
 8:   alignas(alignment) storage_t data{};
 9:
10:   template <typename... Args> auto construct(Args &&...args) -> T * {
11:     ptr = /* somewhere in the storage */;
12:     return std::construct_at<T>(ptr, args...);
13:   }
14: };
```

A couple of variable templates, and a familiar (abridged) implementation.

The result

Using `start_detached` is idiomatic in embedded code.

Using `sync_wait` is (ought to be) abnormal. No blocking!

We still have the regular embedded engineering problems:

- using bounded statically-allocated memory
- care over re-entrancy

And we can deal with them in the normal way(s).

Epilogue

Closing thoughts

Thanks for sticking with me

The strength of the senders model is not just in scaling *up*,
but in scaling *down* expressively and efficiently.

Senders are totally implementable and reasonable in the embedded space.
Implementing is fun.

Using senders for embedded is a promising way to tackle problems like priority inversion, and offers a composable, error- and cancellation-aware API.

Ultimately we still have to well-engineer the issues of hard limits on memory.
And we can.