# A Wording Strategy for Inlinable Receivers

Document Number: P3986R0
Date: 2025-03-07
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

## Abstract

This paper proposes a strategy for wording the changes proposed by P3425 [1].

## Background

When a sender and receiver are connected the result is an operation state. That operation state, once started, is the locus of an asynchronous operation which, upon completion, must notify its consumer of said completion (§33.3 [exec.async.ops]). The way this notification is accomplished is by sending a completion signal to the receiver provided when connecting the sender and receiver.

Absent other information the above formulation facially requires that the operation state contain the receiver: The receiver is a C++ object and must be passed as an argument to `std::execution::set_value` (§33.7.2 [exec.set.value]), `::set_error` (§33.7.3 [exec.set.error]), or `::set_done` (§33.7.4 [exec.set.stopped]) to complete the asynchronous operation. Nothing about the above formulation provides the asynchronous operation with the means to synthesize a receiver on the fly, and therefore it must be stored if a completion signal is to be sent thereto (which it must be as a core requirement of `std::execution`'s asynchronous model, see §33.3 [exec.async.ops], paragraph 2, bullet 3).

Senders and receivers can be understood by analogy with regular synchronous functions [2] with a receiver modeling the return from a regular synchronous function. A regular synchronous function return moves control back to the caller, and therefore needs to have some way of locating this caller. This motivates the "frame pointer." In the asynchronous domain asynchronous operations have operation states rather than stack frames, but the principle still applies: The receiver needs to know the location of the operation state to which asynchronous control flows upon completion of a child operation.

Looking at normal asynchronous use cases, wherein asynchronous operations are nested (and therefore so too are their operation states), and putting the above concretely we conclude that the modal receiver has exactly one data member which is a pointer or reference to an operation

state, that it is transitively a child of the operation state which it refers to, and that therefore the offset of the stored pointer is a compile time constant. That is the storage of the pointer which is the sole data member of this modal receiver is unnecessary.

The above is the primary finding of P3425.

Unfortunately the wording strategy chosen for `std::execution` was to specify it in terms of code. Rather than describing the effects of components with prose, literal C++ code is written in the standard and the component is required to behave as if implemented by that code. This means that not only are all consequences of the provided code mandated by the standard (intentional, desirable, or otherwise), but also that attempting to change components of `std::execution` presents the same kind of mechanical burden as refactoring actual code.

The latter of the above-mentioned issues has caused issues for P3425's progress as it would require a sweeping "refactoring" of the wording of most or all `std::execution` algorithms. This paper considers an alternate approach.

# Discussion

The situation is not quite as dire as is presented in the preceding section. Parts of the behavior of `std::execution` are presented in prose. For example the standard makes no attempt to present the computation of the completion signatures of an asynchronous operation in the form of code. Instead blanket wording is provided (§33.9.9 [exec.getcomplsigs]):

*"Let `rcvr` be an rvalue whose type `Rcvr` models receiver, and let `Sndr` be the type of a sender such that `sender_in<Sndr, env_of_t<Rcvr>>` is `true`. Let `Sigs...` be the template arguments of the `completion_signatures` specialization named by `completion_signatures_of_t<Sndr, env_of_t<Rcvr>>`. Let `CSO` be a completion function. If sender `Sndr` or its operation state cause the expression `CSO(rcvr, args...)` to be potentially evaluated then there shall be a signature `Sig` in `Sigs...` such that*

*"*`MATCHING-SIG(decayed-typeof<CSO>(decltype(args)...), Sig)`

*"is `true`."*

The above has the effect that implementers of `std::execution` must write code which computes the completion functions potentially-evaluated by the algorithms they are implementing, but does not provide code which performs that computation.

A similar strategy can be used to word P3425:

- Define `std::execution::inlinable_receiver` (as P3425 already does),
- Allow or require, via blanket wording, receivers specified by the standard to model `std::execution::inlinable_receiver`, and

- Allow or require, via blanket wording, implementations of algorithms specified by the standard to:
    - Allow the lifetime of the receiver provided to `std::execution::connect` to end therein without being propagated to the operation state, and
    - Allow the resynthesis of that receiver later, as needed, via the `std::execution::inlinable_receiver` protocol

The above-described strategy is proposed by this paper. Separate wording is presented for the “allow” and “require” cases. LEWG should poll between the two.

# Wording

## Common

The wording below is common between the “allow” and “require” cases discussed in the preceding section.

### [execution.syn]

[...]

```
template<class Rcvr, class Completions>
  concept receiver_of = see below;

template<class Rcvr, class ChildOp>
  concept inlinable_receiver =
    receiver<Rcvr> &&
    requires (ChildOp* child) {
      { Rcvr::make_receiver_for(child) } noexcept -> same_as<Rcvr>;
    };

struct set_value_t { unspecified };
struct set_error_t { unspecified };
struct set_stopped_t { unspecified };
```

[...]

Note: The above change is taken from P3425 verbatim.

### [exec.recv.concepts]

[...]

Class types that are marked `final` do not model the `receiver` concept.

The `inlinable_receiver` concept defines the requirements for a receiver that can be reconstructed on-demand from a pointer to the operation-state object created when the receiver was connected to a sender. Given a receiver object, `rcvr`, of type, `Rcvr`, which was connected to a sender, producing an operation-state object, `op`, of type `Op`, and where `Rcvr` models `inlinable_receiver<Op>`, then the expression, `Rcvr::make_receiver_for(addressof(op))`, evaluates to a receiver that is equal to `rcvr`.

*[Note: Such a receiver does not need to be stored as a data-member of `op` as it can be recreated on demand –end note]*

Given objects $O_0 \ldots O_n$, then we say that $O_n$ is "transitively constructed from" $O_0$ if and only if:

- n = 1 and $O_n$ was initialized by decay-copying a reference to $O_0$, or
- n > 1 and $O_{n-1}$ is transitively constructed from $O_0$ and $O_n$ was initialized from a non-const, non-volatile rvalue reference to $O_{n-1}$

Let `e` be some well-formed expression `connect(sndr, rcvr)`. Then we say that `e` "inlines the receiver" if and only if the lifetime of all objects transitively constructed from `rcvr` end within the evaluation of `e`.

*[Note: This prevents such an expression from storing the receiver in the operation-state –end note]*

Let `e` be some well-formed expression `connect(sndr, rcvr)` where:

- `sndr` denotes a sender type defined by this section, and
- `e` inlines the receiver

Then let `op` be the result of the evaluation of `e`, and wherever the specification of the operation associated with `op` contains a glvalue which would denote an object transitively constructed from `rcvr` that glvalue instead denotes the result of `decay_t<decltype(rcvr)>::make_receiver_for(addressof(op))`.

Note: The first added paragraph above (and note thereafter) is taken from P3425 verbatim.

## Allow

The following wording allows, but does not require, implementations to inline receivers.

[exec.recv.concepts]

Let `Rcvr` be a receiver type defined by this section. Given some operation-state type `Op` it is implementation-defined whether `Rcvr` models `inlinable_receiver<Op>`.

Let `Op` be some operation-state type defined by this section, and let `Sndr` and `Rcvr` be types such that `is_same_v<connect_result_t<Sndr, Rcvr>, Op>` is `true`. If `Rcvr` models `inlinable_receiver<Op>` then it is implementation-defined whether the `connect` operation which produces an object of type `Op` inlines the receiver.

## Require

The following wording requires implementations to inline receivers in certain situations.

[exec.recv.concepts]

Let `Rcvr` be a receiver type defined by this section. Given some operation-state type `Op`, `Rcvr` satisfies `inlinable_receiver<Op>` if and only if `Rcvr` has a single data member which:

- Models `inlinable_receiver<Op>`, or
- Is a pointer or reference which, in every initialization of an object of type `Rcvr`, points or refers to a subobject of an object of type `Op`

Let `Op` be some operation-state type defined by this section, and let `Sndr` and `Rcvr` be types such that `is_same_v<connect_result_t<Sndr, Rcvr>, Op>` is `true`. If `Rcvr` models `inlinable_receiver<Op>` then the `connect` operation which produces an object of type `Op` inlines the receiver.

# References

[1] L. Baker. Reducing operation-state sizes for subobject child operations P3425R1
[2] R. Leahy. Of Operation States and Their Lifetimes P3373R2