

# async-resource - aka async-RAII

## Draft Proposal

Document #: D0000R0  
Date: 2023-04-09  
Project: Programming Language C++  
Audience: LEWG Library Evolution  
Reply-to: Kirk Shoop  
<[kirk.shoop@gmail.com](mailto:kirk.shoop@gmail.com)>

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Motivation</b>	<b>2</b>
<b>3</b>	<b>Design</b>	<b>2</b>
3.1	What is an <i>async-resource</i> ? . . . . .	2
3.2	What are the requirements for an <i>async-resource</i> ? . . . . .	2
3.3	What is the concept that an <i>async-resource</i> must satisfy? . . . . .	3
3.3.1	run(), open(), and close() . . . . .	3
3.3.2	run() -> <i>sequence-sender</i> . . . . .	4
3.4	How do these CPOs compose to provide an async resource? . . . . .	5
3.4.1	run(), open(), and close() . . . . .	5
3.4.2	run() -> <i>sequence-sender</i> . . . . .	5
3.5	How do you use an <i>async-resource</i> ? . . . . .	6
<b>4</b>	<b>Why this design?</b>	<b>8</b>
4.1	run(), open(), and close() . . . . .	8
4.1.1	The open cpo . . . . .	8
4.1.2	The run cpo . . . . .	9
4.1.3	The close cpo . . . . .	9
4.1.4	Composition . . . . .	9
4.1.5	Structure . . . . .	9
4.2	run() -> <i>sequence-sender</i> . . . . .	9
4.2.1	The run cpo . . . . .	9
4.2.2	Composition . . . . .	9
4.2.3	Structure . . . . .	9
<b>5</b>	<b>Appendices</b>	<b>10</b>
5.1	Rejected Options: . . . . .	10
5.2	The order of operations when using an async-resource . . . . .	10
5.2.1	One resource . . . . .	10
5.2.2	N resources . . . . .	13
5.2.3	use_resource . . . . .	14
<b>6</b>	<b>References</b>	<b>14</b>

# 1 Introduction

A major precept of [P2300R6] is structured concurrency. The `let_value()` algorithm provides stable storage for values produced by the input sender. What is missing is a way to attach an object to a sender expression such that the object is opened before nested senders start and closed after nested senders complete. This is commonly done in async programs using `std::shared_ptr` to implement ad-hoc garbage collection. Using garbage collection for this purpose removes structure from the code, because the shared ownership is unstructured and allows objects to escape the original scope in which they were created.

This paper describes concepts that would be used to create and cleanup an object within a scope that will contain all senders composed into that scope. These senders have access to a non-owning handle to the resource that is safe to use. These senders can be running on any execution context. An *async-resource* object has only one concern, which is to open before any nested senders start and to close after any nested senders complete. In order to be useful within other asynchronous scopes, the object must not have any blocking functions.

An *async-resource* can be thought of as an *async-RAII* object.

## 2 Motivation

It is becoming apparent that all the sender/receiver features are language features being implemented in library.

Sender/Receiver itself is the implementation of an *async-function*. It supports values and errors and cancellation. It also requires manual memory management in implementations because *async-resources* do not fit inside any single block in the language.

The language has a set of rules that are applied in a code-block to describe when construction and destruction occur, and when values are valid. The language implements those rules.

This paper describes how to implement rules for the construction and destruction of async objects in the library. This paper describes structured construction and destruction of objects in terms of sender/receiver. The `use_resources()` algorithm described in this paper is a library implementation of an async code-block. The `use_resources()` algorithm is somewhat analogous to the `using` keyword in some languages.

## 3 Design

### 3.1 What is an *async-resource*?

An *async-resource* is an object with state that is valid for all the *async-operations* that are nested within the *async\_resource* expression.

Examples include:

- thread
- thread-pool
- io-pool
- buffer-pool
- mutex
- file
- socket
- async-scope

### 3.2 What are the requirements for an *async-resource*?

#### **async construction**

Some objects have *async-functions* to establish a connection, open a file, etc..

The design must allow for *async-functions* to be used during construction - without blocking any threads (this means that C++ constructors are unable to meet this requirement)

### async destruction

Some objects have *async-functions* to teardown a connection, flush a file, etc..

The design must allow for *async-functions* to be used during destruction - without blocking any threads (this means that C++ destructors are unable to meet this requirement)

### structured and correct-by-construction

These are derived from the rules in the language.

The object will not be available until it has been constructed. The object will not be available until the object is contained in an *async-function*. Failures of *async-construction* and *async-destruction* will complete to the containing *async-function* with the error. The object will always complete cleanup before completing to the containing *async-function*. Acquiring an object is a no-fail *async-function*.

### composition

Multiple object will be available at the same time without nesting. Composition will support concurrent *async-construction* of multiple objects. Composition will support concurrent *async-destruction* of multiple objects. Dependencies between objects will be expressed by nesting. Concurrency between objects will be expressed by algorithms like `when_all` and `use_resources()`.

## 3.3 What is the concept that an *async-resource* must satisfy?

There are two options for defining the *async-resource* concept that are described here. Either one will satisfy the requirements.

### 3.3.1 `run()`, `open()`, and `close()`

This option uses three new CPOs in two concepts that describe the lifetime of an *async-resource*.

This option depends only on [\[P2300R6\]](#)

#### 3.3.1.1 `async_resource` Concept:

An *async-resource* stores the state used to open and run a resource.

```
/// @brief the async-resource concept definition
template<class _T>
concept async_resource =
    requires (const _T& __t_clv, _T& __t_lv){
        open_t{__t_clv};
        run_t{__t_lv};
    };

using open_t = /*implementation-defined*/;
/// @brief the open() cpo provides a sender that will complete with a async-resource-token.
/// @details The async-resource-token will be valid until the sender provided
/// by close() is started.
/// The sender provided by open() will complete after the sender provided by
/// run() has completed any async-operation needed to open the resource.
/// The sender provided by open() will not fail.
/// @param async_resource&
/// @returns sender<resource-token>
inline static constexpr open_t open{};
```

```

using run_t = /*implementation-defined*/;
/// @brief the run() cpo provides a sender-of-void.
/// @details The sender provided by run() will start any async-operation
/// needed to open the resource and when all those operations complete
/// then run() will complete the sender provided by open().
/// The sender provided by run() will complete after the sender provided
/// by close() is started and all the async-operation needed to close
/// the async-resource complete and the sender provided by close() is completed.
/// @param async-resource&
/// @returns sender<>
inline static constexpr run_t run{};

```

### 3.3.1.2 async\_resource\_token Concept:

An *async-resource-token* is a non-owning handle to the resource that is provided after the resource has been opened.

The token must be used to close the resource once the resource has been opened.

```

/// @brief the async-resource-token concept definition
template<class _T>
concept async_resource_token =
    requires (const _T& __t_clv){
        close_t{}(__t_clv);
    };

using close_t = /*implementation-defined*/;
/// @brief the close() cpo provides a sender-of-void.
/// @details The sender provided by close() will trigger the sender provided
/// by run() to begin any async-operation needed to close the resource and
/// will complete when all the async-operation complete.
/// The sender provided by close() will not fail.
/// @param async-resource-token&
/// @returns sender<>
inline static constexpr close_t close{};

```

### 3.3.2 run() -> sequence-sender

This option uses one new CPO in one concept that describes the lifetime of an *async-resource*.

This option depends on a paper that adds *sequence-sender* on top of [\[P2300R6\]](#)

```

/// @brief the async-resource concept definition
template<class _T>
concept async_resource =
    requires (_T& __t){
        run_t{}(__t);
    };

using run_t = /*implementation-defined*/;
/// @brief the run() cpo provides a sequence-sender-of-token.
/// @details The sequence-sender provided by run() will produce
/// a run-operation. When the run-operation is started, it will start
/// any async-operation that are needed to open the async-resource.
/// After all those async-operation complete, the run-operation
/// will produce an async-resource-token as the only item in the

```

```

/// sequence.
/// After the sender-expression for the async-resource-token item
/// completes, the run-operation will start any async-operation
/// that are needed to close the async-resource.
/// After all those async-operation complete, the run-operation
/// will complete.
/// @param async-resource&
/// @returns sequence-sender<async-resource-token>
inline static constexpr run_t run{};

```

### 3.4 How do these CPOs compose to provide an async resource?

#### 3.4.1 run(), open(), and close()

The senders returned from `open()` and `run()` produce the *open-operation* and the *run-operation*.

After both of the open and run operations are started, the *run-operation* starts any *async-operation* that are needed to initialize the *async-resource*.

After all those *async-operation* complete, the *open-operation* will complete with the *async-resource-token*.

The *run-operation*, will complete after the following steps:

- the runtime has entered the `main()` function (requires a signal from the runtime)
- any *async-operation* needed to open the *async-resource* has completed
- at this point, the *async-resource* lifetime begins**
- the *open-operation* completes with the *async-resource-token*
- a stop condition is encountered
  - a `stop_token`, provided by the environment of the *open-operation*, is in the `stop_requested()` state
- OR**
- the *close-operation*, produced by the sender returned from `close()`, has been started
- OR**
- the runtime has exited the `main()` function (this requires a signal from the runtime)
- at this point, the *async-resource* lifetime ends**
- any *async-operation* needed to close the *async-resource* have completed
- any *close-operation* completes

#### 3.4.2 run() -> sequence-sender

The *sequence-sender* returned from `run()` produces a *run-operation*.

After the *run-operation* is started, it starts any *async-operation* that are needed to initialize the *async-resource*.

After all those *async-operation* complete, the *run-operation* will emit the *async-resource-token*.

The *run-operation*, will complete after the following steps:

- the runtime has entered the `main()` function (this requires a signal from the runtime)
- any *async-operation* needed to open the *async-resource* has completed
- at this point, the *async-resource* lifetime begins**
- the *async-resource-token* item is emitted

- a stop condition is encountered
    - a `stop_token`, provided by the environment of the *open-operation*, is in the `stop_requested()` state
  - OR
  - the *token-operation*, produced by the sender expression for the *async-resource-token*, has completed
  - OR
  - the runtime has exited the `main()` function (this requires a signal from the runtime)
- at this point, the *async-resource* lifetime ends
- any *async-operation* needed to close the *async-resource* have completed

### 3.5 How do you use an *async-resource*?

Here is a basic example of composing resources using this pattern:

Table 1: basic example

run(), open(), and close()	run() -> <i>sequence-sender</i>
<pre> int main() {   exec::static_thread_pool ctx{1};   exec::counting_scope context;   auto use = ex::when_all(     exec::open(ctx),     exec::open(context))     ex::let_value([&amp;](     ex::scheduler auto sch,     exec::async_scope auto scope){     // async-resource lifetime begins      sender auto begin =       ex::schedule(sch);      sender auto printVoid =       ex::then(begin,         []()noexcept { printf("void\n"); })      exec::spawn(scope, printVoid);      // async-resource lifetime ends     // when close starts     return ex::when_all(       exec::close(sch), exec::close(scope))   });   ex::sync_wait(ex::when_all(     use,     exec::run(ctx),     exec::run(context))); } </pre>	<pre> int main() {   exec::static_thread_pool ctx{1};   exec::counting_scope context;   auto use = ex::zip(     exec::run(ctx),     exec::run(context))     ex::let_value_each([&amp;](     ex::scheduler auto sch,     exec::async_scope auto scope){     // async-resource lifetime begins      sender auto begin =       ex::schedule(sch);      sender auto printVoid =       ex::then(begin,         []()noexcept { printf("void\n"); }));      exec::spawn(scope, printVoid);      // async-resource lifetime ends     // when printVoid completes     return printVoid;   });   ex::sync_wait(use); } </pre>

This pattern correctly scopes the use of the *async-resource* and composes the open, run, and close *async-operations* correctly.

It is possible to compose multiple *async-resources* into the same block or expression.

Table 2: multiple *async-resource* composition example

run(), open(), and close()	run() -> <i>sequence-sender</i>
<pre> stop_source stp; static_thread_pool ctx{1}; async_allocator aa; counting_scope as; async_socket askt; split spl;  auto use = when_all(   open(stp), open(ctx), open(aa),   open(as), open(askt), open(spl))   let_value([](   stop_token stop, auto sched, auto alloc,   auto scope, auto sock, auto spl){   auto env = make_env(empty_env{},     with(get_stop_token, stop),     with(get_scheduler, sched),     with(get_async_allocator, alloc),     with(get_async_scope, scope));   auto [input, output] = spl;   auto producer = produce(input,     async_read_some(sock, MAX_DATA_SIZE));   for (int i = 0; i &lt; 4; ++i) {     spawn(scope,       with_env(env,         on(sched, consume(output))));   }   auto close = when_all(     close(stop), close(sched), close(alloc),     close(scope), close(sock), close(spl));   return finally(     with_env(env,       when_all(producer,         nest(consume(output))),       close);   });  std::this_thread::sync_wait(   when_all(     use,     run(stp), run(ctx), run(aa),     run(as), run(askt), run(spl))); </pre>	<pre> stop_source stp; static_thread_pool ctx{1}; async_allocator aa; counting_scope as; async_socket askt; split spl;  auto use = zip(   run(stp), run(ctx), run(aa),   run(as), run(askt), run(spl))   let_value_each([](   stop_token stop, auto sched, auto alloc,   auto scope, auto sock, auto spl){   auto env = make_env(empty_env{},     with(get_stop_token, stop),     with(get_scheduler, sched),     with(get_async_allocator, alloc),     with(get_async_scope, scope));   auto [input, output] = spl;   auto producer = produce(input,     async_read_some(sock, MAX_DATA_SIZE));   for (int i = 0; i &lt; 4; ++i) {     spawn(scope,       with_env(env,         on(sched, consume(output))));   }   return with_env(env,     when_all(producer,       nest(consume(output)))); });  std::this_thread::sync_wait(use); </pre>

Both of these options fall into a pattern as well. This pattern can be placed in an algorithm. The `use_resources` algorithm changes the above example to look like:

```

std::this_thread::sync_wait(
  use_resources([](
    auto stop, auto sched,

```

```

auto alloc, auto scope,
auto sock, auto split){
    auto [input, output] = split;
    auto env = make_env(empty_env{}),
        with(get_stop_token, stop),
        with(get_scheduler, sched),
        with(get_async_allocator, alloc),
        with(get_async_scope, scope));
    auto producer = produce(input, async_read_some(sock, MAX_DATA_SIZE));
    for (int i = 0; i < 4; ++i) {
        spawn(scope, with_env(env, on(sched, consume(output))));
    }
    return with_env(env, when_all(producer, nest(consume(output))));
},
make_deferred<stop_source>(),
make_deferred<static_thread_pool>(1),
make_deferred<async_allocator>(),
make_deferred<counting_scope>(),
make_deferred<async_socket>(),
make_deferred<split>());

```

## 4 Why this design?

There have been many, many design options explored for the `async_scope`. We had a few variations of a single object with methods, then two objects with methods. It was at this point that a pattern began to form across `stop_source/stop_token`, `execution-context/scheduler`, `async-scope/async-scope-token`.

The patterns model RAII for objects used by *async-functions*.

In C++, RAII works by attaching the constructor and destructor to a block of code in a function. This pattern uses `run()` to represent the block that the object is contained in. The `run()` *async-function* satisfies the structure requirement (only nested *async-functions* use the object) and satisfies the correct-by-construction requirement (the object is not available until the `run()` *async-function* is started).

### 4.1 `run()`, `open()`, and `close()`

The `run/open/close` option uses an *async-function* to store the object (`run()`), another *async-function* to access the object (`open()`), and a final *async-function* to stop the object (`close()`). `open()` is not a constructor and `close()` is not a destructor. `open()` and `close()` are signals. `open()` signals that the object is ready to use and `close()` signals that the object is no longer needed. Any errors encountered in the object cause the `run()` *async-function* to complete with the error, but only after completing any cleanup *async-functions* needed.

#### 4.1.1 The `open cpo`

Existing resources, like `run_loop` and `stop_source` have a method that returns a token. This does not provide for any asynchronous operations that are required before a token is valid.

`open` is an operation that provides the token only after it is valid.

`open` completes when the token is valid. All operations using the token must be nested within the `open` operation.

The receiver passed to the `open` operation is used to query services as needed (allocator, scheduler, stop-token, etc..)



#### 4.1.2 The run cpo

`open` may start before the resource is constructed and completes when the token is valid. `run` starts before the resource is constructed and completes after the token is closed. The `run` operation represents the entire resource. The `run` operation includes construction, open, resource usage, and close. `run` is the owner of the resource, `open` is the token accessor, `close` is the signal to stop the resource.

`open` cannot represent the resource because it will complete before the resource reaches the closed state.

`close` cannot represent the resource because it cannot begin until after `open` has completed.

#### 4.1.3 The close cpo

`close` is used to start any operations that stop the resource and invalidate the token. After the `close` operation completes the `run` operation runs the destructor of the resource and completes.

#### 4.1.4 Composition

The `open` and `close` cpos are not the only way to compose the token into a sender expression.

The benefit provided by the `open` and `close` operations is that a `when_all` of multiple `opens` and a `when_all` of multiple `closes` can be used to access multiple tokens without nesting each token inside the prev.

#### 4.1.5 Structure

The `run`, `open`, and `close` operations provide the token in a structured manner. The token is not available until the `run` operation has started and the `open` operation has completed. The `run` will not complete until the `close` operation is started. This structure makes using the resource correct-by-construction. There is no resource until the `run` and `open` operations are started. The `run` operation will not complete until the `close` operation completes.

Ordering of constructors and destructors is expressed by nesting resources explicitly. Using `when_all` to compose resources concurrently requires that the resources are independent because there is no token to the resource available until the `when_all` completes.

### 4.2 `run()` -> *sequence-sender*

The `run/sequence-sender` option uses an *async-function* to store the object (`run()`). The sequence produces one item that provides access to the object once it is ready to use. When the item has been consumed, `run()` will cleanup the object and complete. Any errors encountered in the object cause the `run()` *async-function* to complete with the error, but only after completing any cleanup *async-functions* needed.

#### 4.2.1 The run cpo

`run` starts before the resource is constructed and completes after all nested *async-functions* have completed and the object has finished any cleanup *async-function*. The `run` operation represents the entire resource. The `run` operation includes construction, resource usage, and destruction. `run` is the owner of the resource.

#### 4.2.2 Composition

Composition is easily achieved using the `zip()` algorithm and the `let_value_each()` algorithm.

#### 4.2.3 Structure

The `run async-function` provides the object in a structured manner. The object is not available until the `run` operation has started. The `run()` *async-function* will not complete until the object is no longer in use. This structure makes using the resource correct-by-construction. There is no resource until the `run()` *async-function* is started. The `run()` *async-function* completes after all nested *async-functions* have completed and the object has finished any cleanup *async-function*.

Ordering of constructors and destructors is expressed by nesting resources explicitly. Using the `zip()` algorithm to compose resources concurrently requires that the resources are independent because there is no token to the resource available until the `zip()` algorithm completes.

## 5 Appendices

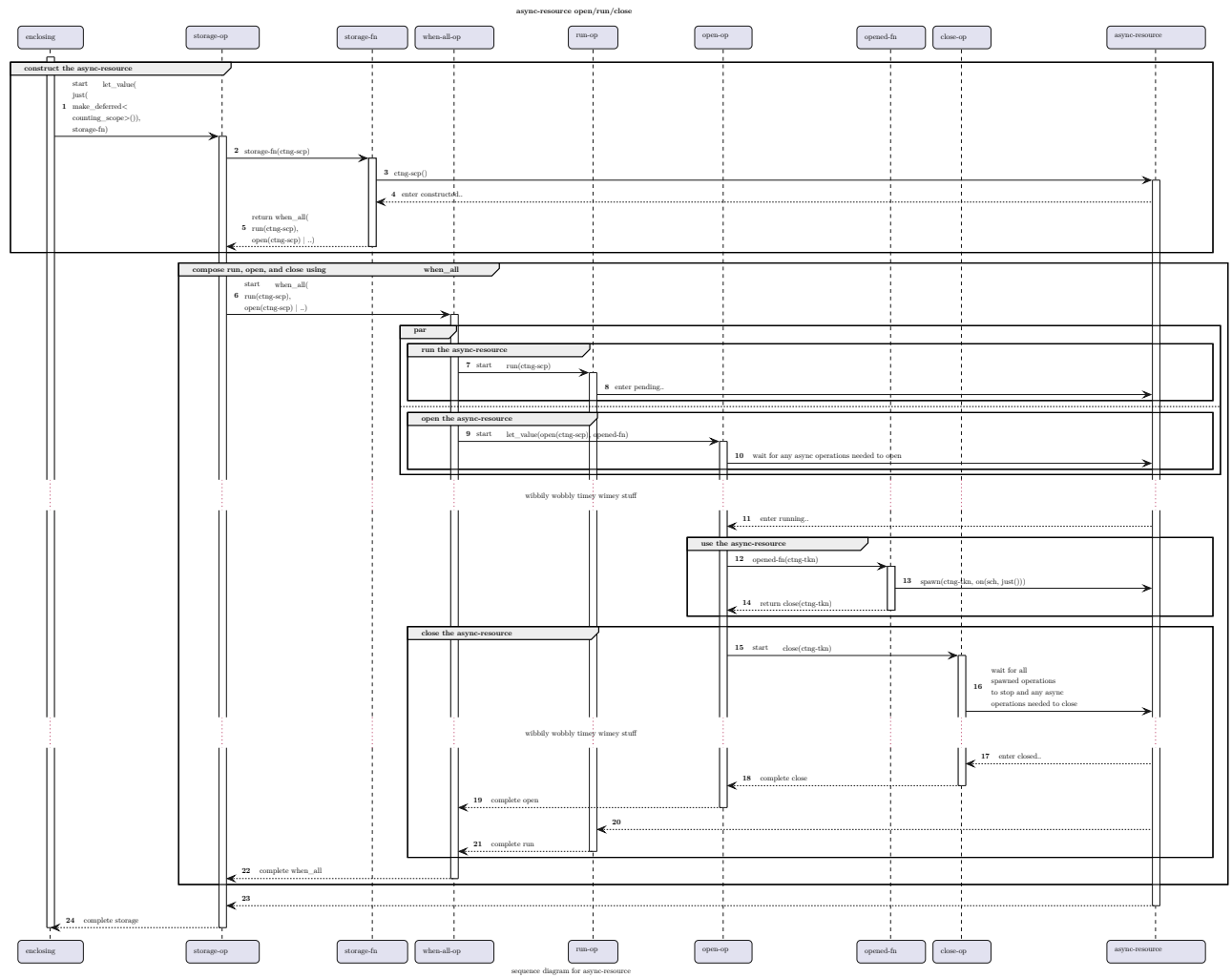
### 5.1 Rejected Options:

- `join()` -> sender:
  - `join()` returns a sender that completes after running any pending *async-operations* followed by running any *async-operations* needed to close the *async-resource*.
  - This option is challenging because it is not correct by construction:
    - Imposes that all users remember to compose `join()` into an `async_scope` and prevent the destructor from running until `join()` completes.
    - Provides no way to run *async-operations* to open the *async-resource*.
- `run((token)->sender)` -> sender:
  - The sender returned from `run()` will complete after the following steps:
    - *async-operations* to open the *async-resource*  
\*\* at this point, *async-resource* lifetime begins \*\*
    - an *async-resource-token* is passed to the provided function
    - the sender returned from the provided function  
\*\* at this point, *async-resource* lifetime ends \*\*
    - any *async-operations* needed to close the *async-resource*
  - This option scopes the use of the *async-resource* and composes the open and close *async-operations* correctly.
  - It is hard to compose multiple *async-resources* into the same block or expression (requires nesting calls to `run()` for each *async-resource*, which also sequences the open and close for each *async-resource*).

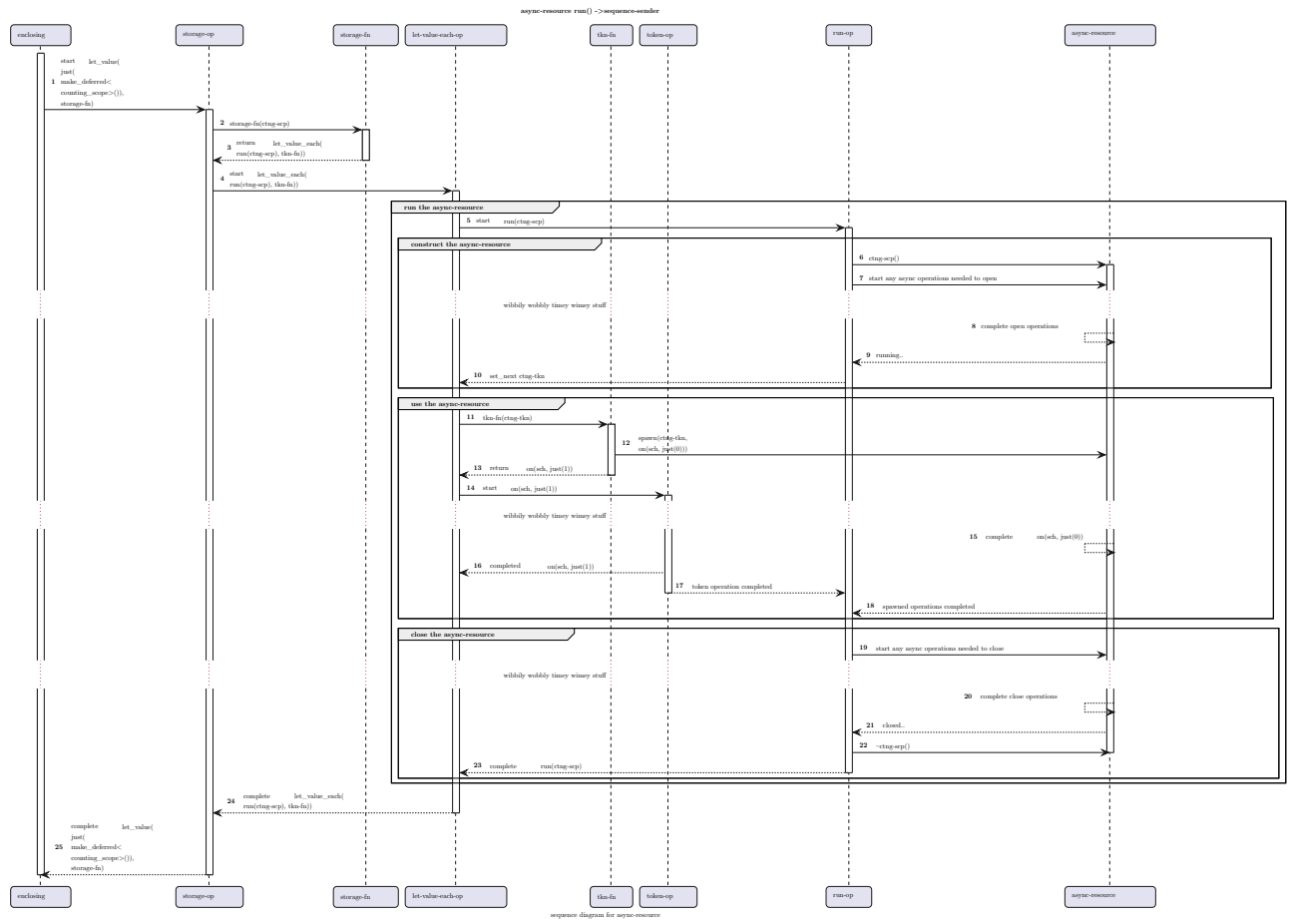
### 5.2 The order of operations when using an `async-resource`

#### 5.2.1 One resource

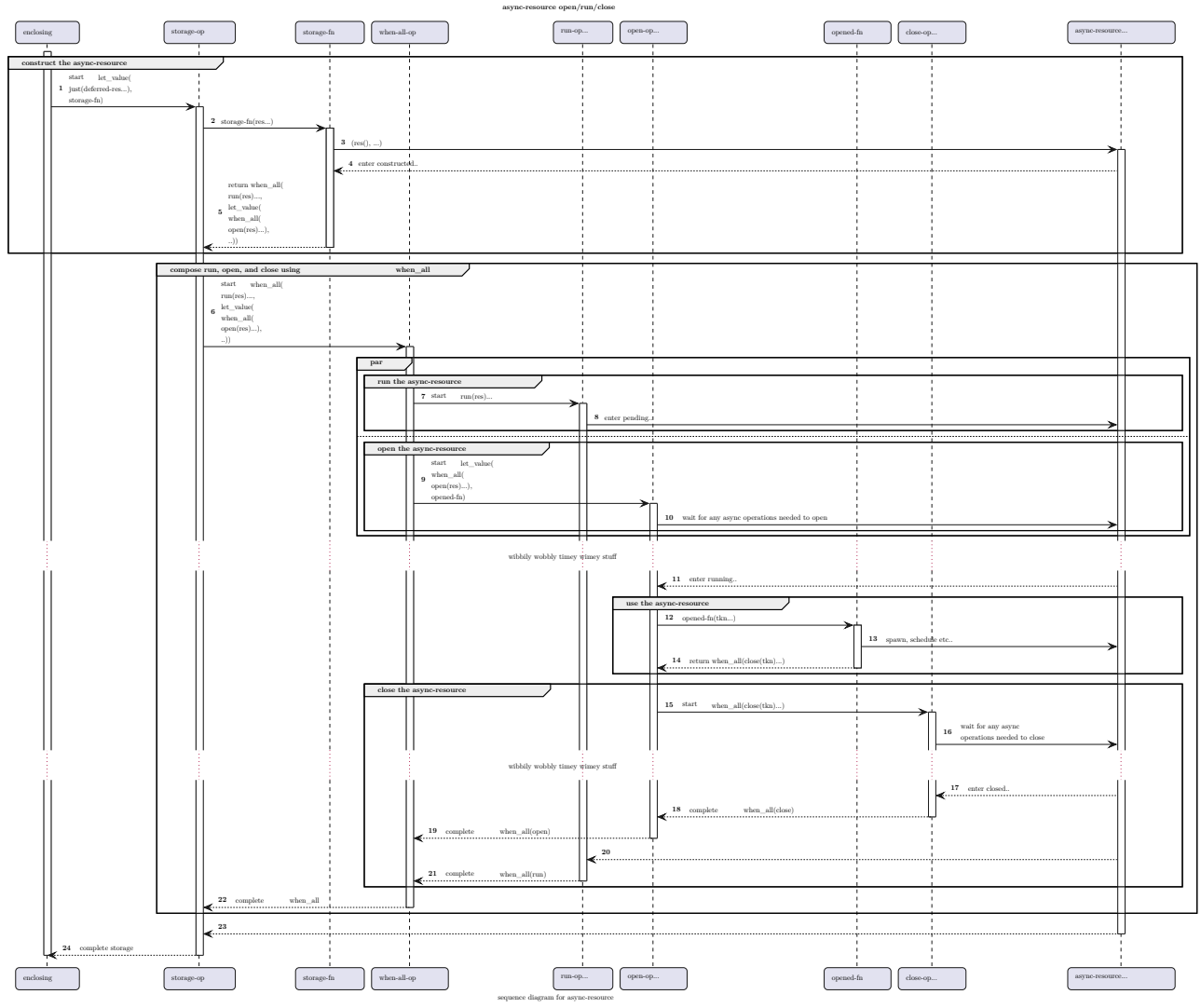
##### 5.2.1.1 `run()`, `open()`, and `close()`



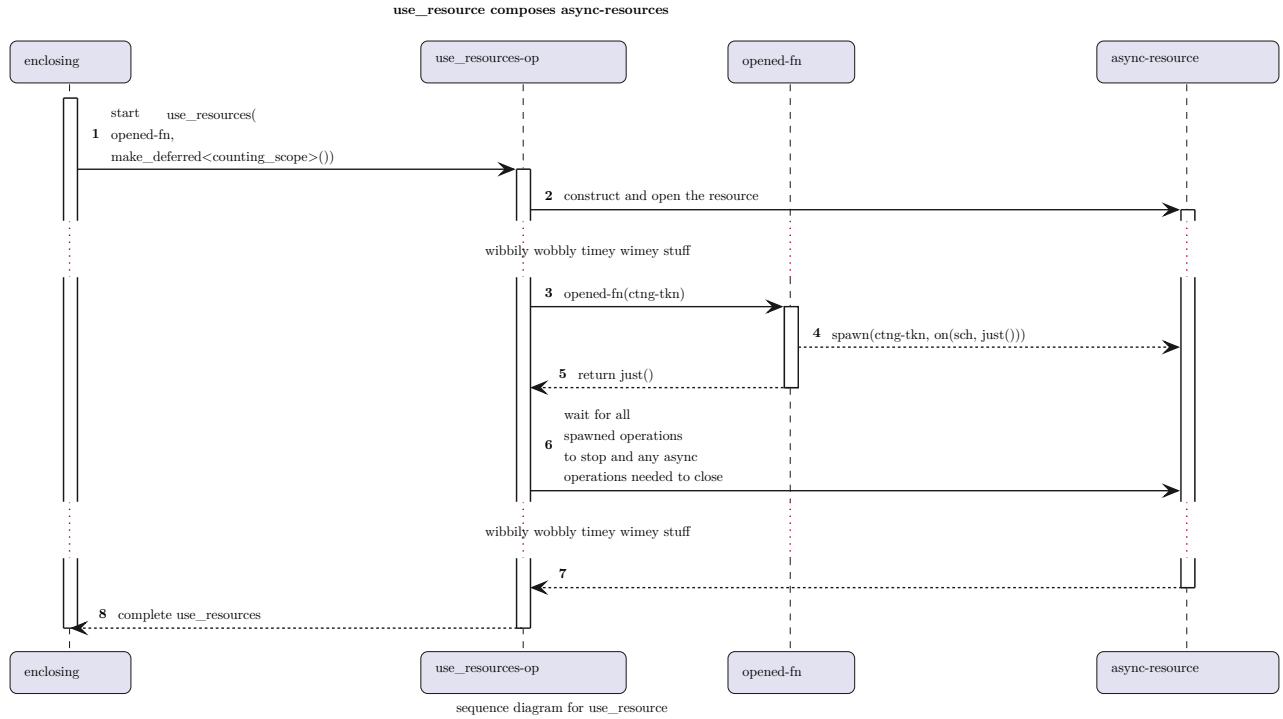
### 5.2.1.2 run() -> sequence-sender



## 5.2.2 N resources



### 5.2.3 use\_resource



## 6 References

- [P2300R6] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2023-01-19. ‘std::execution’.  
<https://wg21.link/p2300r6>