```
/// Common protocol to which all actors
conform.
///
/// The `Actor` protocol generalizes over
all `actor` types. Actor types
/// implicitly conform to this protocol.
///
/// ### Actors and SerialExecutors
/// By default, actors execute tasks on a
shared global concurrency thread pool.
/// This pool is shared by all default
actors and tasks, unless an actor or task
/// specified a more specific executor
requirement.
///
/// It is possible to configure an actor
to use a specific ``SerialExecutor``,
/// as well as impact the scheduling of
default tasks and actors by using
/// a ``TaskExecutor``.
///
/// - SeeAlso: ``SerialExecutor``
/// - SeeAlso: ``TaskExecutor``
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol Actor : AnyObject,
Sendable {

    /// Retrieve the executor for this
actor as an optimized, unowned
    /// reference.
    ///
```

```
    /// This property must always
evaluate to the same executor for a
    /// given actor instance, and holding
on to the actor must keep the
    /// executor alive.
    ///
    /// This property will be implicitly
accessed when work needs to be
    /// scheduled onto this actor.  These
accesses may be merged,
    /// eliminated, and rearranged with
other work, and they may even
    /// be introduced when not strictly
required.  Visible side effects
    /// are therefore strongly
discouraged within this property.
    ///
    /// - SeeAlso: ``SerialExecutor``
    /// - SeeAlso: ``TaskExecutor``
    nonisolated var unownedExecutor:
UnownedSerialExecutor { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Actor {

    /// Stops program execution if the
current task is not executing on this
    /// actor's serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
```

```
    ///
    /// * In playgrounds and `-Onone` builds (the default for Xcode's Debug
    ///   configuration), stops program execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for Xcode's Release configuration), stops
    ///   program execution.
    ///
    /// - Note: This check is performed against the actor's serial executor,
    ///   meaning that / if another actor uses the same serial executor--by using
    ///   that actor's serial executor as its own ``Actor/unownedExecutor``--this
    ///   check will succeed , as from a concurrency safety perspective, the
    ///   serial executor guarantees mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - message: The message to print if the assertion fails.
    ///   - file: The file name to print if the assertion fails. The default is
    ///           where this method was called.
    ///   - line: The line number to print if the assertion fails The default is
    ///           where this method was
```

```
called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    nonisolated public func
preconditionIsolated(_ message:
@autoclosure () -> String = String(),
file: StaticString = #fileID, line: UInt
= #line)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Actor {

    /// Stops program execution if the
current task is not executing on this
    /// actor's serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
    ///
    /// * In playgrounds and `-Onone`
builds (the default for Xcode's Debug
    ///   configuration), stops program
execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for
Xcode's Release configuration),
    ///   the isolation check is not
performed and there are no effects.
```

```
    ///
    /// - Note: This check is performed
against the actor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
    ///   that actor's serial executor as
its own ``Actor/unownedExecutor``--this
    ///   check will succeed , as from a
concurrency safety perspective, the
    ///   serial executor guarantees
mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - message: The message to print
if the assertion fails.
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    nonisolated public func
assertIsolated(_ message: @autoclosure ()
-> String = String(), file: StaticString
= #fileID, line: UInt = #line)
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Actor {

    /// Assume that the current task is
executing on this actor's serial
executor,
    /// or stop program execution
otherwise.
    ///
    /// You call this method to *assume
and verify* that the currently
    /// executing synchronous function is
actually executing on the serial
    /// executor of this actor.
    ///
    /// If that is the case, the
operation is invoked with an `isolated`
version
    /// of the actor, allowing
synchronous access to actor local state
without
    /// hopping through asynchronous
boundaries.
    ///
    /// If the current context is not
running on the actor's serial executor,
or
    /// if the actor is a reference to a
remote actor, this method will crash
    /// with a fatal error (similar to
``preconditionIsolated()``).
```

```
    ///
    /// Note that this check is performed
against the passed in actor's serial
    /// executor, meaning that if another
actor uses the same serial executor--by
    /// using that actor's
``Actor/unownedExecutor`` as its own
    /// ``Actor/unownedExecutor``--this
check will succeed, as from a concurrency
    /// safety perspective, the serial
executor guarantees mutual exclusion of
    /// those two actors.
    ///
    /// This method can only be used from
synchronous functions, as asynchronous
    /// functions should instead perform
a normal method call to the actor, which
    /// will hop task execution to the
target actor if necessary.
    ///
    /// - Note: This check is performed
against the actor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
    ///   another actor's executor as its
own ``Actor/unownedExecutor``
    ///   --this check will succeed , as
from a concurrency safety perspective,
    ///   the serial executor guarantees
mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - operation: the operation that
```

```
will be executed if the current context
    ///                 is executing on
the actors serial executor.
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    /// - Returns: the return value of
the `operation`
    /// - Throws: rethrows the `Error`
thrown by the operation if it threw
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    nonisolated public func
assumeIsolated<T>(_ operation: (isolated
Self) throws -> T, file: StaticString =
#fileID, line: UInt = #line) rethrows ->
T where T : Sendable
}

/// Common marker protocol providing a
shared "base" for both (local) `Actor`
/// and (potentially remote)
`DistributedActor` types.
///
/// The `AnyActor` marker protocol
generalizes over all actor types,
including
```

```
/// distributed ones. In practice, this
protocol can be used to restrict
/// protocols, or generic parameters to
only be usable with actors, which
/// provides the guarantee that calls may
be safely made on instances of given
/// type without worrying about the
thread-safety of it -- as they are
/// guaranteed to follow the actor-style
isolation semantics.
///
/// While both local and distributed
actors are conceptually "actors", there
are
/// some important isolation model
differences between the two, which make
it
/// impossible for one to refine the
other.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@available(*, deprecated, message: "Use
'any Actor' with
'DistributedActor.asLocalActor' instead")
@available(swift, obsoleted: 6.0,
message: "Use 'any Actor' with
'DistributedActor.asLocalActor' instead")
public typealias AnyActor = AnyObject &
Sendable

/// An asynchronous sequence that maps a
given closure over the asynchronous
/// sequence’s elements, omitting results
```

```
that don't return a value.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncCompactMapSequence<Base,
ElementOfResult> where Base :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncCompactMapSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The compact map sequence produces
whatever type of element its
    /// transforming closure produces.
    public typealias Element =
ElementOfResult

    /// The type of the error that can be
produced by the sequence.
    ///
    /// The compact map sequence produces
whatever type of error its
    /// base sequence does.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure
```

```
    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncCompactMapSequence<Base,
ElementOfResult>.Iterator

    /// The iterator that produces
elements of the compact map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        public typealias Element =
ElementOfResult

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the compact map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` calls the
        /// transforming closure on the
received element, returning it if the
        /// transform returns a non-`nil`
```

```
value. If the transform returns `nil`,
        /// this method continues to wait
for further elements until it gets one
        /// that transforms to a non-
`nil` value.
        @inlinable public mutating func
next() async rethrows -> ElementOfResult?

        /// Produces the next element in
the compact map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` calls the
        /// transforming closure on the
received element, returning it if the
        /// transform returns a non-`nil`
value. If the transform returns `nil`,
        /// this method continues to wait
for further elements until it gets one
        /// that transforms to a non-
`nil` value.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncCompactMapSequence<Base,
ElementOfResult>.Iterator.Failure) ->
ElementOfResult?
    }
```

```
    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncCompactMapSequence<Base,
ElementOfResult>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncCompactMapSequence :
@unchecked Sendable where Base :
Sendable, ElementOfResult : Sendable,
Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncCompactMapSequence.Iterator :
@unchecked Sendable where ElementOfResult
: Sendable, Base.AsyncIterator :
Sendable, Base.Element : Sendable {
}

/// An asynchronous sequence which omits
```

```
a specified number of elements from the
/// base asynchronous sequence, then
passes through all remaining elements.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncDropFirstSequence<Base> where Base :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropFirstSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The drop-first sequence produces
whatever type of element its base
    /// iterator produces.
    public typealias Element =
Base.Element

    /// The type of errors produced by
this asynchronous sequence.
    ///
    /// The drop-first sequence produces
whatever type of error its base
    /// sequence produces.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
```

```
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncDropFirstSequence<Base>.Iterator

    /// The iterator that produces
elements of the drop-first sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the drop-first sequence.
        ///
        /// Until reaching the number of
elements to drop, this iterator calls
        /// `next()` on its base iterator
and discards the result. If the base
        /// iterator returns `nil`,
indicating the end of the sequence, this
        /// iterator returns `nil`. After
reaching the number of elements to
        /// drop, this iterator passes
along the result of calling `next()` on
        /// the base iterator.
```

```
        @inlinable public mutating func
next() async rethrows -> Base.Element?

        /// Produces the next element in
the drop-first sequence.
        ///
        /// Until reaching the number of
elements to drop, this iterator calls
        /// `next(isolation:)` on its
base iterator and discards the result. If
the
        /// base iterator returns `nil`,
indicating the end of the sequence, this
        /// iterator returns `nil`. After
reaching the number of elements to drop,
        /// this iterator passes along
the result of calling `next(isolation:)`
on
        /// the base iterator.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncDropFirstSequence<Base>.Itera
tor.Failure) -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }
```

```
    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncDropFirstSequence<Base>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropFirstSequence {

    /// Omits a specified number of
elements from the base asynchronous
sequence,
    /// then passes through all remaining
elements.
    ///
    /// When you call `dropFirst(_:)` on
an asynchronous sequence that is already
    /// an `AsyncDropFirstSequence`, the
returned sequence simply adds the new
    /// drop count to the current drop
count.
    @inlinable public func dropFirst(_
count: Int = 1) ->
AsyncDropFirstSequence<Base>
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropFirstSequence :
Sendable where Base : Sendable,
Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropFirstSequence.Iterator
: Sendable where Base.AsyncIterator :
Sendable, Base.Element : Sendable {
}

/// An asynchronous sequence which omits
elements from the base sequence until a
/// given closure returns false, after
which it passes through all remaining
/// elements.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncDropWhileSequence<Base> where Base :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropWhileSequence :
AsyncSequence {

    /// The type of element produced by
```

```
this asynchronous sequence.
    ///
    /// The drop-while sequence produces
whatever type of element its base
    /// sequence produces.
    public typealias Element =
Base.Element

    /// The type of errors produced by
this asynchronous sequence.
    ///
    /// The drop-while sequence produces
whatever type of error its base
    /// sequence produces.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncDropWhileSequence<Base>.Iterator

    /// The iterator that produces
elements of the drop-while sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
```

```
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the drop-while sequence.
        ///
        /// This iterator calls `next()`
on its base iterator and evaluates the
        /// result with the `predicate`
closure. As long as the predicate returns
        /// `true`, this method returns
`nil`. After the predicate returns
`false`,
        /// for a value received from the
base iterator, this method returns that
        /// value. After that, the
iterator returns values received from its
        /// base iterator as-is, and
never executes the predicate closure
again.
        @inlinable public mutating func
next() async rethrows -> Base.Element?

        /// Produces the next element in
the drop-while sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator
and
        /// evaluates the result with the
`predicate` closure. As long as the
        /// predicate returns `true`,
this method returns `nil`. After the
```

```
predicate
        /// returns `false`, for a value
received from the base iterator, this
        /// method returns that value.
After that, the iterator returns values
        /// received from its base
iterator as-is, and never executes the
predicate
        /// closure again.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncDropWhileSequence<Base>.Itera
tor.Failure) -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates an instance of the drop-
while sequence iterator.
    @inlinable public func
makeAsyncIterator() ->
AsyncDropWhileSequence<Base>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropWhileSequence :
```

```
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncDropWhileSequence.Iterator
: @unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence that
contains, in order, the elements of
/// the base sequence that satisfy a
given predicate.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct AsyncFilterSequence<Base>
where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncFilterSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The filter sequence produces
whatever type of element its base
    /// sequence produces.
```

```
    public typealias Element =
Base.Element

    /// The type of the error that can be
produced by the sequence.
    ///
    /// The filter sequence produces
whatever type of error its
    /// base sequence does.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncFilterSequence<Base>.Iterator

    /// The iterator that produces
elements of the filter sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the filter sequence.
```

```
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns nil.
Otherwise, `next()` evaluates the
        /// result with the `predicate`
closure. If the closure returns `true`,
        /// `next()` returns the received
element; otherwise it awaits the next
        /// element from the base
iterator.
        @inlinable public mutating func
next() async rethrows -> Base.Element?

        /// Produces the next element in
the filter sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns nil.
Otherwise,
        /// `next(isolation:)` evaluates
the result with the `predicate` closure.
If
        /// the closure returns `true`,
`next(isolation:)` returns the received
        /// element; otherwise it awaits
the next element from the base iterator.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
```

```
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncFilterSequence<Base>.Iterator
.Failure) -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncFilterSequence<Base>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncFilterSequence :
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
```

```
6.0, tvOS 13.0, *)
extension AsyncFilterSequence.Iterator :
@unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence that
concatenates the results of calling a
given
/// transformation with each element of
this sequence.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct AsyncFlatMapSequence<Base,
SegmentOfResult> where Base :
AsyncSequence, SegmentOfResult :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncFlatMapSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The flat map sequence produces
the type of element in the asynchronous
    /// sequence produced by the
`transform` closure.
    public typealias Element =
```

```
SegmentOfResult.Element

    /// The type of error produced by
this asynchronous sequence.
    ///
    /// The flat map sequence produces
the type of error in the base
asynchronous
    /// sequence. By construction, the
sequence produced by the `transform`
    /// closure must either produce this
type of error or not produce errors
    /// at all.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncFlatMapSequence<Base,
SegmentOfResult>.Iterator

    /// The iterator that produces
elements of the flat map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
```

```
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the flat map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` calls the
        /// transforming closure on the
received element, takes the resulting
        /// asynchronous sequence, and
creates an asynchronous iterator from it.
        /// `next()` then consumes values
from this iterator until it terminates.
        /// At this point, `next()` is
ready to receive the next value from the
base
        /// sequence.
        @inlinable public mutating func
next() async rethrows ->
SegmentOfResult.Element?

        /// Produces the next element in
the flat map sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns `nil`.
```

```
Otherwise,
        /// `next(isolation:)` calls the
transforming closure on the received
        /// element, takes the resulting
asynchronous sequence, and creates an
        /// asynchronous iterator from
it.  `next(isolation:)` then consumes
values
        /// from this iterator until it
terminates.  At this point,
        /// `next(isolation:)` is ready
to receive the next value from the base
        /// sequence.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncFlatMapSequence<Base,
SegmentOfResult>.Iterator.Failure) ->
SegmentOfResult.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
SegmentOfResult.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
```

```
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncFlatMapSequence<Base,
SegmentOfResult>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncFlatMapSequence :
@unchecked Sendable where Base :
Sendable, SegmentOfResult : Sendable,
Base.Element : Sendable,
SegmentOfResult.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncFlatMapSequence.Iterator :
@unchecked Sendable where SegmentOfResult
: Sendable, Base.AsyncIterator :
Sendable, Base.Element : Sendable,
SegmentOfResult.AsyncIterator : Sendable,
SegmentOfResult.Element : Sendable {
}

/// A type that asynchronously supplies
the values of a sequence one at a
/// time.
///
/// The `AsyncIteratorProtocol` defines
```

```
the type returned by the
/// `makeAsyncIterator()` method of the
`AsyncSequence` protocol. In short,
/// the iterator is what produces the
asynchronous sequence's values. The
/// protocol defines a single
asynchronous method, `next()`, which
either
/// produces the next element of the
sequence, or returns `nil` to signal
/// the end of the sequence.
///
/// To implement your own
`AsyncSequence`, implement a wrapped type
that
/// conforms to `AsyncIteratorProtocol`.
The following example shows a `Counter`
/// type that uses an inner iterator to
monotonically generate `Int` values
/// until reaching a `howHigh` value.
While this example isn't itself
/// asynchronous, it shows the shape of a
custom sequence and iterator, and how
/// to use it as if it were asynchronous:
///
///     struct Counter: AsyncSequence {
///         typealias Element = Int
///         let howHigh: Int
///
///         struct AsyncIterator:
AsyncIteratorProtocol {
///             let howHigh: Int
///             var current = 1
```

```
///
///             mutating func next()
async -> Int? {
///                 // A genuinely
asynchronous implementation uses the
`Task`
///                 // API to check for
cancellation here and return early.
///                 guard current <=
howHigh else {
///                     return nil
///                 }
///
///                 let result = current
///                 current += 1
///                 return result
///             }
///         }
///
///         func makeAsyncIterator() ->
AsyncIterator {
///             return
AsyncIterator(howHigh: howHigh)
///         }
///     }
///
/// At the call site, this looks like:
///
///     for await number in
Counter(howHigh: 10) {
///       print(number, terminator: " ")
///     }
///     // Prints "1 2 3 4 5 6 7 8 9 10 "
```

```
///
/// ### End of Iteration
///
/// The iterator returns `nil` to
indicate the end of the sequence. After
/// returning `nil` (or throwing an
error) from `next()`, the iterator enters
/// a terminal state, and all future
calls to `next()` must return `nil`.
///
/// ### Cancellation
///
/// Types conforming to
`AsyncIteratorProtocol` should use the
cancellation
/// primitives provided by Swift's `Task`
API. The iterator can choose how to
/// handle and respond to cancellation,
including:
///
/// - Checking the `isCancelled` value of
the current `Task` inside `next()`
///   and returning `nil` to terminate
the sequence.
/// - Calling `checkCancellation()` on
the `Task`, which throws a
///   `CancellationError`.
/// - Implementing `next()` with a
///
`withTaskCancellationHandler(handler:oper
ation:)` invocation to
///   immediately react to cancellation.
///
```

```
/// If the iterator needs to clean up on
cancellation, it can do so after
/// checking for cancellation as
described above, or in `deinit` if it's
/// a reference type.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol
AsyncIteratorProtocol<Element, Failure> {

    associatedtype Element

    /// The type of failure produced by
iteration.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    associatedtype Failure : Error = any
Error

    /// Asynchronously advances to the
next element and returns it, or ends the
    /// sequence if there is no next
element.
    ///
    /// - Returns: The next element, if
it exists, or `nil` to signal the end of
    ///   the sequence.
    mutating func next() async throws ->
Self.Element?

    /// Asynchronously advances to the
next element and returns it, or ends the
    /// sequence if there is no next
```

```
element.
    ///
    /// - Returns: The next element, if
it exists, or `nil` to signal the end of
    ///   the sequence.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    mutating func next(isolation actor:
isolated (any Actor)?) async
throws(Self.Failure) -> Self.Element?
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncIteratorProtocol {

    /// Default implementation of
`next()` in terms of `next()`, which is
    /// required to maintain backward
compatibility with existing async
iterators.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws(Self.Failure) ->
Self.Element?
}

/// An asynchronous sequence that maps
the given closure over the asynchronous
/// sequence's elements.
@available(macOS 10.15, iOS 13.0, watchOS
```

```
6.0, tvOS 13.0, *)
public struct AsyncMapSequence<Base,
Transformed> where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncMapSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The map sequence produces
whatever type of element its transforming
    /// closure produces.
    public typealias Element =
Transformed

    /// The type of the error that can be
produced by the sequence.
    ///
    /// The map sequence produces
whatever type of error its
    /// base sequence does.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
```

```
AsyncMapSequence<Base,
Transformed>.Iterator

    /// The iterator that produces
elements of the map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` returns the
result of
        /// calling the transforming
closure on the received element.
        @inlinable public mutating func
next() async rethrows -> Transformed?

        /// Produces the next element in
the map sequence.
        ///
        /// This iterator calls
```

```
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns `nil`.
Otherwise,
        /// `next(isolation:)` returns
the result of calling the transforming
        /// closure on the received
element.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncMapSequence<Base,
Transformed>.Iterator.Failure) ->
Transformed?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Transformed
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
```

```
makeAsyncIterator() ->
AsyncMapSequence<Base,
Transformed>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncMapSequence : @unchecked
Sendable where Base : Sendable,
Transformed : Sendable, Base.Element :
Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncMapSequence.Iterator :
@unchecked Sendable where Transformed :
Sendable, Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence, up to a
specified maximum length,
/// containing the initial elements of a
base asynchronous sequence.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct AsyncPrefixSequence<Base>
where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
extension AsyncPrefixSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The prefix sequence produces
whatever type of element its base
iterator
    /// produces.
    public typealias Element =
Base.Element

    /// The type of the error that can be
produced by the sequence.
    ///
    /// The prefix sequence produces
whatever type of error its
    /// base sequence does.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncPrefixSequence<Base>.Iterator

    /// The iterator that produces
elements of the prefix sequence.
    public struct Iterator :
AsyncIteratorProtocol {
```

```
        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the prefix sequence.
        ///
        /// Until reaching the number of
elements to include, this iterator calls
        /// `next()` on its base iterator
and passes through the result. After
        /// reaching the maximum number
of elements, subsequent calls to `next()`
        /// return `nil`.
        @inlinable public mutating func
next() async rethrows -> Base.Element?

        /// Produces the next element in
the prefix sequence.
        ///
        /// Until reaching the number of
elements to include, this iterator calls
        /// `next(isolation:)` on its
base iterator and passes through the
        /// result. After reaching the
maximum number of elements, subsequent
calls
        /// to `next(isolation:)` return
`nil`.
```

```
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncPrefixSequence<Base>.Iterator
.Failure) -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncPrefixSequence<Base>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncPrefixSequence : Sendable
where Base : Sendable, Base.Element :
Sendable {
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncPrefixSequence.Iterator :
Sendable where Base.AsyncIterator :
Sendable, Base.Element : Sendable {
}

/// An asynchronous sequence, containing
the initial, consecutive
/// elements of the base sequence that
satisfy a given predicate.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncPrefixWhileSequence<Base> where Base
: AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncPrefixWhileSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The prefix-while sequence
produces whatever type of element its
base
    /// iterator produces.
    public typealias Element =
Base.Element
```

```
    /// The type of the error that can be
produced by the sequence.
    ///
    /// The prefix-while sequence
produces whatever type of error its
    /// base sequence does.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public typealias Failure =
Base.Failure

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncPrefixWhileSequence<Base>.Iterator

    /// The iterator that produces
elements of the prefix-while sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The type of failure produced
by iteration.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public typealias Failure =
Base.Failure

        /// Produces the next element in
the prefix-while sequence.
        ///
        /// If the predicate hasn't yet
```

```
failed, this method gets the next element
        /// from the base sequence and
calls the predicate with it. If this call
        /// succeeds, this method passes
along the element. Otherwise, it returns
        /// `nil`, ending the sequence.
        @inlinable public mutating func
next() async rethrows -> Base.Element?

        /// Produces the next element in
the prefix-while sequence.
        ///
        /// If the predicate hasn't yet
failed, this method gets the next element
        /// from the base sequence and
calls the predicate with it. If this call
        /// succeeds, this method passes
along the element. Otherwise, it returns
        /// `nil`, ending the sequence.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async
throws(AsyncPrefixWhileSequence<Base>.Ite
rator.Failure) -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }
```

```
    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncPrefixWhileSequence<Base>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncPrefixWhileSequence :
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncPrefixWhileSequence.Iterator :
@unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// A type that provides asynchronous,
sequential, iterated access to its
/// elements.
///
```

```
/// An `AsyncSequence` resembles the
`Sequence` type --- offering a list of
/// values you can step through one at a
time --- and adds asynchronicity. An
/// `AsyncSequence` may have all, some,
or none of its values available when
/// you first use it. Instead, you use
`await` to receive values as they become
/// available.
///
/// As with `Sequence`, you typically
iterate through an `AsyncSequence` with a
/// `for await`-`in` loop. However,
because the caller must potentially wait
for values,
/// you use the `await` keyword. The
following example shows how to iterate
/// over `Counter`, a custom
`AsyncSequence` that produces `Int`
values from
/// `1` up to a `howHigh` value:
///
///     for await number in
Counter(howHigh: 10) {
///         print(number, terminator: "
")
///     }
///     // Prints "1 2 3 4 5 6 7 8 9 10 "
///
/// An `AsyncSequence` doesn't generate
or contain the values; it just defines
/// how you access them. Along with
defining the type of values as an
```

```
associated
/// type called `Element`, the
`AsyncSequence` defines a
`makeAsyncIterator()`
/// method. This returns an instance of
type `AsyncIterator`. Like the standard
/// `IteratorProtocol`, the
`AsyncIteratorProtocol` defines a single
`next()`
/// method to produce elements. The
difference is that the `AsyncIterator`
/// defines its `next()` method as
`async`, which requires a caller to wait
for
/// the next value with the `await`
keyword.
///
/// `AsyncSequence` also defines methods
for processing the elements you
/// receive, modeled on the operations
provided by the basic `Sequence` in the
/// standard library. There are two
categories of methods: those that return
a
/// single value, and those that return
another `AsyncSequence`.
///
/// Single-value methods eliminate the
need for a `for await`-`in` loop, and
instead
/// let you make a single `await` call.
For example, the `contains(_:)` method
/// returns a Boolean value that
```

```
indicates if a given value exists in the
/// `AsyncSequence`. Given the `Counter`
sequence from the previous example,
/// you can test for the existence of a
sequence member with a one-line call:
///
///     let found = await
Counter(howHigh: 10).contains(5) // true
///
/// Methods that return another
`AsyncSequence` return a type specific to
the
/// method's semantics. For example, the
`.map(_:)` method returns a
/// `AsyncMapSequence` (or a
`AsyncThrowingMapSequence`, if the
closure you
/// provide to the `map(_:)` method can
throw an error). These returned
/// sequences don't eagerly await the
next member of the sequence, which allows
/// the caller to decide when to start
work. Typically, you'll iterate over
/// these sequences with `for await`-
`in`, like the base `AsyncSequence` you
started
/// with. In the following example, the
`map(_:)` method transforms each `Int`
/// received from a `Counter` sequence
into a `String`:
///
///     let stream = Counter(howHigh: 10)
///         .map { $0 % 2 == 0 ? "Even" :
```

```
"Odd" }
///     for await s in stream {
///         print(s, terminator: " ")
///     }
///     // Prints "Odd Even Odd Even Odd
Even Odd Even Odd Even "
///
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol AsyncSequence<Element,
Failure> {

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    associatedtype AsyncIterator :
AsyncIteratorProtocol

    /// The type of element produced by
this asynchronous sequence.
    associatedtype Element where
Self.Element ==
Self.AsyncIterator.Element

    /// The type of errors produced when
iteration over the sequence fails.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    associatedtype Failure = any Error
where Self.Failure ==
Self.AsyncIterator.Failure

    /// Creates the asynchronous iterator
```

```
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    func makeAsyncIterator() ->
Self.AsyncIterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that maps the given closure over the
    /// asynchronous sequence's elements,
omitting results that don't return a
    /// value.
    ///
    /// Use the `compactMap(_:)` method
to transform every element received from
    /// a base asynchronous sequence,
while also discarding any `nil` results
    /// from the closure. Typically, you
use this to transform from one type of
    /// element to another.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
closure provided to the `compactMap(_:)`
```

```
    /// method takes each `Int` and looks
up a corresponding `String` from a
    /// `romanNumeralDict` dictionary.
Because there is no key for `4`, the
closure
    /// returns `nil` in this case, which
`compactMap(_:)` omits from the
    /// transformed asynchronous
sequence.
    ///
    ///     let romanNumeralDict: [Int:
String] =
    ///         [1: "I", 2: "II", 3:
"III", 5: "V"]
    ///
    ///     let stream = Counter(howHigh:
5)
    ///         .compactMap
{ romanNumeralDict[$0] }
    ///     for await numeral in stream {
    ///         print(numeral,
terminator: " ")
    ///     }
    ///     // Prints "I II III V "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns a transformed value
of the
    ///   same or of a different type.
    /// - Returns: An asynchronous
sequence that contains, in order, the
```

```
    ///   non-`nil` elements produced by
the `transform` closure.
    @preconcurrency @inlinable public
func compactMap<ElementOfResult>(_
transform: @escaping @Sendable
(Self.Element) async -> ElementOfResult?)
-> AsyncCompactMapSequence<Self,
ElementOfResult>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Omits a specified number of
elements from the base asynchronous
sequence,
    /// then passes through all remaining
elements.
    ///
    /// Use `dropFirst(_:)` when you want
to drop the first *n* elements from the
    /// base sequence and pass through
the remaining elements.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`dropFirst(_:)` method causes the
modified
    /// sequence to ignore the values `1`
through `3`, and instead emit `4` through
`10`:
```

```
    ///
    ///     for await number in
Counter(howHigh: 10).dropFirst(3) {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "4 5 6 7 8 9 10 "
    ///
    /// If the number of elements to drop
exceeds the number of elements in the
    /// sequence, the result is an empty
sequence.
    ///
    /// - Parameter count: The number of
elements to drop from the beginning of
    ///   the sequence. `count` must be
greater than or equal to zero.
    /// - Returns: An asynchronous
sequence that drops the first `count`
    ///   elements from the base
sequence.
    @inlinable public func dropFirst(_
count: Int = 1) ->
AsyncDropFirstSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Omits elements from the base
asynchronous sequence until a given
closure
```

```
    /// returns false, after which it
passes through all remaining elements.
    ///
    /// Use `drop(while:)` to omit
elements from an asynchronous sequence
until
    /// the element received meets a
condition you specify.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`drop(while:)` method causes the modified
    /// sequence to ignore received
values until it encounters one that is
    /// divisible by `3`:
    ///
    ///     let stream = Counter(howHigh:
10)
    ///         .drop { $0 % 3 != 0 }
    ///     for await number in stream {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "3 4 5 6 7 8 9 10 "
    ///
    /// After the predicate returns
`false`, the sequence never executes it
again,
    /// and from then on the sequence
passes through elements from its
underlying
    /// sequence as-is.
```

```
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
    ///   returns a Boolean value
indicating whether to drop the element
from the
    ///   modified sequence.
    /// - Returns: An asynchronous
sequence that skips over values from the
    ///   base sequence until the
provided closure returns `false`.
    @preconcurrency @inlinable public
func drop(while predicate: @escaping
@Sendable (Self.Element) async -> Bool)
-> AsyncDropWhileSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that contains, in order, the elements of
    /// the base sequence that satisfy
the given predicate.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`filter(_:)` method returns `true` for
even
    /// values and `false` for odd
values, thereby filtering out the odd
```

```
values:
    ///
    ///     let stream = Counter(howHigh:
10)
    ///         .filter { $0 % 2 == 0 }
    ///     for await number in stream {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "2 4 6 8 10 "
    ///
    /// - Parameter isIncluded: A closure
that takes an element of the
    ///   asynchronous sequence as its
argument and returns a Boolean value
    ///   that indicates whether to
include the element in the filtered
sequence.
    /// - Returns: An asynchronous
sequence that contains, in order, the
elements
    ///   of the base sequence that
satisfy the given predicate.
    @preconcurrency @inlinable public
func filter(_ isIncluded: @escaping
@Sendable (Self.Element) async -> Bool)
-> AsyncFilterSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {
```

```
    /// Creates an asynchronous sequence
that concatenates the results of calling
    /// the given transformation with
each element of this sequence.
    ///
    /// Use this method to receive a
single-level asynchronous sequence when
your
    /// transformation produces an
asynchronous sequence for each element.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
transforming closure takes the received
`Int`
    /// and returns a new `Counter` that
counts that high. For example, when the
    /// transform receives `3` from the
base sequence, it creates a new `Counter`
    /// that produces the values `1`,
`2`, and `3`. The `flatMap(_:)` method
    /// "flattens" the resulting
sequence-of-sequences into a single
    /// `AsyncSequence`.
    ///
    ///     let stream = Counter(howHigh:
5)
    ///         .flatMap
{ Counter(howHigh: $0) }
    ///     for await number in stream {
    ///         print(number, terminator:
" ")
```

```
    ///     }
    ///     // Prints "1 1 2 1 2 3 1 2 3
4 1 2 3 4 5 "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns an `AsyncSequence`.
    /// - Returns: A single, flattened
asynchronous sequence that contains all
    ///   elements in all the
asynchronous sequences produced by
`transform`.
    @preconcurrency @inlinable public
func flatMap<SegmentOfResult>(_
transform: @escaping @Sendable
(Self.Element) async -> SegmentOfResult)
-> AsyncFlatMapSequence<Self,
SegmentOfResult> where SegmentOfResult :
AsyncSequence, Self.Failure ==
SegmentOfResult.Failure

    /// Creates an asynchronous sequence
that concatenates the results of calling
    /// the given transformation with
each element of this sequence.
    ///
    /// Use this method to receive a
single-level asynchronous sequence when
your
    /// transformation produces an
asynchronous sequence for each element.
    ///
```

```
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
transforming closure takes the received
`Int`
    /// and returns a new `Counter` that
counts that high. For example, when the
    /// transform receives `3` from the
base sequence, it creates a new `Counter`
    /// that produces the values `1`,
`2`, and `3`. The `flatMap(_:)` method
    /// "flattens" the resulting
sequence-of-sequences into a single
    /// `AsyncSequence`.
    ///
    ///     let stream = Counter(howHigh:
5)
    ///         .flatMap
{ Counter(howHigh: $0) }
    ///     for await number in stream {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "1 1 2 1 2 3 1 2 3
4 1 2 3 4 5 "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns an `AsyncSequence`.
    /// - Returns: A single, flattened
asynchronous sequence that contains all
    ///   elements in all the
```

```
asynchronous sequences produced by
`transform`.
    @preconcurrency @inlinable public
func flatMap<SegmentOfResult>(_
transform: @escaping @Sendable
(Self.Element) async -> SegmentOfResult)
-> AsyncFlatMapSequence<Self,
SegmentOfResult> where SegmentOfResult :
AsyncSequence, SegmentOfResult.Failure ==
Never

    /// Creates an asynchronous sequence
that concatenates the results of calling
    /// the given transformation with
each element of this sequence.
    ///
    /// Use this method to receive a
single-level asynchronous sequence when
your
    /// transformation produces an
asynchronous sequence for each element.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
transforming closure takes the received
`Int`
    /// and returns a new `Counter` that
counts that high. For example, when the
    /// transform receives `3` from the
base sequence, it creates a new `Counter`
    /// that produces the values `1`,
`2`, and `3`. The `flatMap(_:)` method
```

```
    /// "flattens" the resulting
sequence-of-sequences into a single
    /// `AsyncSequence`.
    ///
    ///     let stream = Counter(howHigh:
5)
    ///         .flatMap
{ Counter(howHigh: $0) }
    ///     for await number in stream {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "1 1 2 1 2 3 1 2 3
4 1 2 3 4 5 "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns an `AsyncSequence`.
    /// - Returns: A single, flattened
asynchronous sequence that contains all
    ///   elements in all the
asynchronous sequences produced by
`transform`.
    @preconcurrency @inlinable public
func flatMap<SegmentOfResult>(_
transform: @escaping @Sendable
(Self.Element) async -> SegmentOfResult)
-> AsyncFlatMapSequence<Self,
SegmentOfResult> where SegmentOfResult :
AsyncSequence, Self.Failure == Never,
SegmentOfResult.Failure == Never
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that maps the given closure over the
    /// asynchronous sequence’s elements.
    ///
    /// Use the `map(_:)` method to
transform every element received from a
base
    /// asynchronous sequence. Typically,
you use this to transform from one type
    /// of element to another.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
closure provided to the `map(_:)` method
    /// takes each `Int` and looks up a
corresponding `String` from a
    /// `romanNumeralDict` dictionary.
This means the outer `for await in` loop
    /// iterates over `String` instances
instead of the underlying `Int` values
    /// that `Counter` produces:
    ///
    ///     let romanNumeralDict: [Int:
String] =
    ///         [1: "I", 2: "II", 3:
"III", 5: "V"]
    ///
```

```
    ///     let stream = Counter(howHigh:
5)
    ///         .map
{ romanNumeralDict[$0] ?? "(unknown)" }
    ///     for await numeral in stream {
    ///         print(numeral,
terminator: " ")
    ///     }
    ///     // Prints "I II III (unknown)
V "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns a transformed value
of the
    ///   same or of a different type.
    /// - Returns: An asynchronous
sequence that contains, in order, the
elements
    ///   produced by the `transform`
closure.
    @preconcurrency @inlinable public
func map<Transformed>(_ transform:
@escaping @Sendable (Self.Element) async
-> Transformed) -> AsyncMapSequence<Self,
Transformed>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {
```

```
    /// Returns an asynchronous sequence,
up to the specified maximum length,
    /// containing the initial elements
of the base asynchronous sequence.
    ///
    /// Use `prefix(_:)` to reduce the
number of elements produced by the
    /// asynchronous sequence.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`prefix(_:)` method causes the modified
    /// sequence to pass through the
first six values, then end.
    ///
    ///     for await number in
Counter(howHigh: 10).prefix(6) {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "1 2 3 4 5 6 "
    ///
    /// If the count passed to
`prefix(_:)` exceeds the number of
elements in the
    /// base sequence, the result
contains all of the elements in the
sequence.
    ///
    /// - Parameter count: The maximum
number of elements to return. The value
of
```

```
    ///   `count` must be greater than or
equal to zero.
    /// - Returns: An asynchronous
sequence starting at the beginning of the
    ///   base sequence with at most
`count` elements.
    @inlinable public func prefix(_
count: Int) -> AsyncPrefixSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Returns an asynchronous sequence,
containing the initial, consecutive
    /// elements of the base sequence
that satisfy the given predicate.
    ///
    /// Use `prefix(while:)` to produce
values while elements from the base
    /// sequence meet a condition you
specify. The modified sequence ends when
    /// the predicate closure returns
`false`.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`prefix(while:)` method causes the
modified
    /// sequence to pass along values so
long as they aren't divisible by `2` and
```

```
    /// `3`. Upon reaching `6`, the
sequence ends:
    ///
    ///     let stream = Counter(howHigh:
10)
    ///         .prefix { $0 % 2 != 0 ||
$0 % 3 != 0 }
    ///     for try await number in
stream {
    ///         print(number, terminator:
" ")
    ///     }
    ///     // Prints "1 2 3 4 5 "
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
    ///   returns a Boolean value
indicating whether the element should be
    ///   included in the modified
sequence.
    /// - Returns: An asynchronous
sequence of the initial, consecutive
    ///   elements that satisfy
`predicate`.
    @preconcurrency @inlinable public
func prefix(while predicate: @escaping
@Sendable (Self.Element) async -> Bool)
rethrows ->
AsyncPrefixWhileSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
extension AsyncSequence {

    /// Returns the result of combining
the elements of the asynchronous sequence
    /// using the given closure.
    ///
    /// Use the `reduce(_:_:)` method to
produce a single value from the elements
of
    /// an entire sequence. For example,
you can use this method on an sequence of
    /// numbers to find their sum or
product.
    ///
    /// The `nextPartialResult` closure
executes sequentially with an
accumulating
    /// value initialized to
`initialResult` and each element of the
sequence.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `4`. The
`reduce(_:_:)` method sums the values
    /// received from the asynchronous
sequence.
    ///
    ///     let sum = await
Counter(howHigh: 4)
    ///         .reduce(0) {
    ///             $0 + $1
    ///         }
```

```
    ///     print(sum)
    ///     // Prints "10"
    ///
    ///
    /// - Parameters:
    ///   - initialResult: The value to
use as the initial accumulating value.
    ///     The `nextPartialResult`
closure receives `initialResult` the
first
    ///     time the closure runs.
    ///   - nextPartialResult: A closure
that combines an accumulating value and
    ///     an element of the
asynchronous sequence into a new
accumulating value,
    ///     for use in the next call of
the `nextPartialResult` closure or
    ///     returned to the caller.
    /// - Returns: The final accumulated
value. If the sequence has no elements,
    ///   the result is `initialResult`.
    @inlinable public func
reduce<Result>(_ initialResult: Result, _
nextPartialResult: (_ partialResult:
Result, Self.Element) async throws ->
Result) async rethrows -> Result

    /// Returns the result of combining
the elements of the asynchronous sequence
    /// using the given closure, given a
mutable initial value.
    ///
```

```
    /// Use the `reduce(into:_:)` method to produce a single value from the
    /// elements of an entire sequence. For example, you can use this method on a
    /// sequence of numbers to find their sum or product.
    ///
    /// The `nextPartialResult` closure executes sequentially with an accumulating
    /// value initialized to `initialResult` and each element of the sequence.
    ///
    /// Prefer this method over `reduce(_:_:)` for efficiency when the result is
    /// a copy-on-write type, for example an `Array` or `Dictionary`.
    ///
    /// - Parameters:
    ///   - initialResult: The value to use as the initial accumulating value.
    ///     The `nextPartialResult` closure receives `initialResult` the first
    ///     time the closure executes.
    ///   - nextPartialResult: A closure that combines an accumulating value and
    ///     an element of the asynchronous sequence into a new accumulating value,
    ///     for use in the next call of
```

```
the `nextPartialResult` closure or
    ///     returned to the caller.
    /// - Returns: The final accumulated
value. If the sequence has no elements,
    ///   the result is `initialResult`.
    @inlinable public func
reduce<Result>(into initialResult:
Result, _ updateAccumulatingResult: (_
partialResult: inout Result,
Self.Element) async throws -> Void) async
rethrows -> Result
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Returns a Boolean value that
indicates whether the asynchronous
sequence
    /// contains an element that
satisfies the given predicate.
    ///
    /// You can use the predicate to
check for an element of a type that
doesn’t
    /// conform to the `Equatable`
protocol, or to find an element that
satisfies
    /// a general condition.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
```

```
    /// values from `1` to `10`. The
`contains(where:)` method checks to see
    /// whether the sequence produces a
value divisible by `3`:
    ///
    ///     let containsDivisibleByThree
= await Counter(howHigh: 10)
    ///         .contains { $0 % 3 == 0 }
    ///
print(containsDivisibleByThree)
    ///     // Prints "true"
    ///
    /// The predicate executes each time
the asynchronous sequence produces an
    /// element, until either the
predicate finds a match or the sequence
ends.
    ///
    /// - Parameter predicate: A closure
that takes an element of the asynchronous
    ///   sequence as its argument and
returns a Boolean value that indicates
    ///   whether the passed element
represents a match.
    /// - Returns: `true` if the sequence
contains an element that satisfies
    ///   predicate; otherwise, `false`.
    @inlinable public func contains(where
predicate: (Self.Element) async throws ->
Bool) async rethrows -> Bool

    /// Returns a Boolean value that
indicates whether all elements produced
```

```
by the
    /// asynchronous sequence satisfy the
given predicate.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`allSatisfy(_:)` method checks to see
whether
    /// all elements produced by the
sequence are less than `10`.
    ///
    ///     let allLessThanTen = await
Counter(howHigh: 10)
    ///         .allSatisfy { $0 < 10 }
    ///     print(allLessThanTen)
    ///     // Prints "false"
    ///
    /// The predicate executes each time
the asynchronous sequence produces an
    /// element, until either the
predicate returns `false` or the sequence
ends.
    ///
    /// If the asynchronous sequence is
empty, this method returns `true`.
    ///
    /// - Parameter predicate: A closure
that takes an element of the asynchronous
    ///   sequence as its argument and
returns a Boolean value that indicates
    ///   whether the passed element
satisfies a condition.
```

```
    /// - Returns: `true` if the sequence
contains only elements that satisfy
    ///   `predicate`; otherwise,
`false`.
    @inlinable public func allSatisfy(_
predicate: (Self.Element) async throws ->
Bool) async rethrows -> Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence where
Self.Element : Equatable {

    /// Returns a Boolean value that
indicates whether the asynchronous
sequence
    /// contains the given element.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`contains(_:)` method checks to see
whether
    /// the sequence produces the value
`5`:
    ///
    ///     let containsFive = await
Counter(howHigh: 10)
    ///         .contains(5)
    ///     print(containsFive)
    ///     // Prints "true"
    ///
```

```
    /// - Parameter search: The element
to find in the asynchronous sequence.
    /// - Returns: `true` if the method
found the element in the asynchronous
    ///   sequence; otherwise, `false`.
    @inlinable public func contains(_
search: Self.Element) async rethrows ->
Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Returns the first element of the
sequence that satisfies the given
    /// predicate.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`first(where:)` method returns the first
    /// member of the sequence that's
evenly divisible by both `2` and `3`.
    ///
    ///     let divisibleBy2And3 = await
Counter(howHigh: 10)
    ///         .first { $0 % 2 == 0 &&
$0 % 3 == 0 }
    ///     print(divisibleBy2And3 ??
"none")
    ///     // Prints "6"
    ///
```

```
    /// The predicate executes each time
the asynchronous sequence produces an
    /// element, until either the
predicate finds a match or the sequence
ends.
    ///
    /// - Parameter predicate: A closure
that takes an element of the asynchronous
    ///  sequence as its argument and
returns a Boolean value that indicates
    ///  whether the element is a match.
    /// - Returns: The first element of
the sequence that satisfies `predicate`,
    ///   or `nil` if there is no element
that satisfies `predicate`.
    @inlinable public func first(where
predicate: (Self.Element) async throws ->
Bool) async rethrows -> Self.Element?
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Returns the minimum element in
the asynchronous sequence, using the
given
    /// predicate as the comparison
between elements.
    ///
    /// Use this method when the
asynchronous sequence's values don't
conform
```

```
    /// to `Comparable`, or when you want
to apply a custom ordering to the
    /// sequence.
    ///
    /// The predicate must be a *strict
weak ordering* over the elements. That
is,
    /// for any elements `a`, `b`, and
`c`, the following conditions must hold:
    ///
    /// - `areInIncreasingOrder(a, a)` is
always `false`. (Irreflexivity)
    /// - If `areInIncreasingOrder(a, b)`
and `areInIncreasingOrder(b, c)` are
    ///   both `true`, then
`areInIncreasingOrder(a, c)` is also
    ///   `true`. (Transitive
comparability)
    /// - Two elements are *incomparable*
if neither is ordered before the other
    ///   according to the predicate. If
`a` and `b` are incomparable, and `b`
    ///   and `c` are incomparable, then
`a` and `c` are also incomparable.
    ///   (Transitive incomparability)
    ///
    /// The following example uses an
enumeration of playing cards ranks,
`Rank`,
    /// which ranges from `ace` (low) to
`king` (high). An asynchronous sequence
    /// called `RankCounter` produces all
elements of the array. The predicate
```

```
    /// provided to the `min(by:)` method
sorts ranks based on their `rawValue`:
    ///
    ///     enum Rank: Int {
    ///         case ace = 1, two, three,
four, five, six, seven, eight, nine, ten,
jack, queen, king
    ///     }
    ///
    ///     let min = await RankCounter()
    ///         .min { $0.rawValue <
$1.rawValue }
    ///     print(min ?? "none")
    ///     // Prints "ace"
    ///
    /// - Parameter areInIncreasingOrder:
A predicate that returns `true` if its
    ///   first argument should be
ordered before its second argument;
otherwise,
    ///   `false`.
    /// - Returns: The sequence's minimum
element, according to
    ///   `areInIncreasingOrder`. If the
sequence has no elements, returns `nil`.
    @warn_unqualified_access
    @inlinable public func min(by
areInIncreasingOrder: (Self.Element,
Self.Element) async throws -> Bool) async
rethrows -> Self.Element?

    /// Returns the maximum element in
the asynchronous sequence, using the
```

```
given
    /// predicate as the comparison
between elements.
    ///
    /// Use this method when the
asynchronous sequence's values don't
conform
    /// to `Comparable`, or when you want
to apply a custom ordering to the
    /// sequence.
    ///
    /// The predicate must be a *strict
weak ordering* over the elements. That
is,
    /// for any elements `a`, `b`, and
`c`, the following conditions must hold:
    ///
    /// - `areInIncreasingOrder(a, a)` is
always `false`. (Irreflexivity)
    /// - If `areInIncreasingOrder(a, b)`
and `areInIncreasingOrder(b, c)` are
    ///   both `true`, then
`areInIncreasingOrder(a, c)` is also
    ///   `true`. (Transitive
comparability)
    /// - Two elements are *incomparable*
if neither is ordered before the other
    ///   according to the predicate. If
`a` and `b` are incomparable, and `b`
    ///   and `c` are incomparable, then
`a` and `c` are also incomparable.
    ///   (Transitive incomparability)
    ///
```

```
    /// The following example uses an
enumeration of playing cards ranks,
`Rank`,
    /// which ranges from `ace` (low) to
`king` (high). An asynchronous sequence
    /// called `RankCounter` produces all
elements of the array. The predicate
    /// provided to the `max(by:)` method
sorts ranks based on their `rawValue`:
    ///
    ///     enum Rank: Int {
    ///         case ace = 1, two, three,
four, five, six, seven, eight, nine, ten,
jack, queen, king
    ///     }
    ///
    ///     let max = await RankCounter()
    ///         .max { $0.rawValue <
$1.rawValue }
    ///     print(max ?? "none")
    ///     // Prints "king"
    ///
    /// - Parameter areInIncreasingOrder:
A predicate that returns `true` if its
    ///   first argument should be
ordered before its second argument;
otherwise,
    ///   `false`.
    /// - Returns: The sequence's minimum
element, according to
    ///   `areInIncreasingOrder`. If the
sequence has no elements, returns `nil`.
    @warn_unqualified_access
```

```
    @inlinable public func max(by
areInIncreasingOrder: (Self.Element,
Self.Element) async throws -> Bool) async
rethrows -> Self.Element?
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence where
Self.Element : Comparable {

    /// Returns the minimum element in an
asynchronous sequence of comparable
    /// elements.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`min()` method returns the minimum value
    /// of the sequence.
    ///
    ///     let min = await
Counter(howHigh: 10)
    ///         .min()
    ///     print(min ?? "none")
    ///     // Prints "1"
    ///
    /// - Returns: The sequence's minimum
element. If the sequence has no
    ///   elements, returns `nil`.
    @warn_unqualified_access
    @inlinable public func min() async
rethrows -> Self.Element?
```

```
    /// Returns the maximum element in an
asynchronous sequence of comparable
    /// elements.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`max()` method returns the max value
    /// of the sequence.
    ///
    ///     let max = await
Counter(howHigh: 10)
    ///           .max()
    ///     print(max ?? "none")
    ///     // Prints "10"
    ///
    /// - Returns: The sequence's maximum
element. If the sequence has no
    ///   elements, returns `nil`.
    @warn_unqualified_access
    @inlinable public func max() async
rethrows -> Self.Element?
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that maps an error-throwing closure over
    /// the base sequence's elements,
omitting results that don't return a
```

```
value.
    ///
    /// Use the `compactMap(_:)` method
to transform every element received from
    /// a base asynchronous sequence,
while also discarding any `nil` results
    /// from the closure. Typically, you
use this to transform from one type of
    /// element to another.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
closure provided to the `compactMap(_:)`
    /// method takes each `Int` and looks
up a corresponding `String` from a
    /// `romanNumeralDict` dictionary.
Since there is no key for `4`, the
closure
    /// returns `nil` in this case, which
`compactMap(_:)` omits from the
    /// transformed asynchronous
sequence. When the value is `5`, the
closure
    /// throws `MyError`, terminating the
sequence.
    ///
    ///     let romanNumeralDict: [Int:
String] =
    ///         [1: "I", 2: "II", 3:
"III", 5: "V"]
    ///
    ///     do {
```

```
    ///         let stream =
Counter(howHigh: 5)
    ///             .compactMap { (value)
throws -> String? in
    ///                 if value == 5 {
    ///                     throw
MyError()
    ///                 }
    ///                 return
romanNumeralDict[value]
    ///             }
    ///         for try await numeral in
stream {
    ///             print(numeral,
terminator: " ")
    ///         }
    ///     } catch {
    ///         print("Error: \(error)")
    ///     }
    ///     // Prints "I II III Error:
MyError() "
    ///
    /// - Parameter transform: An error-
throwing mapping closure. `transform`
    ///   accepts an element of this
sequence as its parameter and returns a
    ///   transformed value of the same
or of a different type. If `transform`
    ///   throws an error, the sequence
ends.
    /// - Returns: An asynchronous
sequence that contains, in order, the
    ///   non-`nil` elements produced by
```

```
the `transform` closure. The sequence
    ///   ends either when the base
sequence ends or when `transform` throws
an
    ///   error.
    @preconcurrency @inlinable public
func compactMap<ElementOfResult>(_
transform: @escaping @Sendable
(Self.Element) async throws ->
ElementOfResult?) ->
AsyncThrowingCompactMapSequence<Self,
ElementOfResult>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Omits elements from the base
sequence until a given error-throwing
closure
    /// returns false, after which it
passes through all remaining elements.
    ///
    /// Use `drop(while:)` to omit
elements from an asynchronous sequence
until
    /// the element received meets a
condition you specify. If the closure you
    /// provide throws an error, the
sequence produces no elements and throws
    /// the error instead.
    ///
```

```
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
predicate passed to the `drop(while:)`
    /// method throws an error if it
encounters an even number, and otherwise
    /// returns `true` while it receives
elements less than `5`. Because the
    /// predicate throws when it receives
`2` from the base sequence, this example
    /// throws without ever printing
anything.
    ///
    ///     do {
    ///         let stream =
Counter(howHigh: 10)
    ///             .drop {
    ///                 if $0 % 2 == 0 {
    ///                     throw
EvenError()
    ///                 }
    ///                 return $0 < 5
    ///             }
    ///         for try await number in
stream {
    ///             print(number)
    ///         }
    ///     } catch {
    ///         print(error)
    ///     }
    ///     // Prints "EvenError()"
    ///
    /// After the predicate returns
```

```
`false`, the sequence never executes it
again,
    /// and from then on the sequence
passes through elements from its
underlying
    /// sequence. A predicate that throws
an error also never executes again.
    ///
    /// - Parameter predicate: An error-
throwing closure that takes an element as
    ///   a parameter and returns a
Boolean value indicating whether to drop
the
    ///   element from the modified
sequence.
    /// - Returns: An asynchronous
sequence that skips over values until the
    ///   provided closure returns
`false` or throws an error.
    @preconcurrency @inlinable public
func drop(while predicate: @escaping
@Sendable (Self.Element) async throws ->
Bool) ->
AsyncThrowingDropWhileSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that contains, in order, the elements of
    /// the base sequence that satisfy
```

```
the given error-throwing predicate.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`filter(_:)` method returns `true` for
even
    /// values and `false` for odd
values, thereby filtering out the odd
values,
    /// but also throws an error for
values divisible by 5:
    ///
    ///     do {
    ///         let stream =
Counter(howHigh: 10)
    ///             .filter {
    ///                 if $0 % 5 == 0 {
    ///                     throw
MyError()
    ///                 }
    ///                 return $0 % 2 ==
0
    ///             }
    ///         for try await number in
stream {
    ///             print(number,
terminator: " ")
    ///         }
    ///     } catch {
    ///         print("Error: \(error)")
    ///     }
    ///     // Prints "2 4 Error:
```

```
MyError() "
    ///
    /// - Parameter isIncluded: An error-
throwing closure that takes an element
    ///   of the asynchronous sequence as
its argument and returns a Boolean value
    ///   that indicates whether to
include the element in the filtered
sequence.
    /// - Returns: An asynchronous
sequence that contains, in order, the
elements
    ///   of the base sequence that
satisfy the given predicate. If the
predicate
    ///   throws an error, the sequence
contains only values produced prior to
    ///   the error.
    @preconcurrency @inlinable public
func filter(_ isIncluded: @escaping
@Sendable (Self.Element) async throws ->
Bool) ->
AsyncThrowingFilterSequence<Self>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that concatenates the results of calling
    /// the given error-throwing
transformation with each element of this
```

```
    /// sequence.
    ///
    /// Use this method to receive a
single-level asynchronous sequence when
your
    /// transformation produces an
asynchronous sequence for each element.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
transforming closure takes the received
`Int`
    /// and returns a new `Counter` that
counts that high. For example, when the
    /// transform receives `3` from the
base sequence, it creates a new `Counter`
    /// that produces the values `1`,
`2`, and `3`. The `flatMap(_:)` method
    /// "flattens" the resulting
sequence-of-sequences into a single
    /// `AsyncSequence`. However, when
the closure receives `4`, it throws an
    /// error, terminating the sequence.
    ///
    ///     do {
    ///         let stream =
Counter(howHigh: 5)
    ///             .flatMap { (value) ->
Counter in
    ///                 if value == 4 {
    ///                     throw
MyError()
```

```
    ///                 }
    ///                 return
Counter(howHigh: value)
    ///             }
    ///         for try await number in
stream {
    ///             print(number,
terminator: " ")
    ///         }
    ///     } catch {
    ///         print(error)
    ///     }
    ///     // Prints "1 1 2 1 2 3
MyError() "
    ///
    /// - Parameter transform: An error-
throwing mapping closure. `transform`
    ///   accepts an element of this
sequence as its parameter and returns an
    ///   `AsyncSequence`. If `transform`
throws an error, the sequence ends.
    /// - Returns: A single, flattened
asynchronous sequence that contains all
    ///   elements in all the
asynchronous sequences produced by
`transform`. The
    ///   sequence ends either when the
last sequence created from the last
    ///   element from base sequence
ends, or when `transform` throws an
error.
    @preconcurrency @inlinable public
func flatMap<SegmentOfResult>(_
```

```
transform: @escaping @Sendable
(Self.Element) async throws ->
SegmentOfResult) ->
AsyncThrowingFlatMapSequence<Self,
SegmentOfResult> where SegmentOfResult :
AsyncSequence
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Creates an asynchronous sequence
that maps the given error-throwing
    /// closure over the asynchronous
sequence's elements.
    ///
    /// Use the `map(_:)` method to
transform every element received from a
base
    /// asynchronous sequence. Typically,
you use this to transform from one type
    /// of element to another.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `5`. The
closure provided to the `map(_:)` method
    /// takes each `Int` and looks up a
corresponding `String` from a
    /// `romanNumeralDict` dictionary.
This means the outer `for await in` loop
    /// iterates over `String` instances
```

```
instead of the underlying `Int` values
    /// that `Counter` produces. Also,
the dictionary doesn't provide a key for
    /// `4`, and the closure throws an
error for any key it can't look up, so
    /// receiving this value from
`Counter` ends the modified sequence with
an
    /// error.
    ///
    ///     let romanNumeralDict: [Int:
String] =
    ///         [1: "I", 2: "II", 3:
"III", 5: "V"]
    ///
    ///     do {
    ///         let stream =
Counter(howHigh: 5)
    ///             .map { (value) throws
-> String in
    ///                 guard let roman =
romanNumeralDict[value] else {
    ///                     throw
MyError()
    ///                 }
    ///                 return roman
    ///             }
    ///         for try await numeral in
stream {
    ///             print(numeral,
terminator: " ")
    ///         }
    ///     } catch {
```

```
    ///         print("Error: \(error)")
    ///     }
    ///     // Prints "I II III Error:
MyError() "
    ///
    /// - Parameter transform: A mapping
closure. `transform` accepts an element
    ///   of this sequence as its
parameter and returns a transformed value
of the
    ///   same or of a different type.
`transform` can also throw an error,
which
    ///   ends the transformed sequence.
    /// - Returns: An asynchronous
sequence that contains, in order, the
elements
    ///   produced by the `transform`
closure.
    @preconcurrency @inlinable public
func map<Transformed>(_ transform:
@escaping @Sendable (Self.Element) async
throws -> Transformed) ->
AsyncThrowingMapSequence<Self,
Transformed>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncSequence {

    /// Returns an asynchronous sequence,
containing the initial, consecutive
```

```
    /// elements of the base sequence
that satisfy the given error-throwing
    /// predicate.
    ///
    /// Use `prefix(while:)` to produce
values while elements from the base
    /// sequence meet a condition you
specify. The modified sequence ends when
    /// the predicate closure returns
`false` or throws an error.
    ///
    /// In this example, an asynchronous
sequence called `Counter` produces `Int`
    /// values from `1` to `10`. The
`prefix(_:)` method causes the modified
    /// sequence to pass through values
less than `8`, but throws an
    /// error when it receives a value
that's divisible by `5`:
    ///
    ///     do {
    ///         let stream = try
Counter(howHigh: 10)
    ///             .prefix {
    ///                 if $0 % 5 == 0 {
    ///                     throw
MyError()
    ///                 }
    ///                 return $0 < 8
    ///             }
    ///         for try await number in
stream {
    ///             print(number,
```

```
terminator: " ")
    ///         }
    ///     } catch {
    ///         print("Error: \(error)")
    ///     }
    ///     // Prints "1 2 3 4 Error:
MyError() "
    ///
    /// - Parameter predicate: A error-
throwing closure that takes an element of
    ///   the asynchronous sequence as
its argument and returns a Boolean value
    ///   that indicates whether to
include the element in the modified
sequence.
    /// - Returns: An asynchronous
sequence that contains, in order, the
elements
    ///   of the base sequence that
satisfy the given predicate. If the
predicate
    ///   throws an error, the sequence
contains only values produced prior to
    ///   the error.
    @preconcurrency @inlinable public
func prefix(while predicate: @escaping
@Sendable (Self.Element) async throws ->
Bool) rethrows ->
AsyncThrowingPrefixWhileSequence<Self>
}

/// An asynchronous sequence generated
from a closure that calls a continuation
```

```
/// to produce new elements.
///
/// `AsyncStream` conforms to
`AsyncSequence`, providing a convenient
way to
/// create an asynchronous sequence
without manually implementing an
/// asynchronous iterator. In particular,
an asynchronous stream is well-suited
/// to adapt callback- or delegation-
based APIs to participate with
/// `async`-`await`.
///
/// You initialize an `AsyncStream` with
a closure that receives an
/// `AsyncStream.Continuation`. Produce
elements in this closure, then provide
/// them to the stream by calling the
continuation's `yield(_:)` method. When
/// there are no further elements to
produce, call the continuation's
/// `finish()` method. This causes the
sequence iterator to produce a `nil`,
/// which terminates the sequence. The
continuation conforms to `Sendable`,
which permits
/// calling it from concurrent contexts
external to the iteration of the
/// `AsyncStream`.
///
/// An arbitrary source of elements can
produce elements faster than they are
/// consumed by a caller iterating over
```

```
them. Because of this, `AsyncStream`
/// defines a buffering behavior,
allowing the stream to buffer a specific
/// number of oldest or newest elements.
By default, the buffer limit is
/// `Int.max`, which means the value is
unbounded.
///
/// ### Adapting Existing Code to Use
Streams
///
/// To adapt existing callback code to
use `async`-`await`, use the callbacks
/// to provide values to the stream, by
using the continuation's `yield(_:)`
/// method.
///
/// Consider a hypothetical
`QuakeMonitor` type that provides callers
with
/// `Quake` instances every time it
detects an earthquake. To receive
callbacks,
/// callers set a custom closure as the
value of the monitor's
/// `quakeHandler` property, which the
monitor calls back as necessary.
///
///     class QuakeMonitor {
///         var quakeHandler: ((Quake) ->
Void)?
///
///         func startMonitoring() {…}
```

```
///         func stopMonitoring() {…}
///     }
///
/// To adapt this to use `async`-`await`,
extend the `QuakeMonitor` to add a
/// `quakes` property, of type
`AsyncStream<Quake>`. In the getter for
this
/// property, return an `AsyncStream`,
whose `build` closure -- called at
/// runtime to create the stream -- uses
the continuation to perform the
/// following steps:
///
/// 1. Creates a `QuakeMonitor` instance.
/// 2. Sets the monitor's `quakeHandler`
property to a closure that receives
/// each `Quake` instance and forwards it
to the stream by calling the
/// continuation's `yield(_:)` method.
/// 3. Sets the continuation's
`onTermination` property to a closure
that
/// calls `stopMonitoring()` on the
monitor.
/// 4. Calls `startMonitoring` on the
`QuakeMonitor`.
///
/// ```
/// extension QuakeMonitor {
///
///     static var quakes:
AsyncStream<Quake> {
```

```
///         AsyncStream { continuation in
///             let monitor =
QuakeMonitor()
///             monitor.quakeHandler =
{ quake in
///
continuation.yield(quake)
///             }
///
continuation.onTermination = { @Sendable
_ in
///
monitor.stopMonitoring()
///             }
///             monitor.startMonitoring()
///         }
///     }
/// }
/// ```
///
/// Because the stream is an
`AsyncSequence`, the call point can use
the
/// `for`-`await`-`in` syntax to process
each `Quake` instance as the stream
/// produces it:
///
///     for await quake in
QuakeMonitor.quakes {
///         print("Quake: \(quake.date)")
///     }
///     print("Stream finished.")
///
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct AsyncStream<Element> {

    /// A mechanism to interface between
synchronous code and an asynchronous
    /// stream.
    ///
    /// The closure you provide to the
`AsyncStream` in
    /// `init(_:bufferingPolicy:_:)`
receives an instance of this type when
    /// invoked. Use this continuation to
provide elements to the stream by
    /// calling one of the `yield`
methods, then terminate the stream
normally by
    /// calling the `finish()` method.
    ///
    /// - Note: Unlike other
continuations in Swift,
`AsyncStream.Continuation`
    /// supports escaping.
    public struct Continuation : Sendable
{

        /// A type that indicates how the
stream terminated.
        ///
        /// The `onTermination` closure
receives an instance of this type.
        public enum Termination {
```

```
            /// The stream finished as a
result of calling the continuation's
            ///  `finish` method.
            case finished

            /// The stream finished as a
result of cancellation.
            case cancelled

            /// Returns a Boolean value
indicating whether two values are equal.
            ///
            /// Equality is the inverse
of inequality. For any values `a` and
`b`,
            /// `a == b` implies that
`a != b` is `false`.
            ///
            /// - Parameters:
            ///   - lhs: A value to
compare.
            ///   - rhs: Another value to
compare.
            public static func == (a:
AsyncStream<Element>.Continuation.Termina
tion, b:
AsyncStream<Element>.Continuation.Termina
tion) -> Bool

            /// Hashes the essential
components of this value by feeding them
into the
            /// given hasher.
```

```
            ///
            /// Implement this method to
conform to the `Hashable` protocol. The
            /// components used for
hashing must be the same as the
components compared
            /// in your type's `==`
operator implementation. Call
`hasher.combine(_:)`
            /// with each of these
components.
            ///
            /// - Important: In your
implementation of `hash(into:)`,
            ///   don't call `finalize()`
on the `hasher` instance provided,
            ///   or replace it with a
different instance.
            ///   Doing so may become a
compile-time error in the future.
            ///
            /// - Parameter hasher: The
hasher to use when combining the
components
            ///   of this instance.
            public func hash(into hasher:
inout Hasher)

            /// The hash value.
            ///
            /// Hash values are not
guaranteed to be equal across different
executions of
```

```
            /// your program. Do not save
hash values to use during a future
execution.
            ///
            /// - Important: `hashValue`
is deprecated as a `Hashable`
requirement. To
            ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
            ///   The compiler provides
an implementation for `hashValue` for
you.
            public var hashValue: Int {
get }
        }

        /// A type that indicates the
result of yielding a value to a client,
by
        /// way of the continuation.
        ///
        /// The various `yield` methods
of `AsyncStream.Continuation` return this
        /// type to indicate the success
or failure of yielding an element to the
        /// continuation.
        public enum YieldResult {

            /// The stream successfully
enqueued the element.
            ///
            /// This value represents the
```

```
successful enqueueing of an element,
whether
            /// the stream buffers the
element or delivers it immediately to a
pending
            /// call to `next()`. The
associated value `remaining` is a hint
that
            /// indicates the number of
remaining slots in the buffer at the time
of
            /// the `yield` call.
            ///
            /// - Note: From a thread
safety point of view, `remaining` is a
lower bound
            /// on the number of
remaining slots. This is because a
subsequent call
            /// that uses the `remaining`
value could race on the consumption of
            /// values from the stream.
            case enqueued(remaining: Int)

            /// The stream didn't enqueue
the element because the buffer was full.
            ///
            /// The associated element
for this case is the element dropped by
the stream.
            case dropped(Element)

            /// The stream didn't enqueue
```

```
the element because the stream was in a
            /// terminal state.
            ///
            /// This indicates the stream
terminated prior to calling `yield`,
either
            /// because the stream
finished normally or through
cancellation.
            case terminated
        }

        /// A strategy that handles
exhaustion of a buffer's capacity.
        public enum BufferingPolicy {

            /// Continue to add to the
buffer, without imposing a limit on the
number
            /// of buffered elements.
            case unbounded

            /// When the buffer is full,
discard the newly received element.
            ///
            /// This strategy enforces
keeping at most the specified number of
oldest
            /// values.
            case bufferingOldest(Int)

            /// When the buffer is full,
discard the oldest element in the buffer.
```

```
            ///
            /// This strategy enforces
keeping at most the specified number of
newest
            /// values.
            case bufferingNewest(Int)
        }

        /// Resume the task awaiting the
next iteration point by having it return
        /// normally from its suspension
point with a given element.
        ///
        /// - Parameter value: The value
to yield from the continuation.
        /// - Returns: A `YieldResult`
that indicates the success or failure of
the
        ///   yield operation.
        ///
        /// If nothing is awaiting the
next value, this method attempts to
buffer the
        /// result's element.
        ///
        /// This can be called more than
once and returns to the caller
immediately
        /// without blocking for any
awaiting consumption from the iteration.
        @discardableResult
        public func yield(_ value:
sending Element) ->
```

```
AsyncStream<Element>.Continuation.YieldResult

        /// Resume the task awaiting the next iteration point by having it return
        /// nil, which signifies the end of the iteration.
        ///
        /// Calling this function more than once has no effect. After calling
        /// finish, the stream enters a terminal state and doesn't produce any
        /// additional elements.
        public func finish()

        /// A callback to invoke when canceling iteration of an asynchronous
        /// stream.
        ///
        /// If an `onTermination` callback is set, using task cancellation to
        /// terminate iteration of an `AsyncStream` results in a call to this
        /// callback.
        ///
        /// Canceling an active iteration invokes the `onTermination` callback
        /// first, then resumes by yielding `nil`. This means that you can perform
        /// needed cleanup in the cancellation handler. After reaching a
```

```
terminal
        /// state as a result of
cancellation, the `AsyncStream` sets the
callback
        /// to `nil`.
        public var onTermination:
(@Sendable
(AsyncStream<Element>.Continuation.Termin
ation) -> Void)? { get nonmutating set }
    }

    /// Constructs an asynchronous stream
for an element type, using the
    /// specified buffering policy and
element-producing closure.
    ///
    /// - Parameters:
    ///     - elementType: The type of
element the `AsyncStream` produces.
    ///     - bufferingPolicy: A
`Continuation.BufferingPolicy` value to
    ///       set the stream's buffering
behavior. By default, the stream buffers
an
    ///       unlimited number of
elements. You can also set the policy to
buffer a
    ///       specified number of oldest
or newest elements.
    ///     - build: A custom closure that
yields values to the
    ///       `AsyncStream`. This closure
receives an `AsyncStream.Continuation`
```

```
    ///       instance that it uses to
provide elements to the stream and
terminate the
    ///       stream when finished.
    ///
    /// The `AsyncStream.Continuation`
received by the `build` closure is
    /// appropriate for use in concurrent
contexts. It is thread safe to send and
    /// finish; all calls to the
continuation are serialized. However,
calling
    /// this from multiple concurrent
contexts could result in out-of-order
    /// delivery.
    ///
    /// The following example shows an
`AsyncStream` created with this
    /// initializer that produces 100
random numbers on a one-second interval,
    /// calling `yield(_:)` to deliver
each element to the awaiting call point.
    /// When the `for` loop exits, the
stream finishes by calling the
    /// continuation's `finish()` method.
    ///
    ///     let stream =
AsyncStream<Int>(Int.self,
    ///
bufferingPolicy: .bufferingNewest(5))
{ continuation in
    ///         Task.detached {
    ///             for _ in 0..<100 {
```

```
    ///                     await
Task.sleep(1 * 1_000_000_000)
    ///
continuation.yield(Int.random(in:
1...10))
    ///             }
    ///             continuation.finish()
    ///         }
    ///     }
    ///
    ///     // Call point:
    ///     for await random in stream {
    ///         print(random)
    ///     }
    ///
    public init(_ elementType:
Element.Type = Element.self,
bufferingPolicy limit:
AsyncStream<Element>.Continuation.Bufferi
ngPolicy = .unbounded, _ build:
(AsyncStream<Element>.Continuation) ->
Void)

    /// Constructs an asynchronous stream
from a given element-producing
    /// closure, with an optional closure
to handle cancellation.
    ///
    /// - Parameters:
    ///   - produce: A closure that
asynchronously produces elements for the
    ///     stream.
    ///   - onCancel: A closure to
```

```
execute when canceling the stream's task.
    ///
    /// Use this convenience initializer
when you have an asynchronous function
    /// that can produce elements for the
stream, and don't want to invoke
    /// a continuation manually. This
initializer "unfolds" your closure into
    /// an asynchronous stream. The
created stream handles conformance
    /// to the `AsyncSequence` protocol
automatically, including termination
    /// (either by cancellation or by
returning `nil` from the closure to
finish
    /// iteration).
    ///
    /// The following example shows an
`AsyncStream` created with this
    /// initializer that produces random
numbers on a one-second interval. This
    /// example uses the Swift multiple
trailing closure syntax, which omits
    /// the `unfolding` parameter label.
    ///
    ///     let stream = AsyncStream<Int>
{
    ///         await Task.sleep(1 *
1_000_000_000)
    ///         return Int.random(in:
1...10)
    ///     } onCancel: { @Sendable () in
print("Canceled.") }
```

```
    ///
    ///     // Call point:
    ///     for await random in stream {
    ///         print(random)
    ///     }
    ///
    ///
    @preconcurrency public init(unfolding
produce: @escaping @Sendable () async ->
Element?, onCancel: (@Sendable () ->
Void)? = nil)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncStream : AsyncSequence {

    /// The asynchronous iterator for
iterating an asynchronous stream.
    ///
    /// This type doesn't conform to
`Sendable`. Don't use it from multiple
    /// concurrent contexts. It is a
programmer error to invoke `next()` from
a
    /// concurrent context that contends
with another such call, which
    /// results in a call to
`fatalError()`.
    public struct Iterator :
AsyncIteratorProtocol {

        /// The next value from the
```

```
asynchronous stream.
        ///
        /// When `next()` returns `nil`,
this signifies the end of the
        /// `AsyncStream`.
        ///
        /// It is a programmer error to
invoke `next()` from a
        /// concurrent context that
contends with another such call, which
        /// results in a call to
`fatalError()`.
        ///
        /// If you cancel the task this
iterator is running in while `next()` is
        /// awaiting a value, the
`AsyncStream` terminates. In this case,
`next()`
        /// might return `nil`
immediately, or return `nil` on
subsequent calls.
        public mutating func next() async
-> Element?

        /// The next value from the
asynchronous stream.
        ///
        /// When `next()` returns `nil`,
this signifies the end of the
        /// `AsyncStream`.
        ///
        /// It is a programmer error to
invoke `next()` from a concurrent
```

```
        /// context that contends with
another such call, which results in a
call to
        /// `fatalError()`.
        ///
        /// If you cancel the task this
iterator is running in while `next()`
        /// is awaiting a value, the
`AsyncStream` terminates. In this case,
        /// `next()` might return `nil`
immediately, or return `nil` on
        /// subsequent calls.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public mutating func
next(isolation actor: isolated (any
Actor)?) async -> Element?
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    public func makeAsyncIterator() ->
AsyncStream<Element>.Iterator

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
    public typealias AsyncIterator =
AsyncStream<Element>.Iterator
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncStream {

    /// Initializes a new ``AsyncStream``
and an ``AsyncStream/Continuation``.
    ///
    /// - Parameters:
    ///   - elementType: The element type
of the stream.
    ///   - limit: The buffering policy
that the stream should use.
    /// - Returns: A tuple containing the
stream and its continuation. The
continuation should be passed to the
    /// producer while the stream should
be passed to the consumer.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public static func makeStream(of
elementType: Element.Type = Element.self,
bufferingPolicy limit:
AsyncStream<Element>.Continuation.Bufferi
ngPolicy = .unbounded) -> (stream:
AsyncStream<Element>, continuation:
AsyncStream<Element>.Continuation)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
extension AsyncStream : @unchecked
Sendable where Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncStream.Continuation {

    /// Resume the task awaiting the next
iteration point by having it return
    /// normally from its suspension
point with a given result's success
value.
    ///
    /// - Parameter result: A result to
yield from the continuation.
    /// - Returns: A `YieldResult` that
indicates the success or failure of the
    ///   yield operation.
    ///
    /// If nothing is awaiting the next
value, the method attempts to buffer the
    /// result's element.
    ///
    /// If you call this method
repeatedly, each call returns
immediately, without
    /// blocking for any awaiting
consumption from the iteration.
    @discardableResult
    public func yield(with result:
sending Result<Element, Never>) ->
AsyncStream<Element>.Continuation.YieldRe
```

```
sult

    /// Resume the task awaiting the next
iteration point by having it return
    /// normally from its suspension
point.
    ///
    /// - Returns: A `YieldResult` that
indicates the success or failure of the
    ///   yield operation.
    ///
    /// Use this method with
`AsyncStream` instances whose `Element`
type is
    /// `Void`. In this case, the
`yield()` call unblocks the awaiting
    /// iteration; there is no value to
return.
    ///
    /// If you call this method
repeatedly, each call returns
immediately, without
    /// blocking for any awaiting
consumption from the iteration.
    @discardableResult
    public func yield() ->
AsyncStream<Element>.Continuation.YieldRe
sult where Element == ()
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
```

```
AsyncStream.Continuation.Termination :
Equatable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncStream.Continuation.Termination :
Hashable {
}

/// An asynchronous sequence that maps an
error-throwing closure over the base
/// sequence's elements, omitting results
that don't return a value.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingCompactMapSequence<Base,
ElementOfResult> where Base :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingCompactMapSequence
: AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The compact map sequence produces
whatever type of element its
```

```
    /// transforming closure produces.
    public typealias Element =
ElementOfResult

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The compact map sequence produces
errors from either the base
    /// sequence or the transforming
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingCompactMapSequence<Base,
ElementOfResult>.Iterator

    /// The iterator that produces
elements of the compact map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        public typealias Element =
ElementOfResult

        /// Produces the next element in
the compact map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
```

```
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` calls the
        /// transforming closure on the
received element, returning it if the
        /// transform returns a non-`nil`
value. If the transform returns `nil`,
        /// this method continues to wait
for further elements until it gets one
        /// that transforms to a non-
`nil` value. If calling the closure
throws an
        /// error, the sequence ends and
`next()` rethrows the error.
        @inlinable public mutating func
next() async throws -> ElementOfResult?

        /// Produces the next element in
the compact map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
        /// returns `nil`, `next()`
returns `nil`. Otherwise, `next()`
        /// calls the transforming
closure on the received element,
returning it if
        /// the transform returns a non-
`nil` value. If the transform returns
`nil`,
        /// this method continues to wait
for further elements until it gets one
        /// that transforms to a non-
`nil` value. If calling the closure
```

```
throws an
        /// error, the sequence ends and
`next()` rethrows the error.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws -> ElementOfResult?
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingCompactMapSequence<Base,
ElementOfResult>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingCompactMapSequence
: @unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
extension
AsyncThrowingCompactMapSequence.Iterator
: @unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence which omits
elements from the base sequence until a
/// given error-throwing closure returns
false, after which it passes through
/// all remaining elements.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingDropWhileSequence<Base>
where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingDropWhileSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The drop-while sequence produces
whatever type of element its base
    /// sequence produces.
    public typealias Element =
Base.Element
```

```
    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The drop-while sequence produces
errors from either the base
    /// sequence or the filtering
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingDropWhileSequence<Base>.Iter
ator

    /// The iterator that produces
elements of the drop-while sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the drop-while sequence.
        ///
        /// This iterator calls `next()`
on its base iterator and evaluates the
        /// result with the `predicate`
closure. As long as the predicate returns
        /// `true`, this method returns
`nil`. After the predicate returns
`false`,
        /// for a value received from the
base iterator, this method returns that
```

```
        /// value. After that, the
iterator returns values received from its
        /// base iterator as-is, and
never executes the predicate closure
again.
        /// If calling the closure throws
an error, the sequence ends and `next()`
        /// rethrows the error.
        @inlinable public mutating func
next() async throws -> Base.Element?

        /// Produces the next element in
the drop-while sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator
and
        /// evaluates the result with the
`predicate` closure. As long as the
        /// predicate returns `true`,
this method returns `nil`. After the
predicate
        /// returns `false`, for a value
received from the base iterator, this
        /// method returns that value.
After that, the iterator returns values
        /// received from its base
iterator as-is, and never executes the
predicate
        /// closure again.  If calling
the closure throws an error, the sequence
        /// ends and `next(isolation:)`
rethrows the error.
```

```
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingDropWhileSequence<Base>.Iter
ator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingDropWhileSequence :
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingDropWhileSequence.Iterator :
@unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence that
contains, in order, the elements of
/// the base sequence that satisfy the
given error-throwing predicate.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingFilterSequence<Base> where
Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingFilterSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The filter sequence produces
whatever type of element its base
    /// sequence produces.
    public typealias Element =
```

```
Base.Element

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The filter sequence produces
errors from either the base
    /// sequence or the filtering
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingFilterSequence<Base>.Iterato
r

    /// The iterator that produces
elements of the filter sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the filter sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns nil.
Otherwise, `next()` evaluates the
        /// result with the `predicate`
closure. If the closure returns `true`,
        /// `next()` returns the received
```

```
element; otherwise it awaits the next
        /// element from the base
iterator. If calling the closure throws
an error,
        /// the sequence ends and
`next()` rethrows the error.
        @inlinable public mutating func
next() async throws -> Base.Element?

        /// Produces the next element in
the filter sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns nil.
Otherwise, `next()`
        /// evaluates the result with the
`predicate` closure. If the closure
        /// returns `true`,
`next(isolation:)` returns the received
element;
        /// otherwise it awaits the next
element from the base iterator. If
calling
        /// the closure throws an error,
the sequence ends and `next(isolation:)`
        /// rethrows the error.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
```

```
Actor)?) async throws -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingFilterSequence<Base>.Iterato
r
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingFilterSequence :
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingFilterSequence.Iterator :
```

```
@unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence that
concatenates the results of calling a
given
/// error-throwing transformation with
each element of this sequence.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingFlatMapSequence<Base,
SegmentOfResult> where Base :
AsyncSequence, SegmentOfResult :
AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingFlatMapSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The flat map sequence produces
the type of element in the asynchronous
    /// sequence produced by the
`transform` closure.
    public typealias Element =
SegmentOfResult.Element
```

```
    /// The type of error produced by
this asynchronous sequence.
    ///
    /// The flat map sequence produces
errors from either the base
    /// sequence or the `transform`
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingFlatMapSequence<Base,
SegmentOfResult>.Iterator

    /// The iterator that produces
elements of the flat map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the flat map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns
`nil`. Otherwise, `next()` calls the
        /// transforming closure on the
received element, takes the resulting
        /// asynchronous sequence, and
creates an asynchronous iterator from it.
```

```
        /// `next()` then consumes values
from this iterator until it terminates.
        /// At this point, `next()` is
ready to receive the next value from the
base
        /// sequence. If `transform`
throws an error, the sequence terminates.
        @inlinable public mutating func
next() async throws ->
SegmentOfResult.Element?

        /// Produces the next element in
the flat map sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns `nil`.
Otherwise,
        /// `next(isolation:)` calls the
transforming closure on the received
        /// element, takes the resulting
asynchronous sequence, and creates an
        /// asynchronous iterator from
it.  `next(isolation:)` then consumes
values
        /// from this iterator until it
terminates.  At this point,
        /// `next(isolation:)` is ready
to receive the next value from the base
        /// sequence. If `transform`
throws an error, the sequence terminates.
```

```
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws ->
SegmentOfResult.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
SegmentOfResult.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingFlatMapSequence<Base,
SegmentOfResult>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingFlatMapSequence :
@unchecked Sendable where Base :
Sendable, SegmentOfResult : Sendable,
Base.Element : Sendable,
```

```
SegmentOfResult.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingFlatMapSequence.Iterator :
@unchecked Sendable where SegmentOfResult
: Sendable, Base.AsyncIterator :
Sendable, Base.Element : Sendable,
SegmentOfResult.AsyncIterator : Sendable,
SegmentOfResult.Element : Sendable {
}

/// An asynchronous sequence that maps
the given error-throwing closure over the
/// asynchronous sequence's elements.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingMapSequence<Base,
Transformed> where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingMapSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The map sequence produces
```

```
whatever type of element its the
transforming
    /// closure produces.
    public typealias Element =
Transformed

    /// The type of error produced by
this asynchronous sequence.
    ///
    /// The map sequence produces errors
from either the base
    /// sequence or the `transform`
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingMapSequence<Base,
Transformed>.Iterator

    /// The iterator that produces
elements of the map sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the map sequence.
        ///
        /// This iterator calls `next()`
on its base iterator; if this call
returns
        /// `nil`, `next()` returns nil.
```

```
Otherwise, `next()` returns the result of
        /// calling the transforming
closure on the received element. If
calling
        /// the closure throws an error,
the sequence ends and `next()` rethrows
        /// the error.
        @inlinable public mutating func
next() async throws -> Transformed?

        /// Produces the next element in
the map sequence.
        ///
        /// This iterator calls
`next(isolation:)` on its base iterator;
if this
        /// call returns `nil`,
`next(isolation:)` returns nil.
Otherwise,
        /// `next(isolation:)` returns
the result of calling the transforming
        /// closure on the received
element. If calling the closure throws an
error,
        /// the sequence ends and
`next(isolation:)` rethrows the error.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws -> Transformed?

        @available(iOS 13.0, tvOS 13.0,
```

```
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Transformed
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingMapSequence<Base,
Transformed>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingMapSequence :
@unchecked Sendable where Base :
Sendable, Transformed : Sendable,
Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingMapSequence.Iterator :
@unchecked Sendable where Transformed :
Sendable, Base.AsyncIterator : Sendable,
```

```
Base.Element : Sendable {
}

/// An asynchronous sequence, containing
the initial, consecutive
/// elements of the base sequence that
satisfy the given error-throwing
/// predicate.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingPrefixWhileSequence<Base>
where Base : AsyncSequence {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingPrefixWhileSequence :
AsyncSequence {

    /// The type of element produced by
this asynchronous sequence.
    ///
    /// The prefix-while sequence
produces whatever type of element its
base
    /// iterator produces.
    public typealias Element =
Base.Element

    /// The type of error produced by
this asynchronous sequence.
```

```
    ///
    /// The prefix-while sequence
produces errors from either the base
    /// sequence or the filtering
closure.
    public typealias Failure = any Error

    /// The type of iterator that
produces elements of the sequence.
    public typealias AsyncIterator =
AsyncThrowingPrefixWhileSequence<Base>.It
erator

    /// The iterator that produces
elements of the prefix-while sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the prefix-while sequence.
        ///
        /// If the predicate hasn't
failed yet, this method gets the next
element
        /// from the base sequence and
calls the predicate with it. If this call
        /// succeeds, this method passes
along the element. Otherwise, it returns
        /// `nil`, ending the sequence.
If calling the predicate closure throws
an
        /// error, the sequence ends and
`next()` rethrows the error.
```

```
        @inlinable public mutating func
next() async throws -> Base.Element?

        /// Produces the next element in
the prefix-while sequence.
        ///
        /// If the predicate hasn't
failed yet, this method gets the next
element
        /// from the base sequence and
calls the predicate with it. If this call
        /// succeeds, this method passes
along the element. Otherwise, it returns
        /// `nil`, ending the sequence.
If calling the predicate closure throws
an
        /// error, the sequence ends and
`next(isolation:)` rethrows the error.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        @inlinable public mutating func
next(isolation actor: isolated (any
Actor)?) async throws -> Base.Element?

        @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
        public typealias Element =
Base.Element
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
```

```
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    @inlinable public func
makeAsyncIterator() ->
AsyncThrowingPrefixWhileSequence<Base>.It
erator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingPrefixWhileSequence :
@unchecked Sendable where Base :
Sendable, Base.Element : Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingPrefixWhileSequence.Iterator
: @unchecked Sendable where
Base.AsyncIterator : Sendable,
Base.Element : Sendable {
}

/// An asynchronous sequence generated
from an error-throwing closure that
/// calls a continuation to produce new
elements.
///
```

```
/// `AsyncThrowingStream` conforms to
`AsyncSequence`, providing a convenient
/// way to create an asynchronous
sequence without manually implementing an
/// asynchronous iterator. In particular,
an asynchronous stream is well-suited
/// to adapt callback- or delegation-
based APIs to participate with
/// `async`-`await`.
///
/// In contrast to `AsyncStream`, this
type can throw an error from the awaited
/// `next()`, which terminates the stream
with the thrown error.
///
/// You initialize an
`AsyncThrowingStream` with a closure that
receives an
/// `AsyncThrowingStream.Continuation`.
Produce elements in this closure, then
/// provide them to the stream by calling
the continuation's `yield(_:)` method.
/// When there are no further elements to
produce, call the continuation's
/// `finish()` method. This causes the
sequence iterator to produce a `nil`,
/// which terminates the sequence. If an
error occurs, call the continuation's
/// `finish(throwing:)` method, which
causes the iterator's `next()` method to
/// throw the error to the awaiting call
point. The continuation is `Sendable`,
/// which permits calling it from
```

```
concurrent contexts external to the
iteration
/// of the `AsyncThrowingStream`.
///
/// An arbitrary source of elements can
produce elements faster than they are
/// consumed by a caller iterating over
them. Because of this,
`AsyncThrowingStream`
/// defines a buffering behavior,
allowing the stream to buffer a specific
/// number of oldest or newest elements.
By default, the buffer limit is
/// `Int.max`, which means it's
unbounded.
///
/// ### Adapting Existing Code to Use
Streams
///
/// To adapt existing callback code to
use `async`-`await`, use the callbacks
/// to provide values to the stream, by
using the continuation's `yield(_:)`
/// method.
///
/// Consider a hypothetical
`QuakeMonitor` type that provides callers
with
/// `Quake` instances every time it
detects an earthquake. To receive
callbacks,
/// callers set a custom closure as the
value of the monitor's
```

```
/// `quakeHandler` property, which the
monitor calls back as necessary. Callers
/// can also set an `errorHandler` to
receive asynchronous error notifications,
/// such as the monitor service suddenly
becoming unavailable.
///
///     class QuakeMonitor {
///         var quakeHandler: ((Quake) ->
Void)?
///         var errorHandler: ((Error) ->
Void)?
///
///         func startMonitoring() {…}
///         func stopMonitoring() {…}
///     }
///
/// To adapt this to use `async`–`await`,
extend the `QuakeMonitor` to add a
/// `quakes` property, of type
`AsyncThrowingStream<Quake>`. In the
getter for
/// this property, return an
`AsyncThrowingStream`, whose `build`
closure --
/// called at runtime to create the
stream -- uses the continuation to
/// perform the following steps:
///
/// 1. Creates a `QuakeMonitor` instance.
/// 2. Sets the monitor's `quakeHandler`
property to a closure that receives
/// each `Quake` instance and forwards it
```

```
to the stream by calling the
/// continuation's `yield(_:)` method.
/// 3. Sets the monitor's `errorHandler`
property to a closure that receives
/// any error from the monitor and
forwards it to the stream by calling the
/// continuation's `finish(throwing:)`
method. This causes the stream's
/// iterator to throw the error and
terminate the stream.
/// 4. Sets the continuation's
`onTermination` property to a closure
that
/// calls `stopMonitoring()` on the
monitor.
/// 5. Calls `startMonitoring` on the
`QuakeMonitor`.
///
/// ```
/// extension QuakeMonitor {
///
///     static var throwingQuakes:
AsyncThrowingStream<Quake, Error> {
///         AsyncThrowingStream
{ continuation in
///             let monitor =
QuakeMonitor()
///             monitor.quakeHandler =
{ quake in
///
continuation.yield(quake)
///             }
///             monitor.errorHandler =
```

```
{ error in
///
continuation.finish(throwing: error)
///             }
///
continuation.onTermination = { @Sendable
_ in
///
monitor.stopMonitoring()
///             }
///             monitor.startMonitoring()
///         }
///     }
/// }
/// ```
///
///
/// Because the stream is an
`AsyncSequence`, the call point uses the
/// `for`-`await`-`in` syntax to process
each `Quake` instance as produced by the
stream:
///
///     do {
///         for try await quake in
quakeStream {
///             print("Quake: \
(quake.date)")
///         }
///         print("Stream done.")
///     } catch {
///         print("Error: \(error)")
///     }
```

```
///
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct
AsyncThrowingStream<Element, Failure>
where Failure : Error {

    /// A mechanism to interface between
synchronous code and an asynchronous
    /// stream.
    ///
    /// The closure you provide to the
`AsyncThrowingStream` in
    /// `init(_:bufferingPolicy:_:)`
receives an instance of this type when
    /// invoked. Use this continuation to
provide elements to the stream by
    /// calling one of the `yield`
methods, then terminate the stream
normally by
    /// calling the `finish()` method.
You can also use the continuation's
    /// `finish(throwing:)` method to
terminate the stream by throwing an
error.
    ///
    /// - Note: Unlike other
continuations in Swift,
    ///
`AsyncThrowingStream.Continuation`
supports escaping.
    public struct Continuation : Sendable
{
```

```
        /// A type that indicates how the
stream terminated.
        ///
        /// The `onTermination` closure
receives an instance of this type.
        public enum Termination {

            /// The stream finished as a
result of calling the continuation's
            ///  `finish` method.
            ///
            ///  The associated `Failure`
value provides the error that terminated
            ///  the stream. If no error
occurred, this value is `nil`.
            case finished(Failure?)

            /// The stream finished as a
result of cancellation.
            case cancelled
        }

        /// A type that indicates the
result of yielding a value to a client,
by
        /// way of the continuation.
        ///
        /// The various `yield` methods
of `AsyncThrowingStream.Continuation`
return
        /// this type to indicate the
success or failure of yielding an element
```

```
to
        /// the continuation.
        public enum YieldResult {

            /// The stream successfully
enqueued the element.
            ///
            /// This value represents the
successful enqueueing of an element,
whether
            /// the stream buffers the
element or delivers it immediately to a
pending
            /// call to `next()`. The
associated value `remaining` is a hint
that
            /// indicates the number of
remaining slots in the buffer at the time
of
            /// the `yield` call.
            ///
            /// - Note: From a thread
safety perspective, `remaining` is a
lower bound
            /// on the number of
remaining slots. This is because a
subsequent call
            /// that uses the `remaining`
value could race on the consumption of
            /// values from the stream.
            case enqueued(remaining: Int)

            /// The stream didn't enqueue
```

```
the element because the buffer was full.
            ///
            /// The associated element
for this case is the element that the
stream
            /// dropped.
            case dropped(Element)

            /// The stream didn't enqueue
the element because the stream was in a
            /// terminal state.
            ///
            /// This indicates the stream
terminated prior to calling `yield`,
either
            /// because the stream
finished normally or through
cancellation, or
            /// it threw an error.
            case terminated
        }

        /// A strategy that handles
exhaustion of a buffer’s capacity.
        public enum BufferingPolicy {

            /// Continue to add to the
buffer, treating its capacity as
infinite.
            case unbounded

            /// When the buffer is full,
discard the newly received element.
```

```
            ///
            /// This strategy enforces
keeping the specified amount of oldest
values.
            case bufferingOldest(Int)

            /// When the buffer is full,
discard the oldest element in the buffer.
            ///
            /// This strategy enforces
keeping the specified amount of newest
values.
            case bufferingNewest(Int)
        }

        /// Resume the task awaiting the
next iteration point by having it return
        /// normally from its suspension
point with a given element.
        ///
        /// - Parameter value: The value
to yield from the continuation.
        /// - Returns: A `YieldResult`
that indicates the success or failure of
the
        ///   yield operation.
        ///
        /// If nothing is awaiting the
next value, the method attempts to buffer
the
        /// result's element.
        ///
        /// This can be called more than
```

```
once and returns to the caller
immediately
        /// without blocking for any
awaiting consumption from the iteration.
        @discardableResult
        public func yield(_ value:
sending Element) ->
AsyncThrowingStream<Element,
Failure>.Continuation.YieldResult

        /// Resume the task awaiting the
next iteration point by having it return
        /// nil, which signifies the end
of the iteration.
        ///
        /// - Parameter error: The error
to throw, or `nil`, to finish normally.
        ///
        /// Calling this function more
than once has no effect. After calling
        /// finish, the stream enters a
terminal state and doesn't produce any
additional
        /// elements.
        public func finish(throwing
error: Failure? = nil)

        /// A callback to invoke when
canceling iteration of an asynchronous
        /// stream.
        ///
        /// If an `onTermination`
callback is set, using task cancellation
```

```
to
        /// terminate iteration of an
`AsyncThrowingStream` results in a call
to this
        /// callback.
        ///
        /// Canceling an active iteration
invokes the `onTermination` callback
        /// first, and then resumes by
yielding `nil` or throwing an error from
the
        /// iterator. This means that you
can perform needed cleanup in the
        ///  cancellation handler. After
reaching a terminal state, the
        ///  `AsyncThrowingStream`
disposes of the callback.
        public var onTermination:
(@Sendable (AsyncThrowingStream<Element,
Failure>.Continuation.Termination) ->
Void)? { get nonmutating set }
    }

    /// Constructs an asynchronous stream
for an element type, using the
    /// specified buffering policy and
element-producing closure.
    ///
    /// - Parameters:
    ///   - elementType: The type of
element the `AsyncThrowingStream`
    ///   produces.
    ///   - limit: The maximum number of
```

```
elements to
    ///   hold in the buffer. By default,
this value is unlimited. Use a
    ///   `Continuation.BufferingPolicy`
to buffer a specified number of oldest
    ///   or newest elements.
    ///   - build: A custom closure that
yields values to the
    ///   `AsyncThrowingStream`. This
closure receives an
    ///
`AsyncThrowingStream.Continuation`
instance that it uses to provide
    ///   elements to the stream and
terminate the stream when finished.
    ///
    /// The `AsyncStream.Continuation`
received by the `build` closure is
    /// appropriate for use in concurrent
contexts. It is thread safe to send and
    /// finish; all calls are to the
continuation are serialized. However,
calling
    /// this from multiple concurrent
contexts could result in out-of-order
    /// delivery.
    ///
    /// The following example shows an
`AsyncStream` created with this
    /// initializer that produces 100
random numbers on a one-second interval,
    /// calling `yield(_:)` to deliver
each element to the awaiting call point.
```

```
    /// When the `for` loop exits, the
stream finishes by calling the
    /// continuation's `finish()` method.
If the random number is divisible by 5
    /// with no remainder, the stream
throws a `MyRandomNumberError`.
    ///
    ///     let stream =
AsyncThrowingStream<Int, Error>(Int.self,
    ///
bufferingPolicy: .bufferingNewest(5))
{ continuation in
    ///         Task.detached {
    ///             for _ in 0..<100 {
    ///                 await
Task.sleep(1 * 1_000_000_000)
    ///                 let random =
Int.random(in: 1...10)
    ///                 if random % 5 ==
0 {
    ///
continuation.finish(throwing:
MyRandomNumberError())
    ///                     return
    ///                 } else {
    ///
continuation.yield(random)
    ///                 }
    ///             }
    ///             continuation.finish()
    ///         }
    ///     }
    ///
```

```
    ///     // Call point:
    ///     do {
    ///         for try await random in
stream {
    ///             print(random)
    ///         }
    ///     } catch {
    ///         print(error)
    ///     }
    ///
    public init(_ elementType:
Element.Type = Element.self,
bufferingPolicy limit:
AsyncThrowingStream<Element,
Failure>.Continuation.BufferingPolicy
= .unbounded, _ build:
(AsyncThrowingStream<Element,
Failure>.Continuation) -> Void) where
Failure == any Error

    /// Constructs an asynchronous
throwing stream from a given element-
producing
    /// closure.
    ///
    /// - Parameters:
    ///   - produce: A closure that
asynchronously produces elements for the
    ///    stream.
    ///
    /// Use this convenience initializer
when you have an asynchronous function
    /// that can produce elements for the
```

```
stream, and don't want to invoke
    /// a continuation manually. This
initializer "unfolds" your closure into
    /// a full-blown asynchronous stream.
The created stream handles adherence to
    /// the `AsyncSequence` protocol
automatically. To terminate the stream
with
    /// an error, throw the error from
your closure.
    ///
    /// The following example shows an
`AsyncThrowingStream` created with this
    /// initializer that produces random
numbers on a one-second interval. If the
    /// random number is divisible by 5
with no remainder, the stream throws a
    /// `MyRandomNumberError`.
    ///
    ///     let stream =
AsyncThrowingStream<Int, Error> {
    ///         await Task.sleep(1 *
1_000_000_000)
    ///         let random =
Int.random(in: 1...10)
    ///         if random % 5 == 0 {
    ///             throw
MyRandomNumberError()
    ///         }
    ///         return random
    ///     }
    ///
    ///     // Call point:
```

```
    ///     do {
    ///         for try await random in
stream {
    ///             print(random)
    ///         }
    ///     } catch {
    ///         print(error)
    ///     }
    ///
    @preconcurrency public init(unfolding
produce: @escaping @Sendable () async
throws -> Element?) where Failure == any
Error
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingStream :
AsyncSequence {

    /// The asynchronous iterator for
iterating an asynchronous stream.
    ///
    /// This type is not `Sendable`.
Don't use it from multiple
    /// concurrent contexts. It is a
programmer error to invoke `next()` from
a
    /// concurrent context that contends
with another such call, which
    /// results in a call to
`fatalError()`.
    public struct Iterator :
```

```
AsyncIteratorProtocol {

        /// The next value from the
asynchronous stream.
        ///
        /// When `next()` returns `nil`,
this signifies the end of the
        /// `AsyncThrowingStream`.
        ///
        /// It is a programmer error to
invoke `next()` from a concurrent context
        /// that contends with another
such call, which results in a call to
        ///  `fatalError()`.
        ///
        /// If you cancel the task this
iterator is running in while `next()` is
        /// awaiting a value, the
`AsyncThrowingStream` terminates. In this
case,
        /// `next()` may return `nil`
immediately, or else return `nil` on
        /// subsequent calls.
        public mutating func next() async
throws -> Element?

        /// The next value from the
asynchronous stream.
        ///
        /// When `next()` returns `nil`,
this signifies the end of the
        /// `AsyncThrowingStream`.
        ///
```

```
        /// It is a programmer error to
invoke `next()` from a concurrent
        /// context that contends with
another such call, which results in a
call to
        /// `fatalError()`.
        ///
        /// If you cancel the task this
iterator is running in while `next()`
        /// is awaiting a value, the
`AsyncThrowingStream` terminates. In this
case,
        /// `next()` may return `nil`
immediately, or else return `nil` on
        /// subsequent calls.
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public mutating func
next(isolation actor: isolated (any
Actor)?) async throws(Failure) ->
Element?
    }

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    public func makeAsyncIterator() ->
AsyncThrowingStream<Element,
Failure>.Iterator

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
```

```
    @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
    public typealias AsyncIterator =
AsyncThrowingStream<Element,
Failure>.Iterator
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingStream {

    /// Initializes a new
``AsyncThrowingStream`` and an
``AsyncThrowingStream/Continuation``.
    ///
    /// - Parameters:
    ///   - elementType: The element type
of the stream.
    ///   - failureType: The failure type
of the stream.
    ///   - limit: The buffering policy
that the stream should use.
    /// - Returns: A tuple containing the
stream and its continuation. The
continuation should be passed to the
    /// producer while the stream should
be passed to the consumer.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public static func makeStream(of
elementType: Element.Type = Element.self,
```

```
throwing failureType: Failure.Type =
Failure.self, bufferingPolicy limit:
AsyncThrowingStream<Element,
Failure>.Continuation.BufferingPolicy
= .unbounded) -> (stream:
AsyncThrowingStream<Element, Failure>,
continuation:
AsyncThrowingStream<Element,
Failure>.Continuation) where Failure ==
any Error
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension AsyncThrowingStream :
@unchecked Sendable where Element :
Sendable {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension
AsyncThrowingStream.Continuation {

    /// Resume the task awaiting the next
iteration point by having it return
    /// normally or throw, based on a
given result.
    ///
    /// - Parameter result: A result to
yield from the continuation. In the
    ///   `.success(_:)` case, this
returns the associated value from the
```

```
    ///   iterator's `next()` method. If
the result is the `failure(_:)` case,
    ///   this call terminates the stream
with the result's error, by calling
    ///   `finish(throwing:)`.
    /// - Returns: A `YieldResult` that
indicates the success or failure of the
    ///   yield operation.
    ///
    /// If nothing is awaiting the next
value and the result is success, this
call
    /// attempts to buffer the result's
element.
    ///
    /// If you call this method
repeatedly, each call returns
immediately, without
    /// blocking for any awaiting
consumption from the iteration.
    @discardableResult
    public func yield(with result:
sending Result<Element, Failure>) ->
AsyncThrowingStream<Element,
Failure>.Continuation.YieldResult where
Failure == any Error

    /// Resume the task awaiting the next
iteration point by having it return
    /// normally from its suspension
point.
    ///
    /// - Returns: A `YieldResult` that
```

```
indicates the success or failure of the
    ///   yield operation.
    ///
    /// Use this method with
`AsyncThrowingStream` instances whose
`Element`
    /// type is `Void`. In this case, the
`yield()` call unblocks the
    /// awaiting iteration; there is no
value to return.
    ///
    /// If you call this method
repeatedly, each call returns
immediately,
    /// without blocking for any awaiting
consumption from the iteration.
    @discardableResult
    public func yield() ->
AsyncThrowingStream<Element,
Failure>.Continuation.YieldResult where
Element == ()
}

/// An error that indicates a task was
canceled.
///
/// This error is also thrown
automatically by
`Task.checkCancellation()`,
/// if the current task has been
canceled.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
public struct CancellationError : Error {

    public init()
}

/// A mechanism to interface
/// between synchronous and asynchronous
code,
/// logging correctness violations.
///
/// A *continuation* is an opaque
representation of program state.
/// To create a continuation in
asynchronous code,
/// call the
`withUnsafeContinuation(function:_:)` or
///
`withUnsafeThrowingContinuation(function:
_:)` function.
/// To resume the asynchronous task,
/// call the `resume(returning:)`,
/// `resume(throwing:)`,
/// `resume(with:)`,
/// or `resume()` method.
///
/// - Important: You must call a resume
method exactly once
///   on every execution path throughout
the program.
///
/// Resuming from a continuation more
than once is undefined behavior.
/// Never resuming leaves the task in a
```

```
suspended state indefinitely,
/// and leaks any associated resources.
/// `CheckedContinuation` logs a message
/// if either of these invariants is
violated.
///
/// `CheckedContinuation` performs
runtime checks
/// for missing or multiple resume
operations.
/// `UnsafeContinuation` avoids enforcing
these invariants at runtime
/// because it aims to be a low-overhead
mechanism
/// for interfacing Swift tasks with
/// event loops, delegate methods,
callbacks,
/// and other non-`async` scheduling
mechanisms.
/// However, during development, the
ability to verify that the
/// invariants are being upheld in
testing is important.
/// Because both types have the same
interface,
/// you can replace one with the other in
most circumstances,
/// without making other changes.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct CheckedContinuation<T, E> :
Sendable where E : Error {
```

```
    /// Creates a checked continuation
from an unsafe continuation.
    ///
    /// Instead of calling this
initializer,
    /// most code calls the
`withCheckedContinuation(function:_:)` or
    ///
`withCheckedThrowingContinuation(function
:_:)` function instead.
    /// You only need to initialize
    /// your own `CheckedContinuation<T,
E>` if you already have an
    /// `UnsafeContinuation` you want to
impose checking on.
    ///
    /// - Parameters:
    ///   - continuation: An instance of
`UnsafeContinuation`
    ///     that hasn't yet been resumed.
    ///     After passing the unsafe
continuation to this initializer,
    ///     don't use it outside of this
object.
    ///   - function: A string
identifying the declaration that is the
notional
    ///     source for the continuation,
used to identify the continuation in
    ///     runtime diagnostics related
to misuse of this continuation.
    public init(continuation:
UnsafeContinuation<T, E>, function:
```

```
String = #function)

    /// Resume the task awaiting the
continuation by having it return normally
    /// from its suspension point.
    ///
    /// - Parameter value: The value to
return from the continuation.
    ///
    /// A continuation must be resumed
exactly once. If the continuation has
    /// already been resumed through this
object, then the attempt to resume
    /// the continuation will trap.
    ///
    /// After `resume` enqueues the task,
control immediately returns to
    /// the caller. The task continues
executing when its executor is
    /// able to reschedule it.
    public func resume(returning value:
sending T)

    /// Resume the task awaiting the
continuation by having it throw an error
    /// from its suspension point.
    ///
    /// - Parameter error: The error to
throw from the continuation.
    ///
    /// A continuation must be resumed
exactly once. If the continuation has
    /// already been resumed through this
```

```
object, then the attempt to resume
    /// the continuation will trap.
    ///
    /// After `resume` enqueues the task,
control immediately returns to
    /// the caller. The task continues
executing when its executor is
    /// able to reschedule it.
    public func resume(throwing error: E)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension CheckedContinuation {

    /// Resume the task awaiting the
continuation by having it either
    /// return normally or throw an error
based on the state of the given
    /// `Result` value.
    ///
    /// - Parameter result: A value to
either return or throw from the
    ///   continuation.
    ///
    /// A continuation must be resumed
exactly once. If the continuation has
    /// already been resumed through this
object, then the attempt to resume
    /// the continuation will trap.
    ///
    /// After `resume` enqueues the task,
control immediately returns to
```

```
    /// the caller. The task continues
executing when its executor is
    /// able to reschedule it.
    public func resume<Er>(with result:
sending Result<T, Er>) where E == any
Error, Er : Error

    /// Resume the task awaiting the
continuation by having it either
    /// return normally or throw an error
based on the state of the given
    /// `Result` value.
    ///
    /// - Parameter result: A value to
either return or throw from the
    ///   continuation.
    ///
    /// A continuation must be resumed
exactly once. If the continuation has
    /// already been resumed through this
object, then the attempt to resume
    /// the continuation will trap.
    ///
    /// After `resume` enqueues the task,
control immediately returns to
    /// the caller. The task continues
executing when its executor is
    /// able to reschedule it.
    public func resume(with result:
sending Result<T, E>)

    /// Resume the task awaiting the
continuation by having it return normally
```

```
    /// from its suspension point.
    ///
    /// A continuation must be resumed
exactly once. If the continuation has
    /// already been resumed through this
object, then the attempt to resume
    /// the continuation will trap.
    ///
    /// After `resume` enqueues the task,
control immediately returns to
    /// the caller. The task continues
executing when its executor is
    /// able to reschedule it.
    public func resume() where T == ()
}

/// A mechanism in which to measure time,
and delay work until a given point
/// in time.
///
/// Types that conform to the `Clock`
protocol define a concept of "now" which
/// is the specific instant in time that
property is accessed. Any pair of calls
/// to the `now` property may have a
minimum duration between them - this
/// minimum resolution is exposed by the
`minimumResolution` property to inform
/// any user of the type the expected
granularity of accuracy.
///
/// One of the primary uses for clocks is
to schedule task sleeping. This method
```

```
/// resumes the calling task after a
given deadline has been met or passed
with
/// a given tolerance value. The
tolerance is expected as a leeway around
the
/// deadline. The clock may reschedule
tasks within the tolerance to ensure
/// efficient execution of resumptions by
reducing potential operating system
/// wake-ups. If no tolerance is
specified (i.e. nil is passed in) the
sleep
/// function is expected to schedule with
a default tolerance strategy.
///
/// For more information about specific
clocks see `ContinuousClock` and
/// `SuspendingClock`.
@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
public protocol Clock<Duration> :
Sendable {

    associatedtype Duration where
Self.Duration == Self.Instant.Duration

    associatedtype Instant :
InstantProtocol

    var now: Self.Instant { get }

    var minimumResolution: Self.Duration
```

```
{ get }

    func sleep(until deadline:
Self.Instant, tolerance:
Self.Instant.Duration?) async throws
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension Clock {

    /// Measure the elapsed time to
execute a closure.
    ///
    ///         let clock =
ContinuousClock()
    ///         let elapsed = clock.measure
{
    ///             someWork()
    ///         }
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func measure(_ work: () throws
-> Void) rethrows ->
Self.Instant.Duration

    /// Measure the elapsed time to
execute an asynchronous closure.
    ///
    ///         let clock =
ContinuousClock()
    ///         let elapsed = await
clock.measure {
```

```
    ///             await someWork()
    ///         }
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func measure(isolation:
isolated (any Actor)? = #isolation, _
work: () async throws -> Void) async
rethrows -> Self.Instant.Duration
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension Clock {

    /// Suspends for the given duration.
    ///
    /// Prefer to use the
`sleep(until:tolerance:)` method on
`Clock` if you have
    /// access to an absolute instant.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func sleep(for duration:
Self.Instant.Duration, tolerance:
Self.Instant.Duration? = nil) async
throws
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension Clock where Self ==
ContinuousClock {
```

```
    /// A clock that measures time that
always increments but does not stop
    /// incrementing while the system is
asleep.
    ///
    ///       try await Task.sleep(until:
.now + .seconds(3), clock: .continuous)
    ///
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static var continuous:
ContinuousClock { get }
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension Clock where Self ==
SuspendingClock {

    /// A clock that measures time that
always increments but stops incrementing
    /// while the system is asleep.
    ///
    ///       try await Task.sleep(until:
.now + .seconds(3), clock: .suspending)
    ///
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static var suspending:
SuspendingClock { get }
}

/// A clock that measures time that
```

```
always increments and does not stop
/// incrementing while the system is
asleep.
///
/// `ContinuousClock` can be considered
as a stopwatch style time. The frame of
/// reference of the `Instant` may be
bound to process launch, machine boot or
/// some other locally defined reference
point. This means that the instants are
/// only comparable locally during the
execution of a program.
///
/// This clock is suitable for high
resolution measurements of execution.
@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
public struct ContinuousClock : Sendable
{

    /// A continuous point in time used
for `ContinuousClock`.
    public struct Instant : Codable,
Sendable {

        /// Encodes this value into the
given encoder.
        ///
        /// If the value fails to encode
anything, `encoder` will encode an empty
        /// keyed container in its place.
        ///
        /// This function throws an error
```

```
if any values are invalid for the given
        /// encoder's format.
        ///
        /// - Parameter encoder: The
encoder to write data to.
        public func encode(to encoder:
any Encoder) throws

        /// Creates a new instance by
decoding from the given decoder.
        ///
        /// This initializer throws an
error if reading from the decoder fails,
or
        /// if the data read is corrupted
or otherwise invalid.
        ///
        /// - Parameter decoder: The
decoder to read data from.
        public init(from decoder: any
Decoder) throws
    }

    public init()
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension ContinuousClock : Clock {

    /// The current continuous instant.
    public var now:
ContinuousClock.Instant { get }
```

```
    /// The minimum non-zero resolution
between any two calls to `now`.
    public var minimumResolution:
Duration { get }

    /// The current continuous instant.
    public static var now:
ContinuousClock.Instant { get }

    /// Suspend task execution until a
given deadline within a tolerance.
    /// If no tolerance is specified then
the system may adjust the deadline
    /// to coalesce CPU wake-ups to more
efficiently process the wake-ups in
    /// a more power efficient manner.
    ///
    /// If the task is canceled before
the time ends, this function throws
    /// `CancellationError`.
    ///
    /// This function doesn't block the
underlying thread.
    public func sleep(until deadline:
ContinuousClock.Instant, tolerance:
Duration? = nil) async throws

    @available(iOS 16.0, tvOS 16.0,
watchOS 9.0, macOS 13.0, *)
    public typealias Duration = Duration
}
```

```
@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension ContinuousClock.Instant :
InstantProtocol {

    public static var now:
ContinuousClock.Instant { get }

    public func advanced(by duration:
Duration) -> ContinuousClock.Instant

    public func duration(to other:
ContinuousClock.Instant) -> Duration

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
```

```
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    public func hash(into hasher: inout
Hasher)

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
ContinuousClock.Instant, rhs:
ContinuousClock.Instant) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is less than that of the
second argument.
    ///
    /// This function is the only
requirement of the `Comparable` protocol.
The
    /// remainder of the relational
```

```
operator functions are implemented by the
    /// standard library for any type
that conforms to `Comparable`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func < (lhs:
ContinuousClock.Instant, rhs:
ContinuousClock.Instant) -> Bool

    @inlinable public static func + (lhs:
ContinuousClock.Instant, rhs: Duration)
-> ContinuousClock.Instant

    @inlinable public static func +=
(lhs: inout ContinuousClock.Instant, rhs:
Duration)

    @inlinable public static func - (lhs:
ContinuousClock.Instant, rhs: Duration)
-> ContinuousClock.Instant

    @inlinable public static func -=
(lhs: inout ContinuousClock.Instant, rhs:
Duration)

    @inlinable public static func - (lhs:
ContinuousClock.Instant, rhs:
ContinuousClock.Instant) -> Duration

    @available(iOS 16.0, tvOS 16.0,
```

```
watchOS 9.0, macOS 13.0, *)
    public typealias Duration = Duration

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}

/// A discarding group that contains
dynamically created child tasks.
///
/// To create a discarding task group,
/// call the
``withDiscardingTaskGroup(returning:body:
)`` method.
///
/// Don't use a task group from outside
the task where you created it.
/// In most cases,
/// the Swift type system prevents a task
```

```
group from escaping like that
/// because adding a child task to a task
group is a mutating operation,
/// and mutation operations can't be
performed
/// from a concurrent execution context
like a child task.
///
/// ### Task execution order
/// Tasks added to a task group execute
concurrently, and may be scheduled in
/// any order.
///
/// ### Discarding behavior
/// A discarding task group eagerly
discards and releases its child tasks as
/// soon as they complete. This allows
for the efficient releasing of memory
used
/// by those tasks, which are not
retained for future `next()` calls, as
would
/// be the case with a ``TaskGroup``.
///
/// ### Cancellation behavior
/// A discarding task group becomes
cancelled in one of the following ways:
///
/// - when ``cancelAll()`` is invoked on
it,
/// - when the ``Task`` running this task
group is cancelled.
///
```

```
/// Since a `DiscardingTaskGroup` is a
structured concurrency primitive,
cancellation is
/// automatically propagated through all
of its child-tasks (and their child
/// tasks).
///
/// A cancelled task group can still keep
adding tasks, however they will start
/// being immediately cancelled, and may
act accordingly to this. To avoid adding
/// new tasks to an already cancelled
task group, use
``addTaskUnlessCancelled(priority:body:)`
`
/// rather than the plain
``addTask(priority:body:)`` which adds
tasks unconditionally.
///
/// For information about the language-
level concurrency model that
`DiscardingTaskGroup` is part of,
/// see [Concurrency][concurrency] in
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/Languag
eGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-
book/
///
/// - SeeAlso: ``TaskGroup``
/// - SeeAlso: ``ThrowingTaskGroup``
```

```
/// - SeeAlso:
``ThrowingDiscardingTaskGroup``
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@frozen public struct DiscardingTaskGroup
{

    /// Adds a child task to the group.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(priority: TaskPriority? = nil,
operation: sending @escaping
@isolated(any) () async -> Void)

    /// Adds a child task to the group,
unless the group has been canceled.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
```

```
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async ->
Void) -> Bool

    public mutating func
addTask(operation: sending @escaping
@isolated(any) () async -> Void)

    /// Adds a child task to the group,
unless the group has been canceled.
    ///
    /// - Parameters:
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(operation: sending
@escaping @isolated(any) () async ->
Void) -> Bool

    /// A Boolean value that indicates
whether the group has any remaining
tasks.
    ///
```

```
    /// At the start of the body of a
`withDiscardingTaskGroup(of:returning:bod
y:)` call,
    /// the task group is always empty.
    ///
    /// It's guaranteed to be empty when
returning from that body
    /// because a task group waits for
all child tasks to complete before
returning.
    ///
    /// - Returns: `true` if the group
has no pending tasks; otherwise `false`.
    public var isEmpty: Bool { get }

    /// Cancel all of the remaining tasks
in the group.
    ///
    /// If you add a task to a group
after canceling the group,
    /// that task is canceled immediately
after being added to the group.
    ///
    /// Immediately cancelled child tasks
should therefore cooperatively check for
and
    /// react  to cancellation, e.g. by
throwing an `CancellationError` at their
    /// earliest convenience, or
otherwise handling the cancellation.
    ///
    /// There are no restrictions on
where you can call this method.
```

```
    /// Code inside a child task or even
another task can cancel a group,
    /// however one should be very
careful to not keep a reference to the
    /// group longer than the
`with...TaskGroup(...) { ... }` method
body is executing.
    ///
    /// - SeeAlso: `Task.isCancelled`
    /// - SeeAlso:
`DiscardingTaskGroup.isCancelled`
    public func cancelAll()

    /// A Boolean value that indicates
whether the group was canceled.
    ///
    /// To cancel a group, call the
`DiscardingTaskGroup.cancelAll()` method.
    ///
    /// If the task that's currently
running this group is canceled,
    /// the group is also implicitly
canceled,
    /// which is also reflected in this
property's value.
    public var isCancelled: Bool { get }
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension DiscardingTaskGroup {

    /// Adds a child task to the group
```

```
and enqueue it on the specified executor.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTask()` without passing a value to
the `taskExecutor` parameter,
    ///                   and it will be
inherited automatically.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async ->
Void)

    /// Adds a child task to the group
```

```
and set it up with the passed in task
executor preference,
    /// unless the group has been
canceled.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTask()` without passing a value to
the `taskExecutor` parameter,
    ///                   and it will be
inherited automatically.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(executorPreference
```

```
taskExecutor: (any TaskExecutor)?,
priority: TaskPriority? = nil, operation:
sending @escaping @isolated(any) () async
-> Void) -> Bool
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension DiscardingTaskGroup :
BitwiseCopyable {
}

/// A service that can execute jobs.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol Executor : AnyObject,
Sendable {

    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    func enqueue(_ job: UnownedJob)

    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    @available(*, deprecated, message:
"Implement 'enqueue(_: consuming
ExecutorJob)' instead")
    func enqueue(_ job: consuming Job)

    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    func enqueue(_ job: consuming
ExecutorJob)
```

```
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension Executor {

    public func enqueue(_ job:
UnownedJob)

    public func enqueue(_ job: consuming
ExecutorJob)

    public func enqueue(_ job: consuming
Job)
}

/// A unit of schedulable work.
///
/// Unless you're implementing a
scheduler,
/// you don't generally interact with
jobs directly.
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@frozen public struct ExecutorJob :
~Copyable, Sendable {

    public init(_ job: UnownedJob)

    public init(_ job: Job)

    public var priority: JobPriority {
get }
```

```
    public var description: String {
get }
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension ExecutorJob {

    /// Run this job on the passed in
executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
    ///
    /// Converting a `ExecutorJob` to an
```

```
``UnownedJob`` and invoking
``UnownedJob/runSynchronously(_:)`` on it
multiple times is undefined behavior,
    /// as a job can only ever be run
once, and must not be accessed after it
has been run.
    ///
    /// - Parameter executor: the
executor this job will be semantically
running on.
    @inlinable public func
runSynchronously(on executor:
UnownedSerialExecutor)

    /// Run this job on the passed in
task executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
```

```
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
    ///
    /// Converting a `ExecutorJob` to an
``UnownedJob`` and invoking
``UnownedJob/runSynchronously(_:)` on it
multiple times is undefined behavior,
    /// as a job can only ever be run
once, and must not be accessed after it
has been run.
    ///
    /// - Parameter executor: the
executor this job will be run on.
    ///
    /// - SeeAlso:
``runSynchronously(isolatedTo:taskExecuto
r:)``
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    @inlinable public func
runSynchronously(on executor:
UnownedTaskExecutor)

    /// Run this job isolated to the
passed in serial executor, while
executing it on the specified task
executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
```

```
is for an executor to determine when and
where it
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
    ///
    /// - Parameter serialExecutor: the
executor this job will be semantically
running on.
    /// - Parameter taskExecutor: the
task executor this job will be run on.
    ///
    /// - SeeAlso:
``runSynchronously(on:)``
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    @inlinable public func
runSynchronously(isolatedTo
serialExecutor: UnownedSerialExecutor,
taskExecutor: UnownedTaskExecutor)
}

/// A type that represents a globally-
```

```
unique actor that can be used to isolate
/// various declarations anywhere in the
program.
///
/// A type that conforms to the
`GlobalActor` protocol and is marked with
/// the `@globalActor` attribute can be
used as a custom attribute. Such types
/// are called global actor types, and
can be applied to any declaration to
/// specify that such types are isolated
to that global actor type. When using
/// such a declaration from another actor
(or from nonisolated code),
/// synchronization is performed through
the shared actor instance to ensure
/// mutually-exclusive access to the
declaration.
///
/// ## Custom Actor Executors
/// A global actor uses a custom executor
if it needs to customize its execution
/// semantics, for example, by making
sure all of its invocations are run on a
/// specific thread or dispatch queue.
///
/// This is done the same way as with
normal non-global actors, by declaring a
/// ``Actor/unownedExecutor`` nonisolated
property in the ``ActorType``
/// underlying this global actor.
///
/// It is *not* necessary to override the
```

```
``sharedUnownedExecutor`` static
/// property of the global actor, as its
default implementation already
/// delegates to the
``shared.unownedExecutor``, which is the
most reasonable
/// and correct implementation of this
protocol requirement.
///
/// You can find out more about custom
executors, by referring to the
/// ``SerialExecutor`` protocol's
documentation.
///
/// - SeeAlso: ``SerialExecutor``
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol GlobalActor {

    /// The type of the shared actor
instance that will be used to provide
    /// mutually-exclusive access to
declarations annotated with the given
global
    /// actor type.
    associatedtype ActorType : Actor

    /// The shared actor instance that
will be used to provide mutually-
exclusive
    /// access to declarations annotated
with the given global actor type.
    ///
```

```
    /// The value of this property must
always evaluate to the same actor
    /// instance.
    static var shared: Self.ActorType {
get }

    /// Shorthand for referring to the
`shared.unownedExecutor` of this global
actor.
    ///
    /// When declaring a global actor
with a custom executor, prefer to
implement
    /// the underlying actor's
``Actor/unownedExecutor`` property, and
leave this
    /// `sharedUnownedExecutor` default
implementation in-place as it will simply
    /// delegate to the
`shared.unownedExecutor`.
    ///
    /// The value of this property must
be equivalent to
`shared.unownedExecutor`,
    /// as it may be used by the Swift
concurrency runtime or explicit user code
with
    /// that assumption in mind.
    ///
    /// Returning different executors for
different invocations of this computed
    /// property is also illegal, as it
could lead to inconsistent
```

```
synchronization
    /// of the underlying actor.
    ///
    /// - SeeAlso: ``SerialExecutor``
    static var sharedUnownedExecutor:
UnownedSerialExecutor { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension GlobalActor {

    /// Stops program execution if the
current task is not executing on this
    /// actor's serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
    ///
    /// * In playgrounds and `-Onone`
builds (the default for Xcode's Debug
    ///   configuration), stops program
execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for
Xcode's Release configuration), stops
    ///   program execution.
    ///
    /// - Note: This check is performed
against the actor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
```

```
    ///   that actor's serial executor as
its own ``Actor/unownedExecutor``--this
    ///   check will succeed , as from a
concurrency safety perspective, the
    ///   serial executor guarantees
mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - message: The message to print
if the assertion fails.
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public static func
preconditionIsolated(_ message:
@autoclosure () -> String = String(),
file: StaticString = #fileID, line: UInt
= #line)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension GlobalActor {
```

```
    /// Stops program execution if the
current task is not executing on this
    /// actor's serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
    ///
    /// * In playgrounds and `-Onone`
builds (the default for Xcode's Debug
    ///   configuration), stops program
execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for
Xcode's Release configuration),
    ///   the isolation check is not
performed and there are no effects.
    ///
    /// - Note: This check is performed
against the actor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
    ///   that actor's serial executor as
its own ``Actor/unownedExecutor``--this
    ///   check will succeed , as from a
concurrency safety perspective, the
    ///   serial executor guarantees
mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - message: The message to print
if the assertion fails.
```

```
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public static func assertIsolated(_
message: @autoclosure () -> String =
String(), file: StaticString = #fileID,
line: UInt = #line)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension GlobalActor {

    /// Shorthand for referring to the
`shared.unownedExecutor` of this global
actor.
    ///
    /// When declaring a global actor
with a custom executor, prefer to
implement
    /// the underlying actor's
``Actor/unownedExecutor`` property, and
leave this
```

```
    /// `sharedUnownedExecutor` default
implementation in-place as it will simply
    /// delegate to the
`shared.unownedExecutor`.
    ///
    /// The value of this property must
be equivalent to
`shared.unownedExecutor`,
    /// as it may be used by the Swift
concurrency runtime or explicit user code
with
    /// that assumption in mind.
    ///
    /// Returning different executors for
different invocations of this computed
    /// property is also illegal, as it
could lead to inconsistent
synchronization
    /// of the underlying actor.
    ///
    /// - SeeAlso: ``SerialExecutor``
    public static var
sharedUnownedExecutor:
UnownedSerialExecutor { get }
}

/// Deprecated equivalent of
``ExecutorJob``.
///
/// A unit of schedulable work.
///
/// Unless you're implementing a
scheduler,
```

```
/// you don't generally interact with
jobs directly.
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@available(*, deprecated, renamed:
"ExecutorJob")
@frozen public struct Job : ~Copyable,
Sendable {

    public init(_ job: UnownedJob)

    public init(_ job: ExecutorJob)

    public var priority: JobPriority {
get }

    public var description: String {
get }
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension Job {

    /// Run this job on the passed in
executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
```

```
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
    ///
    /// Converting a `ExecutorJob` to an
``UnownedJob`` and invoking
``UnownedJob/runSynchronously(_:)` on it
multiple times is undefined behavior,
    /// as a job can only ever be run
once, and must not be accessed after it
has been run.
    ///
    /// - Parameter executor: the
executor this job will be semantically
running on.
    @inlinable public func
runSynchronously(on executor:
UnownedSerialExecutor)
}

/// The priority of this job.
///
/// The executor determines how priority
```

```
information affects the way tasks are
scheduled.
/// The behavior varies depending on the
executor currently being used.
/// Typically, executors attempt to run
tasks with a higher priority
/// before tasks with a lower priority.
/// However, the semantics of how
priority is treated are left up to each
/// platform and `Executor`
implementation.
///
/// A ExecutorJob's priority is roughly
equivalent to a `TaskPriority`,
/// however, since not all jobs are
tasks, represented as separate type.
///
/// Conversions between the two
priorities are available as initializers
on the respective types.
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@frozen public struct JobPriority :
Sendable {

    public typealias RawValue = UInt8

    /// The raw priority value.
    public var rawValue:
JobPriority.RawValue
}

@available(macOS 14.0, iOS 17.0, watchOS
```

```
10.0, tvOS 17.0, *)
extension JobPriority : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
JobPriority, rhs: JobPriority) -> Bool

    public static func != (lhs:
JobPriority, rhs: JobPriority) -> Bool
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension JobPriority : Comparable {

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is less than that of the
second argument.
    ///
    /// This function is the only
requirement of the `Comparable` protocol.
```

```
The
    /// remainder of the relational
operator functions are implemented by the
    /// standard library for any type
that conforms to `Comparable`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func < (lhs:
JobPriority, rhs: JobPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is less than or equal to
that of the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func <= (lhs:
JobPriority, rhs: JobPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is greater than that of
the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
```

```
compare.
    public static func > (lhs:
JobPriority, rhs: JobPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is greater than or equal
to that of the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func >= (lhs:
JobPriority, rhs: JobPriority) -> Bool
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension JobPriority : BitwiseCopyable {
}

/// A singleton actor whose executor is
equivalent to the main
/// dispatch queue.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@globalActor final public actor MainActor
: GlobalActor {

    /// The shared actor instance that
will be used to provide mutually-
exclusive
```

```
    /// access to declarations annotated
with the given global actor type.
    ///
    /// The value of this property must
always evaluate to the same actor
    /// instance.
    public static let shared: MainActor

    /// Retrieve the executor for this
actor as an optimized, unowned
    /// reference.
    ///
    /// This property must always
evaluate to the same executor for a
    /// given actor instance, and holding
on to the actor must keep the
    /// executor alive.
    ///
    /// This property will be implicitly
accessed when work needs to be
    /// scheduled onto this actor.  These
accesses may be merged,
    /// eliminated, and rearranged with
other work, and they may even
    /// be introduced when not strictly
required.  Visible side effects
    /// are therefore strongly
discouraged within this property.
    ///
    /// - SeeAlso: ``SerialExecutor``
    /// - SeeAlso: ``TaskExecutor``
    @inlinable nonisolated final public
var unownedExecutor:
```

```
UnownedSerialExecutor { get }

    /// Shorthand for referring to the
`shared.unownedExecutor` of this global
actor.
    ///
    /// When declaring a global actor
with a custom executor, prefer to
implement
    /// the underlying actor's
``Actor/unownedExecutor`` property, and
leave this
    /// `sharedUnownedExecutor` default
implementation in-place as it will simply
    /// delegate to the
`shared.unownedExecutor`.
    ///
    /// The value of this property must
be equivalent to
`shared.unownedExecutor`,
    /// as it may be used by the Swift
concurrency runtime or explicit user code
with
    /// that assumption in mind.
    ///
    /// Returning different executors for
different invocations of this computed
    /// property is also illegal, as it
could lead to inconsistent
synchronization
    /// of the underlying actor.
    ///
    /// - SeeAlso: ``SerialExecutor``
```

```
    @inlinable public static var
sharedUnownedExecutor:
UnownedSerialExecutor { get }

    @inlinable nonisolated final public
func enqueue(_ job: UnownedJob)

    /// The type of the shared actor
instance that will be used to provide
    /// mutually-exclusive access to
declarations annotated with the given
global
    /// actor type.
    @available(iOS 13.0, tvOS 13.0,
watchOS 6.0, macOS 10.15, *)
    public typealias ActorType =
MainActor
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension MainActor {

    /// Execute the given body closure on
the main actor.
    public static func run<T>(resultType:
T.Type = T.self, body: @MainActor
@Sendable () throws -> T) async rethrows
-> T where T : Sendable
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
```

```
extension MainActor {

    /// Assume that the current task is
executing on the main actor's
    /// serial executor, or stop program
execution.
    ///
    /// This method allows to *assume and
verify* that the currently
    /// executing synchronous function is
actually executing on the serial
    /// executor of the MainActor.
    ///
    /// If that is the case, the
operation is invoked with an `isolated`
version
    /// of the actor, / allowing
synchronous access to actor local state
without
    /// hopping through asynchronous
boundaries.
    ///
    /// If the current context is not
running on the actor's serial executor,
or
    /// if the actor is a reference to a
remote actor, this method will crash
    /// with a fatal error (similar to
``preconditionIsolated()``).
    ///
    /// This method can only be used from
synchronous functions, as asynchronous
    /// functions should instead perform
```

```
a normal method call to the actor, which
    /// will hop task execution to the
target actor if necessary.
    ///
    /// - Note: This check is performed
against the MainActor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
    ///
``MainActor/sharedUnownedExecutor`` as
its own
    ///   ``Actor/unownedExecutor``--this
check will succeed , as from a
concurrency
    ///   safety perspective, the serial
executor guarantees mutual exclusion of
    ///   those two actors.
    ///
    /// - Parameters:
    ///   - operation: the operation that
will be executed if the current context
    ///                is executing on
the MainActor's serial executor.
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    /// - Returns: the return value of
```

```
the `operation`
    /// - Throws: rethrows the `Error`
thrown by the operation if it threw
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    public static func
assumeIsolated<T>(_ operation: @MainActor
() throws -> T, file: StaticString =
#fileID, line: UInt = #line) rethrows ->
T where T : Sendable
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@available(*, deprecated, renamed:
"UnownedJob")
public typealias PartialAsyncTask =
UnownedJob

/// A service that executes jobs.
///
/// ### Custom Actor Executors
/// By default, all actor types execute
tasks on a shared global concurrent pool.
/// The global pool does not guarantee
any thread (or dispatch queue) affinity,
/// so actors are free to use different
threads as they execute tasks.
///
/// > The runtime may perform various
optimizations to minimize un-necessary
/// > thread switching.
///
```

```
/// Sometimes it is important to be able
to customize the execution behavior
///  of an actor. For example, when an
actor is known to perform heavy blocking
/// operations (such as IO), and we would
like to keep this work *off* the global
/// shared pool, as blocking it may
prevent other actors from being
responsive.
///
/// You can implement a custom executor,
by conforming a type to the
/// ``SerialExecutor`` protocol, and
implementing the ``enqueue(_:)`` method.
///
/// Once implemented, you can configure
an actor to use such executor by
/// implementing the actor's
``Actor/unownedExecutor`` computed
property.
/// For example, you could accept an
executor in the actor's initializer,
/// store it as a variable (in order to
retain it for the duration of the
/// actor's lifetime), and return it from
the `unownedExecutor` computed
/// property like this:
///
/// ```
/// actor MyActor {
///   let myExecutor: MyExecutor
///
///   // accepts an executor to run this
```

```
actor on.
///   init(executor: MyExecutor) {
///     self.myExecutor = executor
///   }
///
///   nonisolated var unownedExecutor:
UnownedSerialExecutor {
///
self.myExecutor.asUnownedSerialExecutor()
///   }
/// }
/// ```
///
/// It is also possible to use a form of
shared executor, either created as a
/// global or static property, which you
can then re-use for every MyActor
/// instance:
///
/// ```
/// actor MyActor {
///   // Serial executor reused by *all*
instances of MyActor!
///   static let sharedMyActorsExecutor =
MyExecutor() // implements SerialExecutor
///
///
///   nonisolated var unownedExecutor:
UnownedSerialExecutor {
///
Self.sharedMyActorsExecutor.asUnownedSeri
alExecutor()
///   }
```

```
/// }
/// ```
///
/// In the example above, *all* "MyActor"
instances would be using the same
/// serial executor, which would result
in only one of such actors ever being
/// run at the same time. This may be
useful if some of your code has some
/// "specific thread" requirement when
interoperating with non-Swift runtimes
/// for example.
///
/// Since the ``UnownedSerialExecutor``
returned by the `unownedExecutor`
/// property *does not* retain the
executor, you must make sure the lifetime
of
/// it extends beyond the lifetime of any
actor or task using it, as otherwise
/// it may attempt to enqueue work on a
released executor object, causing a
crash.
/// The executor returned by
unownedExecutor *must* always be the same
object,
/// and returning different executors can
lead to unexpected behavior.
///
/// Alternatively, you can also use
existing serial executor implementations,
/// such as Dispatch's
`DispatchSerialQueue` or others.
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public protocol SerialExecutor : Executor
{

    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @available(*, deprecated, message:
"Implement 'enqueue(_: consuming
ExecutorJob)' instead")
    func enqueue(_ job: UnownedJob)

    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    @available(*, deprecated, message:
"Implement 'enqueue(_: consuming
ExecutorJob)' instead")
    func enqueue(_ job: consuming Job)

    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    func enqueue(_ job: consuming
ExecutorJob)

    /// Convert this executor value to
the optimized form of borrowed
    /// executor references.
    func asUnownedSerialExecutor() ->
UnownedSerialExecutor

    /// If this executor has complex
equality semantics, and the runtime needs
to
```

```
    /// compare two executors, it will
first attempt the usual pointer-based
    /// equality / check, / and if it
fails it will compare the types of both
    /// executors, if they are the
same, / it will finally invoke this
method,
    ///  in an
    /// attempt to let the executor
itself decide / if this and the `other`
    /// executor represent the same
serial, exclusive, isolation context.
    ///
    /// This method must be implemented
with great care, as wrongly returning
    /// `true` would allow / code from a
different execution context (e.g. thread)
    /// to execute code which was
intended to be isolated by another actor.
    ///
    /// This check is not used when
performing executor switching.
    ///
    /// This check is used when
performing ``Actor/assertIsolated()``,
    /// ``Actor/preconditionIsolated()``,
``Actor/assumeIsolated()`` and similar
    /// APIs which assert about the same
"exclusive serial execution context".
    ///
    /// - Parameter other: the executor
to compare with.
    /// - Returns: `true`, if `self` and
```

```
the `other` executor actually are
    ///             mutually exclusive and
it is safe—from a concurrency
    ///             perspective—to execute
code assuming one on the other.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    func
isSameExclusiveExecutionContext(other:
Self) -> Bool

    /// Last resort "fallback" isolation
check, called when the concurrency
runtime
    /// is comparing executors e.g.
during ``assumeIsolated()`` and is unable
to prove
    /// serial equivalence between the
expected (this object), and the current
executor.
    ///
    /// During executor comparison, the
Swift concurrency runtime attempts to
compare
    /// current and expected executors in
a few ways (including "complex" equality
    /// between executors (see
``isSameExclusiveExecutionContext(other:)
``), and if all
    /// those checks fail, this method is
invoked on the expected executor.
    ///
    /// This method MUST crash if it is
```

```
unable to prove that the current
execution
    /// context belongs to this executor.
At this point usual executor comparison
would
    /// have already failed, though the
executor may have some external tracking
of
    /// threads it owns, and may be able
to prove isolation nevertheless.
    ///
    /// A default implementation is
provided that unconditionally crashes the
    /// program, and prevents calling
code from proceeding with potentially
    /// not thread-safe execution.
    ///
    /// - Warning: This method must crash
and halt program execution if unable
    ///     to prove the isolation of the
calling context.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    func checkIsolated()
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension SerialExecutor {

    /// Last resort "fallback" isolation
check, called when the concurrency
runtime
```

```
    /// is comparing executors e.g. during ``assumeIsolated()`` and is unable to prove
    /// serial equivalence between the expected (this object), and the current executor.
    ///
    /// During executor comparison, the Swift concurrency runtime attempts to compare
    /// current and expected executors in a few ways (including "complex" equality
    /// between executors (see ``isSameExclusiveExecutionContext(other:)``), and if all
    /// those checks fail, this method is invoked on the expected executor.
    ///
    /// This method MUST crash if it is unable to prove that the current execution
    /// context belongs to this executor. At this point usual executor comparison would
    /// have already failed, though the executor may have some external tracking of
    /// threads it owns, and may be able to prove isolation nevertheless.
    ///
    /// A default implementation is provided that unconditionally crashes the
    /// program, and prevents calling
```

```
code from proceeding with potentially
    /// not thread-safe execution.
    ///
    /// - Warning: This method must crash
and halt program execution if unable
    ///     to prove the isolation of the
calling context.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public func checkIsolated()
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension SerialExecutor {

    /// Convert this executor value to
the optimized form of borrowed
    /// executor references.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public func asUnownedSerialExecutor()
-> UnownedSerialExecutor
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension SerialExecutor {

    /// If this executor has complex
equality semantics, and the runtime needs
to
    /// compare two executors, it will
```

```
first attempt the usual pointer-based
    /// equality / check, / and if it
fails it will compare the types of both
    /// executors, if they are the
same, / it will finally invoke this
method,
    ///  in an
    /// attempt to let the executor
itself decide / if this and the `other`
    /// executor represent the same
serial, exclusive, isolation context.
    ///
    /// This method must be implemented
with great care, as wrongly returning
    /// `true` would allow / code from a
different execution context (e.g. thread)
    /// to execute code which was
intended to be isolated by another actor.
    ///
    /// This check is not used when
performing executor switching.
    ///
    /// This check is used when
performing ``Actor/assertIsolated()``,
    /// ``Actor/preconditionIsolated()``,
``Actor/assumeIsolated()`` and similar
    /// APIs which assert about the same
"exclusive serial execution context".
    ///
    /// - Parameter other: the executor
to compare with.
    /// - Returns: `true`, if `self` and
the `other` executor actually are
```

```
    ///             mutually exclusive and
it is safe—from a concurrency
    ///             perspective—to execute
code assuming one on the other.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public func
isSameExclusiveExecutionContext(other:
Self) -> Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension SerialExecutor {

    /// Stops program execution if the
current task is not executing on this
    /// serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
    ///
    /// * In playgrounds and `-Onone`
builds (the default for Xcode's Debug
    ///   configuration), stops program
execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for
Xcode's Release configuration), stops
    ///   program execution.
    ///
    /// - Note: Because this check is
```

```
performed against the actor's serial
executor,
    ///   if another actor uses the same
serial executor--by using
    ///   that actor's serial executor as
its own ``Actor/unownedExecutor``--this
    ///   check will succeed. From a
concurrency safety perspective, the
    ///   serial executor guarantees
mutual exclusion of those two actors.
    ///
    /// - Parameters:
    ///   - message: The message to print
if the assertion fails.
    ///   - file: The file name to print
if the assertion fails. The default value
is
    ///           the file where this
method was called.
    ///   - line: The line number to
print if the assertion fails The default
value is
    ///           the line where this
method was called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public func preconditionIsolated(_
message: @autoclosure () -> String =
String(), file: StaticString = #fileID,
line: UInt = #line)
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension SerialExecutor {

    /// Stops program execution if the
current task is not executing on this
    /// serial executor.
    ///
    /// This function's effect varies
depending on the build flag used:
    ///
    /// * In playgrounds and `-Onone`
builds (the default for Xcode's Debug
    ///   configuration), stops program
execution in a debuggable state after
    ///   printing `message`.
    ///
    /// * In `-O` builds (the default for
Xcode's Release configuration),
    ///   the isolation check is not
performed and there are no effects.
    ///
    /// - Note: This check is performed
against the actor's serial executor,
    ///   meaning that / if another actor
uses the same serial executor--by using
    ///   that actor's serial executor as
its own ``Actor/unownedExecutor``--this
    ///   check will succeed , as from a
concurrency safety perspective, the
    ///   serial executor guarantees
mutual exclusion of those two actors.
```

```
    ///
    /// - Parameters:
    ///   - message: The message to print
if the assertion fails.
    ///   - file: The file name to print
if the assertion fails. The default is
    ///           where this method was
called.
    ///   - line: The line number to
print if the assertion fails The default
is
    ///           where this method was
called.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 14.0, iOS
17.0, watchOS 10.0, tvOS 17.0)
    public func assertIsolated(_ message:
@autoclosure () -> String = String(),
file: StaticString = #fileID, line: UInt
= #line)
}

/// A clock that measures time that
always increments but stops incrementing
/// while the system is asleep.
///
/// `SuspendingClock` can be considered
as a system awake time clock. The frame
/// of reference of the `Instant` may be
bound machine boot or some other
/// locally defined reference point. This
means that the instants are
```

```
/// only comparable on the same machine
in the same booted session.
///
/// This clock is suitable for high
resolution measurements of execution.
@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
public struct SuspendingClock : Sendable
{

    public struct Instant : Codable,
Sendable {

        /// Encodes this value into the
given encoder.
        ///
        /// If the value fails to encode
anything, `encoder` will encode an empty
        /// keyed container in its place.
        ///
        /// This function throws an error
if any values are invalid for the given
        /// encoder's format.
        ///
        /// - Parameter encoder: The
encoder to write data to.
        public func encode(to encoder:
any Encoder) throws

        /// Creates a new instance by
decoding from the given decoder.
        ///
        /// This initializer throws an
```

```
error if reading from the decoder fails,
or
        /// if the data read is corrupted
or otherwise invalid.
        ///
        /// - Parameter decoder: The
decoder to read data from.
        public init(from decoder: any
Decoder) throws
    }

    public init()
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension SuspendingClock : Clock {

    /// The current instant accounting
for machine suspension.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public var now:
SuspendingClock.Instant { get }

    /// The current instant accounting
for machine suspension.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static var now:
SuspendingClock.Instant { get }

    /// The minimum non-zero resolution
```

```
between any two calls to `now`.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public var minimumResolution:
Duration { get }

    /// Suspend task execution until a
given deadline within a tolerance.
    /// If no tolerance is specified then
the system may adjust the deadline
    /// to coalesce CPU wake-ups to more
efficiently process the wake-ups in
    /// a more power efficient manner.
    ///
    /// If the task is canceled before
the time ends, this function throws
    /// `CancellationError`.
    ///
    /// This function doesn't block the
underlying thread.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func sleep(until deadline:
SuspendingClock.Instant, tolerance:
Duration? = nil) async throws

    @available(iOS 16.0, tvOS 16.0,
watchOS 9.0, macOS 13.0, *)
    public typealias Duration = Duration
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
```

```
extension SuspendingClock.Instant :
InstantProtocol {

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static var now:
SuspendingClock.Instant { get }

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func advanced(by duration:
Duration) -> SuspendingClock.Instant

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func duration(to other:
SuspendingClock.Instant) -> Duration

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
```

```
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public func hash(into hasher: inout
Hasher)

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func == (lhs:
SuspendingClock.Instant, rhs:
SuspendingClock.Instant) -> Bool

    /// Returns a Boolean value
```

```
indicating whether the value of the first
    /// argument is less than that of the
second argument.
    ///
    /// This function is the only
requirement of the `Comparable` protocol.
The
    /// remainder of the relational
operator functions are implemented by the
    /// standard library for any type
that conforms to `Comparable`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func < (lhs:
SuspendingClock.Instant, rhs:
SuspendingClock.Instant) -> Bool

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func + (lhs:
SuspendingClock.Instant, rhs: Duration)
-> SuspendingClock.Instant

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func += (lhs: inout
SuspendingClock.Instant, rhs: Duration)
```

```
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func - (lhs:
SuspendingClock.Instant, rhs: Duration)
-> SuspendingClock.Instant

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func -= (lhs: inout
SuspendingClock.Instant, rhs: Duration)

    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func - (lhs:
SuspendingClock.Instant, rhs:
SuspendingClock.Instant) -> Duration

    @available(iOS 16.0, tvOS 16.0,
watchOS 9.0, macOS 13.0, *)
    public typealias Duration = Duration

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
```

```
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}

/// A unit of asynchronous work.
///
/// When you create an instance of
`Task`,
/// you provide a closure that contains
the work for that task to perform.
/// Tasks can start running immediately
after creation;
/// you don't explicitly start or
schedule them.
/// After creating a task, you use the
instance to interact with it ---
/// for example, to wait for it to
complete or to cancel it.
/// It's not a programming error to
discard a reference to a task
/// without waiting for that task to
finish or canceling it.
/// A task runs regardless of whether you
keep a reference to it.
/// However, if you discard the reference
to a task,
/// you give up the ability
/// to wait for that task's result or
cancel the task.
///
/// To support operations on the current
```

```
task,
/// which can be either a detached task
or child task,
/// `Task` also exposes class methods
like `yield()`.
/// Because these methods are
asynchronous,
/// they're always invoked as part of an
existing task.
///
/// Only code that's running as part of
the task can interact with that task.
/// To interact with the current task,
/// you call one of the static methods on
`Task`.
///
/// A task's execution can be seen as a
series of periods where the task ran.
/// Each such period ends at a suspension
point or the
/// completion of the task.
/// These periods of execution are
represented by instances of
`PartialAsyncTask`.
/// Unless you're implementing a custom
executor,
/// you don't directly interact with
partial tasks.
///
/// For information about the language-
level concurrency model that `Task` is
part of,
/// see [Concurrency][concurrency] in
```

```
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-book/
///
/// Task Cancellation
/// =================
///
/// Tasks include a shared mechanism for
indicating cancellation,
/// but not a shared implementation for
how to handle cancellation.
/// Depending on the work you're doing in
the task,
/// the correct way to stop that work
varies.
/// Likewise,
/// it's the responsibility of the code
running as part of the task
/// to check for cancellation whenever
stopping is appropriate.
/// In a long-task that includes multiple
pieces,
/// you might need to check for
cancellation at several points,
/// and handle cancellation differently
at each point.
/// If you only need to throw an error to
stop the work,
/// call the `Task.checkCancellation()`
```

```
function to check for cancellation.
/// Other responses to cancellation
include
/// returning the work completed so far,
returning an empty result, or returning
`nil`.
///
/// Cancellation is a purely Boolean
state;
/// there's no way to include additional
information
/// like the reason for cancellation.
/// This reflects the fact that a task
can be canceled for many reasons,
/// and additional reasons can accrue
during the cancellation process.
///
/// ### Task closure lifetime
/// Tasks are initialized by passing a
closure containing the code that will be
executed by a given task.
///
/// After this code has run to
completion, the task has completed,
resulting in either
/// a failure or result value, this
closure is eagerly released.
///
/// Retaining a task object doesn't
indefinitely retain the closure,
/// because any references that a task
holds are released
/// after the task completes.
```

```
/// Consequently, tasks rarely need to
capture weak references to values.
///
/// For example, in the following snippet
of code it is not necessary to capture
the actor as `weak`,
/// because as the task completes it'll
let go of the actor reference, breaking
the
/// reference cycle between the Task and
the actor holding it.
///
/// ```
/// struct Work: Sendable {}
///
/// actor Worker {
///     var work: Task<Void, Never>?
///     var result: Work?
///
///     deinit {
///         // even though the task is
still retained,
///         // once it completes it no
longer causes a reference cycle with the
actor
///
///         print("deinit actor")
///     }
///
///     func start() {
///         work = Task {
///             print("start task work")
///             try? await
```

```
Task.sleep(for: .seconds(3))
///             self.result = Work() //
we captured self
///             print("completed task
work")
///             // but as the task
completes, this reference is released
///         }
///         // we keep a strong reference
to the task
///     }
/// }
/// ```
///
/// And using it like this:
///
/// ```
/// await Worker().start()
/// ```
///
/// Note that the actor is only retained
by the start() method's use of `self`,
/// and that the start method immediately
returns, without waiting for the
/// unstructured `Task` to finish. Once
the task is completed and its closure is
/// destroyed, the strong reference to
the actor is also released allowing the
/// actor to deinitialize as expected.
///
/// Therefore, the above call will
consistently result in the following
output:
```

```
///
/// ```other
/// start task work
/// completed task work
/// deinit actor
/// ```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct Task<Success,
Failure> : Sendable where Success :
Sendable, Failure : Error {
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    @available(*, deprecated, message:
"Task.Priority has been removed; use
TaskPriority")
    public typealias Priority =
TaskPriority

    @available(*, deprecated, message:
"Task.Handle has been removed; use Task")
    public typealias Handle = Task

    @available(*, deprecated, message:
"Task.CancellationError has been removed;
use CancellationError")
    public static func
CancellationError() -> CancellationError
```

```
    @available(*, deprecated, renamed:
"yield()")
    public static func suspend() async
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    @available(*, deprecated, message:
"`Task.withCancellationHandler` has been
replaced by `withTaskCancellationHandler`
and will be removed shortly.")
    public static func
withCancellationHandler<T>(handler:
@Sendable () -> Void, operation: () async
throws -> T) async rethrows -> T
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == any Error
{

    @discardableResult
    @available(*, deprecated, message:
"`Task.runDetached` was replaced by
`Task.detached` and will be removed
shortly.")
    public static func
runDetached(priority: TaskPriority? =
```

```
nil, operation: @escaping @Sendable ()
async throws -> Success) -> Task<Success,
Failure>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    @available(*, deprecated, message:
"`Task.Group` was replaced by
`ThrowingTaskGroup` and `TaskGroup` and
will be removed shortly.")
    public typealias Group<TaskResult> =
ThrowingTaskGroup<TaskResult, any Error>
where TaskResult : Sendable

    @available(*, deprecated, message:
"`Task.withGroup` was replaced by
`withThrowingTaskGroup` and
`withTaskGroup` and will be removed
shortly.")
    public static func
withGroup<TaskResult,
BodyResult>(resultType: TaskResult.Type,
returning returnType: BodyResult.Type =
BodyResult.self, body: (inout
Task<Success, Failure>.Group<TaskResult>)
async throws -> BodyResult) async
rethrows -> BodyResult where TaskResult :
Sendable
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task {

    @available(*, deprecated, message:
"get() has been replaced by .value")
    public func get() async throws ->
Success

    @available(*, deprecated, message:
"getResult() has been replaced
by .result")
    public func getResult() async ->
Result<Success, Failure>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == Never {

    @available(*, deprecated, message:
"get() has been replaced by .value")
    public func get() async -> Success
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task {

    /// The result from a throwing task,
after it completes.
    ///
```

```
    /// If the task hasn't completed,
    /// accessing this property waits for
it to complete
    /// and its priority increases to
that of the current task.
    /// Note that this might not be as
effective as
    /// creating the task with the
correct priority,
    /// depending on the executor's
scheduling details.
    ///
    /// If the task throws an error, this
property propagates that error.
    /// Tasks that respond to
cancellation by throwing
`CancellationError`
    /// have that error propagated here
upon cancellation.
    ///
    /// - Returns: The task's result.
    public var value: Success { get async
throws }

    /// The result or error from a
throwing task, after it completes.
    ///
    /// If the task hasn't completed,
    /// accessing this property waits for
it to complete
    /// and its priority increases to
that of the current task.
    /// Note that this might not be as
```

```
effective as
    /// creating the task with the
correct priority,
    /// depending on the executor's
scheduling details.
    ///
    /// - Returns: If the task succeeded,
    ///   `.success` with the task's
result as the associated value;
    ///   otherwise, `.failure` with the
error as the associated value.
    public var result: Result<Success,
Failure> { get async }

    /// Indicates that the task should
stop running.
    ///
    /// Task cancellation is cooperative:
    /// a task that supports cancellation
    /// checks whether it has been
canceled at various points during its
work.
    ///
    /// Calling this method on a task
that doesn't support cancellation
    /// has no effect.
    /// Likewise, if the task has already
run
    /// past the last point where it
would stop early,
    /// calling this method has no
effect.
    ///
```

```
    /// - SeeAlso:
`Task.checkCancellation()`
    public func cancel()
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == Never {

    /// The result from a nonthrowing
task, after it completes.
    ///
    /// If the task hasn't completed yet,
    /// accessing this property waits for
it to complete
    /// and its priority increases to
that of the current task.
    /// Note that this might not be as
effective as
    /// creating the task with the
correct priority,
    /// depending on the executor's
scheduling details.
    ///
    /// Tasks that never throw an error
can still check for cancellation,
    /// but they need to use an approach
like returning `nil`
    /// instead of throwing an error.
    public var value: Success { get async
}
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task : Hashable {

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    public func hash(into hasher: inout
Hasher)

    /// The hash value.
    ///
```

```
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
```

```
Task<Success, Failure>, rhs:
Task<Success, Failure>) -> Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    /// The current task's priority.
    ///
    /// If you access this property
outside of any task,
    /// this queries the system to
determine the
    /// priority at which the current
function is running.
    /// If the system can't provide a
priority,
    /// this property's value is
`Priority.default`.
    public static var currentPriority:
TaskPriority { get }

    /// The current task's base priority.
    ///
    /// If you access this property
outside of any task, this returns nil
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static var basePriority:
TaskPriority? { get }
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == Never {

    /// Runs the given nonthrowing
operation asynchronously
    /// as part of a new top-level task
on behalf of the current actor.
    ///
    /// Use this function when creating
asynchronous work
    /// that operates on behalf of the
synchronous function that calls it.
    /// Like
`Task.detached(priority:operation:)`,
    /// this function creates a separate,
top-level task.
    /// Unlike
`Task.detached(priority:operation:)`,
    /// the task created by
`Task.init(priority:operation:)`
    /// inherits the priority and actor
context of the caller,
    /// so the operation is treated more
like an asynchronous extension
    /// to the synchronous operation.
    ///
    /// You need to keep a reference to
the task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
```

```
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
    @discardableResult
    public init(priority: TaskPriority? =
nil, operation: sending @escaping
@isolated(any) () async -> Success)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == any Error
{

    /// Runs the given throwing operation
asynchronously
    /// as part of a new top-level task
on behalf of the current actor.
    ///
    /// Use this function when creating
asynchronous work
    /// that operates on behalf of the
synchronous function that calls it.
```

```
    /// Like
`Task.detached(priority:operation:)`,
    /// this function creates a separate,
top-level task.
    /// Unlike
`detach(priority:operation:)`,
    /// the task created by
`Task.init(priority:operation:)`
    /// inherits the priority and actor
context of the caller,
    /// so the operation is treated more
like an asynchronous extension
    /// to the synchronous operation.
    ///
    /// You need to keep a reference to
the task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
    @discardableResult
```

```
    public init(priority: TaskPriority? =
nil, operation: sending @escaping
@isolated(any) () async throws ->
Success)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == Never {

    /// Runs the given nonthrowing
operation asynchronously
    /// as part of a new top-level task.
    ///
    /// Don't use a detached task if it's
possible
    /// to model the operation using
structured concurrency features like
child tasks.
    /// Child tasks inherit the parent
task's priority and task-local storage,
    /// and canceling a parent task
automatically cancels all of its child
tasks.
    /// You need to handle these
considerations manually with a detached
task.
    ///
    /// You need to keep a reference to
the detached task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
```

```
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
task.
    ///   - operation: The operation to
perform.
    ///
    /// - Returns: A reference to the
task.
    @discardableResult
    public static func detached(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async ->
Success) -> Task<Success, Failure>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Failure == any Error
{

    /// Runs the given throwing operation
asynchronously
    /// as part of a new top-level task.
    ///
    /// If the operation throws an error,
this method propagates that error.
    ///
```

```
    /// Don't use a detached task if it's
possible
    /// to model the operation using
structured concurrency features like
child tasks.
    /// Child tasks inherit the parent
task's priority and task-local storage,
    /// and canceling a parent task
automatically cancels all of its child
tasks.
    /// You need to handle these
considerations manually with a detached
task.
    ///
    /// You need to keep a reference to
the detached task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
task.
    ///   - operation: The operation to
perform.
    ///
    /// - Returns: A reference to the
task.
```

```
    @discardableResult
    public static func detached(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async throws
-> Success) -> Task<Success, Failure>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    /// Suspends the current task and
allows other tasks to execute.
    ///
    /// A task can voluntarily suspend
itself
    /// in the middle of a long-running
operation
    /// that doesn't contain any
suspension points,
    /// to let other tasks run for a
while
    /// before execution returns to this
task.
    ///
    /// If this task is the highest-
priority task in the system,
    /// the executor immediately resumes
execution of the same task.
    /// As such,
    /// this method isn't necessarily a
way to avoid resource starvation.
```

```
    public static func yield() async
}

/// Task with specified executor
------------------------------------
------
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension Task where Failure == Never {

    /// Runs the given nonthrowing
operation asynchronously
    /// as part of a new top-level task
on behalf of the current actor.
    ///
    /// This overload allows specifying a
preferred ``TaskExecutor`` on which
    /// the `operation`, as well as all
child tasks created from this task will
be
    /// executing whenever possible.
Refer to ``TaskExecutor`` for a detailed
discussion
    /// of the effect of task executors
on execution semantics of asynchronous
code.
    ///
    /// Use this function when creating
asynchronous work
    /// that operates on behalf of the
synchronous function that calls it.
    /// Like
`Task.detached(priority:operation:)`,
```

```
    /// this function creates a separate,
top-level task.
    /// Unlike
`Task.detached(priority:operation:)`,
    /// the task created by
`Task.init(priority:operation:)`
    /// inherits the priority and actor
context of the caller,
    /// so the operation is treated more
like an asynchronous extension
    /// to the synchronous operation.
    ///
    /// You need to keep a reference to
the task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - taskExecutor: the preferred
task executor for this task,
    ///       and any child tasks created
by it. Explicitly passing `nil` is
    ///       interpreted as "no
preference".
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
```

```
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
    /// - SeeAlso:
``withTaskExecutorPreference(_:operation:
)``
    @discardableResult
    public init(executorPreference
taskExecutor: consuming (any
TaskExecutor)?, priority: TaskPriority? =
nil, operation: sending @escaping ()
async -> Success)
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension Task where Failure == any Error
{

    /// Runs the given throwing operation
asynchronously
    /// as part of a new top-level task
on behalf of the current actor.
    ///
    /// Use this function when creating
asynchronous work
    /// that operates on behalf of the
synchronous function that calls it.
    /// Like
`Task.detached(priority:operation:)`,
    /// this function creates a separate,
top-level task.
    /// Unlike
```

```
`detach(priority:operation:)`,
    /// the task created by
`Task.init(priority:operation:)`
    /// inherits the priority and actor
context of the caller,
    /// so the operation is treated more
like an asynchronous extension
    /// to the synchronous operation.
    ///
    /// You need to keep a reference to
the task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - taskExecutor: the preferred
task executor for this task,
    ///       and any child tasks created
by it. Explicitly passing `nil` is
    ///       interpreted as "no
preference".
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
```

```
    /// - SeeAlso:
``withTaskExecutorPreference(_:operation:
)``
    @discardableResult
    public init(executorPreference
taskExecutor: consuming (any
TaskExecutor)?, priority: TaskPriority? =
nil, operation: sending @escaping ()
async throws -> Success)
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension Task where Failure == Never {

    /// Runs the given nonthrowing
operation asynchronously
    /// as part of a new top-level task.
    ///
    /// Don't use a detached task if it's
possible
    /// to model the operation using
structured concurrency features like
child tasks.
    /// Child tasks inherit the parent
task's priority and task-local storage,
    /// and canceling a parent task
automatically cancels all of its child
tasks.
    /// You need to handle these
considerations manually with a detached
task.
    ///
```

```
    /// You need to keep a reference to
the detached task
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - taskExecutor: the preferred
task executor for this task,
    ///       and any child tasks created
by it. Explicitly passing `nil` is
    ///       interpreted as "no
preference".
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
    /// - Returns: A reference to the
newly created task.
    /// - SeeAlso:
``withTaskExecutorPreference(_:operation:
)``
    @discardableResult
    public static func
detached(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
```

```
TaskPriority? = nil, operation: sending
@escaping () async -> Success) ->
Task<Success, Failure>
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension Task where Failure == any Error
{

    /// Runs the given throwing operation
asynchronously
    /// as part of a new top-level task.
    ///
    /// If the operation throws an error,
this method propagates that error.
    ///
    /// Don't use a detached task if it's
possible
    /// to model the operation using
structured concurrency features like
child tasks.
    /// Child tasks inherit the parent
task's priority and task-local storage,
    /// and canceling a parent task
automatically cancels all of its child
tasks.
    /// You need to handle these
considerations manually with a detached
task.
    ///
    /// You need to keep a reference to
the detached task
```

```
    /// if you want to cancel it by
calling the `Task.cancel()` method.
    /// Discarding your reference to a
detached task
    /// doesn't implicitly cancel that
task,
    /// it only makes it impossible for
you to explicitly cancel the task.
    ///
    /// - Parameters:
    ///   - taskExecutor: the preferred
task executor for this task,
    ///       and any child tasks created
by it. Explicitly passing `nil` is
    ///       interpreted as "no
preference".
    ///   - priority: The priority of the
task.
    ///     Pass `nil` to use the
priority from `Task.currentPriority`.
    ///   - operation: The operation to
perform.
    /// - Returns: A reference to the
newly created task.
    /// - SeeAlso:
``withTaskExecutorPreference(_:operation:
)``
    @discardableResult
    public static func
detached(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
TaskPriority? = nil, operation: sending
@escaping () async throws -> Success) ->
```

```
Task<Success, Failure>
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task {

    /// A Boolean value that indicates
whether the task should stop executing.
    ///
    /// After the value of this property
becomes `true`, it remains `true`
indefinitely.
    /// There is no way to uncancel a
task.
    ///
    /// - SeeAlso: `checkCancellation()`
    public var isCancelled: Bool { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    /// A Boolean value that indicates
whether the task should stop executing.
    ///
    /// After the value of this property
becomes `true`, it remains `true`
indefinitely.
    /// There is no way to uncancel a
task.
```

```
    ///
    /// - SeeAlso: `checkCancellation()`
    public static var isCancelled: Bool {
get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    /// Throws an error if the task was
canceled.
    ///
    /// The error is always an instance
of `CancellationError`.
    ///
    /// - SeeAlso: `isCancelled()`
    public static func
checkCancellation() throws
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension Task where Success == Never,
Failure == Never {

    @available(*, deprecated, renamed:
"Task.sleep(nanoseconds:)")
    public static func sleep(_ duration:
UInt64) async

    /// Suspends the current task for at
```

```
least the given duration
    /// in nanoseconds.
    ///
    /// If the task is canceled before
the time ends,
    /// this function throws
`CancellationError`.
    ///
    /// This function doesn't block the
underlying thread.
    public static func sleep(nanoseconds
duration: UInt64) async throws
}

@available(macOS 13.0, iOS 16.0, watchOS
9.0, tvOS 16.0, *)
extension Task where Success == Never,
Failure == Never {

    /// Suspends the current task until
the given deadline within a tolerance.
    ///
    /// If the task is canceled before
the time ends, this function throws
    /// `CancellationError`.
    ///
    /// This function doesn't block the
underlying thread.
    ///
    ///       try await Task.sleep(until:
.now + .seconds(3))
    ///
    @available(macOS 13.0, iOS 16.0,
```

```
watchOS 9.0, tvOS 16.0, *)
    public static func sleep<C>(until
deadline: C.Instant, tolerance:
C.Instant.Duration? = nil, clock: C =
ContinuousClock()) async throws where C :
Clock

    /// Suspends the current task for the
given duration.
    ///
    /// If the task is cancelled before
the time ends, this function throws
    /// `CancellationError`.
    ///
    /// This function doesn't block the
underlying thread.
    ///
    ///       try await
Task.sleep(for: .seconds(3))
    ///
    @available(macOS 13.0, iOS 16.0,
watchOS 9.0, tvOS 16.0, *)
    public static func sleep<C>(for
duration: C.Instant.Duration, tolerance:
C.Instant.Duration? = nil, clock: C =
ContinuousClock()) async throws where C :
Clock
}

/// An executor that may be used as
preferred executor by a task.
///
/// ### Impact of setting a task executor
```

```
preference
/// By default, without setting a task
executor preference, nonisolated
/// asynchronous functions, as well as
methods declared on default actors --
/// that is actors which do not require a
specific executor -- execute on
/// Swift's default global concurrent
executor. This is an executor shared by
/// the entire runtime to execute any
work which does not have strict executor
/// requirements.
///
/// By setting a task executor
preference, either with a
///
``withTaskExecutorPreference(_:operation:
)``, creating a task with a preference
/// (`Task(executorPreference:)`, or
`group.addTask(executorPreference:)`),
the task and all of its child
/// tasks (unless a new preference is
set) will be preferring to execute on
/// the provided task executor.
///
/// Unstructured tasks do not inherit the
task executor.
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
public protocol TaskExecutor : Executor {

    func enqueue(_ job: UnownedJob)
```

```
    @available(*, deprecated, message:
"Implement 'enqueue(_: consuming
ExecutorJob)' instead")
    func enqueue(_ job: consuming Job)

    func enqueue(_ job: consuming
ExecutorJob)

    func asUnownedTaskExecutor() ->
UnownedTaskExecutor
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension TaskExecutor {

    public func asUnownedTaskExecutor()
-> UnownedTaskExecutor
}

/// A group that contains dynamically
created child tasks.
///
/// To create a task group,
/// call the
`withTaskGroup(of:returning:body:)`
method.
///
/// Don't use a task group from outside
the task where you created it.
/// In most cases,
/// the Swift type system prevents a task
group from escaping like that
```

```
/// because adding a child task to a task
group is a mutating operation,
/// and mutation operations can't be
performed
/// from a concurrent execution context
like a child task.
///
/// ### Task execution order
///
/// Tasks added to a task group execute
concurrently, and may be scheduled in
/// any order.
///
/// ### Cancellation behavior
/// A task group becomes cancelled in one
of the following ways:
///
/// - when ``cancelAll()`` is invoked on
it,
/// - when the ``Task`` running this task
group is cancelled.
///
/// Since a `TaskGroup` is a structured
concurrency primitive, cancellation is
/// automatically propagated through all
of its child-tasks (and their child
/// tasks).
///
/// A cancelled task group can still keep
adding tasks, however they will start
/// being immediately cancelled, and may
act accordingly to this. To avoid adding
/// new tasks to an already cancelled
```

```
task group, use
``addTaskUnlessCancelled(priority:body:)``
`
/// rather than the plain
``addTask(priority:body:)`` which adds
tasks unconditionally.
///
/// For information about the language-
level concurrency model that `TaskGroup`
is part of,
/// see [Concurrency][concurrency] in
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/Languag
eGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-
book/
///
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct
TaskGroup<ChildTaskResult> where
ChildTaskResult : Sendable {

    /// Adds a child task to the group.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
```

```
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(priority: TaskPriority? = nil,
operation: sending @escaping
@isolated(any) () async ->
ChildTaskResult)

    /// Adds a child task to the group,
unless the group has been canceled.
    ///
    /// - Parameters:
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async ->
ChildTaskResult) -> Bool

    /// Wait for the next child task to
complete,
    /// and return the value it returned.
```

```
    ///
    /// The values returned by successive
calls to this method
    /// appear in the order that the
tasks *completed*,
    /// not in the order that those tasks
were added to the task group.
    /// For example:
    ///
    ///     group.addTask { 1 }
    ///     group.addTask { 2 }
    ///
    ///     print(await group.next())
    ///     // Prints either "2" or "1".
    ///
    /// If there aren't any pending tasks
in the task group,
    /// this method returns `nil`,
    /// which lets you write the
following
    /// to wait for a single task to
complete:
    ///
    ///     if let first = try await
group.next() {
    ///         return first
    ///     }
    ///
    /// It also lets you write code like
the following
    /// to wait for all the child tasks
to complete,
    /// collecting the values they
```

```
returned:
    ///
    ///     while let value = try await
group.next() {
    ///         collected += value
    ///     }
    ///     return collected
    ///
    /// Awaiting on an empty group
    /// immediate returns `nil` without
suspending.
    ///
    /// You can also use a `for`-`await`-
`in` loop to collect results of a task
group:
    ///
    ///     for await try value in group
{
    ///         collected += value
    ///     }
    ///
    /// Don't call this method from
outside the task
    /// where you created this task
group.
    /// In most cases, the Swift type
system prevents this mistake.
    /// For example, because the
`add(priority:operation:)` method is
mutating,
    /// that method can't be called from
a concurrent execution context like a
child task.
```

```
    ///
    /// - Returns: The value returned by
the next child task that completes.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
    public mutating func next(isolation:
isolated (any Actor)? = #isolation) async
-> ChildTaskResult?

    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    public mutating func next() async ->
ChildTaskResult?

    /// Wait for all of the group's
remaining tasks to complete.
    public mutating func
waitForAll(isolation: isolated (any
Actor)? = #isolation) async

    /// A Boolean value that indicates
whether the group has any remaining
tasks.
    ///
    /// At the start of the body of a
`withTaskGroup(of:returning:body:)` call,
    /// the task group is always empty.
    /// It`s guaranteed to be empty when
returning from that body
    /// because a task group waits for
```

```
all child tasks to complete before
returning.
    ///
    /// - Returns: `true` if the group
has no pending tasks; otherwise `false`.
    public var isEmpty: Bool { get }

    /// Cancel all of the remaining tasks
in the group.
    ///
    /// If you add a task to a group
after canceling the group,
    /// that task is canceled immediately
after being added to the group.
    ///
    /// Immediately cancelled child tasks
should therefore cooperatively check for
and
    /// react  to cancellation, e.g. by
throwing an `CancellationError` at their
    /// earliest convenience, or
otherwise handling the cancellation.
    ///
    /// There are no restrictions on
where you can call this method.
    /// Code inside a child task or even
another task can cancel a group,
    /// however one should be very
careful to not keep a reference to the
    /// group longer than the
`with...TaskGroup(...) { ... }` method
body is executing.
    ///
```

```
    /// - SeeAlso: `Task.isCancelled`
    /// - SeeAlso:
`TaskGroup.isCancelled`
    public func cancelAll()

    /// A Boolean value that indicates
whether the group was canceled.
    ///
    /// To cancel a group, call the
`TaskGroup.cancelAll()` method.
    ///
    /// If the task that's currently
running this group is canceled,
    /// the group is also implicitly
canceled,
    /// which is also reflected in this
property's value.
    public var isCancelled: Bool { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskGroup {

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
    public mutating func add(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> ChildTaskResult)
async -> Bool

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
```

```
    public mutating func spawn(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> ChildTaskResult)

    @available(*, deprecated, renamed:
"addTaskUnlessCancelled(priority:operatio
n:)")
    public mutating func
spawnUnlessCancelled(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> ChildTaskResult) ->
Bool

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
    public mutating func async(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> ChildTaskResult)

    @available(*, deprecated, renamed:
"addTaskUnlessCancelled(priority:operatio
n:)")
    public mutating func
asyncUnlessCancelled(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> ChildTaskResult) ->
Bool
}

/// ==== TaskGroup: AsyncSequence
-----------------------------------
-----
@available(macOS 10.15, iOS 13.0, watchOS
```

```
6.0, tvOS 13.0, *)
extension TaskGroup : AsyncSequence {

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    public typealias AsyncIterator =
TaskGroup<ChildTaskResult>.Iterator

    /// The type of element produced by
this asynchronous sequence.
    public typealias Element =
ChildTaskResult

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    public func makeAsyncIterator() ->
TaskGroup<ChildTaskResult>.Iterator

    /// A type that provides an iteration
interface
    /// over the results of tasks added
to the group.
    ///
    /// The elements returned by this
iterator
    /// appear in the order that the
```

```
tasks *completed*,
    /// not in the order that those tasks
were added to the task group.
    ///
    /// This iterator terminates after
all tasks have completed.
    /// After iterating over the results
of each task,
    /// it's valid to make a new iterator
for the task group,
    /// which you can use to iterate over
the results of new tasks you add to the
group.
    /// For example:
    ///
    ///     group.addTask { 1 }
    ///     for await r in group
{ print(r) }
    ///
    ///     // Add a new child task and
iterate again.
    ///     group.addTask { 2 }
    ///     for await r in group
{ print(r) }
    ///
    /// - SeeAlso: `TaskGroup.next()`
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    public struct Iterator :
AsyncIteratorProtocol {

        public typealias Element =
ChildTaskResult
```

```
        /// Advances to and returns the
result of the next child task.
        ///
        /// The elements returned from
this method
        /// appear in the order that the
tasks *completed*,
        /// not in the order that those
tasks were added to the task group.
        /// After this method returns
`nil`,
        /// this iterator is guaranteed
to never produce more values.
        ///
        /// For more information about
the iteration order and semantics,
        /// see `TaskGroup.next()`.
        ///
        /// - Returns: The value returned
by the next child task that completes,
        ///   or `nil` if there are no
remaining child tasks,
        public mutating func next() async
->
TaskGroup<ChildTaskResult>.Iterator.Eleme
nt?

        /// Advances to and returns the
result of the next child task.
        ///
        /// The elements returned from
this method
```

```
        /// appear in the order that the
tasks *completed*,
        /// not in the order that those
tasks were added to the task group.
        /// After this method returns
`nil`,
        /// this iterator is guaranteed
to never produce more values.
        ///
        /// For more information about
the iteration order and semantics,
        /// see `TaskGroup.next()`.
        ///
        /// - Returns: The value returned
by the next child task that completes,
        ///   or `nil` if there are no
remaining child tasks,
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public mutating func
next(isolation actor: isolated (any
Actor)?) async ->
TaskGroup<ChildTaskResult>.Iterator.Eleme
nt?

        public mutating func cancel()
    }
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension TaskGroup {
```

```
    /// Adds a child task to the group
and enqueue it on the specified executor.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///     Explicitly passing `nil` as
the executor preference is equivalent to
    ///     calling the `addTask` method
without a preference, and effectively
    ///     means to inherit the outer
context's executor preference.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async ->
ChildTaskResult)

    /// Adds a child task to the group
and enqueue it on the specified executor,
unless the group has been canceled.
    ///
    /// - Parameters:
```

```
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTaskUnlessCancelled()` without
passing a value to the `taskExecutor`
parameter,
    ///                   and it will be
inherited automatically.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(executorPreference
taskExecutor: (any TaskExecutor)?,
priority: TaskPriority? = nil, operation:
sending @escaping @isolated(any) () async
-> ChildTaskResult) -> Bool
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskGroup : BitwiseCopyable {
}

/// Wrapper type that defines a task-
local value key.
///
/// A task-local value is a value that
can be bound and read in the context of a
/// ``Task``. It is implicitly carried
with the task, and is accessible by any
/// child tasks it creates (such as
TaskGroup or `async let` created tasks).
///
/// ### Task-local declarations
///
/// Task locals must be declared as
static properties or global properties,
like this:
///
///     enum Example {
///         @TaskLocal
///         static let traceID: TraceID?
///     }
///
///     // Global task local properties
are supported since Swift 6.0:
///     @TaskLocal
///     var contextualNumber: Int = 12
///
/// ### Default values
```

```
/// Reading a task local value when no
value was bound to it results in
returning
/// its default value. For a task local
declared as optional (such as e.g.
`TraceID?`),
/// this defaults to nil, however a
different default value may be defined at
declaration
/// site of the task local, like this:
///
///     enum Example {
///         @TaskLocal
///         static let traceID: TraceID =
TraceID.default
///     }
///
/// The default value is returned
whenever the task-local is read
/// from a context which either: has no
task available to read the value from
/// (e.g. a synchronous function, called
without any asynchronous function in its
call stack),
/// or no value was bound within the
scope of the current task or any of its
parent tasks.
///
/// ### Reading task-local values
/// Reading task local values is simple
and looks the same as-if reading a normal
/// static property:
///
```

```
///     guard let traceID =
Example.traceID else {
///       print("no trace id")
///       return
///     }
///     print(traceID)
///
/// It is possible to perform task-local
value reads from either asynchronous
/// or synchronous functions.
///
/// ### Binding task-local values
/// Task local values cannot be `set`
directly and must instead be bound using
/// the scoped `$traceID.withValue()
{ ... }` operation. The value is only
bound
/// for the duration of that scope, and
is available to any child tasks which
/// are created within that scope.
///
/// Detached tasks do not inherit task-
local values, however tasks created using
/// the `Task { ... }` initializer do
inherit task-locals by copying them to
the
/// new asynchronous task, even though it
is an un-structured task.
///
/// ### Using task local values outside
of tasks
/// It is possible to bind and read task
local values outside of tasks.
```

```
///
/// This comes in handy within
synchronous functions which are not
guaranteed
/// to be called from within a task. When
binding a task-local value from
/// outside of a task, the runtime will
set a thread-local in which the same
/// storage mechanism as used within
tasks will be used. This means that you
/// can reliably bind and read task local
values without having to worry
/// about the specific calling context,
e.g.:
///
///     func enter() {
///         Example.
$traceID.withValue("1234") {
///             read() // always "1234",
regardless if enter() was called from
inside a task or not:
///     }
///
///     func read() -> String {
///         if let value = Self.traceID {
///             "\(value)"
///         } else {
///             "<no value>"
///         }
///     }
///
///     // 1) Call `enter` from non-Task
code
```

```
///     //    e.g. synchronous main() or
non-Task thread (e.g. a plain pthread)
///     enter()
///
///     // 2) Call 'enter' from Task
///     Task {
///         enter()
///     }
///
/// In either cases listed above, the
binding and reading of the task-local
value works as expected.
///
/// ### Examples
///
///
///     enum Example {
///         @TaskLocal
///         static var traceID: TraceID?
///     }
///
///     func read() -> String {
///         if let value = Self.traceID {
///             "\(value)"
///         } else {
///             "<no value>"
///         }
///     }
///
///     await Example.
$traceID.withValue(1234) { // bind the
value
///       print("traceID: \
```

```
(Example.traceID)") // traceID: 1234
///       read() // traceID: 1234
///
///       async let id = read() // async
let child task, traceID: 1234
///
///       await withTaskGroup(of:
String.self) { group in
///           group.addTask { read() } //
task group child task, traceID: 1234
///           return await group.next()!
///       }
///
///       Task { // unstructured tasks do
inherit task locals by copying
///         read() // traceID: 1234
///       }
///
///       Task.detached { // detached
tasks do not inherit task-local values
///         read() // traceID: nil
///       }
///     }
///
/// - SeeAlso: ``TaskLocal-macro``
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
final public class TaskLocal<Value> :
Sendable, CustomStringConvertible where
Value : Sendable {

    public init(wrappedValue
defaultValue: Value)
```

```
    /// Gets the value currently bound to
this task-local from the current task.
    ///
    /// If no current task is available
in the context where this call is made,
    /// or if the task-local has no value
bound, this will return the
`defaultValue`
    /// of the task local.
    final public func get() -> Value

    /// Binds the task-local to the
specific value for the duration of the
asynchronous operation.
    ///
    /// The value is available throughout
the execution of the operation closure,
    /// including any `get` operations
performed by child-tasks created during
the
    /// execution of the operation
closure.
    ///
    /// If the same task-local is bound
multiple times, be it in the same task,
or
    /// in specific child tasks, the more
specific (i.e. "deeper") binding is
    /// returned when the value is read.
    ///
    /// If the value is a reference type,
it will be retained for the duration of
```

```
    /// the operation closure.
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
    @discardableResult
    @inlinable final public func
withValue<R>(_ valueDuringOperation:
Value, operation: () async throws -> R,
isolation: isolated (any Actor)? =
#isolation, file: String = #fileID, line:
UInt = #line) async rethrows -> R

    /// Binds the task-local to the
specific value for the duration of the
    /// synchronous operation.
    ///
    /// The value is available throughout
the execution of the operation closure,
    /// including any `get` operations
performed by child-tasks created during
the
    /// execution of the operation
closure.
    ///
    /// If the same task-local is bound
multiple times, be it in the same task,
or
    /// in specific child tasks, the
"more specific" binding is returned when
the
    /// value is read.
```

```
    ///
    /// If the value is a reference type,
it will be retained for the duration of
    /// the operation closure.
    @discardableResult
    @inlinable final public func
withValue<R>(_ valueDuringOperation:
Value, operation: () throws -> R, file:
String = #fileID, line: UInt = #line)
rethrows -> R

    final public var projectedValue:
TaskLocal<Value>

    final public var wrappedValue: Value
{ get }

    /// A textual representation of this
instance.
    ///
    /// Calling this property directly is
discouraged. Instead, convert an
    /// instance of any type to a string
by using the `String(describing:)`
    /// initializer. This initializer
works with any type, and uses the custom
    /// `description` property for types
that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point:
CustomStringConvertible {
    ///         let x: Int, y: Int
```

```
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string
in the assignment to `s` uses the
    /// `Point` type's `description`
property.
    final public var description: String
{ get }
}

/// Macro that introduces a ``TaskLocal-
class`` binding.
///
/// For information about task-local
bindings, see ``TaskLocal-class``.
///
/// - SeeAlso: ``TaskLocal-class``
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@attached(accessor) @attached(peer,
names: prefixed(`$`)) public macro
TaskLocal() = #externalMacro(module:
"SwiftMacros", type: "TaskLocalMacro")
```

```
/// The priority of a task.
///
/// The executor determines how priority
information affects the way tasks are
scheduled.
/// The behavior varies depending on the
executor currently being used.
/// Typically, executors attempt to run
tasks with a higher priority
/// before tasks with a lower priority.
/// However, the semantics of how
priority is treated are left up to each
/// platform and `Executor`
implementation.
///
/// Child tasks automatically inherit
their parent task's priority.
/// Detached tasks created by
`detach(priority:operation:)` don't
inherit task priority
/// because they aren't attached to the
current task.
///
/// In some situations the priority of a
task is elevated ---
/// that is, the task is treated as it if
had a higher priority,
/// without actually changing the
priority of the task:
///
/// - If a task runs on behalf of an
actor,
///   and a new higher-priority task is
```

```
enqueued to the actor,
///   then the actor's current task is
temporarily elevated
///   to the priority of the enqueued
task.
///   This priority elevation allows the
new task
///   to be processed at the priority it
was enqueued with.
/// - If a a higher-priority task calls
the `get()` method,
///   then the priority of this task
increases until the task completes.
///
/// In both cases, priority elevation
helps you prevent a low-priority task
/// from blocking the execution of a high
priority task,
/// which is also known as *priority
inversion*.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct TaskPriority :
RawRepresentable, Sendable {

    /// The raw type that can be used to
represent all values of the conforming
    /// type.
    ///
    /// Every distinct value of the
conforming type has a corresponding
unique
    /// value of the `RawValue` type, but
```

```
there may be values of the `RawValue`
    /// type that don't have a
corresponding value of the conforming
type.
    public typealias RawValue = UInt8

    /// The corresponding value of the
raw type.
    ///
    /// A new instance initialized with
`rawValue` will be equivalent to this
    /// instance. For example:
    ///
    ///     enum PaperSize: String {
    ///         case A4, A5, Letter,
Legal
    ///     }
    ///
    ///     let selectedSize =
PaperSize.Letter
    ///     print(selectedSize.rawValue)
    ///     // Prints "Letter"
    ///
    ///     print(selectedSize ==
PaperSize(rawValue:
selectedSize.rawValue)!)
    ///     // Prints "true"
    public var rawValue: UInt8

    /// Creates a new instance with the
specified raw value.
    ///
    /// If there is no value of the type
```

```
that corresponds with the specified raw
    /// value, this initializer returns
`nil`. For example:
    ///
    ///     enum PaperSize: String {
    ///         case A4, A5, Letter,
Legal
    ///     }
    ///
    ///     print(PaperSize(rawValue:
"Legal"))
    ///     // Prints
"Optional("PaperSize.Legal")"
    ///
    ///     print(PaperSize(rawValue:
"Tabloid"))
    ///     // Prints "nil"
    ///
    /// - Parameter rawValue: The raw
value to use for the new instance.
    public init(rawValue: UInt8)

    public static let high: TaskPriority

    public static var medium:
TaskPriority { get }

    public static let low: TaskPriority

    public static let userInitiated:
TaskPriority

    public static let utility:
```

```
TaskPriority

    public static let background:
TaskPriority

    @available(*, deprecated, renamed:
"medium")
    public static let `default`:
TaskPriority
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension TaskPriority {

    /// Convert this
``UnownedJob/Priority`` to a
``TaskPriority``.
    ///
    /// Most values are directly
interchangeable, but this initializer
reserves the right to fail for certain
values.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public init?(_ p: JobPriority)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskPriority {

    @available(*, deprecated, message:
```

```
"unspecified priority will be removed;
use nil")
    public static var unspecified:
TaskPriority { get }

    @available(*, deprecated, message:
"userInteractive priority will be
removed")
    public static var userInteractive:
TaskPriority { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskPriority : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
TaskPriority, rhs: TaskPriority) -> Bool

    public static func != (lhs:
TaskPriority, rhs: TaskPriority) -> Bool
```

```
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskPriority : Comparable {

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is less than that of the
second argument.
    ///
    /// This function is the only
requirement of the `Comparable` protocol.
The
    /// remainder of the relational
operator functions are implemented by the
    /// standard library for any type
that conforms to `Comparable`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func < (lhs:
TaskPriority, rhs: TaskPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is less than or equal to
that of the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
```

```
    ///   - rhs: Another value to
compare.
    public static func <= (lhs:
TaskPriority, rhs: TaskPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is greater than that of
the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func > (lhs:
TaskPriority, rhs: TaskPriority) -> Bool

    /// Returns a Boolean value
indicating whether the value of the first
    /// argument is greater than or equal
to that of the second argument.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func >= (lhs:
TaskPriority, rhs: TaskPriority) -> Bool
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension TaskPriority :
```

```
CustomStringConvertible {

    /// A textual representation of this
instance.
    ///
    /// Calling this property directly is
discouraged. Instead, convert an
    /// instance of any type to a string
by using the `String(describing:)`
    /// initializer. This initializer
works with any type, and uses the custom
    /// `description` property for types
that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point:
CustomStringConvertible {
    ///         let x: Int, y: Int
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string
in the assignment to `s` uses the
    /// `Point` type's `description`
property.
```

```
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public var description: String {
get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension TaskPriority : Codable {
}

/// A throwing discarding group that
contains dynamically created child tasks.
///
/// To create a discarding task group,
/// call the
``withDiscardingTaskGroup(returning:body:
)`` method.
///
/// Don't use a task group from outside
the task where you created it.
/// In most cases,
/// the Swift type system prevents a task
group from escaping like that
/// because adding a child task to a task
group is a mutating operation,
/// and mutation operations can't be
performed
/// from a concurrent execution context
like a child task.
///
/// ### Task execution order
/// Tasks added to a task group execute
```

```
concurrently, and may be scheduled in
/// any order.
///
/// ### Discarding behavior
/// A discarding task group eagerly
discards and releases its child tasks as
/// soon as they complete. This allows
for the efficient releasing of memory
used
/// by those tasks, which are not
retained for future `next()` calls, as
would
/// be the case with a ``TaskGroup``.
///
/// ### Cancellation behavior
/// A throwing discarding task group
becomes cancelled in one of the following
ways:
///
/// - when ``cancelAll()`` is invoked on
it,
/// - when an error is thrown out of the
`withThrowingDiscardingTaskGroup { ... }`
closure,
/// - when the ``Task`` running this task
group is cancelled.
///
/// But also, and uniquely in
*discarding* task groups:
/// - when *any* of its child tasks
throws.
///
/// The group becoming cancelled
```

```
automatically, and cancelling all of its
child tasks,
/// whenever *any* child task throws an
error is a behavior unique to discarding
task groups,
/// because achieving such semantics is
not possible otherwise, due to the
missing `next()` method
/// on discarding groups. Accumulating
task groups can implement this by
manually polling `next()`
/// and deciding to `cancelAll()` when
they decide an error should cause the
group to become cancelled,
/// however a discarding group cannot
poll child tasks for results and
therefore assumes that child
/// task throws are an indication of a
group wide failure. In order to avoid
such behavior,
/// use a ``DiscardingTaskGroup`` instead
of a throwing one, or catch specific
errors in
/// operations submitted using `addTask`
///
/// Since a `ThrowingDiscardingTaskGroup`
is a structured concurrency primitive,
cancellation is
/// automatically propagated through all
of its child-tasks (and their child
/// tasks).
///
/// A cancelled task group can still keep
```

```
adding tasks, however they will start
/// being immediately cancelled, and may
act accordingly to this. To avoid adding
/// new tasks to an already cancelled
task group, use
``addTaskUnlessCancelled(priority:body:)`
`
/// rather than the plain
``addTask(priority:body:)`` which adds
tasks unconditionally.
///
/// For information about the language-
level concurrency model that
`DiscardingTaskGroup` is part of,
/// see [Concurrency][concurrency] in
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/Languag
eGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-
book/
///
/// - SeeAlso: ``TaskGroup``
/// - SeeAlso: ``ThrowingTaskGroup``
/// - SeeAlso: ``DiscardingTaskGroup``
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@frozen public struct
ThrowingDiscardingTaskGroup<Failure>
where Failure : Error {

    public mutating func
```

```
addTask(priority: TaskPriority? = nil,
operation: sending @escaping
@isolated(any) () async throws -> Void)

    public mutating func
addTaskUnlessCancelled(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async throws
-> Void) -> Bool

    /// A Boolean value that indicates
whether the group has any remaining
tasks.
    ///
    /// At the start of the body of a
`withThrowingDiscardingTaskGroup(returnin
g:body:)` call,
    /// the task group is always empty.
    ///
    /// It's guaranteed to be empty when
returning from that body
    /// because a task group waits for
all child tasks to complete before
returning.
    ///
    /// - Returns: `true` if the group
has no pending tasks; otherwise `false`.
    public var isEmpty: Bool { get }

    /// Cancel all of the remaining tasks
in the group.
    ///
    /// If you add a task to a group
```

```
after canceling the group,
    /// that task is canceled immediately
after being added to the group.
    ///
    /// Immediately cancelled child tasks
should therefore cooperatively check for
and
    /// react  to cancellation, e.g. by
throwing an `CancellationError` at their
    /// earliest convenience, or
otherwise handling the cancellation.
    ///
    /// There are no restrictions on
where you can call this method.
    /// Code inside a child task or even
another task can cancel a group,
    /// however one should be very
careful to not keep a reference to the
    /// group longer than the
`with...TaskGroup(...) { ... }` method
body is executing.
    ///
    /// - SeeAlso: `Task.isCancelled`
    /// - SeeAlso:
`ThrowingDiscardingTaskGroup.isCancelled`
    public func cancelAll()

    /// A Boolean value that indicates
whether the group was canceled.
    ///
    /// To cancel a group, call the
`ThrowingDiscardingTaskGroup.cancelAll()`
method.
```

```
    ///
    /// If the task that's currently
running this group is canceled,
    /// the group is also implicitly
canceled,
    /// which is also reflected in this
property's value.
    public var isCancelled: Bool { get }
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension ThrowingDiscardingTaskGroup {

    /// Adds a child task to the group
and set it up with the passed in task
executor preference.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTask()` without passing a value to
the `taskExecutor` parameter,
    ///                   and it will be
inherited automatically.
```

```
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async throws
-> Void)

    /// Adds a child task to the group
and set it up with the passed in task
executor preference,
    /// unless the group has been
canceled.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTask()` without passing a value to
```

```
the `taskExecutor` parameter,
    ///                     and it will be
inherited automatically.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
addTaskUnlessCancelled(executorPreference
taskExecutor: (any TaskExecutor)?,
priority: TaskPriority? = nil, operation:
sending @escaping @isolated(any) () async
throws -> Void) -> Bool
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension ThrowingDiscardingTaskGroup :
BitwiseCopyable {
}

/// A group that contains throwing,
dynamically created child tasks.
///
/// To create a throwing task group,
/// call the
```

```
`withThrowingTaskGroup(of:returning:body:)` method.
///
/// Don't use a task group from outside the task where you created it.
/// In most cases,
/// the Swift type system prevents a task group from escaping like that
/// because adding a child task to a task group is a mutating operation,
/// and mutation operations can't be performed
/// from concurrent execution contexts like a child task.
///
/// ### Task execution order
/// Tasks added to a task group execute concurrently, and may be scheduled in
/// any order.
///
/// ### Cancellation behavior
/// A task group becomes cancelled in one of the following ways:
///
/// - when ``cancelAll()`` is invoked on it,
/// - when an error is thrown out of the `withThrowingTaskGroup(...) { }` closure,
/// - when the ``Task`` running this task group is cancelled.
///
/// Since a `ThrowingTaskGroup` is a structured concurrency primitive,
```

```
cancellation is
/// automatically propagated through all
of its child-tasks (and their child
/// tasks).
///
/// A cancelled task group can still keep
adding tasks, however they will start
/// being immediately cancelled, and may
act accordingly to this. To avoid adding
/// new tasks to an already cancelled
task group, use
``addTaskUnlessCancelled(priority:body:)`
`
/// rather than the plain
``addTask(priority:body:)`` which adds
tasks unconditionally.
///
/// For information about the language-
level concurrency model that
`ThrowingTaskGroup` is part of,
/// see [Concurrency][concurrency] in
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/Languag
eGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-
book/
///
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct
ThrowingTaskGroup<ChildTaskResult,
```

```
Failure> where ChildTaskResult :
Sendable, Failure : Error {

    /// Wait for all of the group's
remaining tasks to complete.
    ///
    /// If any of the tasks throw, the
*first* error thrown is captured
    /// and re-thrown by this method
although the task group is *not*
cancelled
    /// when this happens.
    ///
    /// ### Cancelling the task group on
first error
    ///
    /// If you want to cancel the task
group, and all "sibling" tasks,
    /// whenever any of child tasks
throws an error, use the following
pattern instead:
    ///
    /// ```
    /// while !group.isEmpty {
    ///     do {
    ///         try await group.next()
    ///     } catch is CancellationError
{
    ///         // we decide that
cancellation errors thrown by children,
    ///         // should not cause
cancellation of the entire group.
    ///         continue;
```

```
    ///     } catch {
    ///         // other errors though we
print and cancel the group,
    ///         // and all of the
remaining child tasks within it.
    ///         print("Error: \(error)")
    ///         group.cancelAll()
    ///     }
    /// }
    /// assert(group.isEmpty())
    /// ```
    ///
    /// - Throws: The *first* error that
was thrown by a child task during
draining all the tasks.
    ///           This first error is
stored until all other tasks have
completed, and is re-thrown afterwards.
    public mutating func
waitForAll(isolation: isolated (any
Actor)? = #isolation) async throws

    /// Adds a child task to the group.
    ///
    /// This method doesn't throw an
error, even if the child task does.
    /// Instead, the corresponding call
to `ThrowingTaskGroup.next()` rethrows
that error.
    ///
    /// - Parameters:
    ///   - overridingPriority: The
priority of the operation task.
```

```
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(priority: TaskPriority? = nil,
operation: sending @escaping
@isolated(any) () async throws ->
ChildTaskResult)

    /// Adds a child task to the group,
unless the group has been canceled.
    ///
    /// This method doesn't throw an
error, even if the child task does.
    /// Instead, the corresponding call
to `ThrowingTaskGroup.next()` rethrows
that error.
    ///
    /// - Parameters:
    ///   - overridingPriority: The
priority of the operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
```

```
    public mutating func
addTaskUnlessCancelled(priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async throws
-> ChildTaskResult) -> Bool

    /// Wait for the next child task to
complete,
    /// and return the value it returned
or rethrow the error it threw.
    ///
    /// The values returned by successive
calls to this method
    /// appear in the order that the
tasks *completed*,
    /// not in the order that those tasks
were added to the task group.
    /// For example:
    ///
    ///     group.addTask { 1 }
    ///     group.addTask { 2 }
    ///
    ///     print(await group.next())
    ///     // Prints either "2" or "1".
    ///
    /// If there aren't any pending tasks
in the task group,
    /// this method returns `nil`,
    /// which lets you write the
following
    /// to wait for a single task to
complete:
    ///
```

```
    ///     if let first = try await
group.next() {
    ///         return first
    ///     }
    ///
    /// It also lets you write code like
the following
    /// to wait for all the child tasks
to complete,
    /// collecting the values they
returned:
    ///
    ///     while let first = try await
group.next() {
    ///         collected += value
    ///     }
    ///     return collected
    ///
    /// Awaiting on an empty group
    /// immediately returns `nil` without
suspending.
    ///
    /// You can also use a `for`-`await`-
`in` loop to collect results of a task
group:
    ///
    ///     for try await value in group
{
    ///         collected += value
    ///     }
    ///
    /// If the next child task throws an
error
```

```
    /// and you propagate that error from
this method
    /// out of the body of a call to the
    ///
`ThrowingTaskGroup.withThrowingTaskGroup(
of:returning:body:)` method,
    /// then all remaining child tasks in
that group are implicitly canceled.
    ///
    /// Don't call this method from
outside the task
    /// where this task group was
created.
    /// In most cases, the Swift type
system prevents this mistake;
    /// for example, because the
`add(priority:operation:)` method is
mutating,
    /// that method can't be called from
a concurrent execution context like a
child task.
    ///
    /// - Returns: The value returned by
the next child task that completes.
    ///
    /// - Throws: The error thrown by the
next child task that completes.
    ///
    /// - SeeAlso: `nextResult()`
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    @backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
```

```
2.0)
    public mutating func next(isolation:
isolated (any Actor)? = #isolation) async
throws -> ChildTaskResult?

    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
    public mutating func next() async
throws -> ChildTaskResult?

    /// Wait for the next child task to
complete,
    /// and return a result containing
either
    /// the value that the child task
returned or the error that it threw.
    ///
    /// The values returned by successive
calls to this method
    /// appear in the order that the
tasks *completed*,
    /// not in the order that those tasks
were added to the task group.
    /// For example:
    ///
    ///     group.addTask { 1 }
    ///     group.addTask { 2 }
    ///
    ///     guard let result = await
group.nextResult() else {
    ///         return  // No task to
wait on, which won't happen in this
example.
```

```
    ///     }
    ///
    ///     switch result {
    ///     case .success(let value):
print(value)
    ///     case .failure(let error):
print("Failure: \(error)")
    ///     }
    ///     // Prints either "2" or "1".
    ///
    /// If the next child task throws an
error
    /// and you propagate that error from
this method
    /// out of the body of a call to the
    ///
`ThrowingTaskGroup.withThrowingTaskGroup(
of:returning:body:)` method,
    /// then all remaining child tasks in
that group are implicitly canceled.
    ///
    /// - Returns: A `Result.success`
value
    ///   containing the value that the
child task returned,
    ///   or a `Result.failure` value
    ///   containing the error that the
child task threw.
    ///
    /// - SeeAlso: `next()`
    public mutating func
nextResult(isolation: isolated (any
Actor)? = #isolation) async ->
```

```
Result<ChildTaskResult, Failure>?

    /// A Boolean value that indicates
whether the group has any remaining
tasks.
    ///
    /// At the start of the body of a
`withThrowingTaskGroup(of:returning:body:
)` call,
    /// the task group is always empty.
    ///
    /// It's guaranteed to be empty when
returning from that body
    /// because a task group waits for
all child tasks to complete before
returning.
    ///
    /// - Returns: `true` if the group
has no pending tasks; otherwise `false`.
    public var isEmpty: Bool { get }

    /// Cancel all of the remaining tasks
in the group.
    ///
    /// If you add a task to a group
after canceling the group,
    /// that task is canceled immediately
after being added to the group.
    ///
    /// Immediately cancelled child tasks
should therefore cooperatively check for
and
    /// react  to cancellation, e.g. by
```

```
throwing an `CancellationError` at their
    /// earliest convenience, or
otherwise handling the cancellation.
    ///
    /// There are no restrictions on
where you can call this method.
    /// Code inside a child task or even
another task can cancel a group,
    /// however one should be very
careful to not keep a reference to the
    /// group longer than the
`with...TaskGroup(...) { ... }` method
body is executing.
    ///
    /// - SeeAlso: `Task.isCancelled`
    /// - SeeAlso:
`ThrowingTaskGroup.isCancelled`
    public func cancelAll()

    /// A Boolean value that indicates
whether the group was canceled.
    ///
    /// To cancel a group, call the
`ThrowingTaskGroup.cancelAll()` method.
    ///
    /// If the task that's currently
running this group is canceled,
    /// the group is also implicitly
canceled,
    /// which is also reflected in this
property's value.
    public var isCancelled: Bool { get }
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension ThrowingTaskGroup {

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
    public mutating func add(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws ->
ChildTaskResult) async -> Bool

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
    public mutating func spawn(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws ->
ChildTaskResult)

    @available(*, deprecated, renamed:
"addTaskUnlessCancelled(priority:operatio
n:)")
    public mutating func
spawnUnlessCancelled(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws ->
ChildTaskResult) -> Bool

    @available(*, deprecated, renamed:
"addTask(priority:operation:)")
    public mutating func async(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws ->
```

```
ChildTaskResult)

    @available(*, deprecated, renamed:
"addTaskUnlessCancelled(priority:operatio
n:)")
    public mutating func
asyncUnlessCancelled(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws ->
ChildTaskResult) -> Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension ThrowingTaskGroup :
AsyncSequence {

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    public typealias AsyncIterator =
ThrowingTaskGroup<ChildTaskResult,
Failure>.Iterator

    /// The type of element produced by
this asynchronous sequence.
    public typealias Element =
ChildTaskResult

    /// Creates the asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    ///
```

```
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
    /// elements of the asynchronous
sequence.
    public func makeAsyncIterator() ->
ThrowingTaskGroup<ChildTaskResult,
Failure>.Iterator

    /// A type that provides an iteration
interface
    /// over the results of tasks added
to the group.
    ///
    /// The elements returned by this
iterator
    /// appear in the order that the
tasks *completed*,
    /// not in the order that those tasks
were added to the task group.
    ///
    /// This iterator terminates after
all tasks have completed successfully,
    /// or after any task completes by
throwing an error.
    /// If a task completes by throwing
an error,
    /// it doesn't return any further
task results.
    /// After iterating over the results
of each task,
    /// it's valid to make a new iterator
for the task group,
    /// which you can use to iterate over
```

```
the results of new tasks you add to the
group.
    /// You can also make a new iterator
to resume iteration
    /// after a child task throws an
error.
    /// For example:
    ///
    ///     group.addTask { 1 }
    ///     group.addTask { throw
SomeError }
    ///     group.addTask { 2 }
    ///
    ///     do {
    ///         // Assuming the child
tasks complete in order, this prints "1"
    ///         // and then throws an
error.
    ///         for try await r in group
{ print(r) }
    ///     } catch {
    ///         // Resolve the error.
    ///     }
    ///
    ///     // Assuming the child tasks
complete in order, this prints "2".
    ///     for try await r in group
{ print(r) }
    ///
    /// - SeeAlso:
`ThrowingTaskGroup.next()`
    @available(macOS 10.15, iOS 13.0,
watchOS 6.0, tvOS 13.0, *)
```

```
    public struct Iterator :
AsyncIteratorProtocol {

        public typealias Element =
ChildTaskResult

        /// Advances to and returns the
result of the next child task.
        ///
        /// The elements returned from
this method
        /// appear in the order that the
tasks *completed*,
        /// not in the order that those
tasks were added to the task group.
        /// After this method returns
`nil`,
        /// this iterator is guaranteed
to never produce more values.
        ///
        /// For more information about
the iteration order and semantics,
        /// see
`ThrowingTaskGroup.next()`
        ///
        /// - Throws: The error thrown by
the next child task that completes.
        ///
        /// - Returns: The value returned
by the next child task that completes,
        ///   or `nil` if there are no
remaining child tasks,
        public mutating func next() async
```

```
throws ->
ThrowingTaskGroup<ChildTaskResult,
Failure>.Iterator.Element?

        /// Advances to and returns the
result of the next child task.
        ///
        /// The elements returned from
this method
        /// appear in the order that the
tasks *completed*,
        /// not in the order that those
tasks were added to the task group.
        /// After this method returns
`nil`,
        /// this iterator is guaranteed
to never produce more values.
        ///
        /// For more information about
the iteration order and semantics,
        /// see
`ThrowingTaskGroup.next()`
        ///
        /// - Throws: The error thrown by
the next child task that completes.
        ///
        /// - Returns: The value returned
by the next child task that completes,
        ///   or `nil` if there are no
remaining child tasks,
        @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
        public mutating func
```

```
next(isolation actor: isolated (any
Actor)?) async throws(Failure) ->
ThrowingTaskGroup<ChildTaskResult,
Failure>.Iterator.Element?

        public mutating func cancel()
    }
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension ThrowingTaskGroup {

    /// Adds a child task to the group
and enqueue it on the specified executor.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///                   If `nil` is
passed explicitly, tht parent task's
executor preference (if any),
    ///                   will be
ignored. In order to inherit the parent
task's executor preference
    ///                   invoke
`addTask()` without passing a value to
the `taskExecutor` parameter,
    ///                   and it will be
inherited automatically.
    ///   - priority: The priority of the
operation task.
```

```
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    public mutating func
addTask(executorPreference taskExecutor:
(any TaskExecutor)?, priority:
TaskPriority? = nil, operation: sending
@escaping @isolated(any) () async throws
-> ChildTaskResult)

    /// Adds a child task to the group
and enqueue it on the specified executor,
unless the group has been canceled.
    ///
    /// - Parameters:
    ///   - taskExecutor: The task
executor that the child task should be
started on and keep using.
    ///   - priority: The priority of the
operation task.
    ///     Omit this parameter or pass
`.unspecified`
    ///     to set the child task's
priority to the priority of the group.
    ///   - operation: The operation to
execute as part of the task group.
    /// - Returns: `true` if the child
task was added to the group;
    ///   otherwise `false`.
    public mutating func
```

```
addTaskUnlessCancelled(executorPreference
taskExecutor: (any TaskExecutor)?,
priority: TaskPriority? = nil, operation:
sending @escaping @isolated(any) () async
throws -> ChildTaskResult) -> Bool
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension ThrowingTaskGroup :
BitwiseCopyable {
}

/// A unit of schedulable work.
///
/// Unless you're implementing a
scheduler,
/// you don't generally interact with
jobs directly.
///
/// An `UnownedJob` must be eventually
run *exactly once* using
``runSynchronously(on:)``.
/// Not doing so is effectively going to
leak and "hang" the work that the job
represents (e.g. a ``Task``).
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct UnownedJob :
Sendable {

    /// Create an `UnownedJob` whose
lifetime must be managed carefully until
```

```
it is run exactly once.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public init(_ job: Job)

    /// Create an `UnownedJob` whose
lifetime must be managed carefully until
it is run exactly once.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public init(_ job: ExecutorJob)

    /// The priority of this job.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public var priority: JobPriority {
get }

    /// Run this job on the passed in
executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
```

```
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// - Parameter executor: the
executor this job will be semantically
running on.
    @inlinable public func
runSynchronously(on executor:
UnownedSerialExecutor)

    /// Run this job isolated to the
passed task executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
    /// wants to run the job and then
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
```

```
    ///
    /// Converting a `ExecutorJob` to an
``UnownedJob`` and invoking
``UnownedJob/runSynchronously(_:)` on it
multiple times is undefined behavior,
    /// as a job can only ever be run
once, and must not be accessed after it
has been run.
    ///
    /// - Parameter executor: the task
executor this job will be run on.
    ///
    /// - SeeAlso:
``runSynchronously(isolatedTo:taskExecuto
r:)``
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    @inlinable public func
runSynchronously(on executor:
UnownedTaskExecutor)

    /// Run this job isolated to the
passed in serial executor, while
executing it on the specified task
executor.
    ///
    /// This operation runs the job on
the calling thread and *blocks* until the
job completes.
    /// The intended use of this method
is for an executor to determine when and
where it
    /// wants to run the job and then
```

```
call this method on it.
    ///
    /// The passed in executor reference
is used to establish the executor context
for the job,
    /// and should be the same executor
as the one semantically calling the
`runSynchronously` method.
    ///
    /// This operation consumes the job,
preventing it accidental use after it has
been run.
    ///
    /// Converting a `ExecutorJob` to an
``UnownedJob`` and invoking
    ///
``UnownedJob/runSynchronously(isolatedTo:
taskExecutor:)` on it multiple times
    /// is undefined behavior, as a job
can only ever be run once, and must not
be
    /// accessed after it has been run.
    ///
    /// - Parameter serialExecutor: the
executor this job will be semantically
running on.
    /// - Parameter taskExecutor: the
task executor this job will be run on.
    ///
    /// - SeeAlso:
``runSynchronously(on:)``
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
```

```
    @inlinable public func
runSynchronously(isolatedTo
serialExecutor: UnownedSerialExecutor,
taskExecutor: UnownedTaskExecutor)
}

@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
extension UnownedJob :
CustomStringConvertible {

    /// A textual representation of this
instance.
    ///
    /// Calling this property directly is
discouraged. Instead, convert an
    /// instance of any type to a string
by using the `String(describing:)`
    /// initializer. This initializer
works with any type, and uses the custom
    /// `description` property for types
that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point:
CustomStringConvertible {
    ///         let x: Int, y: Int
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
```

```
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string
in the assignment to `s` uses the
    /// `Point` type's `description`
property.
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public var description: String {
get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnownedJob : BitwiseCopyable {
}

/// An unowned reference to a serial
executor (a `SerialExecutor`
/// value).
///
/// This is an optimized type used
internally by the core scheduling
/// operations.  It is an unowned
reference to avoid unnecessary
/// reference-counting work even when
working with actors abstractly.
/// Generally there are extra constraints
imposed on core operations
/// in order to allow this.  For example,
```

```
keeping an actor alive must
/// also keep the actor's associated
executor alive; if they are
/// different objects, the executor must
be referenced strongly by the
/// actor.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct
UnownedSerialExecutor : Sendable {

    @inlinable public init(_ executor:
Builtin.Executor)

    @inlinable public init<E>(ordinary
executor: E) where E : SerialExecutor

    /// Opts the executor into complex
"same exclusive execution context"
equality checks.
    ///
    /// This means what when asserting or
assuming executors, and the current and
expected
    /// executor are not the same
instance (by object equality), the
runtime may invoke
    /// `isSameExclusiveExecutionContext`
in order to compare the executors for
equality.
    ///
    /// Implementing such complex
equality can be useful if multiple
```

```
executor instances
    /// actually use the same underlying
serialization context and can be
therefore
    /// safely treated as the same serial
exclusive execution context (e.g.
multiple
    /// dispatch queues targeting the
same serial queue).
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    @inlinable public
init<E>(complexEquality executor: E)
where E : SerialExecutor
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnownedSerialExecutor :
BitwiseCopyable {
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
@frozen public struct UnownedTaskExecutor
: Sendable {

    @inlinable public init(_ executor:
Builtin.Executor)

    @inlinable public init<E>(ordinary
executor: E) where E : TaskExecutor
}
```

```
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension UnownedTaskExecutor : Equatable
{

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    @inlinable public static func ==
(lhs: UnownedTaskExecutor, rhs:
UnownedTaskExecutor) -> Bool
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension UnownedTaskExecutor :
BitwiseCopyable {
}

/// A mechanism to interface
/// between synchronous and asynchronous
code,
/// without correctness checking.
```

```
///
/// A *continuation* is an opaque
representation of program state.
/// To create a continuation in
asynchronous code,
/// call the `withUnsafeContinuation(_:)`
or
/// `withUnsafeThrowingContinuation(_:)`
function.
/// To resume the asynchronous task,
/// call the `resume(returning:)`,
/// `resume(throwing:)`,
/// `resume(with:)`,
/// or `resume()` method.
///
/// - Important: You must call a resume
method exactly once
///   on every execution path throughout
the program.
///   Resuming from a continuation more
than once is undefined behavior.
///   Never resuming leaves the task in a
suspended state indefinitely,
///   and leaks any associated resources.
///
/// `CheckedContinuation` performs
runtime checks
/// for missing or multiple resume
operations.
/// `UnsafeContinuation` avoids enforcing
these invariants at runtime
/// because it aims to be a low-overhead
mechanism
```

```
/// for interfacing Swift tasks with
/// event loops, delegate methods,
callbacks,
/// and other non-`async` scheduling
mechanisms.
/// However, during development, the
ability to verify that the
/// invariants are being upheld in
testing is important.
/// Because both types have the same
interface,
/// you can replace one with the other in
most circumstances,
/// without making other changes.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@frozen public struct
UnsafeContinuation<T, E> : Sendable where
E : Error {

    /// Resume the task that's awaiting
the continuation
    /// by returning the given value.
    ///
    /// - Parameter value: The value to
return from the continuation.
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
```

```
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume(returning value:
sending T) where E == Never

    /// Resume the task that's awaiting
the continuation
    /// by returning the given value.
    ///
    /// - Parameter value: The value to
return from the continuation.
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume(returning value:
sending T)

    /// Resume the task that's awaiting
the continuation
```

```
    /// by throwing the given error.
    ///
    /// - Parameter error: The error to
throw from the continuation.
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume(throwing error:
consuming E)
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnsafeContinuation {

    /// Resume the task that's awaiting
the continuation
    /// by returning or throwing the
given result value.
    ///
    /// - Parameter result: The result.
    ///   If it contains a `.success`
value,
```

```
    ///   the continuation returns that
value;
    ///   otherwise, it throws the
`.error` value.
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume<Er>(with result:
sending Result<T, Er>) where E == any
Error, Er : Error

    /// Resume the task that's awaiting
the continuation
    /// by returning or throwing the
given result value.
    ///
    /// - Parameter result: The result.
    ///   If it contains a `.success`
value,
    ///   the continuation returns that
value;
    ///   otherwise, it throws the
`.error` value.
```

```
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume(with result:
sending Result<T, E>)

    /// Resume the task that's awaiting
the continuation by returning.
    ///
    /// A continuation must be resumed
exactly once.
    /// If the continuation has already
resumed,
    /// then calling this method results
in undefined behavior.
    ///
    /// After calling this method,
    /// control immediately returns to
the caller.
    /// The task continues executing
    /// when its executor schedules it.
    public func resume() where T == ()
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnsafeContinuation :
BitwiseCopyable {
}

/// An unsafe reference to the current
task.
///
/// To get an instance of
`UnsafeCurrentTask` for the current task,
/// call the
`withUnsafeCurrentTask(body:)` method.
/// Don't store an unsafe task reference
/// for use outside that method's
closure.
/// Storing an unsafe reference doesn't
affect the task's actual life cycle,
/// and the behavior of accessing an
unsafe task reference
/// outside of the
`withUnsafeCurrentTask(body:)` method's
closure isn't defined.
///
/// Only APIs on `UnsafeCurrentTask` that
are also part of `Task`
/// are safe to invoke from a task other
than
/// the task that this
`UnsafeCurrentTask` instance refers to.
/// Calling other APIs from another task
is undefined behavior,
```

```
/// breaks invariants in other parts of
the program running on this task,
/// and may lead to crashes or data loss.
///
/// For information about the language-
level concurrency model that
`UnsafeCurrentTask` is part of,
/// see [Concurrency][concurrency] in
[The Swift Programming Language][tspl].
///
/// [concurrency]:
https://docs.swift.org/swift-book/Languag
eGuide/Concurrency.html
/// [tspl]: https://docs.swift.org/swift-
book/
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public struct UnsafeCurrentTask {

    /// A Boolean value that indicates
whether the current task was canceled.
    ///
    /// After the value of this property
becomes `true`, it remains `true`
indefinitely.
    /// There is no way to uncancel a
task.
    ///
    /// - SeeAlso: `checkCancellation()`
    public var isCancelled: Bool { get }

    /// The current task's priority.
    ///
```

```
    /// - SeeAlso: `TaskPriority`
    /// - SeeAlso: `Task.currentPriority`
    public var priority: TaskPriority {
get }

    /// The current task's base priority.
    ///
    /// - SeeAlso: `TaskPriority`
    /// - SeeAlso: `Task.basePriority`
    @available(macOS 14.0, iOS 17.0,
watchOS 10.0, tvOS 17.0, *)
    public var basePriority: TaskPriority
{ get }

    /// Cancel the current task.
    public func cancel()
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnsafeCurrentTask : Hashable {

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
```

```
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    public func hash(into hasher: inout
Hasher)

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}
```

```
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
extension UnsafeCurrentTask : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
UnsafeCurrentTask, rhs:
UnsafeCurrentTask) -> Bool
}

@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
extension UnsafeCurrentTask {

    /// The current ``TaskExecutor``
preference, if this task has one
configured.
    ///
    /// The executor may be used to
compare for equality with an expected
executor preference.
```

```
    ///
    /// The lifetime of an executor is
not guaranteed by an
``UnownedTaskExecutor``,
    /// so accessing it must be handled
with great case -- and the program must
use other
    /// means to guarantee the executor
remains alive while it is in use.
    @available(macOS 15.0, iOS 18.0,
watchOS 11.0, tvOS 18.0, visionOS 2.0, *)
    public var unownedTaskExecutor:
UnownedTaskExecutor? { get }
}

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@available(*, deprecated, message:
"please use UnsafeContinuation<...,
Error>")
public typealias
UnsafeThrowingContinuation<T> =
UnsafeContinuation<T, any Error>

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@available(*, deprecated, message:
"`async` was replaced by `Task.init` and
will be removed shortly.")
@discardableResult
public func async<T>(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> T) -> Task<T,
```

```
Never> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@available(*, deprecated, message:
"`async` was replaced by `Task.init` and
will be removed shortly.")
@discardableResult
public func async<T>(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws -> T) ->
Task<T, any Error> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@discardableResult
@available(*, deprecated, message:
"`asyncDetached` was replaced by
`Task.detached` and will be removed
shortly.")
public func asyncDetached<T>(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> T) -> Task<T,
Never> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@discardableResult
@available(*, deprecated, message:
"`asyncDetached` was replaced by
`Task.detached` and will be removed
shortly.")
public func asyncDetached<T>(priority:
```

```
TaskPriority? = nil, operation: @escaping
@Sendable () async throws -> T) ->
Task<T, any Error> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@discardableResult
@available(*, deprecated, message:
"`detach` was replaced by `Task.detached`
and will be removed shortly.")
public func detach<T>(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async -> T) -> Task<T,
Never> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@discardableResult
@available(*, deprecated, message:
"`detach` was replaced by `Task.detached`
and will be removed shortly.")
public func detach<T>(priority:
TaskPriority? = nil, operation: @escaping
@Sendable () async throws -> T) ->
Task<T, any Error> where T : Sendable

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public func extractIsolation<each Arg,
Result>(_ fn: @escaping @isolated(any)
(repeat each Arg) async throws -> Result)
-> (any Actor)?
```

```
/// The global concurrent executor that
is used by default for Swift Concurrency
/// tasks.
///
/// The executor's implementation is
platform dependent.
/// By default it uses a fixed size pool
of threads and should not be used for
/// blocking operations which do not
guarantee forward progress as doing so
may
/// prevent other tasks from being
executed and render the system
unresponsive.
///
/// You may pass this executor explicitly
to a ``Task`` initializer as a task
/// executor preference, in order to
ensure and document that task be executed
/// on the global executor, instead e.g.
inheriting the enclosing actor's
/// executor. Refer to
``withTaskExecutorPreference(_:operation:
)`` for a
/// detailed discussion of task executor
preferences.
///
/// Customizing the global concurrent
executor is currently not supported.
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
public var globalConcurrentExecutor: any
TaskExecutor { get }
```

```
/// Produce a reference to the actor to
which the enclosing code is
/// isolated, or `nil` if the code is
nonisolated.
///
/// If the type annotation provided for
`#isolation` is not `(any Actor)?`,
/// the type must match the enclosing
actor type. If no type annotation is
/// provided, the type defaults to `(any
Actor)?`.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@freestanding(expression) public macro
isolation<T>() -> T =
Builtin.IsolationMacro

@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public func
swift_deletedAsyncMethodError() async

/// Invokes the passed in closure with a
checked continuation for the current
task.
///
/// The body of the closure executes
synchronously on the calling task, and
once it returns
/// the calling task is suspended. It is
possible to immediately resume the task,
or escape the
```

```
/// continuation in order to complete it
afterwards, which will then resume the
suspended task.
///
/// You must invoke the continuation's
`resume` method exactly once.
///
/// Missing to invoke it (eventually)
will cause the calling task to remain
suspended
/// indefinitely which will result in the
task "hanging" as well as being leaked
with
/// no possibility to destroy it.
///
/// The checked continuation offers
detection of mis-use, and dropping the
last reference
/// to it, without having resumed it will
trigger a warning. Resuming a
continuation twice
/// is also diagnosed and will cause a
crash.
///
/// - Parameters:
///   - function: A string identifying
the declaration that is the notional
///     source for the continuation, used
to identify the continuation in
///     runtime diagnostics related to
misuse of this continuation.
///   - body: A closure that takes a
`CheckedContinuation` parameter.
```

```
/// - Returns: The value continuation is
resumed with.
///
/// - SeeAlso:
`withCheckedThrowingContinuation(function
:_:)`
/// - SeeAlso:
`withUnsafeContinuation(function:_:)`
/// - SeeAlso:
`withUnsafeThrowingContinuation(function:
_:)`
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
@inlinable public func
withCheckedContinuation<T>(isolation:
isolated (any Actor)? = #isolation,
function: String = #function, _ body:
(CheckedContinuation<T, Never>) -> Void)
async -> sending T

/// Invokes the passed in closure with a
checked continuation for the current
task.
///
/// The body of the closure executes
synchronously on the calling task, and
once it returns
/// the calling task is suspended. It is
possible to immediately resume the task,
or escape the
```

```
/// continuation in order to complete it
afterwards, which will then resume the
suspended task.
///
/// If `resume(throwing:)` is called on
the continuation, this function throws
that error.
///
/// You must invoke the continuation's
`resume` method exactly once.
///
/// Missing to invoke it (eventually)
will cause the calling task to remain
suspended
/// indefinitely which will result in the
task "hanging" as well as being leaked
with
/// no possibility to destroy it.
///
/// The checked continuation offers
detection of mis-use, and dropping the
last reference
/// to it, without having resumed it will
trigger a warning. Resuming a
continuation twice
/// is also diagnosed and will cause a
crash.
///
/// - Parameters:
///   - function: A string identifying
the declaration that is the notional
///     source for the continuation, used
to identify the continuation in
```

```
///     runtime diagnostics related to
misuse of this continuation.
///   - body: A closure that takes a
`CheckedContinuation` parameter.
/// - Returns: The value continuation is
resumed with.
///
/// - SeeAlso:
`withCheckedContinuation(function:_:)`
/// - SeeAlso:
`withUnsafeContinuation(function:_:)`
/// - SeeAlso:
`withUnsafeThrowingContinuation(function:
_:)`
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
@inlinable public func
withCheckedThrowingContinuation<T>(isolat
ion: isolated (any Actor)? = #isolation,
function: String = #function, _ body:
(CheckedContinuation<T, any Error>) ->
Void) async throws -> sending T

/// Starts a new scope that can contain a
dynamic number of child tasks.
///
/// Unlike a ``TaskGroup``, the child
tasks as well as their results are
/// discarded as soon as the tasks
complete. This prevents the discarding
```

```
/// task group from accumulating many
results waiting to be consumed, and is
/// best applied in situations where the
result of a child task is some form
/// of side-effect.
///
/// A group waits for all of its child
tasks
/// to complete before it returns. Even
cancelled tasks must run until
/// completion before this function
returns.
/// Cancelled child tasks cooperatively
react to cancellation and attempt
/// to return as early as possible.
/// After this function returns, the task
group is always empty.
///
/// It is not possible to explicitly
await completion of child-tasks,
/// however the group will automatically
await *all* child task completions
/// before returning from this function:
///
/// ```
/// await withDiscardingTaskGroup(...)
{ group in
///   group.addTask { /* slow-task */ }
///   // slow-task executes...
/// }
/// // guaranteed that slow-task has
completed and the group is empty &
destroyed
```

```
/// ```
///
/// Task Group Cancellation
/// ======================
///
/// You can cancel a task group and all
of its child tasks
/// by calling the
``TaskGroup/cancelAll()`` method on the
task group,
/// or by canceling the task in which the
group is running.
///
/// If you call
`addTask(priority:operation:)` to create
a new task in a canceled group,
/// that task is immediately canceled
after creation.
/// Alternatively, you can call
`asyncUnlessCancelled(priority:operation:
)`,
/// which doesn't create the task if the
group has already been canceled
/// Choosing between these two functions
/// lets you control how to react to
cancellation within a group:
/// some child tasks need to run
regardless of cancellation,
/// but other tasks are better not even
being created
/// when you know they can't produce
useful results.
///
```

```
/// Because the tasks you add to a group
with this method are nonthrowing,
/// those tasks can't respond to
cancellation by throwing
`CancellationError`.
/// The tasks must handle cancellation in
some other way,
/// such as returning the work completed
so far, returning an empty result, or
returning `nil`.
/// For tasks that need to handle
cancellation by throwing an error,
/// use the
`withThrowingDiscardingTaskGroup(returnin
g:body:)` method instead.
///
/// - SeeAlso:
``withThrowingDiscardingTaskGroup(returni
ng:body:)``
@available(macOS 14.0, iOS 17.0, watchOS
10.0, tvOS 17.0, *)
@backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
@inlinable public func
withDiscardingTaskGroup<GroupResult>(retu
rning returnType: GroupResult.Type =
GroupResult.self, isolation: isolated
(any Actor)? = #isolation, body: (inout
DiscardingTaskGroup) async ->
GroupResult) async -> GroupResult

@available(macOS 10.15, iOS 13.0, watchOS
```

```
6.0, tvOS 13.0, *)
@available(*, deprecated, renamed:
"withTaskCancellationHandler(operation:on
Cancel:)")
public func
withTaskCancellationHandler<T>(handler:
@Sendable () -> Void, operation: () async
throws -> T) async rethrows -> T

/// Execute an operation with a
cancellation handler that's immediately
/// invoked if the current task is
canceled.
///
/// This differs from the operation
cooperatively checking for cancellation
/// and reacting to it in that the
cancellation handler is _always_ and
/// _immediately_ invoked when the task
is canceled. For example, even if the
/// operation is running code that never
checks for cancellation, a cancellation
/// handler still runs and provides a
chance to run some cleanup code:
///
/// ```
/// await withTaskCancellationHandler {
///   var sum = 0
///   while condition {
///     sum += 1
///   }
///   return sum
/// } onCancel: {
```

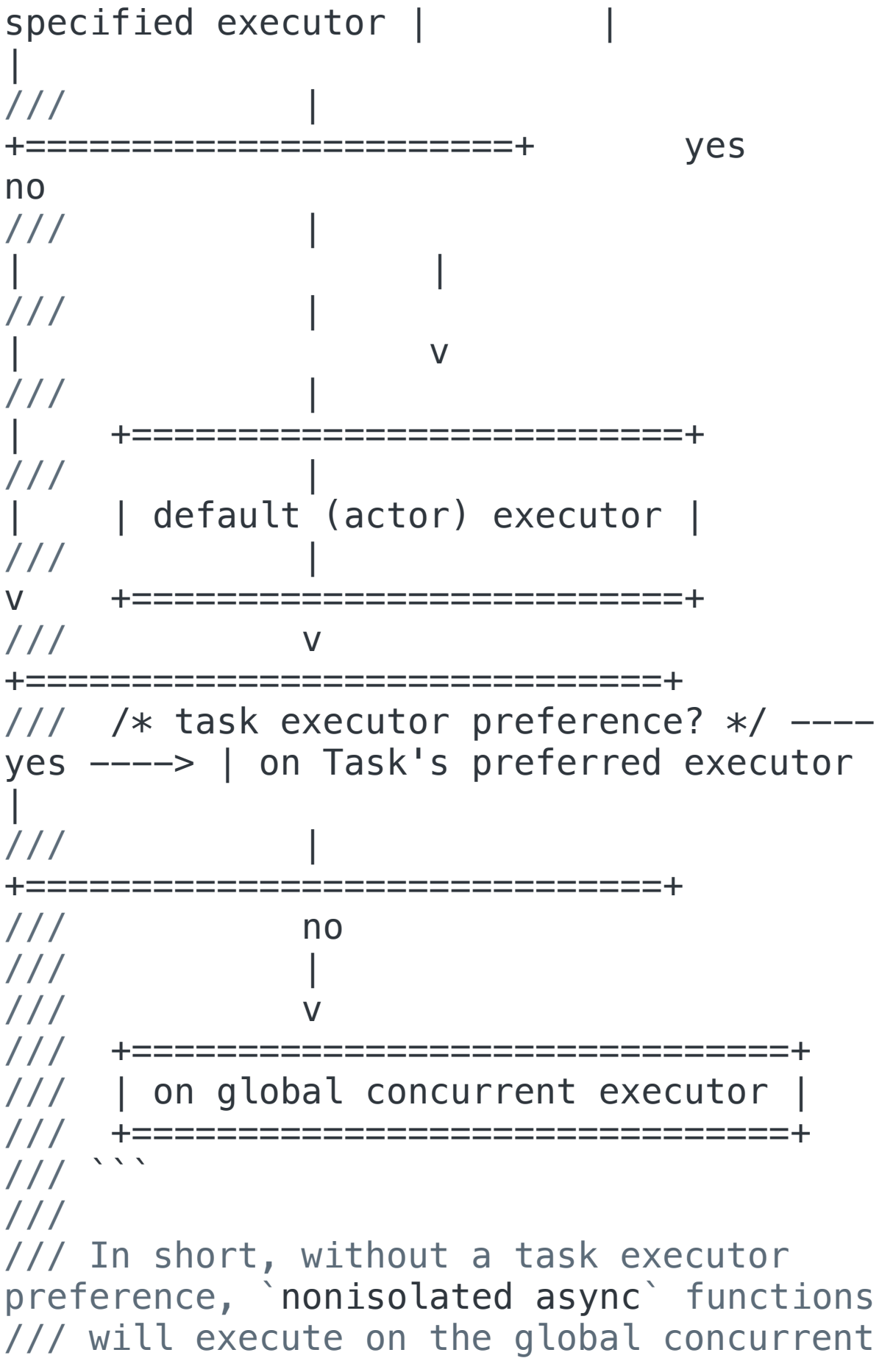
```
///   // This onCancel closure might
execute concurrently with the operation.
///   condition.cancel()
/// }
/// ```
///
/// ### Execution order and semantics
/// The `operation` closure is always
invoked, even when the
///
`withTaskCancellationHandler(operation:on
Cancel:)` method is called from a task
/// that was already cancelled.
///
/// When
`withTaskCancellationHandler(operation:on
Cancel:)` is used in a task that has
already been
/// cancelled, the cancellation handler
will be executed
/// immediately before the `operation`
closure gets to execute.
///
/// This allows the cancellation handler
to set some external "cancelled" flag
/// that the operation may be
*atomically* checking for in order to
avoid
/// performing any actual work once the
operation gets to run.
///
/// The `operation` closure executes on
the calling execution context, and
```

```
doesn't
/// suspend or change execution context
unless code contained within the closure
/// does so. In other words, the
potential suspension point of the
///
`withTaskCancellationHandler(operation:on
Cancel:)` never suspends by itself before
/// executing the operation.
///
/// If cancellation occurs while the
operation is running, the cancellation
/// handler executes *concurrently* with
the operation.
///
/// ### Cancellation handlers and locks
///
/// Cancellation handlers which acquire
locks must take care to avoid deadlock.
/// The cancellation handler may be
invoked while holding internal locks
/// associated with the task or other
tasks.  Other operations on the task,
such
/// as resuming a continuation, may
acquire these same internal locks.
/// Therefore, if a cancellation handler
must acquire a lock, other code should
/// not cancel tasks or resume
continuations while holding that lock.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@backDeployed(before: macOS 15.0, iOS
```

```
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
public func
withTaskCancellationHandler<T>(operation:
() async throws -> T, onCancel handler:
@Sendable () -> Void, isolation: isolated
(any Actor)? = #isolation) async rethrows
-> T

/// Configure the current task
hierarchy's task executor preference to
the passed ``TaskExecutor``,
/// and execute the passed in closure by
immediately hopping to that executor.
///
/// ### Task executor preference
semantics
/// Task executors influence _where_
nonisolated asynchronous functions, and
default actor methods execute.
/// The preferred executor will be used
whenever possible, rather than hopping to
the global concurrent pool.
///
/// For an in depth discussion of this
topic, see ``TaskExecutor``.
///
/// ### Disabling task executor
preference
/// Passing `nil` as executor means
disabling any preference preference (if
it was set) and the task hierarchy
/// will execute without any executor
```

```
preference until a different preference
is set.
///
/// ### Asynchronous function execution
semantics in presence of task executor
preferences
/// The following diagram illustrates on
which executor an `async` function will
/// execute, in presence (or lack
thereof) a task executor preference.
///
/// ```
/// [ func / closure ] - /* where should
it execute? */
///                                  |
///                          +-------------+
+============================+
///            +-------- | is isolated? |
- yes -> | actor has unownedExecutor |
///            |         +-------------+
+============================+
///            |
|                |
///            |
yes               no
///            |
|                |
///            |
v                v
///            |
+======================+    /* task
executor preference? */
///            |                  | on
```




```
/// ```
///
/// In short, without a task executor
preference, `nonisolated async` functions
/// will execute on the global concurrent
```

```
executor. If a task executor preference
/// is present, such `nonisolated async`
functions will execute on the preferred
/// task executor.
///
/// Isolated functions semantically
execute on the actor they are isolated
to,
/// however if such actor does not
declare a custom executor (it is a
"default
/// actor") in presence of a task
executor preference, tasks executing on
this
/// actor will use the preferred executor
as source of threads to run the task,
/// while isolated on the actor.
///
/// ### Example
///
///     Task {
///       // case 0) "no task executor
preference"
///
///       // default task executor
///       // ...
///       await
SomeDefaultActor().hello() // default
executor
///       await
ActorWithCustomExecutor().hello() //
'hello' executes on actor's custom
executor
```

```
///
///       // child tasks execute on
default executor:
///       async let x = ...
///       await withTaskGroup(of:
Int.self) { group in g.addTask { 7 } }
///
///       await
withTaskExecutorPreference(specific) {
///         // case 1) 'specific' task
executor preference
///
///         // 'specific' task executor
///         // ...
///         await
SomeDefaultActor().hello() // 'hello'
executes on 'specific' executor
///         await
ActorWithCustomExecutor().hello() //
'hello' executes on actor's custom
executor (same as case 0)
///
///         // child tasks execute on
'specific' task executor:
///         async let x = ...
///         await withTaskGroup(of:
Int.self) { group in
///           group.addTask { 7 } //
child task executes on 'specific'
executor
///
group.addTask(executorPreference:
globalConcurrentExecutor) { 13 } // child
```

```
task executes on global concurrent
executor
///         }
///
///         // disable the task executor
preference:
///         await
withTaskExecutorPreference(globalConcurre
ntExecutor) {
///           // equivalent to case 0)
preference is globalConcurrentExecutor
///
///           // default task executor
///           // ...
///           await
SomeDefaultActor().hello() // default
executor (same as case 0)
///           await
ActorWithCustomExecutor().hello() //
'hello' executes on actor's custom
executor (same as case 0)
///
///           // child tasks execute on
default executor (same as case 0):
///           async let x = ...
///           await withTaskGroup(of:
Int.self) { group in g.addTask { 7 } }
///         }
///       }
///     }
///
/// - Parameters:
///   - taskExecutor: the executor to use
```

```
as preferred task executor for this
///     operation, and any child tasks
created inside the `operation` closure.
///     If `nil` it is interpreted as "no
preference" and calling this method
///     will have no impact on execution
semantics of the `operation`
///   - operation: the operation to
execute on the passed executor
/// - Returns: the value returned from
the `operation` closure
/// - Throws: if the operation closure
throws
/// - SeeAlso: ``TaskExecutor``
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
public func withTaskExecutorPreference<T,
Failure>(_ taskExecutor: (any
TaskExecutor)?, isolation: isolated (any
Actor)? = #isolation, operation: () async
throws(Failure) -> T) async
throws(Failure) -> T where Failure :
Error

/// Starts a new scope that can contain a
dynamic number of child tasks.
///
/// A group waits for all of its child
tasks
/// to complete or be canceled before it
returns.
/// After this function returns, the task
group is always empty.
```

```
///
/// To collect the results of the group's
child tasks,
/// you can use a `for`-`await`-`in`
loop:
///
///     var sum = 0
///     for await result in group {
///         sum += result
///     }
///
/// If you need more control or only a
few results,
/// you can call `next()` directly:
///
///     guard let first = await
group.next() else {
///         group.cancelAll()
///         return 0
///     }
///     let second = await
group.next() ?? 0
///     group.cancelAll()
///     return first + second
///
/// Task Group Cancellation
/// ======================
///
/// You can cancel a task group and all
of its child tasks
/// by calling the `cancelAll()` method
on the task group,
/// or by canceling the task in which the
```

```
group is running.
///
/// If you call
`addTask(priority:operation:)` to create
a new task in a canceled group,
/// that task is immediately canceled
after creation.
/// Alternatively, you can call
`addTaskUnlessCancelled(priority:operatio
n:)`,
/// which doesn't create the task if the
group has already been canceled
/// Choosing between these two functions
/// lets you control how to react to
cancellation within a group:
/// some child tasks need to run
regardless of cancellation,
/// but other tasks are better not even
being created
/// when you know they can't produce
useful results.
///
/// Because the tasks you add to a group
with this method are nonthrowing,
/// those tasks can't respond to
cancellation by throwing
`CancellationError`.
/// The tasks must handle cancellation in
some other way,
/// such as returning the work completed
so far, returning an empty result, or
returning `nil`.
/// For tasks that need to handle
```

```
cancellation by throwing an error,
/// use the
`withThrowingTaskGroup(of:returning:body:
)` method instead.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
@inlinable public func
withTaskGroup<ChildTaskResult,
GroupResult>(of childTaskResultType:
ChildTaskResult.Type, returning
returnType: GroupResult.Type =
GroupResult.self, isolation: isolated
(any Actor)? = #isolation, body: (inout
TaskGroup<ChildTaskResult>) async ->
GroupResult) async -> GroupResult where
ChildTaskResult : Sendable

/// Starts a new scope that can contain a
dynamic number of child tasks.
///
/// Unlike a ``ThrowingTaskGroup``, the
child tasks as well as their results are
/// discarded as soon as the tasks
complete. This prevents the discarding
/// task group from accumulating many
results waiting to be consumed, and is
/// best applied in situations where the
result of a child task is some form
/// of side-effect.
///
```

```
/// A group waits for all of its child
tasks
/// to complete before it returns. Even
cancelled tasks must run until
/// completion before this function
returns.
/// Cancelled child tasks cooperatively
react to cancellation and attempt
/// to return as early as possible.
/// After this function returns, the task
group is always empty.
///
/// It is not possible to explicitly
await completion of child-tasks,
/// however the group will automatically
await *all* child task completions
/// before returning from this function:
///
/// ```
/// try await
withThrowingDiscardingTaskGroup(of:
Void.self) { group in
///   group.addTask { /* slow-task */ }
///   // slow-task executes...
/// }
/// // guaranteed that slow-task has
completed and the group is empty &
destroyed
/// ```
///
/// Task Group Cancellation
/// ======================
///
```

```
/// You can cancel a task group and all
of its child tasks
/// by calling the
``TaskGroup/cancelAll()`` method on the
task group,
/// or by canceling the task in which the
group is running.
///
/// If you call
`addTask(priority:operation:)` to create
a new task in a canceled group,
/// that task is immediately canceled
after creation.
/// Alternatively, you can call
`asyncUnlessCancelled(priority:operation:
)`,
/// which doesn't create the task if the
group has already been canceled
/// Choosing between these two functions
/// lets you control how to react to
cancellation within a group:
/// some child tasks need to run
regardless of cancellation,
/// but other tasks are better not even
being created
/// when you know they can't produce
useful results.
///
/// Error Handling and Implicit
Cancellation
///
=======================================
///
```

```
/// Since it is not possible to
explicitly await individual task
completions,
/// it is also not possible to "re-throw"
an error thrown by one of the child
/// tasks using the same pattern as one
would in a ``ThrowingTaskGroup``:
///
/// ```
/// // ThrowingTaskGroup, pattern not
applicable to ThrowingDiscardingTaskGroup
/// try await withThrowingTaskGroup(of:
Void.self) { group in
///   group.addTask { try boom() }
///   try await group.next() // re-throws
"boom"
/// }
/// ```
///
/// Since discarding task groups don't
have access to `next()`, this pattern
/// cannot be used.
/// Instead,
/// a *throwing discarding task group
implicitly cancels itself whenever any
/// of its child tasks throws*.
///
/// The *first error* thrown inside such
task group
/// is then retained and thrown
/// out of the
`withThrowingDiscardingTaskGroup` method
when it returns.
```

```
///
/// ```
/// try await
withThrowingDiscardingTaskGroup { group
in
///   group.addTask { try boom(1) }
///   group.addTask { try boom(2,
after: .seconds(5)) }
///   group.addTask { try boom(3,
after: .seconds(5)) }
/// }
/// ```
///
/// Generally, this suits the typical use
cases of a
/// discarding task group well, however,
if you want to prevent specific
/// errors from canceling the group you
can catch them inside the child
/// task's body like this:
///
/// ```
/// try await
withThrowingDiscardingTaskGroup { group
in
///   group.addTask {
///     do {
///       try boom(1)
///     } catch is HarmlessError {
///       return
///     }
///   }
///   group.addTask {
```

```
///     try boom(2, after: .seconds(5))
///   }
/// }
/// ```
@available(macOS 14.0, iOS 17.0, watchOS 10.0, tvOS 17.0, *)
@backDeployed(before: macOS 15.0, iOS 18.0, watchOS 11.0, tvOS 18.0, visionOS 2.0)
@inlinable public func withThrowingDiscardingTaskGroup<GroupResult>(returning returnType: GroupResult.Type = GroupResult.self, isolation: isolated (any Actor)? = #isolation, body: (inout ThrowingDiscardingTaskGroup<any Error>) async throws -> GroupResult) async throws -> GroupResult

/// Starts a new scope that can contain a dynamic number of throwing child tasks.
///
/// A group waits for all of its child tasks
/// to complete before it returns. Even cancelled tasks must run until
/// completion before this function returns.
/// Cancelled child tasks cooperatively react to cancellation and attempt
/// to return as early as possible.
/// After this function returns, the task group is always empty.
```

```
///
/// To collect the results of the group's
child tasks,
/// you can use a `for`-`await`-`in`
loop:
///
///     var sum = 0
///     for try await result in group {
///         sum += result
///     }
///
/// If you need more control or only a
few results,
/// you can call `next()` directly:
///
///     guard let first = try await
group.next() else {
///         group.cancelAll()
///         return 0
///     }
///     let second = await
group.next() ?? 0
///     group.cancelAll()
///     return first + second
///
/// Task Group Cancellation
/// ======================
///
/// You can cancel a task group and all
of its child tasks
/// by calling the `cancelAll()` method
on the task group,
/// or by canceling the task in which the
```

```
group is running.
///
/// If you call
`addTask(priority:operation:)` to create
a new task in a canceled group,
/// that task is immediately canceled
after creation.
/// Alternatively, you can call
`addTaskUnlessCancelled(priority:operatio
n:)`,
/// which doesn't create the task if the
group has already been canceled
/// Choosing between these two functions
/// lets you control how to react to
cancellation within a group:
/// some child tasks need to run
regardless of cancellation,
/// but other tasks are better not even
being created
/// when you know they can't produce
useful results.
///
/// Error Handling
/// ==============
///
/// Throwing an error in one of the child
tasks of a task group
/// doesn't immediately cancel the other
tasks in that group.
/// However,
/// throwing out of the `body` of the
`withThrowingTaskGroup` method does
cancel
```

```
/// the group, and all of its child
tasks.
/// For example,
/// if you call `next()` in the task
group and propagate its error,
/// all other tasks are canceled.
/// For example, in the code below,
/// nothing is canceled and the group
doesn't throw an error:
///
///     try await
withThrowingTaskGroup(of: Void.self)
{ group in
///         group.addTask { throw
SomeError() }
///     }
///
/// In contrast, this example throws
`SomeError`
/// and cancels all of the tasks in the
group:
///
///     try await
withThrowingTaskGroup(of: Void.self)
{ group in
///         group.addTask { throw
SomeError() }
///         try await group.next()
///     }
///
/// An individual task throws its error
/// in the corresponding call to
`Group.next()`,
```

```
/// which gives you a chance to handle
the individual error
/// or to let the group rethrow the
error.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
@backDeployed(before: macOS 15.0, iOS
18.0, watchOS 11.0, tvOS 18.0, visionOS
2.0)
@inlinable public func
withThrowingTaskGroup<ChildTaskResult,
GroupResult>(of childTaskResultType:
ChildTaskResult.Type, returning
returnType: GroupResult.Type =
GroupResult.self, isolation: isolated
(any Actor)? = #isolation, body: (inout
ThrowingTaskGroup<ChildTaskResult, any
Error>) async throws -> GroupResult)
async rethrows -> GroupResult where
ChildTaskResult : Sendable

/// Invokes the passed in closure with a
unsafe continuation for the current task.
///
/// The body of the closure executes
synchronously on the calling task, and
once it returns
/// the calling task is suspended. It is
possible to immediately resume the task,
or escape the
/// continuation in order to complete it
afterwards, which will then resume the
suspended task.
```

```
///
/// You must invoke the continuation's
`resume` method exactly once.
///
/// Missing to invoke it (eventually)
will cause the calling task to remain
suspended
/// indefinitely which will result in the
task "hanging" as well as being leaked
with
/// no possibility to destroy it.
///
/// Unlike the "checked" continuation
variant, the `UnsafeContinuation` does
not
/// detect or diagnose any kind of
misuse, so you need to be extra careful
to avoid
/// calling `resume` twice or forgetting
to call resume before letting go of the
/// continuation object.
///
/// - Parameter fn: A closure that takes
an `UnsafeContinuation` parameter.
/// - Returns: The value continuation is
resumed with.
///
/// - SeeAlso:
`withUnsafeThrowingContinuation(function:
_:)`
/// - SeeAlso:
`withCheckedContinuation(function:_:)`
/// - SeeAlso:
```

```
`withCheckedThrowingContinuation(function
:_:)`
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public func
withUnsafeContinuation<T>(isolation:
isolated (any Actor)? = #isolation, _ fn:
(UnsafeContinuation<T, Never>) -> Void)
async -> sending T

/// Calls a closure with an unsafe
reference to the current task.
///
/// If you call this function from the
body of an asynchronous function,
/// the unsafe task handle passed to the
closure is always non-`nil`
/// because an asynchronous function
always runs in the context of a task.
/// However, if you call this function
from the body of a synchronous function,
/// and that function isn't executing in
the context of any task,
/// the unsafe task handle is `nil`.
///
/// Don't store an unsafe task reference
/// for use outside this method's
closure.
/// Storing an unsafe reference doesn't
affect the task's actual life cycle,
/// and the behavior of accessing an
unsafe task reference
/// outside of the
```

```
`withUnsafeCurrentTask(body:)` method's
closure isn't defined.
/// There's no safe way to retrieve a
reference to the current task
/// and save it for long-term use.
/// To query the current task without
saving a reference to it,
/// use properties like
`currentPriority`.
/// If you need to store a reference to a
task,
/// create an unstructured task using
`Task.detached(priority:operation:)`
instead.
///
/// - Parameters:
///   - body: A closure that takes an
`UnsafeCurrentTask` parameter.
///     If `body` has a return value,
///     that value is also used as the
return value
///     for the
`withUnsafeCurrentTask(body:)` function.
///
/// - Returns: The return value, if any,
of the `body` closure.
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public func
withUnsafeCurrentTask<T>(body:
(UnsafeCurrentTask?) throws -> T)
rethrows -> T
```

```
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
public func
withUnsafeCurrentTask<T>(body:
(UnsafeCurrentTask?) async throws -> T)
async rethrows -> T

/// Invokes the passed in closure with a
unsafe continuation for the current task.
///
/// The body of the closure executes
synchronously on the calling task, and
once it returns
/// the calling task is suspended. It is
possible to immediately resume the task,
or escape the
/// continuation in order to complete it
afterwards, which will then resume the
suspended task.
///
/// If `resume(throwing:)` is called on
the continuation, this function throws
that error.
///
/// You must invoke the continuation's
`resume` method exactly once.
///
/// Missing to invoke it (eventually)
will cause the calling task to remain
suspended
/// indefinitely which will result in the
task "hanging" as well as being leaked
with
```

```
/// no possibility to destroy it.
///
/// Unlike the "checked" continuation
variant, the `UnsafeContinuation` does
not
/// detect or diagnose any kind of
misuse, so you need to be extra careful
to avoid
/// calling `resume` twice or forgetting
to call resume before letting go of the
/// continuation object.
///
/// - Parameter fn: A closure that takes
an `UnsafeContinuation` parameter.
/// - Returns: The value continuation is
resumed with.
///
/// - SeeAlso:
`withUnsafeContinuation(function:_:)`
/// - SeeAlso:
`withCheckedContinuation(function:_:)`
/// - SeeAlso:
`withCheckedThrowingContinuation(function
:_:)`
@available(macOS 10.15, iOS 13.0, watchOS
6.0, tvOS 13.0, *)
public func
withUnsafeThrowingContinuation<T>(isolati
on: isolated (any Actor)? = #isolation, _
fn: (UnsafeContinuation<T, any Error>) ->
Void) async throws -> sending T

/// The global concurrent executor that
```

```
is used by default for Swift Concurrency
/// tasks.
///
/// The executor's implementation is
platform dependent.
/// By default it uses a fixed size pool
of threads and should not be used for
/// blocking operations which do not
guarantee forward progress as doing so
may
/// prevent other tasks from being
executed and render the system
unresponsive.
///
/// You may pass this executor explicitly
to a ``Task`` initializer as a task
/// executor preference, in order to
ensure and document that task be executed
/// on the global executor, instead e.g.
inheriting the enclosing actor's
/// executor. Refer to
``withTaskExecutorPreference(_:operation:
)`` for a
/// detailed discussion of task executor
preferences.
///
/// Customizing the global concurrent
executor is currently not supported.
@available(macOS 15.0, iOS 18.0, watchOS
11.0, tvOS 18.0, visionOS 2.0, *)
public let globalConcurrentExecutor: any
TaskExecutor { get }
```