Document number: P3813R0
Date: 2025-09-10
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

`execution::task::valueless()`

Abstract
This paper proposes to add a `valueless` member function to `execution::task`.

Tony Table

Before	Proposed

`task<int> t = …;`
`//is it safe to use t?`
`auto res = sync_wait(t);`
	`task<int> t = …;`
	`contract_assert(not t.valueless());`
	`auto res = sync_wait(t);`

Revisions
R0: Initial version

Motivation
Types with *empty states* aren't extraordinary in C++, but types with *unrecoverable empty states* are. The former category includes pointers, `optional`, `variant`[2] and `string`. The latter category is defined by `indirect`, `polymorphic`, `generator` and `execution::task`.

In general objects in an *unrecoverable empty state* can only destroyed or assigned to, as all their other operations have a precondition on the object not being in this state. Even though this state shouldn't be observed in a valid program, we've added `valueless_after_move` to `indirect` and `polymorphic` to guard against it.

There is currently no way to check for this state in `generator` or `task`. We originally wanted to propose an API for both of these types, but after examining existing implementations, we've decided to move extensions to `generator` into a future paper.

Design Space
A `task` can be in the *unrecoverable empty state* only if it was involved in a move operation, or `connect` was previously called. As there is little value in differentiating these two reasons for the *empty state*, we are proposing only one member function:

```
bool valueless() const noexcept;
```

We don't propose a `task::state::valueless` as contrary to `task`, `task::state` is not user-facing and will almost exclusively be used in the context of executors.

---

[1] RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

[2] `variant` has an extraordinary, yet *recoverable empty state*.

Impact on the Standard

This proposal is a pure library addition. Existing standard library classes are modified in a non-ABI-breaking way.

Proposed Wording

Wording is relative to [N5014]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[version.syn]

```
#define __cpp_lib_task YYYYYMM~~202506~~L // also in <execution>
```

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]

[exec.task]

??.??.?.? Class template `task` **[task.class]**

```
namespace std::execution {
  template<class T, class Environment>
  class task {
    …
    ~task();

    bool valueless() const noexcept;

    template<receiver Rcvr>
      state<Rcvr> connect(Rcvr&& rcvr);
  };
}
```

[task.members]

??.??.?.? `task` members **[task.members]**

```
~task();
```

2 *Effects*: Equivalent to:

```
if (handle)
  handle.destroy();
```

```
bool valueless() const noexcept;
```

3 *Returns*: `bool(handle)` is `false`.

```
template<receiver Rcvr>
  state<Rcvr> connect(Rcvr&& recv);
```

~~3~~4 *Preconditions*: ~~`bool(handle)`~~`valueless()` is ~~`true`~~`false`.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work.