

started_by

Document Number: P4394R0

Date: 2026-10-18

Reply-to: Robert Leahy <rleahy@rleahy.ca>

Audience: SG1, LEWG

Abstract

This paper proposes `std::execution::started_by` which allows asynchronous operations to be specified from only their completion signatures and an initiating function.

Motivation

Consider `ares_getaddrinfo`, an asynchronous operation provided by `libcares`, which is a C library:

```
typedef void (*ares_addrinfo_callback)(void *arg, int status,
                                       int timeouts,
                                       struct ares_addrinfo *result);

void ares_getaddrinfo(ares_channel_t *channel, const char *name,
                    const char* service,
                    const struct ares_addrinfo_hints *hints,
                    ares_addrinfo_callback callback,
                    void *arg);
```

It may be brought into the penumbra of `std::execution` as follows:

```
struct ares_getaddrinfo_sender {
    using sender_concept = std::execution::sender_tag;

    ::ares_channel_t* channel;
    const char* name;
    const char* service;
    const ::ares_addrinfo_hints* hints;

    template<class Self, class... Env>
    static constexpr auto get_completion_signatures() {
        return std::execution::completion_signatures<
            std::execution::set_value_t(::ares_addrinfo*),
            std::execution::set_error_t(int)>{};
    }
};
```

```

}

template<class Rcvr>
struct operation {
    using operation_state_concept = std::execution::operation_state_tag;

    ::ares_channel_t* channel;
    const char* name;
    const char* service;
    const ::ares_addrinfo_hints* hints;
    Rcvr rcvr;

    void start() & noexcept {
        ::ares_getaddrinfo(
            channel,
            name,
            service,
            hints,
            [](void* arg, int status, int, ::ares_addrinfo* result) noexcept {
                auto& self = *static_cast<operation*>(arg);
                if (status == ARES_SUCCESS) {
                    std::execution::set_value(std::move(self.rcvr), result);
                } else {
                    std::execution::set_error(std::move(self.rcvr), status);
                }
            },
            this);
    }
};

template<class Rcvr>
auto connect(Rcvr rcvr) const noexcept {
    return operation<Rcvr>{
        channel,
        name,
        service,
        hints,
        std::move(rcvr)};
};

std::execution::sender auto ares_getaddrinfo(
    ::ares_channel_t* channel,

```

```

    const char* name,
    const char* service,
    const ::ares_addrinfo_hints* hints) noexcept
{
    return ares_getaddrinfo_sender{channel, name, service, hints};
}

```

Note that this is almost entirely boilerplate to bind the selected parameters (`channel`, `name`, `service`, and `hints`) into a sender, allow that sender to be connected, and provide the necessary shell of an operation state. While none of this boilerplate is particularly error prone it is tedious, verbose, and distracting, and therefore could benefit from abstraction.

Design

`started_by` is sender factory with the following signature:

```

struct empty-base {};

template<valid-completion-signatures Signatures,
        class Base = empty-base,
        class Initiation,
        class... Args>
std::execution::sender auto started_by(
    Initiation&& init,
    Args&&... args);

```

The operation state formed by connecting a `started_by` sender has the following form:

```

struct operation-state : Base {
    using operation_state_concept = std::execution::operation_state_tag;

    // See below
    void complete(Tag, Args&&...) noexcept;
};

```

Where there is a `complete` member function for each completion signature `Tag(Args...)`. The completion signatures are supplied by the first template parameter to `started_by`.

The first parameter is the “initiation.” A unary invocable which, when the resulting asynchronous operation is initiated, is invoked with an lvalue reference to the operation state. The initiation is expected to arrange for the `complete` member function of the operation state to be invoked exactly once, at the conclusion of the resulting asynchronous operation, in accordance with the “receiver contract”.

The trailing parameters are arguments which will be used to initialize the Base base class subobject of the operation state.

Example

An implementation of the earlier wrapper:

```
std::execution::sender auto ares_getaddrinfo(
    ::ares_channel_t* channel,
    const char* name,
    const char* service,
    const ::ares_addrinfo_hints* hints) noexcept
{
    using signatures = std::execution::completion_signatures<
        std::execution::set_value_t(::ares_addrinfo*),
        std::execution::set_error_t(int)>;

    return std::execution::started_by<signatures>(
        [channel, name, service, hints]<class Op>(Op& op) noexcept {
            ::ares_getaddrinfo(
                channel, name, service, hints,
                [](void* arg, int status, int, ::ares_addrinfo* result) noexcept {
                    auto& op = *static_cast<Op*>(arg);
                    if (status == ARES_SUCCESS) {
                        op.complete(std::execution::set_value, std::move(result));
                    } else {
                        op.complete(std::execution::set_error, std::move(status));
                    }
                },
                std::addressof(op));
        });
}
```

A striking improvement. The entirety of the code is now concerned with the particular concern of adapting `ares_getaddrinfo`.

Note the above operation completely ignores stop requests. That is the responsibility of a different algorithm and therefore a topic for another paper [1].

Discussion

Lineage

`std::execution::started_by` is the result of decomposing an algorithm called `externally_completed` which the author has presented in several conference talks [2][3][4][5].

The other half of the above-mentioned decomposition is left to a different paper [1].

Trivial Closure Pointers

An operation state remains valid and stable in memory for the duration of an asynchronous operation. This is similar to the way in which synchronous stack frames remain valid and stable in memory for the duration of a synchronous function call. As such the initiation of an operation modelled by `started_by` can retain and use the provided operation state reference beyond the return therefrom, through until it invokes `complete` on said operation state (whereupon the asynchronous operation ends and the operation state becomes “invalid”).

The above has two important consequences for convenience and correctness:

- Since the initiation is a subobject of the operation state it can depend on its own validity until it completes the asynchronous operation it is implementing (it is of particular interest that the captures of a lambda initiation remain thusly valid)
- When interoperating with C-style APIs a “closure pointer” can be provisioned easily (i.e. a pointer to the operation state)

The second bullet is on display in the adaptation of `ares_getaddrinfo` above which makes use of `started_by`:

```
return std::execution::started_by<signatures>(
    [channel, name, service, hints]<class Op>(Op& op) noexcept {
        ::ares_getaddrinfo(
            channel, name, service, hints,
            [](void* arg, int status, int, ::ares_addrinfo* result) noexcept {
                auto& op = *static_cast<Op*>(arg);
                // ...
            },
            std::addressof(op));
    });
```

Note that since this is a pointer to the operation state it enables said operation state to be recovered on the other side of a C-style API.

As is the case everywhere in `std::execution` these conveniences require no allocation due to the “receiver contract.” Whosoever receives the operation state from `std::execution::connect` is obligated, if they choose to start the operation, to ensure the validity thereof until its completion.

Explicit Completion Signatures

The fact that a consumer of `started_by` needs to provide the completion signatures of the operation they’re implementing at first seems heterodox: Other adaptors and factories in `std::execution` don’t work this way.

However there is no other way to obtain the necessary information. Invocation of the initiation reports completion of the operation through the operation state. The C++ language does not provide a way to extract therefrom the list of completion signatures which are actually used. Other prior art in this space, such as `exec::just_from` et al. from Nvidia’s `stdexec`, rely on a probe function and synchronous invocation to extract this information. `started_by`, on the other hand, permits asynchronous execution.

Therefore users are expected to enumerate the ways in which their operation will eventually complete by providing an instantiation of `std::execution::completion_signatures` which includes every completion signature that the operation may produce. Since the proposed wording in this paper uses that information to provide overloads of `complete` on the operation state failure to enumerate each completion signature which will be used will result in a compilation error.

Optional Base Class

No exploration heretofore, including the motivating example, justifies the optional, second template parameter: The base class from which the operation state shall publicly and unambiguously derive.

The fact that lambda captures can be relied on throughout the lifetime of the asynchronous operation could be construed as an argument against this template parameter: Data members can be, in effect, added to the operation state by lambda capture.

In order to provide the useful properties of operation states (see above) an operation state must not be moved during the lifetime of its corresponding asynchronous operation. The standard goes even further and demands that users never move or copy any of its operation states whether the corresponding asynchronous operation is in progress or otherwise (§33.8.1 [exec.opstate.general]):

“The program is ill-formed if it performs a copy or move construction or assignment operation on an operation state object created by connecting a library-provided sender.”

Therefore operation states may, in general, contain immovable subobjects. If an operation implemented in terms of `started_by` wishes to avail itself of such objects they cannot be stored in the initiation (i.e. as lambda captures) as the initiation must be:

- Decay-copied into the sender, and then
- Moved or copied (depending on the cv- and ref-qualification with which the sender is connected) into the operation state

This immediately motivates the optional, second template parameter.

There is however another, less obvious reason: Type erased completion.

`started_by` operation states are specified to supply a non-template `complete` member function for each possible completion. They are also specified to derive publicly and unambiguously from the type indicated by the optional, second template parameter. Therefore for some completion signature `Tag(Args...)` the supplied base class can be, for example:

```
struct base {  
    virtual void complete(Tag, Args&&...) noexcept = 0;  
};
```

Note that despite GCC and Clang's `-Wnon-virtual-dtor` this formulation is perfectly legitimate: The virtual function provides a type erased way to interact with an object whose lifetime is managed non-polymorphically.

An example of the usefulness of this technique is the following example realization of `std::execution::run_loop` (note that this implementation does not have stop support):

```
class run_loop {  
    struct operation_base {  
        operation_base* next = nullptr;  
  
        virtual void complete(std::execution::set_value_t) noexcept = 0;  
    };  
  
    std::mutex mutex_;  
    std::condition_variable cv_;  
    operation_base* head_ = nullptr;  
    operation_base* tail_ = nullptr;  
    bool finishing_ = false;  
  
public:  
    class scheduler {  
        run_loop* loop_;  
    public:
```

```

using scheduler_concept = std::execution::scheduler_tag;

explicit scheduler(run_loop& loop) noexcept : loop_(&loop) {}

friend bool operator==(scheduler, scheduler) = default;

std::execution::sender auto schedule() const {
    using signatures = std::execution::completion_signatures<
        std::execution::set_value_t()>;

    return std::execution::started_by<signatures, operation_base>(
        [&loop = *loop_](auto& op) noexcept {
            const std::lock_guard _(loop.mutex_);
            if (loop.tail_) {
                loop.tail_->next = &op;
            } else {
                loop.head_ = &op;
            }
            loop.tail_ = &op;
            loop.cv_.notify_one();
        });
}

};

scheduler get_scheduler() noexcept {
    return scheduler{*this};
}

void finish() noexcept {
    const std::lock_guard _(mutex_);
    finishing_ = true;
    cv_.notify_all();
}

void run() noexcept {
    std::unique_lock lock(mutex_);
    for (;;) {
        cv_.wait(lock, [&]() noexcept { return finishing_ || head_; });
        if (!head_) {
            return;
        }

        const auto op = head_;

```

```

        head_ = op->next;
        if (!head_) {
            tail_ = nullptr;
        }

        lock.unlock();
        op->complete(std::execution::set_value);
        lock.lock();
    }
}
};

```

Initiation as an Invocable

The discussion above regarding the transformative power of customizing the operation state's base class raises a question: Why not mandate such a base class be specified, and supply the initiation as a member function thereof?

There are two problems with the above formulation:

- For simple adaptations it is inconvenient to have to write an entire base class (especially given how much more convenient lambda captures are as opposed to data members)
- In the case where type erasure is not desired the initiation implementation needs the type of the derived operation state to call `complete`

Structured Completion Requirements

`std::execution` requires structured completion (§33.3 [exec.async.ops], emphasis added):

*“After [the] start operation [of an asynchronous operation] has begun executing, **exactly one completion operation** shall execute.”*

And `started_by`, being a `std::execution` operation, requires the same, and therefore may not be a suitable tool for integrating asynchronous operations which are not themselves structured.

This is not a defect. Structured primitives are difficult [6] or impossible [7] to bootstrap from unstructured primitives.

Note that wording is likely necessary to articulate normatively that invocation of the initiation satisfies the “receiver contract.” A note has been left below to consult with LWG as to how to achieve this.

Receiver-Dependent Initiation

The initiation is provided with a reference to the operation state that contains it. This is not a problem in practice but does create friction with the way in which `std::execution` performs type checking: By throwing an exception from `std::execution::get_completion_signatures` [8].

The reason for this is that the type of the operation state depends on the type of the receiver. But `std::execution::get_completion_signatures` is parameterized on the type of the environment, not the type of the receiver.

This paper therefore proposes biphasic type checking for `started_by`:

- Within *check-types* invocability is checked using an “archetype” receiver, which is an established practice elsewhere in `std::execution` [9].
- `connect` reestablishes invocability with the actual type of the receiver, and verifies that the throwing-ness of such invocation is identical to that computed with the archetype

The latter bullet is to ensure completion signatures can be accurately computed: If the initiation possibly throws `started_by` must add

`std::execution::set_error_t(std::exception_ptr)` to its list of completion signatures.

Since `std::execution::get_completion_signatures` only receives the type of the environment this computation is performed with only the environment’s type, and is therefore performed with the archetype receiver. Requiring the throwing-ness to remain unchanged when the actual receiver becomes known ensures that the advertised completion signatures accurately describe the fully-instantiated operation.

Proposal

[execution.syn]

```
namespace std::execution {  
    [...]
```

```
    // [exec.factories], sender factories
```

```
    struct default-started-by-base {}; // exposition only
```

```
    struct just_t { unspecified };  
    struct just_error_t { unspecified };  
    struct just_stopped_t { unspecified };  
    struct schedule_t { unspecified };  
    template<valid-completion-signatures Signatures,
```

```

        class Base = default-started-by-base>
        struct started_by_t { unspecified };

        inline constexpr just_t just{};
        inline constexpr just_error_t just_error{};
        inline constexpr just_stopped_t just_stopped{};
        inline constexpr schedule_t schedule{};
        inline constexpr unspecified read_env{};
        template<valid-completion-signatures Signatures,
                class Base = default-started-by-base>
        inline constexpr started_by_t started_by{};

        template<scheduler Sch>
        using schedule_result_t = decltype(schedule(declval<Sch>()));

        [...]
    }

```

[exec.started_by]

Note: This is a new section. This depends on P4354 [10].

`started_by` is a sender factory whose asynchronous operations are initiated by invoking a user-provided function object.

The name `started_by` denotes a customization point object.

For a subexpression `init` and a pack of subexpressions `args`, let `Init` be `decltype((init))` and let `Args` be the pack of types `decltype((args))...`

The expression `started_by<Signatures, Base>(init, args...)` is ill-formed if any of the following is true:

- `Signatures` does not satisfy *valid-completion-signatures*,
- `Base` is not a class type or is final,
- `Init` does not satisfy *movable-value*, or
- `(movable-value<Args> && ...)` is false.

Otherwise, `started_by<Signatures, Base>(init, args...)` is expression-equivalent to:

```
make-sender(started_by<Signatures, Base>, product-type{init, args...})
```

The expression `started_by<Signatures>(init)` is expression-equivalent to `started_by<Signatures, default-started-by-base>(init)`.

Let *started-by-state-base* denote the following exposition-only class template:

```
template<class Derived, class Signatures, class Base>
struct started-by-state-base : Base {
    using Base::Base;

    template<class... Args>
        requires is_aggregate_v<Base> && is_constructible_v<Base, Args...>
        constexpr started-by-state-base(Args&&... args)
            noexcept(is_nothrow_constructible_v<Base, Args...>)
            : Base(std::forward<Args&&>(args)...) {}
};
```

see below

Let *State* denote the specialization *started-by-state-base*<Derived, Base, Signatures>. Let *Sigs...* be the template arguments of the *completion_signatures* specialization denoted by Signatures. For each completion signature *Tag*(Args...) in *Sigs...*, *State* has a public member function with a definition equivalent to:

```
constexpr void complete(Tag tag, Args&&... args) noexcept {
    static_cast<Derived&>(*this).complete(std::forward<Args>(args)...);
}
```

Let *started-by-state* denote the following exposition-only class template:

```
template<class Signatures, class Initiation, class Base, class Rcvr>
struct started-by-state : started-by-state-base<
    started-by-state<Signatures, Initiation, Base, Rcvr>,
    Signatures,
    Base>
{
    using operation_state_concept = operation_state_tag;

    using base-t = started-by-state-base<
        started-by-state,
        Signatures,
        Base>; // exposition only

    Rcvr rcvr; // exposition only
    Initiation init; // exposition only

    template<class... Args>
        requires constructible_from<base-t, Args...>
};
```

```

constexpr started-by-state(Initiation init, Rcvr rcvr, Args&&... args)
    noexcept(
        is_nothrow_move_constructible_v<Initiation> &&
        is_nothrow_constructible_v<base-t, Args...>)
    : Base(std::forward<Args>(args)...),
      rcvr(std::move(rcvr)),
      init(std::move(init)) {}

constexpr void start() & noexcept {
    try {
        INVOKE(std::move(init), *this);
    } catch (...) {
        if constexpr (!is_nothrow_invocable_v<Initiation, started-by-state>) {
            set_error(std::move(rcvr), current_exception());
        }
    }
}

template<class Tag, class... Args>
constexpr void complete(Tag tag, Args&&... args) noexcept {
    tag(std::move(rcvr), std::forward<Args>(args)...);
}
};

```

Let *archetype-receiver* denote the following exposition-only class template:

```

template<class Env>
struct archetype-receiver {
    using receiver_concept = receiver_tag;

    template<class... ResultDatums>
    constexpr void set_value(ResultDatums&&...) && noexcept {}
    template<class ResultDatum>
    constexpr void set_error(ResultDatum&&) && noexcept {}
    constexpr void set_stopped() && noexcept {}

    constexpr Env get_env() const noexcept {}
};

```

The exposition-only class template *impls-for* ([exec.snd.expos]) is specialized for *started_by_t* as follows:

```

namespace std::execution {
    template<class Signatures, class Base>

```

```

struct impls-for<started_by_t> : default-impls {
    static constexpr auto connect = see below;

    template<class Sndr, class... Env>
        static constexpr void check-types();
};
}

```

```

template<class Sndr, class... Env>
    static constexpr void check-types();

```

Equivalent to:

```

if constexpr (sizeof...(Env) == 1) {
    constexpr auto base = template_arguments_of(^tag-type<Sndr>)[1];

    using data_type = data-type<Sndr>;
    using raw_initiation_type = decltype(get<0>(declval<data_type>()));
    using initiation_type = decay_t<raw_initiation_type>;
    using receiver_type = archetype-receiver<Env...>;
    using operation_state_type = started-by-state<Signatures,
                                                initiation_type,
                                                typename [:base:],
                                                receiver_type>;

    constexpr auto args = [] {
        constexpr auto types = template_arguments_of(^data_type);
        vector<meta::info> result{^raw_initiation_type, ^receiver_type};
        template for (constexpr auto type : types | views::drop(1)) {
            result.push_back(
                ^^decltype(std::forward_like<Sndr>(declval<[:type:]>())));
        }
        return result;
    }();

    if constexpr (!is_constructible_type(^operation_state_type, args)) {
        throw unspecified-exception{};
    }

    if constexpr (!is_invocable_v<initiation_type, operation_state_type&>) {
        throw unspecified-exception{};
    }
}

```

The member *impls-for*<started_by_t>::connect is initialized with a callable object equivalent to the following lambda expression:

```
[<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr rcvr)
noexcept(see below)
requires see below
{
    auto& [_ , data] = sndr;
    auto& [init, ...args] = data;

    constexpr auto targ = template_arguments_of(^tag-type<Sndr>);
    using signatures = [:targ[0]:];
    using base = [:targ[1]:];

    return started-by-state<signatures,
                                decay_t<decltype(init)>,
                                base,
                                Rcvr(std::forward_like<Sndr>(init),
                                    std::move(rcvr),
                                    std::forward_like<Sndr>(args)...);
}
```

The expression in the noexcept clause is equivalent to noexcept(*e*) where *e* is the expression evaluated by the return statement.

Let *Data* denote *data-type*<Sndr>, let *Initiation* denote *decay_t*<decltype(get<0>(declval<Data>()))>, and let *Env* denote *env_of_t*<Rcvr>. Let *targ* denote *template_arguments_of*<(^tag-type<Sndr>)>, let *Signatures* denote *[:targ[0]:]*, and let *Base* denote *[:targ[1]:]*. Let *State* denote *started-by-state*<Signatures, Initiation, Base, Rcvr> and let *ArchetypeState* denote *started-by-state*<Signatures, Initiation, Base, archetype-receiver<Env>>. The expression in the requires-clause is equivalent to:

```
is_invocable_v<Initiation, State&> &&
(is_nothrow_invocable_v<Initiation, State&> ==
 is_nothrow_invocable_v<Initiation, ArchetypeState&>)
```

Requirements on the Initiation

It is likely that library wording is necessary to demand that invocation of initiation starts an asynchronous operation which conforms to the requirements of *std::execution*. The author does not know the shape of such wording and therefore would like to obtain that from LWG when/if the time comes.

Implementation Experience

The author has implemented the predecessor to this algorithm, `externally_completed`, and used it in the implementation of an `io_uring` execution context [2][3], and integration of libcares asynchronous operations into the `std::execution` ecosystem [4].

In making `externally_completed` ready for proposal the author realized that it could be split in two: `started_by` (proposed by this paper) and `stopped_by` [1].

`started_by` has been implemented against Nvidia's `stdexec` [11].

References

- [1] R. Leahy. `stopped_by` P4390R0
- [2] R. Leahy. Towards Async Everything Part 1: Senders as the Lowest Layer C++Now 2026
- [3] R. Leahy. Towards Async Everything Part 2: Scopes, Construction, and Destruction C++Now 2026
- [4] R. Leahy. The Async Pattern Graveyard ACCU on Sea 2026
- [5] R. Leahy. What Are We Synchronizing? CppCon 2026
- [6] R. Leahy. `std::execution` in Asio Codebases: Adopting Senders Without a Rewrite CppCon 2025
- [7] R. Leahy. `user_data` Is a Return Address or I'm Not Unreasonable for Expecting Functions to Return C++Now 2026
- [8] E. Niebler. High-Quality Sender Diagnostics with `Constexpr` Exceptions P3557R3
- [9] R. Leahy. When Do You Know `connect` Doesn't Throw? P3388R3
- [10] R. Leahy. No More `transform_sender` Workarounds P4354R0
- [11] <https://github.com/NVIDIA/stdexec/pull/2348>