1 Thread support library [thread]

1.1 General [jthread.general]

1 The following subclauses describe components to create and manage threads (??), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 1.

Table 1 — Thread support library summary

	Subclause	Header(s)
1.2	Requirements	
1.3	Threads	`<thread>`
1.4	Interrupt Tokens	`<interrupt_token>`
1.5	Joining Threads	`<jthread>`

??	Mutual exclusion	`<mutex>`
		`<shared_mutex>`

1.6	Condition variables	`<condition_variable>`
??	Futures	`<future>`

1.2 Requirements [thread.req]

1.3 Threads [thread.threads]

1.4 Interrupt Tokens [thread.interrupt_token]

1 1.4 describes components that can be used to asynchonously signal an interrupt. The interrupt can only be signaled once.

1.4.1 Header `<interrupt_token>` synopsis [thread.interrupt_token.syn]

```
namespace std {
  // 1.4.2 class interrupted
  class interrupted;

  // 1.4.1 class interrupt_token
  class interrupt_token;
}
```

1.4.2 Class `interrupted` [interrupted]

```
namespace std {
  class interrupted {
  public:
    explicit interrupted() noexcept;
    explicit interrupted(const interrupted&) noexcept;
    interrupted& operator=(const interrupted&) noexcept;
    const char* what() const noexcept;
  };
}
```

1 The class `interrupted` defines the type of objects thrown as exceptions by C++ standard library components, and certain expressions, to report a signaled interrupt. [*Note*: This class is not derived from class `exception`. — *end note*]

`interrupted() noexcept;`

2 *Effects:* Constructs an object of class `interrupted`.

```
interrupted(const interrupted&) noexcept;
interrupted& operator=(const interrupted&) noexcept;
```

3 *Effects:* Copies an object of class `interrupted`.

4 *Ensures:* If `*this` and `rhs` both have dynamic type `interrupted` then the value of the expression `strcmp(what(), rhs.what())` shall equal 0.

`const char* what() const noexcept;`

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**). The return value remains valid until the exception object from which it is obtained is destroyed or a non-`const` member function of the exception object is called.

1.4.2.1 Class `interrupt_token` [interrupt_token]

1 The class `interrupt_token` implements semantics of shared ownership of a token to signal interrupts. An interrupt can only be signaled once. All owners can signal an interrupt, provided the token is valid. All owners can check whether an interrupt was signaled. The last remaining owner of the token is responsible for destroying the object.

```
namespace std {
  class interrupt_token {
  public:
    explicit interrupt_token() noexcept;
    explicit interrupt_token(bool initial_state);

    interrupt_token(const interrupt_token&) noexcept;
    interrupt_token(interrupt_token&&) noexcept;
    interrupt_token& operator=(const interrupt_token&) noexcept;
    interrupt_token& operator=(interrupt_token&&) noexcept;
    void swap(interrupt_token&) noexcept;
```

```
      // 1.4.2.5 interrupt handling:
      bool valid() const noexcept;
      bool is_interrupted() const noexcept;
      bool interrupt();
      void throw_if_interrupted() const;
    }
  }

  bool operator== (const interrupt_token& lhs, const interrupt_token& rhs);
  bool operator!= (const interrupt_token& lhs, const interrupt_token& rhs);
```

Calls to `interrupt()`, `is_interrupted()`, and `throw_if_interrupted()` are atomic operations(6.8.2.1p3 **??**) on an atomic object contained in the interrupt_token. Hence concurrent calls to these functions do not introduce data races. A call to `interrupt()` synchronizes with any call to `interrupt()`, `is_interrupted()`, or `throw_if_interrupted()` that observes the interrupt (and hence returns `true` or throws).

1.4.2.2 `interrupt_token` constructors [interrupt_token.constr]

`interrupt_token() noexcept;`

1 *Effects:* Constructs a new `interrupt_token` object that can't be used to signal interrupts.

2 *Ensures:* `valid() == false`.

`interrupt_token(bool initial_state) noexcept;`

3 *Effects:* Constructs a new `interrupt_token` object that can signal interrupts.

4 *Ensures:* `valid() == true` and `is_interrupted() == initial_state`.

`interrupt_token(const interrupt_token& it) noexcept;`

5 *Effects:* If `it` is not valid, constructs an `interrupt_token` object that is not valid; otherwise, constructs an `interrupt_token` that shares the ownership of the interrupt signal with `it`.

6 *Ensures:* `valid() == true`.

`interrupt_token(interrupt_token&& it) noexcept;`

7 *Effects:* Move constructs an object of type `interrupt_token` from `it`.

8 *Ensures:* `*this` shall contain the old value of `it`. `it.valid() == false`.

1.4.2.3 `interrupt_token` assign [interrupt_token.assign]

`interrupt_token& operator=(const interrupt_token& it) noexcept;`

1 *Effects:* Equivalent to: `interrupt_token(it).swap(*this);`

2 *Returns:* `*this`.

`interrupt_token& operator=(interrupt_token&& it) noexcept;`

3 *Effects:* Equivalent to: `interrupt_token(std::move(it)).swap(*this);`

4 *Returns:* `*this`.

1.4.2.4 `interrupt_token` swap [interrupt_token.swap]

`void swap(interrupt_token& it) noexcept;`

1 *Effects:* Swaps the state of `*this` and `it`.

1.4.2.5 `interrupt_token` members [interrupt_token.mem]

`bool valid() const noexcept;`

1 *Returns:* `true` if the interrupt token can be used to signal interrupts.

`bool is_interrupted() const noexcept;`

2 *Returns:* `true` if initialized with `true` or initialized with `false` and `interrupt()` was called by one of the owners.

`bool interrupt();`

3 *Requires:* `valid() == true`

4 *Effects:* `is_interrupted() == true`.

5 *Returns:* The value of `is_interrupted()` prior to the call.

`void throw_if_interrupted() const;`

6 *Effects:* Equivalent to:

```
if (is_interrupted())
  throw interrupted();
```

1.4.2.6 interrupt_token comparisons [interrupt_token.cmp]

`bool operator== (const interrupt_token& lhs, const interrupt_token& rhs);`

1 *Returns:* `!lhs.valid() && !rhs.valid()` or whether `lhs` and `rhs` refer to the same interrupt token (copied or moved from the same initial object).

`bool operator!= (const interrupt_token& lhs, const interrupt_token& rhs);`

2 *Returns:* `!(lhs==rhs)`.

1.5 Joining Threads [thread.jthreads]

[*Editorial note:* This color signals differences to class `std::thread`.]

1 1.5 describes components that can be used to create and manage threads with the ability to signal interrupts to cooperatively cancel the running thread.

1.5.1 Header `<jthread>` synopsis [thread.jthread.syn]

```
#include <interrupt_token>

namespace std {
  // 1.5.2 class jthread
  class jthread;

  void swap(jthread& x, jthread& y) noexcept;

  // 1.5.3 class jthread
  namespace this_thread {
    static bool is_interrupted() noexcept;
    static void throw_if_interrupted();
    static interrupt_token get_interrupt_token() noexcept;
    static interrupt_token exchange_interrupt_token(const interrupt_token&) noexcept;
  }
}
```

1.5.2 Class `jthread` [thread.jthread.class]

1 The class `jthread` provides a mechanism on top of class `thread` (??), which the additional ability to signal interrupts and let the destructor `join()` if still `joinable()`. The functionality is identical to class `thread` except where otherwise specified.

```
namespace std {
  class jthread {
  public:
    // types
    using id = thread::id;
    using native_handle_type = thread::native_handle_type;

    // construct/copy/destroy
    jthread() noexcept;
    template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
    ~jthread();
    jthread(const jthread&) = delete;
    jthread(jthread&&) noexcept;
    jthread& operator=(const jthread&) = delete;
    jthread& operator=(jthread&&) noexcept;

    // members
    void swap(jthread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();
    id get_id() const noexcept;
    native_handle_type native_handle();     // see ??

    // interrupt token handling
    interrupt_token get_original_interrupt_token() const noexcept;
    bool interrupt() noexcept;

    // static members
    static unsigned int hardware_concurrency() noexcept;

  private:
    interrupt_token itoken;                 // exposition only
  };
}
```

1.5.2.1 jthread constructors [thread.jthread.constr]

`jthread() noexcept;`

1 *Effects:* Constructs a `jthread` object that does not represent a thread of execution.

2 *Ensures:* `get_id() == id()` and `itoken.valid() == false`.

`template<class F, class... Args> explicit jthread(F&& f, Args&&... args);`

3 *Requires:* `F` and each T_i in `Args` shall satisfy the *Cpp17MoveConstructible* requirements. *INVOKE*(*DECAY_COPY*(std::forward<F>(f)), *DECAY_COPY*(std::forward<Args>(args))...) (??) shall be a valid expression.

4 *Remarks:* This constructor shall not participate in overload resolution if `remove_cvref_t<F>` is the same type as `std::jthread`.

5 *Effects:* Constructs an object of type `jthread`. The new thread of execution executes *INVOKE*(*DECAY_COPY*(std::forward<F>(f)), *DECAY_COPY*(std::forward<Args>(args))...) with the calls to *DECAY_COPY* being evaluated in the constructing thread. Any return value from this invocation is ignored. [*Note*: This implies that any exceptions not thrown from the invocation of the copy of `f` will be thrown in the constructing thread, not the new thread. — *end note*] If the invocation of *INVOKE*(*DECAY_COPY*(std::forward<F>(f)), *DECAY_COPY*(std::forward<Args>(args))...) terminates with an uncaught exception, `terminate` shall be called.

An uncaught `interrupted` exception in the started thread of execution will silently be ignored. [*Note*: Thus, an uncaught exception thrown by `throw_if_interrupted()` will cause the started thread to end silently. — *end note*]

6 *Synchronization:* The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

7 *Ensures:* `get_id() != id()`. `itoken.valid() == true`. `*this` represents the newly started thread. In the started thread of execution `this_thread::thread_itoken` is an `interrupt_token` equal to `itoken`. [*Note*: Note that the calling thread can signal an interrupt only once, because it can't replace this interrupt token. — *end note*]

8 *Throws:* `system_error` if unable to start the new thread.

9 *Error conditions:*

(9.1) — `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

`jthread(jthread&& x) noexcept;`

10 *Effects:* Constructs an object of type `jthread` from `x`, and sets `x` to a default constructed state.

11 *Ensures:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction. `itoken` yields the value of `x.itoken` prior to the start of construction and `x.itoken.valid() == false`.

1.5.2.2 jthread destructor [thread.jthread.destr]

`~jthread();`

1 If `joinable()`, calls `interrupt()` and `join()`. Otherwise, has no effects. [*Note*: Operations on `*this` are not synchronized. — *end note*]

1.5.2.3 jthread assignment [thread.jthread.assign]

`jthread& operator=(jthread&& x) noexcept;`

1 *Effects:* If `joinable()`, calls `interrupt()` and `join()`. Assigns the state of `x` to `*this` and sets `x` to a default constructed state.

2 *Ensures:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment. `itoken` yields the value of `x.itoken` prior to the assignment and `x.itoken.valid() == false`.

3 *Returns:* `*this`.

1.5.2.4 jthread interrupt members [thread.jthread.interrupt]

```
interrupt_token get_original_interrupt_token() const noexcept
```

1 *Effects:* Equivalent to: `return itoken;`

```
bool interrupt() noexcept;
```

2 *Effects:* Equivalent to: `return itoken.interrupt();`

1.5.3 Namespace this_thread Interrupt Handling [thread.jthread.this]

To be able to deal with signaled interrupt `this_thread` provides the following access to an `interrupt_token` (1.4.2.1):

```
namespace std::this_thread {
  interrupt_token thread_itoken;                  // exposition only
  static interrupt_token get_interrupt_token() noexcept;
  static bool is_interrupted() noexcept;
  static void throw_if_interrupted();
  static interrupt_token exchange_interrupt_token(const interrupt_token&) noexcept;
}
```

For any thread of execution, `thread_itoken` is default initialized unless the thread was started with a constructor of class `jthread` (??).

```
interrupt_token get_interrupt_token() noexcept;
```

1 *Returns:* `this_thread::thread_itoken`.

```
bool is_interrupted() noexcept;
```

2 *Returns:* `this_thread::get_interrupt_token().is_interrupted()`.

```
void throw_if_interrupted();
```

3 *Effects:* Equivalent to: `this_thread::get_interrupt_token().throw_if_interrupted();`

```
interrupt_token exchange_interrupt_token(const interrupt_token& it) noexcept;
```

4 *Effects:* Equivalent to: `this_thread::itoken = it;`

5 *Returns:* `this_thread::thread_itoken` prior to the exchange.

[*Note*: With this exchange `this_thread::get_interrupt_token()` will no longer signal interrupts from the calling threads until the returned token is restored. — *end note*]

1.6 Condition variables [thread.condition]

1.6.1 Class condition_variable [thread.condition.condvar]

```
namespace std {
  class condition_variable {
  public:
    condition_variable();
    ~condition_variable();

    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;

    void notify_one() noexcept;
    void notify_all() noexcept;

    void wait(unique_lock<mutex>& lock);
    template<class Predicate>
      void wait(unique_lock<mutex>& lock, Predicate pred);

    template<class Clock, class Duration>
      cv_status wait_until(unique_lock<mutex>& lock,
                           const chrono::time_point<Clock, Duration>& abs_time);
    template<class Clock, class Duration, class Predicate>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time,
                      Predicate pred);

    template<class Rep, class Period>
      cv_status wait_for(unique_lock<mutex>& lock,
                         const chrono::duration<Rep, Period>& rel_time);
    template<class Rep, class Period, class Predicate>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred);
    // 1.6.1.1 dealing with interrupts:

    // throw std::interrupted if this_thread::is_interrupted():
    void iwait(unique_lock<mutex>& lock);

    template<class Clock, class Duration, class Predicate>
      bool iwait_until(unique_lock<mutex>& lock,
                       const chrono::time_point<Clock, Duration>& abs_time,
                       Predicate pred);
    template<class Rep, class Period, class Predicate>
      bool iwait_for(unique_lock<mutex>& lock,
                     const chrono::duration<Rep, Period>& rel_time,
                     Predicate pred);

    // return false if itoken.is_interrupted():
    template <class Predicate>
      bool wait_until(unique_lock<mutex>& lock,
                      Predicate pred,
                      interrupt_token itoken);
    template <class Clock, class Duration, class Predicate>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time
                      Predicate pred,
                      interrupt_token itoken);
    template <class Rep, class Period, class Predicate>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred,
                    interrupt_token itoken);
    using native_handle_type = implementation-defined;           // see ??
    native_handle_type native_handle();                          // see ??
  };
}
```

1 The class `condition_variable` shall be a standard-layout class (??).

```
condition_variable();
```

2 *Effects:* Constructs an object of type `condition_variable`.

3 *Throws:* `system_error` when an exception is required (??).

4 *Error conditions:*

(4.1) — `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

```
~condition_variable();
```

5 *Requires:* There shall be no thread blocked on `*this`. [*Note*: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. — *end note*]

6 *Effects:* Destroys the object.

```
void notify_one() noexcept;
```

7 *Effects:* If any threads are blocked waiting for `*this`, unblocks one of those threads.

```
void notify_all() noexcept;
```

8 *Effects:* Unblocks all threads that are blocked waiting for `*this`.

```
void wait(unique_lock<mutex>& lock);
```

9 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(9.1) — no other thread is waiting on this `condition_variable` object or

(9.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

10 *Effects:*

(10.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(10.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.

(10.3) — The function will unblock when signaled by a call to `notify_one()` or a call to `notify_all()`, or spuriously.

11 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

12 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

13 *Throws:* Nothing.

```
template<class Predicate>
  void wait(unique_lock<mutex>& lock, Predicate pred);
```

14 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(14.1) — no other thread is waiting on this `condition_variable` object or

(14.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

15 *Effects:* Equivalent to:

```
while (!pred())
  wait(lock);
```

16 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

17 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

18 *Throws:* Any exception thrown by `pred`.

```
template<class Clock, class Duration>
  cv_status wait_until(unique_lock<mutex>& lock,
                       const chrono::time_point<Clock, Duration>& abs_time);
```

19 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(19.1) — no other thread is waiting on this `condition_variable` object or

(19.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

20 *Effects:*

(20.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(20.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.

(20.3) — The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (??) specified by `abs_time`, or spuriously.

(20.4) — If the function exits via an exception, `lock.lock()` shall be called prior to exiting the function.

21 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

22 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

23 *Returns:* `cv_status::timeout` if the absolute timeout (??) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

24 *Throws:* Timeout-related exceptions (??).

```
template<class Rep, class Period>
  cv_status wait_for(unique_lock<mutex>& lock,
                     const chrono::duration<Rep, Period>& rel_time);
```

25 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(25.1) — no other thread is waiting on this `condition_variable` object or

(25.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

26 *Effects:* Equivalent to:

```
  return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

27 *Returns:* `cv_status::timeout` if the relative timeout (??) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

28 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

29 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

30 *Throws:* Timeout-related exceptions (??).

```
template<class Clock, class Duration, class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time,
                  Predicate pred);
```

31 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(31.1) — no other thread is waiting on this `condition_variable` object or

(31.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

32 *Effects:* Equivalent to:

```
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
return true;
```

33 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

34 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

35 [*Note*: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered. — *end note*]

36 *Throws:* Timeout-related exceptions (??) or any exception thrown by `pred`.

```
template<class Rep, class Period, class Predicate>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred);
```

37 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(37.1) — no other thread is waiting on this `condition_variable` object or

(37.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

38 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

39 [*Note*: There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — *end note*]

40 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

41 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

42 [*Note*: The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

43 *Throws:* Timeout-related exceptions (??) or any exception thrown by `pred`.

1.6.1.1 interrupt_token handling [thread.condition.interrupted]

The following functions respect the state of the `interrupt_token` passed as argument or returned by `this_thread::get_interrupt_token()`. The functions starting with `i` (`iwait()`, `iwait_until()`, `iwait_for()` throw `std::interrupted` on a signaled interrupt. The other functions listed here (`wait_until()`, `wait_for()` return `false` on a signaled interrupt (if the predicate evaluates to `false`).

[*Editorial note:* This color signals differences to the corresponding `wait...()` function.]

```
template<class Predicate>
  void iwait(unique_lock<mutex>& lock, Predicate pred);
```

1 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(1.1) — no other thread is waiting on this `condition_variable` object or

(1.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

2 *Effects:* Registers `*this` to get notified if an interrupt is signaled on `this_thread::get_interrupt_token()` and then equivalent to:

```
while(!pred()) {
  this_thread::throw_if_interrupted();
  cv.wait(lock, [&pred] {
                  return pred() || this_thread::is_interrupted();
                });
}
```

3 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (**??**). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

4 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

5 *Throws:* Any exception thrown by `pred` or exception `interrupted` if `this_thread::is_interrupted()`.

6 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called `is_interrupted()`.

```
template<class Clock, class Duration, class Predicate>
  bool iwait_until(unique_lock<mutex>& lock,
                   const chrono::time_point<Clock, Duration>& abs_time,
                   Predicate pred);
```

7 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(7.1) — no other thread is waiting on this `condition_variable` object or

(7.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

8 *Effects:* Registers `*this` to get notified if an interrupt is signaled on `this_thread::get_interrupt_token()` and then equivalent to:

```
while(!pred() && Clock::now() < abs_time) {
  this_thread::throw_if_interrupted();
  cv.wait_until(lock, abs_time,
                [&pred] {
                  return pred() || this_thread::is_interrupted();
                });
}
return pred();
```

9 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (**??**). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

10 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

11 [*Note*: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered or an interrupt was signaled. — *end note*]

12 *Throws:* Timeout-related exceptions (**??**) or any exception thrown by `pred` or exception `interrupted` if `this_thread::is_interrupted()`.

13 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called `is_interrupted()`.

```
template<class Rep, class Period, class Predicate>
  bool iwait_for(unique_lock<mutex>& lock,
                 const chrono::duration<Rep, Period>& rel_time,
                 Predicate pred);
```

14 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(14.1) — no other thread is waiting on this `condition_variable` object or

(14.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

15 *Effects:* Equivalent to:

```
this_thread::throw_if_interrupted();
return iwait_until(lock,
                   std::chrono::steady_clock::now() + rel_time,
                   std::move(pred));
```

16 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (**??**). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

17 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

18 [*Note*: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered or an interrupt was signaled. — *end note*]

19 *Throws:* Timeout-related exceptions (??) or any exception thrown by pred or exception interrupted if this_thread::is_interrupted().

20 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called is_interrupted().

[*Editorial note:* This color signals differences to the corresponding wait() function.]

```
template <class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  Predicate pred,
                  interrupt_token itoken);
```

21 *Requires:* lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

(21.1) — no other thread is waiting on this condition_variable object or

(21.2) — lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until , iwait, iwait_for, or iwait_until) threads.

22 *Effects:* Registers *this to get notified if an interrupt is signaled on itoken and then equivalent to:

```
while(!pred() && !itoken.is_interrupted()) {
  cv.wait(lock, [&pred, &itoken] {
                  return pred() || itoken.is_interrupted();
                });
}
return pred();
```

23 [*Note*: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or an interrupt was signaled. — *end note*]

24 *Ensures:* lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

25 *Remarks:* If the function fails to meet the postcondition, terminate() shall be called (??). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

26 *Throws:* Any exception thrown by pred.

27 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called is_interrupted().

[*Editorial note:* This color signals differences to the corresponding wait_until() function without interrupt token.]

```
template <class Clock, class Duration, class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time
                  Predicate pred,
                  interrupt_token itoken);
```

28 *Requires:* lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

(28.1) — no other thread is waiting on this condition_variable object or

(28.2) — lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until , iwait, iwait_for, or iwait_until) threads.

29 *Effects:* Registers *this to get notified if an interrupt is signaled on itoken and then equivalent to:

```
while(!pred() && !itoken.is_interrupted() && Clock::now() < abs_time) {
  cv.wait_until(lock,
                abs_time,
                [&pred, &itoken] {
                  return pred() || itoken.is_interrupted();
                });
}
return pred();
```

30 [*Note*: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or an interrupt was signaled. — *end note*]

31 *Ensures:* lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

32 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (**??**). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

33 *Throws:* Timeout-related exceptions (**??**) or any exception thrown by `pred`.

34 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called `is_interrupted()`.

[*Editorial note:* This color signals differences to the corresponding `wait_for()` function without interrupt token.]

```
template <class Rep, class Period, class Predicate>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred,
                interrupt_token itoken);
```

35 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(35.1) — no other thread is waiting on this `condition_variable` object or

(35.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until` , `iwait`, `iwait_for`, or `iwait_until`) threads.

36 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred),
                  std::move(itoken));
```

37 [*Note*: The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered or an interrupt was signaled. — *end note*]

38 [*Note*: There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — *end note*]

39 *Ensures:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

40 *Remarks:* If the function fails to meet the postcondition, `terminate()` shall be called (**??**). [*Note*: This can happen if the re-locking of the mutex throws an exception. — *end note*]

41 *Throws:* Timeout-related exceptions (**??**) or any exception thrown by `pred` or `interrupted` if an interrupt was signaled.

42 *Synchronization:* If the function returns with an interrupted status, their synchronization behavior is as though it called `is_interrupted()`.