A Universal Continuation Model

Document Number:	P4126R0
Date:	2026-03-18
Audience:	EWG, SG1, LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Klemens Morgenstern klemens.d.morgenstern@gmail.com C++ Alliance Proposal Team

Table of Contents

Abstract

Senders pay a frame allocation to enter the awaitable protocol. They do not have to.

The IoAwaitable protocol (P4003R1[1]) defines a contract between a coroutine and an I/O reactor: the coroutine suspends, the reactor performs the operation, and the executor resumes the coroutine when the result is ready. The only way to obtain a `coroutine_handle<>` today is from a coroutine, and a coroutine requires a frame allocation. A sender pipeline that wants to invoke an IoAwaitable must allocate a coroutine frame to get a handle - even though the sender already has its own operation state and does not need a frame.

This paper is additive. It does not take anything away from senders, from coroutines, or from any existing design. It gives senders something they do not have today: zero-allocation access to every awaitable ever written - timers, mutexes, channels, semaphores, I/O operations, and anything else the ecosystem produces. It gives awaitable authors a new consumer base without modifying a single line of their code.

This paper traces the history of alternative coroutine designs that explored the boundary between type erasure and type visibility, observes that C++ can support multiple coroutine models serving different domains, and explores language-level options to let senders invoke awaitables without allocating a coroutine frame. The goal is one I/O implementation consumed by both coroutines and senders with zero allocation overhead.

Revision History

R0: March 2026 (post-Croydon mailing)

- Initial version.

---

1. Disclosure

This paper is part of the Network Endeavor (P4100R0[2]), a project to bring networking to C++29 using a coroutine-native approach. The author developed and maintains Corosio[3] and Capy[4] and believes coroutine-native I/O is the correct foundation for networking in C++. Coroutine-native I/O does not target compile-time work graphs. The author provides information, asks nothing, and serves at the pleasure of the chair.

This paper seeks input from EWG, SG1, LEWG, and the sender/receiver community. The ideas are presented for discussion, not as a finished proposal. The author invites collaboration from compiler implementers, language designers, and anyone who has thought about the boundary between coroutines and senders.

---

2. The Goal

One I/O implementation. Both coroutines and senders consume it. Zero allocation for either path.

2.1 The Executor

The coroutine executor (P4003R1[1]) has two operations:

```
coroutine_handle<>
dispatch(coroutine_handle<> h) const;

void post(coroutine_handle<> h) const;
```

`post` schedules a handle for later resumption on the execution context. `dispatch` may resume the handle inline - it returns a `coroutine_handle<>` that the caller's `await_suspend` returns to the compiler for symmetric transfer. If `dispatch` defers, it returns `noop_coroutine()`.

The reactor does not call `h.resume()` directly. When an I/O operation completes, the reactor calls the executor - either `executor.dispatch(h)` or `executor.post(h)`. The executor is the policy point that determines how and where the handle resumes.

2.2 The Awaitable

Consider `read_some`, the canonical IoAwaitable:

```
struct read_some_awaitable {
    stream& s_;
    mutable_buffer buf_;
    io_result<size_t> result_;

    bool await_ready() noexcept
    {
        return false;
    }

    coroutine_handle<>
    await_suspend(
        coroutine_handle<> h,
        io_env const* env) noexcept
    {
        s_.impl_->async_read(
            buf_, h, env);
        return noop_coroutine();
    }

    io_result<size_t>
    await_resume() noexcept
    {
        return result_;
    }
};
```

A coroutine user writes:

```
auto [ec, n] = co_await stream.read_some(buf);
```

The compiler provides the handle. When the I/O completes, the reactor calls the executor, and the executor resumes the coroutine. Zero user effort.

A sender pipeline wants to do the same thing - pass a handle to `await_suspend`, have the executor resume it when the I/O completes. The sender is not a coroutine. It has a receiver. It has operation state. It does not have a frame. It needs a `coroutine_handle<>` that, when the executor calls `.resume()`, invokes a function on the sender's own state.

The handle is the only thing the executor sees. It calls `.resume()`. It does not know or care whether the handle points at a coroutine frame or a callback.

3. The Problem

The only way to obtain a `coroutine_handle<>` today is from a coroutine. A coroutine requires a frame allocation. The sender-to-awaitable bridge in P4092R0[5], "Consuming Senders from Coroutine-Native Code," demonstrates this: the bridge creates a coroutine whose sole purpose is to hold a handle that the reactor can resume. The coroutine frame is the tax.

P4092R0[5] Appendix A shows the bridge implementation. The `bridge_task` coroutine exists to produce a `coroutine_handle<>`. The coroutine body calls `co_await` on the IoAwaitable, and the `await_suspend` receives the handle from the compiler. The bridge works. It allocates a coroutine frame per I/O operation.

For a coroutine user, the frame allocation is the cost of doing business - the frame holds the coroutine's local variables, its suspension-point state, and its promise. The frame earns its allocation. For a sender pipeline, the frame holds nothing the sender needs. The sender already has its own operation state. The frame is overhead.

One allocation per I/O operation. For high-throughput networking - millions of operations per second - that matters.

4. The Shape of an I/O Operation

An I/O operation can take one of two shapes: a sender or an awaitable. The choice determines which consumption model pays a tax and which runs at zero cost.

If the I/O operation is a sender, coroutines consume it through `co_await` on the sender. The sender's `connect` produces an operation state. The coroutine must store that operation state somewhere - typically in the coroutine frame or in a bridge object. The sender's completion calls `set_value` on a receiver, which must resume the coroutine. The machinery to connect a sender to a coroutine - `execution::task`, or a bridge like the one in P4092R0[5] - is the tax coroutines pay. P3552R3[22], "Add a Coroutine Task Type," is this tax made standard: it type-erases the operation state, allocates, and converts an `error_code` to `exception_ptr` through `AS-EXCEPT-PTR`.

If the I/O operation is an awaitable, coroutines consume it directly. `co_await stream.read_some(buf)` is the language feature working as designed. The compiler provides the handle. The awaitable suspends the coroutine. The reactor completes the operation. The executor resumes the coroutine. No bridge. No type erasure. No allocation beyond the coroutine frame that the coroutine already needs for its own state.

The asymmetry is structural. Coroutines are a language feature. Awaitables are the native protocol of that language feature. A coroutine consuming an awaitable is zero-cost by construction. A coroutine consuming a sender requires a bridge - and every bridge has a cost.

The question is whether senders can consume an awaitable at zero cost. Today they cannot - they need a coroutine frame to get a handle. This paper proposes to eliminate that cost. If a callback handle exists, senders consume awaitables at zero cost too.

The awaitable is the right shape for an I/O operation because it is the shape that makes the language feature free, and this paper makes it free for senders as well.

---

5. The Timeline

The tension between "frame hidden from the caller" and "frame visible to the caller" has been present since the earliest coroutine proposals. This section traces the published record so the reader can follow the trail.

Year	Paper	Author(s)	Design
2015	N4453[6]	Kohlhoff	Resumable Expressions. Single `resumable` keyword, "suspend down."
2015	P0114R0[7]	Kohlhoff	Resumable Expressions (revised).
2015	P0158R0[8]	Allsop et al.	Coroutines belong in a TS. Argued for more time.
2018	P0057R8[9]	Nishanov	Coroutines TS. Frame-erased. The design that shipped.
2018	P0973R0[10]	Romer, Dennett	Coroutines TS Use Cases and Design Issues. Critique: implicit allocation, hidden frame.
2018	P1063R0[11]	Romer, Dennett, Carruth	Core Coroutines. Frame-visible alternative. Expose minimal primitives.
2018	P1134R0[12]	Falco	An Elegant Coroutine Abstraction. Library-only stackless coroutines.
2018	P1342R0[13]	Baker	Unifying Coroutines TS and Core Coroutines. Attempted compromise.

Year	Paper	Author(s)	Design
2018	P1362R0[14]	Nishanov	Incremental Approach: Coroutine TS + Core Coroutines.
2019	P1492R0[15]	Smith, Vandevoorde et al.	Language and implementation impact of coroutine proposals.
2019	P1493R0[16]	Romer, Nishanov, Baker, Mihailov	Coroutines: Use-cases and Trade-offs.
2019	P0912R5[17]	Nishanov	Merge Coroutines TS into C++20. The frame-erased model ships.
2026	P0876R22[18]	Kowalke, Goodspeed	`fiber_context`. Stackful coroutines. Complementary, not competing.
2024	P3203R0[19]	Morgenstern	Implementation defined coroutine extensions. Legalizes `coroutine_handle` specialization.

The committee explored both frame-erased and frame-visible designs. It chose frame-erased. That was the right choice for I/O - type erasure through `coroutine_handle<>` gives type-erased streams, split compilation, and ABI stability. The I/O library compiles once.

Senders need frame visibility. The sender pipeline owns its operation state, knows its size at compile time, and inlines it. A coroutine frame that the sender cannot see, cannot size, and cannot place is a foreign object in the sender's world.

The two needs are not in conflict.

---

6. Three Kinds of Coroutines

C++ has shipped multiple models for parallel execution (`std::execution_policy` and P2300R10[20] senders with `bulk`), multiple models for formatted output (`iostream` and `std::format`), and multiple models for error handling (exceptions and `error_code`). Multiple coroutine models serving different domains is consistent with the committee's practice. Unlike `iostream` and `std::format` , which overlap significantly, the three coroutine models serve non-overlapping domains: stackful coroutines serve deep suspension through coroutine-unaware

APIs, frame-erased coroutines serve type-erased I/O with split compilation and ABI stability, and frame-visible coroutines (if they ever exist) serve compile-time work graphs that need the frame in the type system. Each addresses a use case the others structurally cannot.

6.1 Stackful (fibers)

P0876R22[18] (Kowalke) proposes `fiber_context` - stackful coroutines that maintain a separate stack and support deep suspension. Stackful coroutines address a different set of use cases than stackless coroutines: interacting with coroutine-unaware APIs, deep call chains that suspend at arbitrary depth, and integration with legacy code. The committee has been working on this for over a decade. It is complementary, not competing.

6.2 Stackless, frame-erased (C++20 coroutines)

C++20 coroutines type-erase the frame through `coroutine_handle<>`. The promise type is invisible to the caller. The caller sees only a handle. This is ideal for I/O: type erasure gives type-erased streams, split compilation, and ABI stability. The I/O library compiles once. Transport changes do not break the ABI.

`std::execution` (P2300R10[20]) provides compile-time sender composition, structured concurrency guarantees, and a customization point model that enables heterogeneous dispatch. These are real achievements. The sender model serves GPU dispatch, parallel algorithms, and infrastructure well.

6.3 Stackless, frame-visible

Senders want to see everything in the type system. The frame is part of the operation state. The sender pipeline owns the frame, knows its size, and can inline it.

Romer, Dennett, and Carruth identified this need in P1063R0[11], "Core Coroutines." Their proposal sought to expose minimal coroutine primitives that map directly to the underlying implementation, giving the caller direct access to the coroutine frame in the C++ type system. Baker attempted to unify the two models in P1342R0[13].

If C++ had frame-visible stackless coroutines, senders could invoke IoAwaitables by constructing the frame inline in their operation state. No allocation. The frame is part of the sender's storage.

This is a large language change. This paper names it as the deeper solution and invites exploration. The pragmatic solution follows.

---

7. The Pragmatic Solution: Callback Handles

Even without frame-visible coroutines, the immediate problem is solvable.

What a sender needs is a `coroutine_handle<>` that, when `.resume()` is called, invokes a function on the sender's operation state. The minimum viable representation is two pointers: a function pointer and a data pointer. No frame. No promise. No suspension points. No heap allocation. When the executor calls `.resume()`, it calls the function with the data pointer. When `.destroy()` is called, it is a no-op - the sender owns its own lifetime.

7.1 Symmetric Transfer

The coroutine executor's `dispatch` returns a `coroutine_handle<>` to enable symmetric transfer - the compiler tail-calls the returned handle, avoiding stack buildup in coroutine chains. A sender pipeline is not a coroutine. It does not have an `await_suspend` that the compiler can tail-call out of. Symmetric transfer is a coroutine mechanism that senders do not need.

The sender provides an executor that maps `dispatch` to `post`:

```
struct sender_executor {
    underlying_executor exec_;

    void post(coroutine_handle<> h) const
    {
        exec_.post(h);
    }

    coroutine_handle<>
    dispatch(coroutine_handle<> h) const
    {
        exec_.post(h);
        return noop_coroutine();
    }
};
```

When the reactor completes an I/O operation and calls `executor.dispatch(h)`, the sender's executor posts the handle for later resumption and returns `noop_coroutine()`. The callback handle is resumed from the event loop. No symmetric transfer. No stack buildup. The executor is the policy point - a sender-provided executor that collapses `dispatch` to `post` is a policy choice, not a limitation.

7.2 The Sender Path

A sender pipeline would use a callback handle like this:

```
struct callback_frame {
    void (*resume)(callback_frame*);
    void (*destroy)(callback_frame*);
    void* data;
};

template<class Receiver>
struct read_op_state {
    stream& s_;
    mutable_buffer buf_;
    Receiver rcvr_;
    read_some_awaitable aw_;
    io_env env_;
    callback_frame cb_;

    static void on_resume(
        callback_frame* p) noexcept
    {
        auto* self = static_cast<
            read_op_state*>(p->data);
        auto result =
            self->aw_.await_resume();
        set_value(
            std::move(self->rcvr_),
            result);
    }

    void start() noexcept
    {
        if (aw_.await_ready()) {
            set_value(
                std::move(rcvr_),
                aw_.await_resume());
            return;
        }

        cb_.resume = &on_resume;
        cb_.destroy =
            +[](callback_frame*) {};
        cb_.data = this;

        auto h =
            coroutine_handle<>::from_address(
                &cb_);
        aw_.await_suspend(h, &env_);
```

```
    }
};
```

The `callback_frame` struct has `resume` and `destroy` function pointers at offsets 0 and 1 - matching the coroutine frame layout that all three major compilers use. `coroutine_handle<>::from_address(&cb_)` produces a handle whose `.resume()` calls the function pointer at offset 0. The awaitable cannot tell the difference between this handle and one from a real coroutine.

The `await_ready` check is a no-op for `read_some_awaitable` (which always returns `false`), but a general sender-to-awaitable bridge must respect the full awaitable protocol.

The `io_env` carries the sender's executor. The awaitable submits the operation to the reactor. The reactor calls the executor. The executor calls `.resume()` on the callback handle. The sender's completion function runs. No coroutine frame was allocated.

---

8. Prior Art: P3203R0

P3203R0[19] (Morgenstern, 2024), "Implementation defined coroutine extensions," was presented at Sofia (June 2025) by Niall Douglas on behalf of the author. The paper proposes changing the standard's prohibition on specializing `coroutine_handle` from undefined behavior to implementation defined behavior.

P3203R0[19] documents that all three major compilers (MSVC, GCC, Clang) use the same coroutine frame layout:

```
struct coroutine_frame {
    void (*resume)(coroutine_frame*);
    void (*destroy)(coroutine_frame*);
    promise_type promise;
};
```

The `.resume()` member function of `coroutine_handle<>` calls the function pointer at offset 0. A user-provided struct with the same two-pointer prefix works on every compiler today.

P3203R0[19] identifies the same use case this paper describes: allowing non-coroutine code to provide a `coroutine_handle` that participates in the awaitable protocol. Morgenstern demonstrates this in Boost.Cobalt for Python bindings and stackful coroutine integration.

The wording change in P3203R0[19] is the legal prerequisite for the callback handle approach described in Section 9.

---

9. The Design

The approach requires no `coroutine_handle` specialization, no factory function, and no compiler-generated frame. The user defines a struct whose first two members match the coroutine frame prefix, and `coroutine_handle<>::from_address` does the rest.

9.1 The Callback Frame

The user defines a struct with `resume` and `destroy` function pointers at offsets 0 and 1:

```
struct callback_frame {
    void (*resume)(callback_frame*);
    void (*destroy)(callback_frame*);
    void* data;
};
```

The struct's first two members match the coroutine frame layout documented by P3203R0[19]. `coroutine_handle<>::from_address(&cb)` produces a `coroutine_handle<>` that, when `.resume()` is called, calls the function pointer at offset 0. The awaitable receives this handle. It cannot tell the difference between this handle and one from a real coroutine.

The `destroy` pointer is a no-op - the sender owns its own lifetime. The `data` pointer points back to the operation state. Three pointers. Twenty-four bytes on a 64-bit platform. No heap allocation.

9.2 The Type-Erasure Constraint

Any callback handle must be convertible to `coroutine_handle<void>`. This is non-negotiable. Awaitables accept `coroutine_handle<>`. Executors traffic in `coroutine_handle<>`. The handle is the type-erased boundary between the awaitable and its consumer.

A `coroutine_handle<>` is a pointer to a frame. The storage for that frame must come from somewhere. A factory function cannot conjure storage without allocating - and allocation is the cost this paper eliminates. The compiler cannot rewrite the user's struct into something else. The only zero-allocation path is: the user provides the storage, and `coroutine_handle<>` points directly at it.

This means the standard must mandate the two-pointer prefix layout - `resume` and `destroy` function pointers at offsets 0 and 1 - so that `from_address` on a user-provided struct produces a valid handle. There is no alternative design that avoids allocation.

9.3 What the Standard Must Do

P3203R0[19] formalizes what all three major compilers already do. The coroutine frame layout with two function pointers at the front is not an implementation accident - it is the layout every compiler chose independently, and it is the layout that makes `coroutine_handle<>::from_address` work. The question is whether the committee will mandate it.

The standard could also provide convenience wrappers on top of the mandated layout - a standard `callback_frame` type, a factory function that fills in the pointers, or a named concept that constrains the prefix. These are API sugar. The layout mandate is the prerequisite. Without it, none of them can produce a zero-allocation `coroutine_handle<>` from user-owned storage.

---

10. What This Enables

A callback handle gives senders a zero-allocation entry into the awaitable protocol. The consequences go beyond I/O.

- **The entire awaitable ecosystem opens to senders.** Every IoAwaitable anyone has written - timers, mutexes, channels, semaphores, file I/O, database queries, HTTP clients - becomes consumable by sender pipelines at zero allocation cost. Awaitable authors change nothing. Sender authors gain a new universe of composable operations. The sender ecosystem and the awaitable ecosystem merge.
- **One I/O implementation.** The I/O library implements each operation once as an IoAwaitable. Coroutine users `co_await` it. Sender users invoke `await_suspend` with a callback handle. Both go through the same reactor, the same executor, the same platform implementation.
- **Zero-allocation bridge.** The sender-to-awaitable bridge in P4092R0[5] currently allocates a coroutine frame. With a callback handle, the bridge is two pointers.
- **Type-erased streams.** Streams are type-erasable because the executor is type-erased and the handle is type-erased. Both coroutines and senders see the same stream type.
- **Split compilation.** The I/O library compiles once. It does not need to know whether its caller is a coroutine or a sender pipeline.
- **ABI stability.** Transport changes do not break the ABI. The handle is the boundary.

The I/O library does not need two APIs - one for coroutines and one for senders. It needs one API and two ways to produce a handle.

---

11. Anticipated Objections

Q: This breaks the coroutine abstraction.

A: The reactor already treats the handle as opaque. It calls `.resume()`. It does not inspect the frame, access the promise, or query the suspension point. A callback handle is a `coroutine_handle<>` that does less, not more.

Q: Senders should use their own I/O protocol.

A: They can. This gives them a zero-cost entry into the awaitable protocol when they need I/O. Two protocols for the same socket operation means two implementations to maintain, two sets of bugs, and two surfaces to audit.

Q: Just use a coroutine.

A: That is one allocation per I/O operation. For high-throughput networking - millions of operations per second - that matters. The sender pipeline already has operation state. Allocating a frame to hold nothing the sender needs is overhead.

Q: Frame-visible coroutines are too ambitious.

A: Section 6.3 names frame-visible coroutines as the deeper solution. Section 7 presents the pragmatic fallback. A callback handle solves the immediate problem without a large language change.

Q: C++ should have only one kind of coroutine.

A: C++ has multiple models for parallel execution, formatted output, and error handling. Multiple coroutine models serving different domains is consistent with the committee's practice.

Acknowledgments

The author thanks Gor Nishanov for the C++20 coroutine model and its explicit support for task type diversity; Christopher Kohlhoff for the original continuation framing in P0113R0[21] and for Resumable Expressions, which explored the boundary between type erasure and type visibility before most of the committee was thinking about it; Geoff Romer, James Dennett, and Chandler Carruth for P1063R0[11], which identified the frame-visibility need with precision; Lewis Baker for P1342R0[13], which attempted to unify the two models; Klemens Morgenstern for

P3203R0[19], which removes the legal barrier and documents the ABI reality; Niall Douglas for presenting P3203R0[19] at Sofia; Oliver Kowalke and Nat Goodspeed for a decade of work on stackful coroutines; and Steve Gerbino and Mungo Gill for Capy[4] and Corosio[3] implementation work.

References

1. P4003R1 - "Coroutines for I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026). https://wg21.link/p4003r1
2. P4100R0 - "The Network Endeavor: Coroutine-Native I/O for C++29" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026). https://wg21.link/p4100r0
3. cppalliance/corosio - Coroutine-native networking library. https://github.com/cppalliance/corosio
4. cppalliance/capy - Coroutine I/O primitives library. https://github.com/cppalliance/capy
5. P4092R0 - "Consuming Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026). https://wg21.link/p4092r0
6. N4453 - "Resumable Expressions" (Christopher Kohlhoff, 2015). https://wg21.link/n4453
7. P0114R0 - "Resumable Expressions" (Christopher Kohlhoff, 2015). https://wg21.link/p0114r0
8. P0158R0 - "Coroutines belong in a TS" (Jamie Allsop, Jonathan Wakely, Christopher Kohlhoff, Anthony Williams, Roger Orr, Andy Sawyer, Jonathan Coe, Arash Partow, 2015). https://wg21.link/p0158r0
9. P0057R8 - "Working Draft, C++ Extensions for Coroutines" (Gor Nishanov, 2018). https://wg21.link/p0057r8
10. P0973R0 - "Coroutines TS Use Cases and Design Issues" (Geoff Romer, James Dennett, 2018). https://wg21.link/p0973r0
11. P1063R0 - "Core Coroutines" (Geoff Romer, James Dennett, Chandler Carruth, 2018). https://wg21.link/p1063r0
12. P1134R0 - "An Elegant Coroutine Abstraction" (Vinnie Falco, 2018). https://vinniefalco.github.io/papers/drafts/d1134r0.html
13. P1342R0 - "Unifying Coroutines TS and Core Coroutines" (Lewis Baker, 2018). https://wg21.link/p1342r0
14. P1362R0 - "Incremental Approach: Coroutine TS + Core Coroutines" (Gor Nishanov, 2018). https://wg21.link/p1362r0
15. P1492R0 - "Language and implementation impact of coroutine proposals" (Richard Smith, Daveed Vandevoorde et al., 2019). https://wg21.link/p1492r0

16. P1493R0 - “Coroutines: Use-cases and Trade-offs” (Geoffrey Romer, Gor Nishanov, Lewis Baker, Mihail Mihailov, 2019). https://wg21.link/p1493r0

17. P0912R5 - “Merge Coroutines TS into C++20 working draft” (Gor Nishanov, 2019). https://wg21.link/p0912r5

18. P0876R22 - “fiber_context - fibers without scheduler” (Oliver Kowalke, Nat Goodspeed, 2026). https://wg21.link/p0876r22

19. P3203R0 - “Implementation defined coroutine extensions” (Klemens Morgenstern, 2024). https://wg21.link/p3203r0

20. P2300R10 - “std::execution” (Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 2024). https://wg21.link/p2300r10

21. P0113R0 - “Executors and Asynchronous Operations, Revision 2” (Christopher Kohlhoff, 2015). https://wg21.link/p0113r0

22. P3552R3 - “Add a Coroutine Task Type” (Dietmar Kühl, Maikel Nadolski, 2025). https://wg21.link/p3552r3