A Wording Strategy for Inlinable Receivers

Document Number: P3986R1
Date: 2025-03-24
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LWG

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.

Note that LEWG decided on the “allow” case in Croydon 2026. The “allow” case is reflected in the subsequent wording.

Wording

[execution.syn]

[...]

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

template<class Rcvr, class ChildOp>
  concept inlinable_receiver = see below;

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]

Add to end of section:

```
namespace std::execution {

  template<class Rcvr, class ChildOp>
    concept inlinable_receiver =
      receiver<Rcvr> &&
```

```
      requires (ChildOp* child) {
        { remove_cvref_t<Rcvr>::make_receiver_for(child) } noexcept ->
          same_as<remove_cvref_t<Rcvr>>;
      };

}
```

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`, `Rcvr` models `inlinable_receiver<Op>` only if the expression `Rcvr::make_receiver_for(addressof(op))` evaluates to a receiver that is equal to `rcvr` ([concepts.equality]).

[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]

`ChildOp` may be an incomplete type.

Given objects $O_0 \ldots O_n$, O_n is *transitively constructed from* O_0 if:

- `remove_cvref_t<decltype(`O_n`)>` denotes the same type as `remove_cvref_t<decltype(`O_0`)>`, and
- Either
  - O_1 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)`. `e` *inlines the receiver `rcvr`* if the lifetimes of all objects transitively constructed from `rcvr` during the evaluation of `e` end within the evaluation of `e`.

[Note: This means such an expression does not store 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 document, and
- `e` inlines the receiver `rcvr`

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 applying the temporary materialization conversion to the expression `remove_cvref_t<decltype(rcvr)>::make_receiver_for(addressof(op))`.

Let `StdRcvr` be a receiver type defined by this document. Given some operation state type `Op` it is unspecified whether `StdRcvr` models `inlinable_receiver<Op>`.

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

Review History

R0

Presented to LEWG in Croydon 2026-03-23. The following polls were taken:

POLL: We support the “Common” + “Require” option of the wording in the paper P3986R0 (building upon P3425R0’s encouragement for a change) (vs. the “Common” + “Allow”).

Note the meaning of the votes:

SF, F: “Common” + “Require”
N: Don’t mind between the two
A, SA: “Common” + “Allow”.

SF	F	N	A	SA
2	5	2	15	5

Attendance: 47 IP, 12 online
# of Authors: 1
Author’s Position: A
Outcome: Stronger support for the “Allow” version of the wording

POLL: Approve the wording under “Common” + “Allow” sections of the wording in the paper P3986R0 (building upon P3425R0’s encouragement for a change), to resolve “CA-318 33 [exec] Reduce operation-state sizes for child operations P3425R0”.

SF	F	N	A	SA
11	12	0	5	0

Attendance: 47 IP, 10 online
# of Authors: 1
Author's Position: F
Outcome: Consensus in favor

Revision History

R1

- Removed "require" option (as LEWG voted against it)
- Adopted LWG feedback

References

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