```
import Combine
import TargetConditionals
import _Builtin_stdbool
import _Builtin_stdint
import _Concurrency
import _StringProcessing
import _SwiftConcurrencyShims
import _string
import os_availability
import os_object
import sys_types
import unistd

public var DISPATCH_API_VERSION: Int32 {
get }

public var DISPATCH_SWIFT3_OVERLAY: Int32
{ get }

public var MSEC_PER_SEC: UInt64 { get }

public var NSEC_PER_SEC: UInt64 { get }

public var NSEC_PER_MSEC: UInt64 { get }

public var USEC_PER_SEC: UInt64 { get }

public var NSEC_PER_USEC: UInt64 { get }

/**
 * @typedef dispatch_time_t
 *
 * @abstract
```

```
 * A somewhat abstract representation of
time; where zero means "now" and
 * DISPATCH_TIME_FOREVER means "infinity"
and every value in between is an
 * opaque encoding.
 */
public typealias dispatch_time_t = UInt64

@available(macOS 10.14, *)
public var DISPATCH_WALLTIME_NOW: UInt {
get }

public var DISPATCH_TIME_NOW: UInt64 {
get }

public var DISPATCH_TIME_FOREVER: UInt64
{ get }

open class DispatchObject : OS_object {
}

public typealias dispatch_object_t =
DispatchObject

extension DispatchObject {

    /**
     * @function dispatch_activate
     *
     * @abstract
     * Activates the specified dispatch
object.
     *
```

```
     * @discussion
     * Dispatch objects such as queues
and sources may be created in an inactive
     * state. Objects in this state have
to be activated before any blocks
     * associated with them will be
invoked.
     *
     * The target queue of inactive
objects can be changed using
     * dispatch_set_target_queue().
Change of target queue is no longer
permitted
     * once an initially inactive object
has been activated.
     *
     * Calling dispatch_activate() on an
active object has no effect.
     * Releasing the last reference count
on an inactive object is undefined.
     *
     * @param object
     * The object to be activated.
     * The result of passing NULL in this
parameter is undefined.
     */
    @available(macOS 10.12, *)
    public func activate()

    /**
     * @function dispatch_suspend
     *
     * @abstract
```

```
     * Suspends the invocation of blocks
on a dispatch object.
     *
     * @discussion
     * A suspended object will not invoke
any blocks associated with it. The
     * suspension of an object will occur
after any running block associated with
     * the object completes.
     *
     * Calls to dispatch_suspend() must
be balanced with calls
     * to dispatch_resume().
     *
     * @param object
     * The object to be suspended.
     * The result of passing NULL in this
parameter is undefined.
     */
    @available(macOS 10.6, *)
    public func suspend()

    /**
     * @function dispatch_resume
     *
     * @abstract
     * Resumes the invocation of blocks
on a dispatch object.
     *
     * @discussion
     * Dispatch objects can be suspended
with dispatch_suspend(), which increments
     * an internal suspension count.
```

```
     * dispatch_resume() is the inverse operation,
     * and consumes suspension counts. When the last suspension count is consumed,
     * blocks associated with the object will be invoked again.
     *
     * For backward compatibility reasons, dispatch_resume() on an inactive and not
     * otherwise suspended dispatch source object has the same effect as calling
     * dispatch_activate(). For new code, using dispatch_activate() is preferred.
     *
     * If the specified object has zero suspension count and is not an inactive
     * source, this function will result in an assertion and the process being
     * terminated.
     *
     * @param object
     * The object to be resumed.
     * The result of passing NULL in this parameter is undefined.
     */
    @available(macOS 10.6, *)
    public func resume()

    /**
     * @function
```

```
dispatch_set_target_queue
     *
     * @abstract
     * Sets the target queue for the
given object.
     *
     * @discussion
     * An object's target queue is
responsible for processing the object.
     *
     * When no quality of service class
and relative priority is specified for a
     * dispatch queue at the time of
creation, a dispatch queue's quality of
service
     * class is inherited from its target
queue. The dispatch_get_global_queue()
     * function may be used to obtain a
target queue of a specific quality of
     * service class, however the use of
dispatch_queue_attr_make_with_qos_class()
     * is recommended instead.
     *
     * Blocks submitted to a serial queue
whose target queue is another serial
     * queue will not be invoked
concurrently with blocks submitted to the
target
     * queue or to any other queue with
that same target queue.
     *
     * The result of introducing a cycle
into the hierarchy of target queues is
```

```
     * undefined.
     *
     * A dispatch source's target queue
specifies where its event handler and
     * cancellation handler blocks will
be submitted.
     *
     * A dispatch I/O channel's target
queue specifies where where its I/O
     * operations are executed. If the
channel's target queue's priority is set
to
     *
DISPATCH_QUEUE_PRIORITY_BACKGROUND, then
the I/O operations performed by
     * dispatch_io_read() or
dispatch_io_write() on that queue will be
     * throttled when there is I/O
contention.
     *
     * For all other dispatch object
types, the only function of the target
queue
     * is to determine where an object's
finalizer function is invoked.
     *
     * In general, changing the target
queue of an object is an asynchronous
     * operation that doesn't take effect
immediately, and doesn't affect blocks
     * already associated with the
specified object.
     *
```

```
     * However, if an object is inactive
at the time dispatch_set_target_queue()
is
     * called, then the target queue
change takes effect immediately, and will
     * affect blocks already associated
with the specified object. After an
     * initially inactive object has been
activated, calling
     * dispatch_set_target_queue()
results in an assertion and the process
being
     * terminated.
     *
     * If a dispatch queue is active and
targeted by other dispatch objects,
     * changing its target queue results
in undefined behavior.  Instead, it is
     * recommended to create dispatch
objects in an inactive state, set up the
     * relevant target queues and then
activate them.
     *
     * @param object
     * The object to modify.
     * The result of passing NULL in this
parameter is undefined.
     *
     * @param queue
     * The new target queue for the
object. The queue is retained, and the
     * previous target queue, if any, is
released.
```

```
     * If queue is
DISPATCH_TARGET_QUEUE_DEFAULT, set the
object's target queue
     * to the default target queue for
the given object type.
     */
    @available(macOS 10.6, *)
    public func setTarget(queue:
dispatch_queue_t?)
}

open class DispatchQueue :
DispatchObject, @unchecked Sendable {
}

extension DispatchQueue {

    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
```

```
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let concurrent:
DispatchQueue.Attributes

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        public static let
initiallyInactive:
DispatchQueue.Attributes

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchQueue.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
```

```
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchQueue.Attributes

        /// 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 =
UInt64
    }

    public enum GlobalQueuePriority :
Sendable {

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
```

```
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case high

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case `default`

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case low

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
```

```
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case background

        /// 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:
DispatchQueue.GlobalQueuePriority, b:
DispatchQueue.GlobalQueuePriority) ->
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 }
    }

    public enum AutoreleaseFrequency :
Sendable {

        case inherit

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        case workItem

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        case never

        /// 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:
DispatchQueue.AutoreleaseFrequency, b:
```

```
DispatchQueue.AutoreleaseFrequency) ->
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 }
    }

    @preconcurrency public class func
concurrentPerform(iterations: Int,
execute work: @Sendable (Int) -> Void)

    public class var main: DispatchQueue
{ get }

    @available(macOS, deprecated: 10.10)
    @available(iOS, deprecated: 8.0)
    @available(tvOS, deprecated)
    @available(watchOS, deprecated)
    public class func global(priority:
```

```
DispatchQueue.GlobalQueuePriority) ->
DispatchQueue

    @available(macOS 10.10, iOS 8.0, *)
    public class func global(qos:
DispatchQoS.QoSClass = .default) ->
DispatchQueue

    @preconcurrency public class func
getSpecific<T>(key:
DispatchSpecificKey<T>) -> T? where T :
Sendable

    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes: DispatchQueue.Attributes =
[], autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .inherit, target: DispatchQueue? = nil)

    public var label: String { get }

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `async(execute:)`, however
    /// `sync(execute:)` will not return
until the work item has finished.
    ///
    /// Work items submitted to a queue
with `sync(execute:)` do not observe
```

```
certain
    /// queue attributes of that queue
when invoked (such as autorelease
frequency
    /// and QoS class).
    ///
    /// Calls to `sync(execute:)`
targeting the current queue will result
    /// in deadlock. Use of
`sync(execute:)` is also subject to the
same
    /// multi-party deadlock problems
that may result from the use of a mutex.
    /// Use of `async(execute:)` is
preferred.
    ///
    /// As an optimization,
`sync(execute:)` invokes the work item on
the thread which
    /// submitted it, except when the
queue is the main queue or
    /// a queue targetting it.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func sync(execute workItem:
DispatchWorkItem)
```

```
    ///
    /// Submits a work item for
asynchronous execution on a dispatch
queue.
    ///
    /// `async(execute:)` is the
fundamental mechanism for submitting
    /// work items to a dispatch queue.
    ///
    /// Calls to `async(execute:)` always
return immediately after the work item
has
    /// been submitted, and never wait
for the work item to be invoked.
    ///
    /// The target queue determines
whether the work item will be invoked
serially or
    /// concurrently with respect to
other work items submitted to that same
queue.
    /// Serial queues are processed
concurrently with respect to each other.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func async(execute workItem:
```

```
DispatchWorkItem)

    ///
    /// Submits a work item for
synchronous execution on a dispatch
queue.
    ///
    /// Submits a work item to a dispatch
queue like `async(execute:)`, however
    /// `asyncAndWait(execute:)` will not
return until the work item has finished.
    ///
    /// `asyncAndWait(excute:)` is
subject to deadlock under the same
conditions
    /// as `sync(execute:)`.
`asyncAndWait(execute:)` differs from
    /// `sync(execute:)` in the following
ways:
    ///
    ///   * Work items submitted to a
queue with `asyncAndWait` observe all
    ///     queue attributes of that
queue when invoked (including autorelease
    ///     frequency or DispatchQoS
class).
    ///
    ///   * Work items submitted to a
queue with `asyncAndWait` are not
    ///     guaranteed to run on the
calling thread.
    ///
    ///     If the queue the work is
```

```
submitted to already has a thread
    ///     servicing it, the servicing
thread will execute the work item
    ///     submitted via `asyncAndWait`.
If the queue the work is submitted
    ///     to does not have any threads
servicing it, the calling thread
    ///     will execute the work item.
As an exception, if the queue the work
    ///     is submitted to doesn't
target a global concurrent queue (for
example
    ///     because it targets the main
queue or a custom priority workloop),
    ///     then the work item will never
be invoked by the thread calling
    ///     `asyncAndWait(execute:)`.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso: `sync(execute:)`
    ///
    @available(macOS 10.14, iOS 12.0, *)
    public func asyncAndWait(execute
workItem: DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue and associates it with the given
    /// dispatch group. The dispatch
group may be used to wait for the
completion
```

```
    /// of the work items it references.
    ///
    /// - parameter group: the dispatch
group to associate with the submitted
block.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func async(group:
DispatchGroup, execute workItem:
DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue and optionally associates it with a
    /// dispatch group. The dispatch
group may be used to wait for the
completion
    /// of the work items it references.
    ///
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter group: the dispatch
group to associate with the submitted
    /// work item. If this is `nil`, the
work item is not associated with a group.
```

```
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func asyncUnsafe(group:
DispatchGroup? = nil, qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)

    ///
    /// Submits a work item to a dispatch
queue and optionally associates it with a
    /// dispatch group. The dispatch
group may be used to wait for the
completion
    /// of the work items it references.
    ///
    /// This method enforces the work
```

```
item to be sendable.
    ///
    /// - parameter group: the dispatch
group to associate with the submitted
    /// work item. If this is `nil`, the
work item is not associated with a group.
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    @preconcurrency public func
async(group: DispatchGroup? = nil, qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    /// Submits a work item for
synchronous execution on a dispatch
queue.
    ///
```

```
    /// Submits a work item to a dispatch
queue like `asyncAndWait(execute:)`,
    /// and returns the value, of type
`T`, returned by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    @available(macOS 10.14, iOS 12.0,
tvOS 12.0, watchOS 5.0, *)
    public func asyncAndWait<T>(execute
work: () throws -> T) rethrows -> T

    /// Submits a work item for
synchronous execution on a dispatch
queue.
    ///
    /// Submits a work item to a dispatch
queue like `asyncAndWait(execute:)`,
    /// and returns the value, of type
`T`, returned by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
```

```
    @available(macOS 10.14, iOS 12.0,
tvOS 12.0, watchOS 5.0, *)
    public func asyncAndWait<T>(flags:
DispatchWorkItemFlags, execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `sync(execute:)`, and returns
    /// the value, of type `T`, returned
by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso: `sync(execute:)`
    ///
    public func sync<T>(execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `sync(execute:)`, and returns
    /// the value, of type `T`, returned
by that work item.
    ///
```

```
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    public func sync<T>(flags:
DispatchWorkItemFlags, execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
```

```
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(deadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func
asyncAfterUnsafe(deadline: DispatchTime,
qos: DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This method enforces the work
item to be sendable.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
```

```
    /// given as a `DispatchTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(deadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchTime`
    ///
    @preconcurrency public func
asyncAfter(deadline: DispatchTime, qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
```

```
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(wallDeadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchWallTime`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func
asyncAfterUnsafe(wallDeadline:
DispatchWallTime, qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)
```

```
    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This method enforces the work
item to be sendable.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(wallDeadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchWallTime`
    ///
    @preconcurrency public func
asyncAfter(wallDeadline:
DispatchWallTime, qos: DispatchQoS
```

```
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchTime`.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso:
`asyncAfter(deadline:qos:flags:execute:)`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func asyncAfter(deadline:
DispatchTime, execute: DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
```

```
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso:
`asyncAfter(wallDeadline:qos:flags:execut
e:)`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func asyncAfter(wallDeadline:
DispatchWallTime, execute:
DispatchWorkItem)

    @available(macOS 10.10, iOS 8.0, *)
    public var qos: DispatchQoS { get }

    @preconcurrency public func
getSpecific<T>(key:
DispatchSpecificKey<T>) -> T? where T :
Sendable

    @preconcurrency public func
setSpecific<T>(key:
DispatchSpecificKey<T>, value: T?) where
T : Sendable
}

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

    /// The scheduler time type used by
the dispatch queue.
    public struct SchedulerTimeType :
```

```
Strideable, Codable, Hashable, Sendable {

        /// The dispatch time represented
by this type.
        public var dispatchTime:
DispatchTime

        /// Creates a dispatch queue time
type instance.
        ///
        /// - Parameter time: The
dispatch time to represent.
        public init(_ time: DispatchTime)

        /// 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

        /// 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

        /// Returns the distance to
another dispatch queue time.
        ///
        /// - Parameter other: Another
dispatch queue time.
        /// - Returns: The time interval
between this time and the provided time.
        public func distance(to other:
DispatchQueue.SchedulerTimeType) ->
DispatchQueue.SchedulerTimeType.Stride

        /// Returns a dispatch queue
scheduler time calculated by advancing
this instance’s time by the given
interval.
        ///
        /// - Parameter n: A time
interval to advance.
        /// - Returns: A dispatch queue
time advanced by the given interval from
this instance’s time.
        public func advanced(by n:
```

```
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType

        /// 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 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:
DispatchQueue.SchedulerTimeType, rhs:
DispatchQueue.SchedulerTimeType) -> Bool

        /// A type that represents the
distance between two values.
        public struct Stride :
SchedulerTimeIntervalConvertible,
Comparable, SignedNumeric,
ExpressibleByFloatLiteral, Hashable,
Codable {

            /// If created via floating
point literal, the value is converted to
nanoseconds via multiplication.
```

```
            public typealias
FloatLiteralType = Double

            /// Nanoseconds, same as
DispatchTimeInterval.
            public typealias
IntegerLiteralType = Int

            /// A type that can represent
the absolute value of any possible value
of the
            /// conforming type.
            public typealias Magnitude =
Int

            /// The value of this time
interval in nanoseconds.
            public var magnitude: Int

            /// A `DispatchTimeInterval`
created with the value of this type in
nanoseconds.
            public var timeInterval:
DispatchTimeInterval { get }

            /// Creates a dispatch queue
time interval from the given dispatch
time interval.
            ///
            /// - Parameter timeInterval:
A dispatch time interval.
            public init(_ timeInterval:
DispatchTimeInterval)
```

```
            /// Creates a dispatch queue
time interval from a floating-point
seconds value.
            ///
            /// - Parameter value: The
number of seconds, as a `Double`.
            public init(floatLiteral
value: Double)

            /// Creates a dispatch queue
time interval from an integer seconds
value.
            ///
            /// - Parameter value: The
number of seconds, as an `Int`.
            public init(integerLiteral
value: Int)

            /// Creates a dispatch queue
time interval from a binary integer type
representing a number of seconds.
            ///
            /// If `source` cannot be
exactly represented, the resulting time
interval is `nil`.
            /// - Parameter source: A
binary integer representing a time
interval.
            public init?<T>(exactly
source: T) where T : BinaryInteger

            /// 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:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> Bool

            /// Multiplies two values and
produces their product.
            ///
            /// The multiplication
operator (`*`) calculates the product of
its two
            /// arguments. For example:
            ///
            ///     2 * 3
```

```
// 6
            ///     100 * 21
// 2100
            ///     -10 * 15
// -150
            ///     3.5 * 2.25
// 7.875
            ///
            /// You cannot use `*` with
arguments of different types. To multiply
values
            /// of different types,
convert one of the values to the other
value's type.
            ///
            ///     let x: Int8 = 21
            ///     let y: Int = 1000000
            ///     Int(x) * y
// 21000000
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
            ///   - rhs: The second value
to multiply.
            public static func * (lhs:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Adds two values and
produces their sum.
```

```
            ///
            /// The addition operator
(`+`) calculates the sum of its two
arguments. For
            /// example:
            ///
            ///     1 + 2
// 3
            ///     -10 + 15
// 5
            ///     -15 + -5
// -20
            ///     21.5 + 3.25
// 24.75
            ///
            /// You cannot use `+` with
arguments of different types. To add
values of
            /// different types, convert
one of the values to the other value's
type.
            ///
            ///     let x: Int8 = 21
            ///     let y: Int = 1000000
            ///     Int(x) + y
// 1000021
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func + (lhs:
```

```
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Subtracts one value from
another and produces their difference.
            ///
            /// The subtraction operator
(`-`) calculates the difference of its
two
            /// arguments. For example:
            ///
            ///     8 - 3
// 5
            ///     -10 - 5
// -15
            ///     100 - -5
// 105
            ///     10.5 - 100.0
// -89.5
            ///
            /// You cannot use `-` with
arguments of different types. To subtract
values
            /// of different types,
convert one of the values to the other
value's type.
            ///
            ///     let x: UInt8 = 21
            ///     let y: UInt = 1000000
            ///     y - UInt(x)
// 999979
```

```
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func - (lhs:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Subtracts the second
value from the first and stores the
difference in the
            /// left-hand-side variable.
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func -= (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)

            /// Multiplies two values and
stores the result in the left-hand-side
            /// variable.
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
```

```
            ///   - rhs: The second value
to multiply.
            public static func *= (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)

            /// Adds two values and
stores the result in the left-hand-side
variable.
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func += (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)

            /// Converts the specified
number of seconds, as a floating-point
value, into an instance of this scheduler
time type.
            public static func seconds(_
s: Double) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of seconds into an instance of
```

```
this scheduler time type.
            public static func seconds(_
s: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of milliseconds into an instance
of this scheduler time type.
            public static func
milliseconds(_ ms: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of microseconds into an instance
of this scheduler time type.
            public static func
microseconds(_ us: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of nanoseconds into an instance of
this scheduler time type.
            public static func
nanoseconds(_ ns: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// 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 == (a:
DispatchQueue.SchedulerTimeType.Stride,
b:
DispatchQueue.SchedulerTimeType.Stride)
-> Bool

            /// 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

            /// 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 }

            /// 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
        }
```

```
        /// 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 }
    }

    /// Options that affect the operation
of the dispatch queue scheduler.
    public struct SchedulerOptions :
Sendable {

        /// The dispatch queue quality of
service.
        public var qos: DispatchQoS

        /// The dispatch queue work item
flags.
        public var flags:
```

```
DispatchWorkItemFlags

        /// The dispatch group, if any,
that should be used for performing
actions.
        public var group: DispatchGroup?

        public init(qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], group:
DispatchGroup? = nil)
    }

    /// The minimum tolerance allowed by
the scheduler.
    public var minimumTolerance:
DispatchQueue.SchedulerTimeType.Stride {
get }

    /// This scheduler’s definition of
the current moment in time.
    public var now:
DispatchQueue.SchedulerTimeType { get }

    /// Performs the action at the next
possible opportunity.
    public func schedule(options:
DispatchQueue.SchedulerOptions?, _
action: @escaping () -> Void)

    /// Performs the action at some time
after the specified date.
    public func schedule(after date:
```

```
DispatchQueue.SchedulerTimeType,
tolerance:
DispatchQueue.SchedulerTimeType.Stride,
options: DispatchQueue.SchedulerOptions?,
_ action: @escaping () -> Void)

    /// Performs the action at some time
after the specified date, at the
specified frequency, optionally taking
into account tolerance if possible.
    public func schedule(after date:
DispatchQueue.SchedulerTimeType,
interval:
DispatchQueue.SchedulerTimeType.Stride,
tolerance:
DispatchQueue.SchedulerTimeType.Stride,
options: DispatchQueue.SchedulerOptions?,
_ action: @escaping () -> Void) -> any
Cancellable
}

extension
DispatchQueue.GlobalQueuePriority :
Equatable {
}

extension
DispatchQueue.GlobalQueuePriority :
Hashable {
}

extension
DispatchQueue.AutoreleaseFrequency :
```

```
Equatable {
}

extension
DispatchQueue.AutoreleaseFrequency :
Hashable {
}

public typealias dispatch_queue_t =
DispatchQueue

open class OS_dispatch_queue_global :
DispatchQueue, @unchecked Sendable {
}

public typealias dispatch_queue_global_t
= OS_dispatch_queue_global

@available(macOS 14.0, *)
open class _DispatchSerialExecutorQueue :
DispatchQueue, @unchecked Sendable {
}

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

    public func enqueue(_ job: consuming
ExecutorJob)

    /// Convert this executor value to
the optimized form of borrowed
```

```
    /// executor references.
    public func asUnownedSerialExecutor()
-> UnownedSerialExecutor

    /// 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.
    public func checkIsolated()
}

public typealias
dispatch_queue_serial_executor_t =
_DispatchSerialExecutorQueue

@available(macOS 10.14, *)
open class DispatchSerialQueue :
_DispatchSerialExecutorQueue, @unchecked
Sendable {
}

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

    public struct Attributes : OptionSet,
Sendable {
```

```
        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
```

```
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let
initiallyInactive:
DispatchSerialQueue.Attributes

        /// The type of the elements of
```

```
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
ArrayLiteralElement =
DispatchSerialQueue.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchSerialQueue.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
```

```
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }

    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes:
DispatchSerialQueue.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, target: DispatchQueue? =
nil)
}

public typealias dispatch_queue_serial_t
= DispatchSerialQueue

open class OS_dispatch_queue_main :
DispatchSerialQueue, @unchecked Sendable
{
}

public typealias dispatch_queue_main_t =
OS_dispatch_queue_main

@available(macOS 10.14, *)
open class DispatchConcurrentQueue :
DispatchQueue, @unchecked Sendable {
}

@available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
```

```
extension DispatchConcurrentQueue {

    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
```

```
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let
```

```
initiallyInactive:
DispatchConcurrentQueue.Attributes

        /// The type of the elements of
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
ArrayLiteralElement =
DispatchConcurrentQueue.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchConcurrentQueue.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }

    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes:
DispatchConcurrentQueue.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, target: DispatchQueue? =
nil)
}

public typealias
dispatch_queue_concurrent_t =
DispatchConcurrentQueue

extension DispatchQueue {

    @available(macOS 10.6, *)
    public func sync(execute block: () ->
Void)

    @available(macOS 10.14, *)
    public func asyncAndWait(execute
block: () -> Void)
}
```

```
public var DISPATCH_APPLY_AUTO_AVAILABLE:
Int32 { get }

@available(macOS 10.6, *)
public var _dispatch_main_q: <<error
type>>

public var DISPATCH_QUEUE_PRIORITY_HIGH:
Int32 { get }

public var
DISPATCH_QUEUE_PRIORITY_DEFAULT: Int32 {
get }

public var DISPATCH_QUEUE_PRIORITY_LOW:
Int32 { get }

public var
DISPATCH_QUEUE_PRIORITY_BACKGROUND: Int32
{ get }

public typealias dispatch_queue_attr_t =
__OS_dispatch_queue_attr

@available(macOS 10.7, *)
public var
_dispatch_queue_attr_concurrent: <<error
type>>

public enum
__dispatch_autorelease_frequency_t :
UInt, @unchecked Sendable {
```

```
}

/**
 * @function dispatch_main
 *
 * @abstract
 * Execute blocks submitted to the main
queue.
 *
 * @discussion
 * This function "parks" the main thread
and waits for blocks to be submitted
 * to the main queue. This function never
returns.
 *
 * Applications that call
NSApplicationMain() or CFRunLoopRun() on
the
 * main thread do not need to call
dispatch_main().
 */
@available(macOS 10.6, *)
public func dispatchMain() -> Never

@available(macOS 14.4, *)
public func dispatch_allow_send_signals(_
preserve_signum: Int32) -> Int32

public struct __dispatch_block_flags_t :
OptionSet, @unchecked Sendable {

    public init(rawValue: UInt)
}
```

```
open class DispatchSource :
DispatchObject, @unchecked Sendable {
}

extension DispatchSource {

    public struct MachSendEvent :
OptionSet, RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
```

```
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let dead:
DispatchSource.MachSendEvent

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.MachSendEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.MachSendEvent

        /// 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 = UInt
    }

    public struct MemoryPressureEvent :
OptionSet, RawRepresentable,
CustomStringConvertible {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
```

```
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let normal:
DispatchSource.MemoryPressureEvent

        public static let warning:
DispatchSource.MemoryPressureEvent

        public static let critical:
DispatchSource.MemoryPressureEvent

        public static let all:
DispatchSource.MemoryPressureEvent

        /// 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.
        public var description: String {
get }

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.MemoryPressureEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
```

```
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.MemoryPressureEvent

        /// 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 = UInt
    }

    public struct ProcessEvent :
OptionSet, RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
```

```
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let exit:
DispatchSource.ProcessEvent

        public static let fork:
DispatchSource.ProcessEvent

        public static let exec:
DispatchSource.ProcessEvent

        public static let signal:
DispatchSource.ProcessEvent

        public static let all:
DispatchSource.ProcessEvent
```

```
        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.ProcessEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.ProcessEvent

        /// 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 = UInt
    }

    public struct TimerFlags : OptionSet,
```

```
RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
```

```
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let strict:
DispatchSource.TimerFlags

        /// The type of the elements of
```

```
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.TimerFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.TimerFlags

        /// 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 = UInt
    }

    public struct FileSystemEvent :
OptionSet, RawRepresentable {
```

```
        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
```

```
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let delete:
DispatchSource.FileSystemEvent

        public static let write:
DispatchSource.FileSystemEvent
```

```
        public static let extend:
DispatchSource.FileSystemEvent

        public static let attrib:
DispatchSource.FileSystemEvent

        public static let link:
DispatchSource.FileSystemEvent

        public static let rename:
DispatchSource.FileSystemEvent

        public static let revoke:
DispatchSource.FileSystemEvent

        public static let funlock:
DispatchSource.FileSystemEvent

        public static let all:
DispatchSource.FileSystemEvent

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.FileSystemEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
```

```
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.FileSystemEvent

        /// 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 = UInt
    }

    public class func
makeMachSendSource(port: mach_port_t,
eventMask: DispatchSource.MachSendEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceMachSend

    public class func
makeMachReceiveSource(port: mach_port_t,
queue: DispatchQueue? = nil) -> any
DispatchSourceMachReceive

    public class func
```

```
makeMemoryPressureSource(eventMask:
DispatchSource.MemoryPressureEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceMemoryPressure

    public class func
makeProcessSource(identifier: pid_t,
eventMask: DispatchSource.ProcessEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceProcess

    public class func
makeReadSource(fileDescriptor: Int32,
queue: DispatchQueue? = nil) -> any
DispatchSourceRead

    public class func
makeSignalSource(signal: Int32, queue:
DispatchQueue? = nil) -> any
DispatchSourceSignal

    public class func
makeTimerSource(flags:
DispatchSource.TimerFlags = [], queue:
DispatchQueue? = nil) -> any
DispatchSourceTimer

    public class func
makeUserDataAddSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataAdd

    public class func
```

```
makeUserDataOrSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataOr

    public class func
makeUserDataReplaceSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataReplace

    public class func
makeFileSystemObjectSource(fileDescriptor
: Int32, eventMask:
DispatchSource.FileSystemEvent, queue:
DispatchQueue? = nil) -> any
DispatchSourceFileSystemObject

    public class func
makeWriteSource(fileDescriptor: Int32,
queue: DispatchQueue? = nil) -> any
DispatchSourceWrite
}

public typealias dispatch_source_t =
DispatchSource

public protocol DispatchSourceProtocol :
NSObjectProtocol {
}

extension DispatchSourceProtocol {

    public typealias
DispatchSourceHandler =
```

```
@convention(block) () -> Void

    public func setEventHandler(qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)

    @available(macOS 10.10, iOS 8.0, *)
    public func setEventHandler(handler:
DispatchWorkItem)

    public func setCancelHandler(qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)

    @available(macOS 10.10, iOS 8.0, *)
    public func setCancelHandler(handler:
DispatchWorkItem)

    public func
setRegistrationHandler(qos: DispatchQoS =
.unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)

    @available(macOS 10.10, iOS 8.0, *)
    public func
setRegistrationHandler(handler:
DispatchWorkItem)

    @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
```

```
    public func activate()

    public func cancel()

    public func resume()

    public func suspend()

    public var handle: UInt { get }

    public var mask: UInt { get }

    public var data: UInt { get }

    public var isCancelled: Bool { get }
}

extension DispatchSource :
DispatchSourceProtocol {
}

@available(macOS 10.6, *)
public var
_dispatch_source_type_data_add: <<error
type>>

public protocol DispatchSourceUserDataAdd
: DispatchSourceProtocol {
}

extension DispatchSourceUserDataAdd {

    /// @function add
```

```
    ///
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_ADD
    /// and submits its event handler
block to its target queue.
    ///
    /// @param data
    /// The value to add to the current
pending data. A value of zero has no
effect
    /// and will not result in the
submission of the event handler block.
    public func add(data: UInt)
}

extension DispatchSource :
DispatchSourceUserDataAdd {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_data_or:
<<error type>>

public protocol
DispatchSourceUserDataOr :
DispatchSourceProtocol {
}

extension DispatchSourceUserDataOr {

    /// @function or
```

```
    ///
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_OR and
    /// submits its event handler block
to its target queue.
    ///
    /// @param data
    /// The value to OR into the current
pending data. A value of zero has no
effect
    /// and will not result in the
submission of the event handler block.
    public func or(data: UInt)
}

extension DispatchSource :
DispatchSourceUserDataOr {
}

@available(macOS 10.13, *)
public var
_dispatch_source_type_data_replace:
<<error type>>

public protocol
DispatchSourceUserDataReplace :
DispatchSourceProtocol {
}

extension DispatchSourceUserDataReplace {
```

```
    /// @function replace
    ///
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_REPLACE
    /// and submits its event handler
block to its target queue.
    ///
    /// @param data
    /// The value that will replace the
current pending data. A value of zero
will be stored
    /// but will not result in the
submission of the event handler block.
    public func replace(data: UInt)
}

extension DispatchSource :
DispatchSourceUserDataReplace {
}

@available(macOS 10.6, *)
public var
_dispatch_source_type_mach_send: <<error
type>>

public protocol DispatchSourceMachSend :
DispatchSourceProtocol {
}

extension DispatchSourceMachSend {
```

```
    public var handle: mach_port_t {
get }

    public var data:
DispatchSource.MachSendEvent { get }

    public var mask:
DispatchSource.MachSendEvent { get }
}

extension DispatchSource :
DispatchSourceMachSend {
}

@available(macOS 10.6, *)
public var
_dispatch_source_type_mach_recv: <<error
type>>

public protocol DispatchSourceMachReceive
: DispatchSourceProtocol {
}

extension DispatchSourceMachReceive {

    public var handle: mach_port_t {
get }
}

extension DispatchSource :
DispatchSourceMachReceive {
}
```

```
@available(macOS 10.9, *)
public var
_dispatch_source_type_memorypressure:
<<error type>>

public protocol
DispatchSourceMemoryPressure :
DispatchSourceProtocol {
}

extension DispatchSourceMemoryPressure {

    public var data:
DispatchSource.MemoryPressureEvent {
get }

    public var mask:
DispatchSource.MemoryPressureEvent {
get }
}

extension DispatchSource :
DispatchSourceMemoryPressure {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_proc:
<<error type>>

public protocol DispatchSourceProcess :
DispatchSourceProtocol {
}
```

```
extension DispatchSourceProcess {

    public var handle: pid_t { get }

    public var data:
DispatchSource.ProcessEvent { get }

    public var mask:
DispatchSource.ProcessEvent { get }
}

extension DispatchSource :
DispatchSourceProcess {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_read:
<<error type>>

public protocol DispatchSourceRead :
DispatchSourceProtocol {
}

extension DispatchSource :
DispatchSourceRead {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_signal:
<<error type>>

public protocol DispatchSourceSignal :
DispatchSourceProtocol {
```

```
}

extension DispatchSource :
DispatchSourceSignal {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_timer:
<<error type>>

public protocol DispatchSourceTimer :
DispatchSourceProtocol {
}

extension DispatchSourceTimer {

    ///
    /// Sets the deadline and leeway for
a timer event that fires once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared and the next
timer event will occur at `deadline`.
    ///
    /// Delivery of the timer event may
be delayed by the system in order to
improve power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
```

```
under the control of the system.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    /// - note: Delivery of the timer
event does not cancel the timer source.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
```

```
renamed:
"schedule(deadline:repeating:leeway:)")
    public func scheduleOneshot(deadline:
DispatchTime, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline and leeway for
a timer event that fires once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared and the next
timer event will occur at `wallDeadline`.
    ///
    /// Delivery of the timer event may
be delayed by the system in order to
improve power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
```

```
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    /// - note: Delivery of the timer
event does not cancel the timer source.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
    public func
scheduleOneshot(wallDeadline:
DispatchWallTime, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
```

```
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`interval` units of
    /// time thereafter until the timer
source is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N * interval`,
the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
```

```
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter interval: the
interval for the timer.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(deadline:repeating:leeway:)")
    public func
scheduleRepeating(deadline: DispatchTime,
interval: DispatchTimeInterval, leeway:
```

```
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`interval` seconds
    /// thereafter until the timer source
is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N * interval`,
the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
```

```
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter interval: the
interval for the timer in seconds.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
```

```
"schedule(deadline:repeating:leeway:)")
    public func
scheduleRepeating(deadline: DispatchTime,
interval: Double, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `interval` units of
    /// time thereafter until the timer
source is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
```

```
timer fires at `wallDeadline + N * interval`, the upper
    /// limit is the smaller of `leeway` and `interval/2`.
    ///
    /// The lower limit to the allowable delay may vary with process state such as visibility of the
    /// application UI. If the timer source was created with flags `TimerFlags.strict`, the system
    /// will make a best effort to strictly observe the provided `leeway` value, even if it is smaller
    /// than the current lower limit. Note that a minimal amount of delay is to be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect if the timer source has already been canceled.
    ///
    /// - parameter wallDeadline: the time at which the timer event will be delivered, subject to the
    ///     leeway and other considerations described above. The deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter interval: the interval for the timer.
    /// - parameter leeway: the leeway
```

```
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
    public func
scheduleRepeating(wallDeadline:
DispatchWallTime, interval:
DispatchTimeInterval, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `interval` seconds
    /// thereafter until the timer source
is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
```

```
under the control of the system.
    ///
    /// For the initial timer fire at `wallDeadline`, the upper limit to the allowable delay is set to
    /// `leeway`. For the subsequent timer fires at `wallDeadline + N * interval`, the upper
    /// limit is the smaller of `leeway` and `interval/2`.
    ///
    /// The lower limit to the allowable delay may vary with process state such as visibility of the
    /// application UI. If the timer source was created with flags `TimerFlags.strict`, the system
    /// will make a best effort to strictly observe the provided `leeway` value, even if it is smaller
    /// than the current lower limit. Note that a minimal amount of delay is to be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect if the timer source has already been canceled.
    ///
    /// - parameter wallDeadline: the time at which the timer event will be delivered, subject to the
    ///     leeway and other
```

```
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter interval: the
interval for the timer in seconds.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
    public func
scheduleRepeating(wallDeadline:
DispatchWallTime, interval: Double,
leeway: DispatchTimeInterval
= .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`repeating` units of
    /// time thereafter until the timer
source is canceled. If the value of
`repeating` is `.never`,
    /// or is defaulted, the timer fires
only once.
```

```
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
```

```
    /// Calling this method has no effect if the timer source has already been canceled.
    ///
    /// - parameter deadline: the time at which the first timer event will be delivered, subject to the
    ///     leeway and other considerations described above. The deadline is based on Mach absolute
    ///     time.
    /// - parameter repeating: the repeat interval for the timer, or `.never` if the timer should fire
    ///     only once.
    /// - parameter leeway: the leeway for the timer.
    ///
    @available(swift 4)
    public func schedule(deadline: DispatchTime, repeating interval: DispatchTimeInterval = .never, leeway: DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, repeat interval and leeway for a timer event.
    ///
    /// Once this function returns, any pending source data accumulated for the previous timer values
    /// has been cleared. The next timer event will occur at `deadline` and every
```

```
`repeating` seconds
    /// thereafter until the timer source
is canceled. If the value of `repeating`
is `.infinity`,
    /// the timer fires only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
```

```
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter repeating: the repeat
interval for the timer in seconds, or
`.infinity` if the timer
    ///     should fire only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(deadline:
DispatchTime, repeating interval: Double,
leeway: DispatchTimeInterval
= .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event.
    ///
```

```
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `repeating` units of
    /// time thereafter until the timer
source is canceled. If the value of
`repeating` is `.never`,
    /// or is defaulted, the timer fires
only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
```

```
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter repeating: the repeat
interval for the timer, or `.never` if
the timer should fire
    ///     only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(wallDeadline:
DispatchWallTime, repeating interval:
DispatchTimeInterval = .never, leeway:
```

```
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event
that fires at least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `repeating` seconds
    /// thereafter until the timer source
is canceled. If the value of `repeating`
is `.infinity`,
    /// the timer fires only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
repeating`, the upper
```

```
    /// limit is the smaller of `leeway` and `repeating/2`.
    ///
    /// The lower limit to the allowable delay may vary with process state such as visibility of the
    /// application UI. If the timer source was created with flags `TimerFlags.strict`, the system
    /// will make a best effort to strictly observe the provided `leeway` value, even if it is smaller
    /// than the current lower limit. Note that a minimal amount of delay is to be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect if the timer source has already been canceled.
    ///
    /// - parameter wallDeadline: the time at which the timer event will be delivered, subject to the
    ///     leeway and other considerations described above. The deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter repeating: the repeat interval for the timer in secondss, or `.infinity` if the timer
    /// should fire only once.
    /// - parameter leeway: the leeway
```

```
for the timer.
    ///
    @available(swift 4)
    public func schedule(wallDeadline:
DispatchWallTime, repeating interval:
Double, leeway: DispatchTimeInterval
= .nanoseconds(0))
}

extension DispatchSource :
DispatchSourceTimer {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_vnode:
<<error type>>

public protocol
DispatchSourceFileSystemObject :
DispatchSourceProtocol {
}

extension DispatchSourceFileSystemObject
{

    public var handle: Int32 { get }

    public var data:
DispatchSource.FileSystemEvent { get }

    public var mask:
DispatchSource.FileSystemEvent { get }
}
```

```
extension DispatchSource :
DispatchSourceFileSystemObject {
}

@available(macOS 10.6, *)
public var _dispatch_source_type_write:
<<error type>>

public protocol DispatchSourceWrite :
DispatchSourceProtocol {
}

extension DispatchSource :
DispatchSourceWrite {
}

public var DISPATCH_MACH_SEND_DEAD: Int32
{ get }

/**
 * @typedef
dispatch_source_mach_recv_flags_t
 * Type of dispatch_source_mach_recv
flags
 */
public typealias
dispatch_source_mach_recv_flags_t = UInt

public var
DISPATCH_MEMORYPRESSURE_NORMAL: Int32 {
get }
```

```
public var DISPATCH_MEMORYPRESSURE_WARN:
Int32 { get }

public var
DISPATCH_MEMORYPRESSURE_CRITICAL: Int32 {
get }

public var DISPATCH_PROC_EXIT: UInt32 {
get }

public var DISPATCH_PROC_FORK: Int32 {
get }

public var DISPATCH_PROC_EXEC: Int32 {
get }

public var DISPATCH_PROC_SIGNAL: Int32 {
get }

public var DISPATCH_VNODE_DELETE: Int32 {
get }

public var DISPATCH_VNODE_WRITE: Int32 {
get }

public var DISPATCH_VNODE_EXTEND: Int32 {
get }

public var DISPATCH_VNODE_ATTRIB: Int32 {
get }

public var DISPATCH_VNODE_LINK: Int32 {
get }
```

```
public var DISPATCH_VNODE_RENAME: Int32 {
get }

public var DISPATCH_VNODE_REVOKE: Int32 {
get }

public var DISPATCH_VNODE_FUNLOCK: Int32
{ get }

public var DISPATCH_TIMER_STRICT: Int32 {
get }

open class DispatchGroup :
DispatchObject, @unchecked Sendable {
}

/// dispatch_group
extension DispatchGroup {

    public func notify(qos: DispatchQoS =
.unspecified, flags:
DispatchWorkItemFlags = [], queue:
DispatchQueue, execute work: @escaping
@convention(block) () -> Void)

    @available(macOS 10.10, iOS 8.0, *)
    public func notify(queue:
DispatchQueue, work: DispatchWorkItem)

    public func wait()

    public func wait(timeout:
```

```
DispatchTime) -> DispatchTimeoutResult

    public func wait(wallTimeout timeout:
DispatchWallTime) ->
DispatchTimeoutResult
}

public typealias dispatch_group_t =
DispatchGroup

extension DispatchGroup {

    /**
     * @function dispatch_group_create
     *
     * @abstract
     * Creates new group with which
blocks may be associated.
     *
     * @discussion
     * This function creates a new group
with which blocks may be associated.
     * The dispatch group may be used to
wait for the completion of the blocks it
     * references. The group object
memory is freed with dispatch_release().
     *
     * @result
     * The newly created group, or NULL
on failure.
     */
    @available(macOS 10.6, *)
    public /*not inherited*/ init()
```

```
    /**
     * @function dispatch_group_enter
     *
     * @abstract
     * Manually indicate a block has
entered the group
     *
     * @discussion
     * Calling this function indicates
another block has joined the group
through
     * a means other than
dispatch_group_async(). Calls to this
function must be
     * balanced with
dispatch_group_leave().
     *
     * @param group
     * The dispatch group to update.
     * The result of passing NULL in this
parameter is undefined.
     */
    @available(macOS 10.6, *)
    public func enter()

    /**
     * @function dispatch_group_leave
     *
     * @abstract
     * Manually indicate a block in the
group has completed
     *
```

```
     * @discussion
     * Calling this function indicates
block has completed and left the dispatch
     * group by a means other than
dispatch_group_async().
     *
     * @param group
     * The dispatch group to update.
     * The result of passing NULL in this
parameter is undefined.
     */
    @available(macOS 10.6, *)
    public func leave()
}

open class DispatchSemaphore :
DispatchObject, @unchecked Sendable {
}

/// dispatch_semaphore
extension DispatchSemaphore {

    @discardableResult
    public func signal() -> Int

    public func wait()

    public func wait(timeout:
DispatchTime) -> DispatchTimeoutResult

    public func wait(wallTimeout:
DispatchWallTime) ->
DispatchTimeoutResult
```

```
}

public typealias dispatch_semaphore_t =
DispatchSemaphore

extension DispatchSemaphore {

    /**
     * @function
dispatch_semaphore_create
     *
     * @abstract
     * Creates new counting semaphore
with an initial value.
     *
     * @discussion
     * Passing zero for the value is
useful for when two threads need to
reconcile
     * the completion of a particular
event. Passing a value greater than zero
is
     * useful for managing a finite pool
of resources, where the pool size is
equal
     * to the value.
     *
     * @param value
     * The starting value for the
semaphore. Passing a value less than zero
will
     * cause NULL to be returned.
     *
```

```
     * @result
     * The newly created semaphore, or
NULL on failure.
     */
    @available(macOS 10.6, *)
    public /*not inherited*/ init(value:
Int)
}

public var DISPATCH_ONCE_INLINE_FASTPATH:
Int32 { get }

public func __builtin_expect(_: Int, _:
Int) -> Int

public func __builtin_assume(_: Bool)

open class __DispatchData : NSObject {
}

public typealias dispatch_data_t =
__DispatchData

@available(macOS 10.7, *)
public var _dispatch_data_empty: <<error
type>>

open class DispatchIO : DispatchObject,
@unchecked Sendable {
}

extension DispatchIO {
```

```
    public enum StreamType : UInt,
Sendable {

        case stream

        case random

        /// 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: UInt)
```

```
        /// 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 = UInt

        /// 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: UInt { get }
    }

    public struct CloseFlags : OptionSet,
RawRepresentable, Sendable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from the given raw value.
        ///
        /// This initializer always succeeds, even if the value passed as `rawValue`
        /// exceeds the static properties declared as part of the option set. This
        /// example creates an instance of `ShippingOptions` with a raw value beyond
        /// the highest element, with a bit mask that effectively contains all the
        /// declared static members.
        ///
        ///     let extraOptions = ShippingOptions(rawValue: 255)
        ///
        /// print(extraOptions.isStrictSuperset(of: .all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw value of the option set to create. Each bit
        ///   of `rawValue` potentially represents an element of the option set,
        ///   though raw values may include bits that are not defined as
```

```
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let stop:
DispatchIO.CloseFlags

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchIO.CloseFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchIO.CloseFlags

        /// 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 = UInt
    }

    public struct IntervalFlags :
OptionSet, RawRepresentable, Sendable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
```

```
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public init(nilLiteral: ())

        public static let strictInterval:
DispatchIO.IntervalFlags

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchIO.IntervalFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchIO.IntervalFlags

        /// 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 = UInt
    }

    public class func
read(fromFileDescriptor: Int32,
maxLength: Int, runningHandlerOn queue:
DispatchQueue, handler: @escaping (_
data: DispatchData, _ error: Int32) ->
Void)

    public class func
write(toFileDescriptor: Int32, data:
DispatchData, runningHandlerOn queue:
DispatchQueue, handler: @escaping (_
data: DispatchData?, _ error: Int32) ->
Void)

    public convenience init(type:
DispatchIO.StreamType, fileDescriptor:
Int32, queue: DispatchQueue,
cleanupHandler: @escaping (_ error:
Int32) -> Void)

    @available(swift 4)
    public convenience init?(type:
DispatchIO.StreamType, path:
UnsafePointer<Int8>, oflag: Int32, mode:
mode_t, queue: DispatchQueue,
```

```
cleanupHandler: @escaping (_ error:
Int32) -> Void)

    public convenience init(type:
DispatchIO.StreamType, io: DispatchIO,
queue: DispatchQueue, cleanupHandler:
@escaping (_ error: Int32) -> Void)

    public func read(offset: off_t,
length: Int, queue: DispatchQueue,
ioHandler: @escaping (_ done: Bool, _
data: DispatchData?, _ error: Int32) ->
Void)

    public func write(offset: off_t,
data: DispatchData, queue: DispatchQueue,
ioHandler: @escaping (_ done: Bool, _
data: DispatchData?, _ error: Int32) ->
Void)

    public func setInterval(interval:
DispatchTimeInterval, flags:
DispatchIO.IntervalFlags = [])

    public func close(flags:
DispatchIO.CloseFlags = [])
}

extension DispatchIO.StreamType :
Equatable {
}

extension DispatchIO.StreamType :
```

```
Hashable {
}

extension DispatchIO.StreamType :
RawRepresentable {
}

public typealias dispatch_io_t =
DispatchIO

public var DISPATCH_IO_STREAM: Int32 {
get }

public var DISPATCH_IO_RANDOM: Int32 {
get }

extension DispatchIO {

    /**
     * @function dispatch_io_barrier
     * Schedule a barrier operation on
the specified I/O channel; all previously
     * scheduled operations on the
channel will complete before the provided
     * barrier block is enqueued onto the
global queue determined by the channel's
     * target queue, and no subsequently
scheduled operations will start until the
     * barrier block has returned.
     *
     * If multiple channels are
associated with the same file descriptor,
a barrier
```

```
     * operation scheduled on any of
these channels will act as a barrier
across all
     * channels in question, i.e. all
previously scheduled operations on any of
the
     * channels will complete before the
barrier block is enqueued, and no
     * operations subsequently scheduled
on any of the channels will start until
the
     * barrier block has returned.
     *
     * While the barrier block is
running, it may safely operate on the
channel's
     * underlying file descriptor with
fsync(2), lseek(2) etc. (but not
close(2)).
     *
     * @param channel   The dispatch I/O
channel to schedule the barrier on.
     * @param barrier   The barrier
block.
     */
    @available(macOS 10.7, *)
    public func barrier(execute barrier:
@escaping () -> Void)

    /**
     * @function
dispatch_io_get_descriptor
     * Returns the file descriptor
```

```
underlying a dispatch I/O channel.
     *
     * Will return -1 for a channel
closed with dispatch_io_close() and for a
     * channel associated with a path
name that has not yet been open(2)ed.
     *
     * If called from a barrier block
scheduled on a channel associated with a
path
     * name that has not yet been
open(2)ed, this will trigger the channel
open(2)
     * operation and return the resulting
file descriptor.
     *
     * @param channel   The dispatch I/O
channel to query.
     * @result      The file descriptor
underlying the channel, or -1.
     */
    @available(macOS 10.7, *)
    public var fileDescriptor: Int32 {
get }

    /**
     * @function
dispatch_io_set_high_water
     * Set a high water mark on the I/O
channel for all operations.
     *
     * The system will make a best effort
to enqueue I/O handlers with partial
```

```
     * results as soon the number of
bytes processed by an operation (i.e.
read or
     * written) reaches the high water
mark.
     *
     * The size of data objects passed to
I/O handlers for this channel will never
     * exceed the specified high water
mark.
     *
     * The default value for the high
water mark is unlimited (i.e. SIZE_MAX).
     *
     * @param channel   The dispatch I/O
channel on which to set the policy.
     * @param high_water The number of
bytes to use as a high water mark.
     */
    @available(macOS 10.7, *)
    public func setLimit(highWater
high_water: Int)

    /**
     * @function
dispatch_io_set_low_water
     * Set a low water mark on the I/O
channel for all operations.
     *
     * The system will process (i.e. read
or write) at least the low water mark
     * number of bytes for an operation
before enqueueing I/O handlers with
```

```
partial
     * results.
     *
     * The size of data objects passed to
intermediate I/O handler invocations for
     * this channel (i.e. excluding the
final invocation) will never be smaller
than
     * the specified low water mark,
except if the channel has an interval
with the
     * DISPATCH_IO_STRICT_INTERVAL flag
set or if EOF or an error was
encountered.
     *
     * I/O handlers should be prepared to
receive amounts of data significantly
     * larger than the low water mark in
general. If an I/O handler requires
     * intermediate results of fixed
size, set both the low and and the high
water
     * mark to that size.
     *
     * The default value for the low
water mark is unspecified, but must be
assumed
     * to be such that intermediate
handler invocations may occur.
     * If I/O handler invocations with
partial results are not desired, set the
     * low water mark to SIZE_MAX.
     *
```

```
     * @param channel    The dispatch I/O
channel on which to set the policy.
     * @param low_water  The number of
bytes to use as a low water mark.
     */
    @available(macOS 10.7, *)
    public func setLimit(lowWater
low_water: Int)
}

public var DISPATCH_IO_STOP: Int32 {
get }

public var DISPATCH_IO_STRICT_INTERVAL:
Int32 { get }

@available(macOS 10.14, *)
open class DispatchWorkloop :
_DispatchSerialExecutorQueue, @unchecked
Sendable {
}

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

    ///
    /// Workloop attributes to customize
at creation time.
    ///
    /// This is an empty set today; but,
support for additional attributes could
be
```

```
    /// added in the future.
    ///
    /// The reason this exists is it has
SPI only attribute to create an initially
    /// inactive workloop. (See
DispatchWorkloop.Attributes.initiallyInac
tive)
    /// The goal is to future proof our
internal clients that create an inactive
workloop,
    /// set properties such as scheduler
priority or QoS Class that are SPI only,
followed by
    /// activation of the workloop.
    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
```

```
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        /// The type of the elements of
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
ArrayLiteralElement =
DispatchWorkloop.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchWorkloop.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }

    ///
    /// Initializes an instance of
DispatchWorkloop
    ///
    /// - parameter label: A string label
to attach to the workloop.
    /// - parameter attributes:
Additional workloop attributes to
customize.
    ///   (See
DispatchWorkloop.Attributes).
    /// - parameter autoreleaseFrequency:
Autorelease frequency to assign to the
workloop.
    ///   See
DispatchQueue.AutoreleaseFrequency.
```

```
Defaults to
AutoreleaseFrequency.workItem.
    /// - parameter osWorkgroup: OS
Workgroup to assign to the workloop.
    public convenience init(label:
String, attributes:
DispatchWorkloop.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, osWorkgroup: WorkGroup? =
nil)
}

public typealias dispatch_workloop_t =
DispatchWorkloop

public func + (time: DispatchTime,
interval: DispatchTimeInterval) ->
DispatchTime

public func + (time: DispatchTime,
seconds: Double) -> DispatchTime

public func + (time: DispatchWallTime,
interval: DispatchTimeInterval) ->
DispatchWallTime

public func + (time: DispatchWallTime,
seconds: Double) -> DispatchWallTime

public func - (time: DispatchTime,
interval: DispatchTimeInterval) ->
DispatchTime
```

```
public func - (time: DispatchTime,
seconds: Double) -> DispatchTime

public func - (time: DispatchWallTime,
interval: DispatchTimeInterval) ->
DispatchWallTime

public func - (time: DispatchWallTime,
seconds: Double) -> DispatchWallTime

public struct DispatchData :
RandomAccessCollection {

    /// A type that provides the
collection's iteration interface and
    /// encapsulates its iteration state.
    ///
    /// By default, a collection conforms
to the `Sequence` protocol by
    /// supplying `IndexingIterator` as
its associated `Iterator`
    /// type.
    public typealias Iterator =
DispatchDataIterator

    /// A type that represents a position
in the collection.
    ///
    /// Valid indices consist of the
position of every element and a
    /// "past the end" position that's
not valid for use as a subscript
```

```
    /// argument.
    public typealias Index = Int

    /// A type that represents the
indices that are valid for subscripting
the
    /// collection, in ascending order.
    public typealias Indices =
DefaultIndices<DispatchData>

    public static let empty: DispatchData

    public enum Deallocator {

        /// Use `free`
        case free

        /// Use `munmap`
        case unmap

        /// A custom deallocator
        case custom(DispatchQueue?,
@convention(block) () -> Void)
    }

    /// Initialize a `Data` with copied
memory content.
    ///
    /// - parameter bytes: A pointer to
the memory. It will be copied.
    /// - parameter count: The number of
bytes to copy.
    @available(swift, deprecated: 4,
```

```
message: "Use init(bytes:
UnsafeRawBufferPointer) instead")
    public init(bytes buffer:
UnsafeBufferPointer<UInt8>)

    /// Initialize a `Data` with copied
memory content.
    ///
    /// - parameter bytes: A pointer to
the memory. It will be copied.
    /// - parameter count: The number of
bytes to copy.
    public init(bytes buffer:
UnsafeRawBufferPointer)

    /// Initialize a `Data` without
copying the bytes.
    ///
    /// - parameter bytes: A pointer to
the bytes.
    /// - parameter count: The size of
the bytes.
    /// - parameter deallocator:
Specifies the mechanism to free the
indicated buffer.
    @available(swift, deprecated: 4,
message: "Use init(bytesNoCopy:
UnsafeRawBufferPointer, deallocater:
Deallocator) instead")
    public init(bytesNoCopy bytes:
UnsafeBufferPointer<UInt8>, deallocator:
DispatchData.Deallocator = .free)
```

```
    /// Initialize a `Data` without
copying the bytes.
    ///
    /// - parameter bytes: A pointer to
the bytes.
    /// - parameter count: The size of
the bytes.
    /// - parameter deallocator:
Specifies the mechanism to free the
indicated buffer.
    public init(bytesNoCopy bytes:
UnsafeRawBufferPointer, deallocator:
DispatchData.Deallocator = .free)

    /// The number of elements in the
collection.
    ///
    /// To check whether a collection is
empty, use its `isEmpty` property
    /// instead of comparing `count` to
zero. Unless the collection guarantees
    /// random-access performance,
calculating `count` can be an O(*n*)
    /// operation.
    ///
    /// - Complexity: O(1) if the
collection conforms to
    ///   `RandomAccessCollection`;
otherwise, O(*n*), where *n* is the
length
    ///   of the collection.
    public var count: Int { get }
```

```
    public func withUnsafeBytes<Result,
ContentType>(body:
(UnsafePointer<ContentType>) throws ->
Result) rethrows -> Result

    @available(swift 4.2)
    public func enumerateBytes(_ block:
(_ buffer: UnsafeBufferPointer<UInt8>, _
byteIndex: Int, _ stop: inout Bool) ->
Void)

    /// Append bytes to the data.
    ///
    /// - parameter bytes: A pointer to
the bytes to copy in to the data.
    /// - parameter count: The number of
bytes to copy.
    @available(swift, deprecated: 4,
message: "Use append(_:
UnsafeRawBufferPointer) instead")
    public mutating func append(_ bytes:
UnsafePointer<UInt8>, count: Int)

    /// Append bytes to the data.
    ///
    /// - parameter bytes: A pointer to
the bytes to copy in to the data.
    /// - parameter count: The number of
bytes to copy.
    public mutating func append(_ bytes:
UnsafeRawBufferPointer)

    /// Append data to the data.
```

```
    ///
    /// - parameter data: The data to
append to this data.
    public mutating func append(_ other:
DispatchData)

    /// Append a buffer of bytes to the
data.
    ///
    /// - parameter buffer: The buffer of
bytes to append. The size is calculated
from `SourceType` and `buffer.count`.
    public mutating func
append<SourceType>(_ buffer:
UnsafeBufferPointer<SourceType>)

    /// Copy the contents of the data to
a pointer.
    ///
    /// - parameter pointer: A pointer to
the buffer you wish to copy the bytes
into.
    /// - parameter count: The number of
bytes to copy.
    /// - warning: This method does not
verify that the contents at pointer have
enough space to hold `count` bytes.
    @available(swift, deprecated: 4,
message: "Use copyBytes(to:
UnsafeMutableRawBufferPointer, count:
Int) instead")
    public func copyBytes(to pointer:
UnsafeMutablePointer<UInt8>, count: Int)
```

```
    /// Copy the contents of the data to
a pointer.
    ///
    /// - parameter pointer: A pointer to
the buffer you wish to copy the bytes
into. The buffer must be large
    /// enough to hold `count` bytes.
    /// - parameter count: The number of
bytes to copy.
    public func copyBytes(to pointer:
UnsafeMutableRawBufferPointer, count:
Int)

    /// Copy a subset of the contents of
the data to a pointer.
    ///
    /// - parameter pointer: A pointer to
the buffer you wish to copy the bytes
into.
    /// - parameter range: The range in
the `Data` to copy.
    /// - warning: This method does not
verify that the contents at pointer have
enough space to hold the required number
of bytes.
    @available(swift, deprecated: 4,
message: "Use copyBytes(to:
UnsafeMutableRawBufferPointer, from:
Range<Index>) instead")
    public func copyBytes(to pointer:
UnsafeMutablePointer<UInt8>, from range:
Range<DispatchData.Index>)
```

```
    /// Copy a subset of the contents of
the data to a pointer.
    ///
    /// - parameter pointer: A pointer to
the buffer you wish to copy the bytes
into. The buffer must be large
    /// enough to hold `count` bytes.
    /// - parameter range: The range in
the `Data` to copy.
    public func copyBytes(to pointer:
UnsafeMutableRawBufferPointer, from
range: Range<DispatchData.Index>)

    /// Copy the contents of the data
into a buffer.
    ///
    /// This function copies the bytes in
`range` from the data into the buffer. If
the count of the `range` is greater than
`MemoryLayout<DestinationType>.stride *
buffer.count` then the first N bytes will
be copied into the buffer.
    /// - precondition: The range must be
within the bounds of the data. Otherwise
`fatalError` is called.
    /// - parameter buffer: A buffer to
copy the data into.
    /// - parameter range: A range in the
data to copy into the buffer. If the
range is empty, this function will return
0 without copying anything. If the range
is nil, as much data as will fit into
```

```
`buffer` is copied.
    /// - returns: Number of bytes copied
into the destination buffer.
    public func
copyBytes<DestinationType>(to buffer:
UnsafeMutableBufferPointer<DestinationTyp
e>, from range:
Range<DispatchData.Index>? = nil) -> Int

    /// Sets or returns the byte at the
specified index.
    public subscript(index:
DispatchData.Index) -> UInt8 { get }

    /// Accesses a contiguous subrange of
the collection's elements.
    ///
    /// The accessed slice uses the same
indices for the same elements as the
    /// original collection uses. Always
use the slice's `startIndex` property
    /// instead of assuming that its
indices start at a particular value.
    ///
    /// This example demonstrates getting
a slice of an array of strings, finding
    /// the index of one of the strings
in the slice, and then using that index
    /// in the original array.
    ///
    ///     let streets = ["Adams",
"Bryant", "Channing", "Douglas",
"Evarts"]
```

```
    ///     let streetsSlice =
streets[2 ..< streets.endIndex]
    ///     print(streetsSlice)
    ///     // Prints "["Channing",
"Douglas", "Evarts"]"
    ///
    ///     let index =
streetsSlice.firstIndex(of:
"Evarts")    // 4
    ///     print(streets[index!])
    ///     // Prints "Evarts"
    ///
    /// - Parameter bounds: A range of
the collection's indices. The bounds of
    ///   the range must be valid indices
of the collection.
    ///
    /// - Complexity: O(1)
    public subscript(bounds: Range<Int>)
-> Slice<DispatchData> { get }

    /// Return a new copy of the data in
a specified range.
    ///
    /// - parameter range: The range to
copy.
    public func subdata(in range:
Range<DispatchData.Index>) ->
DispatchData

    public func region(location: Int) ->
(data: DispatchData, offset: Int)
```

```
    /// The position of the first element
in a nonempty collection.
    ///
    /// If the collection is empty,
`startIndex` is equal to `endIndex`.
    public var startIndex:
DispatchData.Index { get }

    /// The collection's "past the end"
position---that is, the position one
    /// greater than the last valid
subscript argument.
    ///
    /// When you need a range that
includes the last element of a
collection, use
    /// the half-open range operator
(`..<`) with `endIndex`. The `..<`
operator
    /// creates a range that doesn't
include the upper bound, so it's always
    /// safe to use with `endIndex`. For
example:
    ///
    ///     let numbers = [10, 20, 30,
40, 50]
    ///     if let index =
numbers.firstIndex(of: 30) {
    ///         print(numbers[index ..<
numbers.endIndex])
    ///     }
    ///     // Prints "[30, 40, 50]"
    ///
```

```
    /// If the collection is empty,
`endIndex` is equal to `startIndex`.
    public var endIndex:
DispatchData.Index { get }

    /// Returns the position immediately
before the given index.
    ///
    /// - Parameter i: A valid index of
the collection. `i` must be greater than
    ///   `startIndex`.
    /// - Returns: The index value
immediately before `i`.
    public func index(before i:
DispatchData.Index) -> DispatchData.Index

    /// Returns the position immediately
after the given index.
    ///
    /// The successor of an index must be
well defined. For an index `i` into a
    /// collection `c`, calling
`c.index(after: i)` returns the same
index every
    /// time.
    ///
    /// - Parameter i: A valid index of
the collection. `i` must be less than
    ///   `endIndex`.
    /// - Returns: The index value
immediately after `i`.
    public func index(after i:
DispatchData.Index) -> DispatchData.Index
```

```
    /// An iterator over the contents of
the data.
    ///
    /// The iterator will increment byte-
by-byte.
    public func makeIterator() ->
DispatchData.Iterator

    /// A type representing the
sequence's elements.
    public typealias Element = UInt8

    /// A collection representing a
contiguous subrange of this collection's
    /// elements. The subsequence shares
indices with the original collection.
    ///
    /// The default subsequence type for
collections that don't define their own
    /// is `Slice`.
    public typealias SubSequence =
Slice<DispatchData>
}

public struct DispatchDataIterator :
IteratorProtocol, Sequence {

    /// The type of element traversed by
the iterator.
    public typealias Element = UInt8

    /// Advance to the next element and
```

```
return it, or `nil` if no next
    /// element exists.
    public mutating func next() ->
DispatchData.Element?

    /// A type that provides the
sequence's iteration interface and
    /// encapsulates its iteration state.
    public typealias Iterator =
DispatchDataIterator
}

/// dispatch_assert
@available(macOS 10.12, iOS 10.0, tvOS
10.0, watchOS 3.0, *)
public enum DispatchPredicate : Sendable
{

    case onQueue(DispatchQueue)

    case onQueueAsBarrier(DispatchQueue)

    case notOnQueue(DispatchQueue)
}

/// qos_class_t
public struct DispatchQoS : Equatable,
Sendable {

    public let qosClass:
DispatchQoS.QoSClass

    public let relativePriority: Int
```

```
    @available(macOS 10.10, iOS 8.0, *)
    public static let background:
DispatchQoS

    @available(macOS 10.10, iOS 8.0, *)
    public static let utility:
DispatchQoS

    @available(macOS 10.10, iOS 8.0, *)
    public static let `default`:
DispatchQoS

    @available(macOS 10.10, iOS 8.0, *)
    public static let userInitiated:
DispatchQoS

    @available(macOS 10.10, iOS 8.0, *)
    public static let userInteractive:
DispatchQoS

    public static let unspecified:
DispatchQoS

    public enum QoSClass : Sendable {

        @available(macOS 10.10, iOS 8.0,
*)
        case background

        @available(macOS 10.10, iOS 8.0,
*)
        case utility
```

```
        @available(macOS 10.10, iOS 8.0,
*)
        case `default`

        @available(macOS 10.10, iOS 8.0,
*)
        case userInitiated

        @available(macOS 10.10, iOS 8.0,
*)
        case userInteractive

        case unspecified

        @available(macOS 10.10, iOS 8.0,
*)
        public init?(rawValue:
qos_class_t)

        @available(macOS 10.10, iOS 8.0,
*)
        public var rawValue: qos_class_t
{ get }

        /// 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:
DispatchQoS.QoSClass, b:
DispatchQoS.QoSClass) -> 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 }
    }

    public init(qosClass:
DispatchQoS.QoSClass, relativePriority:
Int)

    /// 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:
DispatchQoS, b: DispatchQoS) -> Bool
}

extension DispatchQoS.QoSClass :
Equatable {
}

extension DispatchQoS.QoSClass : Hashable
{
}

final public class DispatchSpecificKey<T>
{

    public init()
}

extension DispatchSpecificKey : Sendable
where T : Sendable {
}

public struct DispatchTime : Comparable,
Sendable {
```

```
    public let rawValue: dispatch_time_t

    public static func now() ->
DispatchTime

    public static let distantFuture:
DispatchTime

    /// Creates a `DispatchTime` relative
to the system clock that
    /// ticks since boot.
    ///
    /// - Parameters:
    ///   - uptimeNanoseconds: The number
of nanoseconds since boot, excluding
    ///                        time the
system spent asleep
    /// - Returns: A new `DispatchTime`
    /// - Discussion: This clock is the
same as the value returned by
    ///
`mach_absolute_time` when converted into
nanoseconds.
    ///                On some platforms,
the nanosecond value is rounded up to a
    ///                multiple of the
Mach timebase, using the conversion
factors
    ///                returned by
`mach_timebase_info()`. The nanosecond
equivalent
    ///                of the rounded
```

```
result can be obtained by reading the
    ///                 `uptimeNanoseconds`
property.
    ///                Note that
`DispatchTime(uptimeNanoseconds: 0)` is
    ///                 equivalent to
`DispatchTime.now()`, that is, its value
    ///                 represents the
number of nanoseconds since boot
(excluding
    ///                 system sleep time),
not zero nanoseconds since boot.
    public init(uptimeNanoseconds:
UInt64)

    public var uptimeNanoseconds: UInt64
{ get }
}

extension DispatchTime {

    /// 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 < (a:
DispatchTime, b: DispatchTime) -> Bool

    /// 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:
DispatchTime, b: DispatchTime) -> Bool
}

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

    public func distance(to other:
DispatchTime) -> DispatchTimeInterval

    public func advanced(by n:
DispatchTimeInterval) -> DispatchTime
```

```
}

/// Represents a time interval that can
be used as an offset from a
`DispatchTime`
/// or `DispatchWallTime`.
///
/// For example:
///     let inOneSecond =
DispatchTime.now() +
DispatchTimeInterval.seconds(1)
///
/// If the requested time interval is
larger then the internal representation
/// permits, the result of adding it to a
`DispatchTime` or `DispatchWallTime`
/// is `DispatchTime.distantFuture` and
`DispatchWallTime.distantFuture`
/// respectively. Such time intervals
compare as equal:
///
///     let t1 =
DispatchTimeInterval.seconds(Int.max)
///     let t2 =
DispatchTimeInterval.milliseconds(Int.max
)
///     let result = t1 == t2   // true
public enum DispatchTimeInterval :
Equatable, Sendable {

    case seconds(Int)

    case milliseconds(Int)
```

```
    case microseconds(Int)

    case nanoseconds(Int)

    case never

    /// 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:
DispatchTimeInterval, rhs:
DispatchTimeInterval) -> Bool
}

///
@frozen public enum DispatchTimeoutResult
: Sendable {

    case success

    case timedOut

    /// 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:
DispatchTimeoutResult, b:
DispatchTimeoutResult) -> 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 }
}

extension DispatchTimeoutResult :
Equatable {
}

extension DispatchTimeoutResult :
Hashable {
```

```
}

extension DispatchTimeoutResult :
BitwiseCopyable {
}

public struct DispatchWallTime :
Comparable, Sendable {

    public let rawValue: dispatch_time_t

    public static func now() ->
DispatchWallTime

    public static let distantFuture:
DispatchWallTime

    public init(timespec: timespec)
}

extension DispatchWallTime {

    /// 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 < (a:
DispatchWallTime, b: DispatchWallTime) ->
Bool

    /// 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:
DispatchWallTime, b: DispatchWallTime) ->
Bool
}

@available(macOS 10.10, iOS 8.0, *)
public class DispatchWorkItem {

    public init(qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], block:
```

```
@escaping @convention(block) () -> Void)

    @available(macOS 11.3, iOS 14.5,
watchOS 7.4, tvOS 14.5, *)
    public init(flags:
DispatchWorkItemFlags = [], block:
@escaping @convention(block) () -> Void)

    public func perform()

    public func wait()

    public func wait(timeout:
DispatchTime) -> DispatchTimeoutResult

    public func wait(wallTimeout:
DispatchWallTime) ->
DispatchTimeoutResult

    public func notify(qos: DispatchQoS =
.unspecified, flags:
DispatchWorkItemFlags = [], queue:
DispatchQueue, execute: @escaping
@convention(block) () -> Void)

    public func notify(queue:
DispatchQueue, execute: DispatchWorkItem)

    public func cancel()

    public var isCancelled: Bool { get }
}
```

```
public struct DispatchWorkItemFlags :
OptionSet, RawRepresentable, Sendable {

    /// 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 let rawValue: UInt

    /// Creates a new option set from the
given raw value.
    ///
    /// This initializer always succeeds,
even if the value passed as `rawValue`
    /// exceeds the static properties
declared as part of the option set. This
```

```
    /// example creates an instance of
`ShippingOptions` with a raw value beyond
    /// the highest element, with a bit
mask that effectively contains all the
    /// declared static members.
    ///
    ///     let extraOptions =
ShippingOptions(rawValue: 255)
    ///
print(extraOptions.isStrictSuperset(of: .
all))
    ///     // Prints "true"
    ///
    /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
    ///   of `rawValue` potentially
represents an element of the option set,
    ///   though raw values may include
bits that are not defined as distinct
    ///   values of the `OptionSet` type.
    public init(rawValue: UInt)

    public static let barrier:
DispatchWorkItemFlags

    @available(macOS 10.10, iOS 8.0, *)
    public static let detached:
DispatchWorkItemFlags

    @available(macOS 10.10, iOS 8.0, *)
    public static let
assignCurrentContext:
```

```
DispatchWorkItemFlags

    @available(macOS 10.10, iOS 8.0, *)
    public static let noQoS:
DispatchWorkItemFlags

    @available(macOS 10.10, iOS 8.0, *)
    public static let inheritQoS:
DispatchWorkItemFlags

    @available(macOS 10.10, iOS 8.0, *)
    public static let enforceQoS:
DispatchWorkItemFlags

    /// The type of the elements of an
array literal.
    public typealias ArrayLiteralElement
= DispatchWorkItemFlags

    /// The element type of the option
set.
    ///
    /// To inherit all the default
implementations from the `OptionSet`
protocol,
    /// the `Element` type must be
`Self`, the default.
    public typealias Element =
DispatchWorkItemFlags

    /// 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 = UInt
}

@available(macOS 10.12, iOS 10.0, tvOS
10.0, watchOS 3.0, *)
public func
dispatchPrecondition(condition:
@autoclosure () -> DispatchPredicate)

extension DispatchQueue {

    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
```

```
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let concurrent:
DispatchQueue.Attributes

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        public static let
initiallyInactive:
DispatchQueue.Attributes

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchQueue.Attributes

        /// The element type of the
```

```
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchQueue.Attributes

        /// 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 =
UInt64
    }

    public enum GlobalQueuePriority :
Sendable {

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
```

```
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case high

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case `default`

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case low

        @available(macOS, deprecated:
10.10, message: "Use qos attributes
instead")
        @available(iOS, deprecated: 8.0,
```

```
message: "Use qos attributes instead")
        @available(tvOS, deprecated,
message: "Use qos attributes instead")
        @available(watchOS, deprecated,
message: "Use qos attributes instead")
        case background

        /// 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:
DispatchQueue.GlobalQueuePriority, b:
DispatchQueue.GlobalQueuePriority) ->
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 }
    }

    public enum AutoreleaseFrequency :
Sendable {

        case inherit

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        case workItem

        @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
        case never

        /// 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:
DispatchQueue.AutoreleaseFrequency, b:
DispatchQueue.AutoreleaseFrequency) ->
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 }
    }

    @preconcurrency public class func
concurrentPerform(iterations: Int,
execute work: @Sendable (Int) -> Void)

    public class var main: DispatchQueue
{ get }

    @available(macOS, deprecated: 10.10)
    @available(iOS, deprecated: 8.0)
```

```
    @available(tvOS, deprecated)
    @available(watchOS, deprecated)
    public class func global(priority:
DispatchQueue.GlobalQueuePriority) ->
DispatchQueue

    @available(macOS 10.10, iOS 8.0, *)
    public class func global(qos:
DispatchQoS.QoSClass = .default) ->
DispatchQueue

    @preconcurrency public class func
getSpecific<T>(key:
DispatchSpecificKey<T>) -> T? where T :
Sendable

    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes: DispatchQueue.Attributes =
[], autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .inherit, target: DispatchQueue? = nil)

    public var label: String { get }

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `async(execute:)`, however
    /// `sync(execute:)` will not return
until the work item has finished.
```

```
    ///
    /// Work items submitted to a queue
with `sync(execute:)` do not observe
certain
    /// queue attributes of that queue
when invoked (such as autorelease
frequency
    /// and QoS class).
    ///
    /// Calls to `sync(execute:)`
targeting the current queue will result
    /// in deadlock. Use of
`sync(execute:)` is also subject to the
same
    /// multi-party deadlock problems
that may result from the use of a mutex.
    /// Use of `async(execute:)` is
preferred.
    ///
    /// As an optimization,
`sync(execute:)` invokes the work item on
the thread which
    /// submitted it, except when the
queue is the main queue or
    /// a queue targetting it.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    @available(macOS 10.10, iOS 8.0, *)
```

```
    public func sync(execute workItem:
DispatchWorkItem)

    ///
    /// Submits a work item for
asynchronous execution on a dispatch
queue.
    ///
    /// `async(execute:)` is the
fundamental mechanism for submitting
    /// work items to a dispatch queue.
    ///
    /// Calls to `async(execute:)` always
return immediately after the work item
has
    /// been submitted, and never wait
for the work item to be invoked.
    ///
    /// The target queue determines
whether the work item will be invoked
serially or
    /// concurrently with respect to
other work items submitted to that same
queue.
    /// Serial queues are processed
concurrently with respect to each other.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
```

```
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func async(execute workItem:
DispatchWorkItem)

    ///
    /// Submits a work item for
synchronous execution on a dispatch
queue.
    ///
    /// Submits a work item to a dispatch
queue like `async(execute:)`, however
    /// `asyncAndWait(execute:)` will not
return until the work item has finished.
    ///
    /// `asyncAndWait(excute:)` is
subject to deadlock under the same
conditions
    /// as `sync(execute:)`.
`asyncAndWait(execute:)` differs from
    /// `sync(execute:)` in the following
ways:
    ///
    ///   * Work items submitted to a
queue with `asyncAndWait` observe all
    ///     queue attributes of that
queue when invoked (including autorelease
    ///     frequency or DispatchQoS
class).
    ///
    ///   * Work items submitted to a
queue with `asyncAndWait` are not
    ///     guaranteed to run on the
```

```
calling thread.
    ///
    ///     If the queue the work is
submitted to already has a thread
    ///     servicing it, the servicing
thread will execute the work item
    ///     submitted via `asyncAndWait`.
If the queue the work is submitted
    ///     to does not have any threads
servicing it, the calling thread
    ///     will execute the work item.
As an exception, if the queue the work
    ///     is submitted to doesn't
target a global concurrent queue (for
example
    ///     because it targets the main
queue or a custom priority workloop),
    ///     then the work item will never
be invoked by the thread calling
    ///     `asyncAndWait(execute:)`.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso: `sync(execute:)`
    ///
    @available(macOS 10.14, iOS 12.0, *)
    public func asyncAndWait(execute
workItem: DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue and associates it with the given
```

```
    /// dispatch group. The dispatch
group may be used to wait for the
completion
    /// of the work items it references.
    ///
    /// - parameter group: the dispatch
group to associate with the submitted
block.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func async(group:
DispatchGroup, execute workItem:
DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue and optionally associates it with a
    /// dispatch group. The dispatch
group may be used to wait for the
completion
    /// of the work items it references.
    ///
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter group: the dispatch
```

```
group to associate with the submitted
    /// work item. If this is `nil`, the
work item is not associated with a group.
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func asyncUnsafe(group:
DispatchGroup? = nil, qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)

    ///
    /// Submits a work item to a dispatch
queue and optionally associates it with a
    /// dispatch group. The dispatch
group may be used to wait for the
completion
```

```
    /// of the work items it references.
    ///
    /// This method enforces the work
item to be sendable.
    ///
    /// - parameter group: the dispatch
group to associate with the submitted
    /// work item. If this is `nil`, the
work item is not associated with a group.
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    @preconcurrency public func
async(group: DispatchGroup? = nil, qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    /// Submits a work item for
```

```
synchronous execution on a dispatch
queue.
    ///
    /// Submits a work item to a dispatch
queue like `asyncAndWait(execute:)`,
    /// and returns the value, of type
`T`, returned by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    @available(macOS 10.14, iOS 12.0,
tvOS 12.0, watchOS 5.0, *)
    public func asyncAndWait<T>(execute
work: () throws -> T) rethrows -> T

    /// Submits a work item for
synchronous execution on a dispatch
queue.
    ///
    /// Submits a work item to a dispatch
queue like `asyncAndWait(execute:)`,
    /// and returns the value, of type
`T`, returned by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
```

```
    /// - SeeAlso:
`asyncAndWait(execute:)`
    ///
    @available(macOS 10.14, iOS 12.0,
tvOS 12.0, watchOS 5.0, *)
    public func asyncAndWait<T>(flags:
DispatchWorkItemFlags, execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `sync(execute:)`, and returns
    /// the value, of type `T`, returned
by that work item.
    ///
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso: `sync(execute:)`
    ///
    public func sync<T>(execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a block for synchronous
execution on this queue.
    ///
    /// Submits a work item to a dispatch
queue like `sync(execute:)`, and returns
```

```
    /// the value, of type `T`, returned
by that work item.
    ///
    /// - parameter flags: flags that
control the execution environment of the
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - returns the value returned by
the work item.
    /// - SeeAlso: `sync(execute:)`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    ///
    public func sync<T>(flags:
DispatchWorkItemFlags, execute work: ()
throws -> T) rethrows -> T

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchTime`.
```

```
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(deadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func
asyncAfterUnsafe(deadline: DispatchTime,
qos: DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This method enforces the work
item to be sendable.
    ///
```

```
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(deadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchTime`
    ///
    @preconcurrency public func
asyncAfter(deadline: DispatchTime, qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
```

```
    /// This function does not enforce
sendability requirement on work item.
    /// If non-sendable objects are
captured by the closure to this method,
    /// clients are responsible for
manually verifying their correctness.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(wallDeadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchWallTime`
    ///
    @available(macOS 14.0, iOS 17.0, tvOS
17.0, watchOS 10.0, *)
    public func
asyncAfterUnsafe(wallDeadline:
DispatchWallTime, qos: DispatchQoS
```

```
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @convention(block) () -> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// This method enforces the work
item to be sendable.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
    /// - parameter qos: the QoS at which
the work item should be executed.
    /// Defaults to
`DispatchQoS.unspecified`.
    /// - parameter flags: flags that
control the execution environment of the
    /// work item.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso: `async(execute:)`
    /// - SeeAlso:
`asyncAfter(wallDeadline:execute:)`
    /// - SeeAlso: `DispatchQoS`
    /// - SeeAlso:
`DispatchWorkItemFlags`
    /// - SeeAlso: `DispatchWallTime`
    ///
```

```
    @preconcurrency public func
asyncAfter(wallDeadline:
DispatchWallTime, qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], execute work:
@escaping @Sendable @convention(block) ()
-> Void)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// - parameter: deadline the time
after which the work item should be
executed,
    /// given as a `DispatchTime`.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso:
`asyncAfter(deadline:qos:flags:execute:)`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func asyncAfter(deadline:
DispatchTime, execute: DispatchWorkItem)

    ///
    /// Submits a work item to a dispatch
queue for asynchronous execution after
    /// a specified time.
    ///
    /// - parameter: deadline the time
```

```
after which the work item should be
executed,
    /// given as a `DispatchWallTime`.
    /// - parameter execute: The work
item to be invoked on the queue.
    /// - SeeAlso:
`asyncAfter(wallDeadline:qos:flags:execut
e:)`
    /// - SeeAlso: `DispatchTime`
    ///
    @available(macOS 10.10, iOS 8.0, *)
    public func asyncAfter(wallDeadline:
DispatchWallTime, execute:
DispatchWorkItem)

    @available(macOS 10.10, iOS 8.0, *)
    public var qos: DispatchQoS { get }

    @preconcurrency public func
getSpecific<T>(key:
DispatchSpecificKey<T>) -> T? where T :
Sendable

    @preconcurrency public func
setSpecific<T>(key:
DispatchSpecificKey<T>, value: T?) where
T : Sendable
}

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

```
    public func enqueue(_ job: consuming
ExecutorJob)

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

    /// 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.
    public func checkIsolated()
}

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

    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
```

```
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let
initiallyInactive:
DispatchSerialQueue.Attributes

        /// The type of the elements of
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
```

```
ArrayLiteralElement =
DispatchSerialQueue.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchSerialQueue.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }
```

```
    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes:
DispatchSerialQueue.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, target: DispatchQueue? =
nil)
}

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

    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
```

```
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        public static let
initiallyInactive:
DispatchConcurrentQueue.Attributes

        /// The type of the elements of
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
ArrayLiteralElement =
DispatchConcurrentQueue.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
```

```
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchConcurrentQueue.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }

    public convenience init(label:
String, qos: DispatchQoS = .unspecified,
attributes:
DispatchConcurrentQueue.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, target: DispatchQueue? =
nil)
}

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

```
17.0, watchOS 10.0, *)
extension DispatchWorkloop {

    ///
    /// Workloop attributes to customize
at creation time.
    ///
    /// This is an empty set today; but,
support for additional attributes could
be
    /// added in the future.
    ///
    /// The reason this exists is it has
SPI only attribute to create an initially
    /// inactive workloop. (See
DispatchWorkloop.Attributes.initiallyInac
tive)
    /// The goal is to future proof our
internal clients that create an inactive
workloop,
    /// set properties such as scheduler
priority or QoS Class that are SPI only,
followed by
    /// activation of the workloop.
    public struct Attributes : OptionSet,
Sendable {

        /// 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 let rawValue: UInt64

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
```

```
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt64)

        /// The type of the elements of
an array literal.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias
ArrayLiteralElement =
DispatchWorkloop.Attributes

        /// The element type of the
option set.
        ///
        /// To inherit all the default
```

```
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias Element =
DispatchWorkloop.Attributes

        /// 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.
        @available(iOS 17.0, tvOS 17.0,
watchOS 10.0, macOS 14.0, *)
        public typealias RawValue =
UInt64
    }

    ///
    /// Initializes an instance of
DispatchWorkloop
    ///
    /// - parameter label: A string label
to attach to the workloop.
```

```
    /// - parameter attributes:
Additional workloop attributes to
customize.
    ///   (See
DispatchWorkloop.Attributes).
    /// - parameter autoreleaseFrequency:
Autorelease frequency to assign to the
workloop.
    ///   See
DispatchQueue.AutoreleaseFrequency.
Defaults to
AutoreleaseFrequency.workItem.
    /// - parameter osWorkgroup: OS
Workgroup to assign to the workloop.
    public convenience init(label:
String, attributes:
DispatchWorkloop.Attributes = [],
autoreleaseFrequency:
DispatchQueue.AutoreleaseFrequency
= .workItem, osWorkgroup: WorkGroup? =
nil)
}

extension DispatchSourceProtocol {

    public typealias
DispatchSourceHandler =
@convention(block) () -> Void

    public func setEventHandler(qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)
```

```
    @available(macOS 10.10, iOS 8.0, *)
    public func setEventHandler(handler:
DispatchWorkItem)

    public func setCancelHandler(qos:
DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)

    @available(macOS 10.10, iOS 8.0, *)
    public func setCancelHandler(handler:
DispatchWorkItem)

    public func
setRegistrationHandler(qos: DispatchQoS =
.unspecified, flags:
DispatchWorkItemFlags = [], handler:
Self.DispatchSourceHandler?)

    @available(macOS 10.10, iOS 8.0, *)
    public func
setRegistrationHandler(handler:
DispatchWorkItem)

    @available(macOS 10.12, iOS 10.0,
tvOS 10.0, watchOS 3.0, *)
    public func activate()

    public func cancel()

    public func resume()
```

```
    public func suspend()

    public var handle: UInt { get }

    public var mask: UInt { get }

    public var data: UInt { get }

    public var isCancelled: Bool { get }
}

extension DispatchSource {

    public struct MachSendEvent :
OptionSet, RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
```

```
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let dead:
DispatchSource.MachSendEvent

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.MachSendEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.MachSendEvent

        /// 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 = UInt
    }

    public struct MemoryPressureEvent :
OptionSet, RawRepresentable,
CustomStringConvertible {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
```

```
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let normal:
DispatchSource.MemoryPressureEvent

        public static let warning:
DispatchSource.MemoryPressureEvent

        public static let critical:
DispatchSource.MemoryPressureEvent

        public static let all:
DispatchSource.MemoryPressureEvent

        /// 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.
        public var description: String {
get }

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.MemoryPressureEvent
```

```
        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.MemoryPressureEvent

        /// 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 = UInt
    }

    public struct ProcessEvent :
OptionSet, RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
```

```
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let exit:
DispatchSource.ProcessEvent

        public static let fork:
DispatchSource.ProcessEvent

        public static let exec:
DispatchSource.ProcessEvent
```

```
        public static let signal:
DispatchSource.ProcessEvent

        public static let all:
DispatchSource.ProcessEvent

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.ProcessEvent

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.ProcessEvent

        /// 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 = UInt
    }

    public struct TimerFlags : OptionSet,
RawRepresentable {

        /// 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 let rawValue: UInt
```

```
        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
```

```
        public init(rawValue: UInt)

        public static let strict:
DispatchSource.TimerFlags

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.TimerFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.TimerFlags

        /// 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 = UInt
    }

    public struct FileSystemEvent :
OptionSet, RawRepresentable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
```

```
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)
```

```
        public static let delete:
DispatchSource.FileSystemEvent

        public static let write:
DispatchSource.FileSystemEvent

        public static let extend:
DispatchSource.FileSystemEvent

        public static let attrib:
DispatchSource.FileSystemEvent

        public static let link:
DispatchSource.FileSystemEvent

        public static let rename:
DispatchSource.FileSystemEvent

        public static let revoke:
DispatchSource.FileSystemEvent

        public static let funlock:
DispatchSource.FileSystemEvent

        public static let all:
DispatchSource.FileSystemEvent

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchSource.FileSystemEvent
```

```
        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchSource.FileSystemEvent

        /// 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 = UInt
    }

    public class func
makeMachSendSource(port: mach_port_t,
eventMask: DispatchSource.MachSendEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceMachSend
```

```
    public class func
makeMachReceiveSource(port: mach_port_t,
queue: DispatchQueue? = nil) -> any
DispatchSourceMachReceive

    public class func
makeMemoryPressureSource(eventMask:
DispatchSource.MemoryPressureEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceMemoryPressure

    public class func
makeProcessSource(identifier: pid_t,
eventMask: DispatchSource.ProcessEvent,
queue: DispatchQueue? = nil) -> any
DispatchSourceProcess

    public class func
makeReadSource(fileDescriptor: Int32,
queue: DispatchQueue? = nil) -> any
DispatchSourceRead

    public class func
makeSignalSource(signal: Int32, queue:
DispatchQueue? = nil) -> any
DispatchSourceSignal

    public class func
makeTimerSource(flags:
DispatchSource.TimerFlags = [], queue:
DispatchQueue? = nil) -> any
DispatchSourceTimer
```

```
    public class func
makeUserDataAddSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataAdd

    public class func
makeUserDataOrSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataOr

    public class func
makeUserDataReplaceSource(queue:
DispatchQueue? = nil) -> any
DispatchSourceUserDataReplace

    public class func
makeFileSystemObjectSource(fileDescriptor
: Int32, eventMask:
DispatchSource.FileSystemEvent, queue:
DispatchQueue? = nil) -> any
DispatchSourceFileSystemObject

    public class func
makeWriteSource(fileDescriptor: Int32,
queue: DispatchQueue? = nil) -> any
DispatchSourceWrite
}

extension DispatchSourceMachSend {

    public var handle: mach_port_t {
get }
```

```
    public var data:
DispatchSource.MachSendEvent { get }

    public var mask:
DispatchSource.MachSendEvent { get }
}

extension DispatchSourceMachReceive {

    public var handle: mach_port_t {
get }
}

extension DispatchSourceMemoryPressure {

    public var data:
DispatchSource.MemoryPressureEvent {
get }

    public var mask:
DispatchSource.MemoryPressureEvent {
get }
}

extension DispatchSourceProcess {

    public var handle: pid_t { get }

    public var data:
DispatchSource.ProcessEvent { get }

    public var mask:
DispatchSource.ProcessEvent { get }
```

```
}

extension DispatchSourceTimer {

    ///
    /// Sets the deadline and leeway for a timer event that fires once.
    ///
    /// Once this function returns, any pending source data accumulated for the previous timer values
    /// has been cleared and the next timer event will occur at `deadline`.
    ///
    /// Delivery of the timer event may be delayed by the system in order to improve power consumption
    /// and system performance. The upper limit to the allowable delay may be configured with the `leeway`
    /// argument; the lower limit is under the control of the system.
    ///
    /// The lower limit to the allowable delay may vary with process state such as visibility of the
    /// application UI. If the timer source was created with flags `TimerFlags.strict`, the system
    /// will make a best effort to strictly observe the provided `leeway` value, even if it is smaller
    /// than the current lower limit.
```

```
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    /// - note: Delivery of the timer
event does not cancel the timer source.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(deadline:repeating:leeway:)")
    public func scheduleOneshot(deadline:
DispatchTime, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline and leeway for
a timer event that fires once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
```

```
previous timer values
    /// has been cleared and the next
timer event will occur at `wallDeadline`.
    ///
    /// Delivery of the timer event may
be delayed by the system in order to
improve power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    /// - note: Delivery of the timer
event does not cancel the timer source.
    ///
```

```
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
    public func
scheduleOneshot(wallDeadline:
DispatchWallTime, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`interval` units of
    /// time thereafter until the timer
source is canceled.
    ///
```

```
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N * interval`,
the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
```

```
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter interval: the
interval for the timer.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(deadline:repeating:leeway:)")
    public func
scheduleRepeating(deadline: DispatchTime,
interval: DispatchTimeInterval, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
```

```
`interval` seconds
    /// thereafter until the timer source
is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N * interval`,
the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
```

```
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter interval: the
interval for the timer in seconds.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(deadline:repeating:leeway:)")
    public func
scheduleRepeating(deadline: DispatchTime,
interval: Double, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
```

```
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `interval` units of
    /// time thereafter until the timer
source is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
interval`, the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
```

```
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter interval: the
interval for the timer.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
    public func
scheduleRepeating(wallDeadline:
DispatchWallTime, interval:
DispatchTimeInterval, leeway:
DispatchTimeInterval = .nanoseconds(0))
```

```
    ///
    /// Sets the deadline, interval and
leeway for a timer event that fires at
least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `interval` seconds
    /// thereafter until the timer source
is canceled.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
interval`, the upper
    /// limit is the smaller of `leeway`
and `interval/2`.
    ///
    /// The lower limit to the allowable
```

```
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter interval: the
interval for the timer in seconds.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift, deprecated: 4,
renamed:
"schedule(wallDeadline:repeating:leeway:)
")
```

```
    public func
scheduleRepeating(wallDeadline:
DispatchWallTime, interval: Double,
leeway: DispatchTimeInterval
= .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`repeating` units of
    /// time thereafter until the timer
source is canceled. If the value of
`repeating` is `.never`,
    /// or is defaulted, the timer fires
only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
```

```
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the first timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter repeating: the repeat
```

```
interval for the timer, or `.never` if
the timer should fire
    ///     only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(deadline:
DispatchTime, repeating interval:
DispatchTimeInterval = .never, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `deadline` and every
`repeating` seconds
    /// thereafter until the timer source
is canceled. If the value of `repeating`
is `.infinity`,
    /// the timer fires only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
```

```
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`deadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `deadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter deadline: the time at
which the timer event will be delivered,
subject to the
```

```
    ///     leeway and other
considerations described above. The
deadline is based on Mach absolute
    ///     time.
    /// - parameter repeating: the repeat
interval for the timer in seconds, or
`.infinity` if the timer
    ///     should fire only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(deadline:
DispatchTime, repeating interval: Double,
leeway: DispatchTimeInterval
= .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
every `repeating` units of
    /// time thereafter until the timer
source is canceled. If the value of
`repeating` is `.never`,
    /// or is defaulted, the timer fires
only once.
    ///
```

```
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption and
    /// system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
```

```
if the timer source has already been
canceled.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter repeating: the repeat
interval for the timer, or `.never` if
the timer should fire
    ///     only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(wallDeadline:
DispatchWallTime, repeating interval:
DispatchTimeInterval = .never, leeway:
DispatchTimeInterval = .nanoseconds(0))

    ///
    /// Sets the deadline, repeat
interval and leeway for a timer event
that fires at least once.
    ///
    /// Once this function returns, any
pending source data accumulated for the
previous timer values
    /// has been cleared. The next timer
event will occur at `wallDeadline` and
```

```
every `repeating` seconds
    /// thereafter until the timer source
is canceled. If the value of `repeating`
is `.infinity`,
    /// the timer fires only once.
    ///
    /// Delivery of a timer event may be
delayed by the system in order to improve
power consumption
    /// and system performance. The upper
limit to the allowable delay may be
configured with the `leeway`
    /// argument; the lower limit is
under the control of the system.
    ///
    /// For the initial timer fire at
`wallDeadline`, the upper limit to the
allowable delay is set to
    /// `leeway`. For the subsequent
timer fires at `wallDeadline + N *
repeating`, the upper
    /// limit is the smaller of `leeway`
and `repeating/2`.
    ///
    /// The lower limit to the allowable
delay may vary with process state such as
visibility of the
    /// application UI. If the timer
source was created with flags
`TimerFlags.strict`, the system
    /// will make a best effort to
strictly observe the provided `leeway`
value, even if it is smaller
```

```
    /// than the current lower limit.
Note that a minimal amount of delay is to
be expected even if
    /// this flag is specified.
    ///
    /// Calling this method has no effect
if the timer source has already been
canceled.
    ///
    /// - parameter wallDeadline: the
time at which the timer event will be
delivered, subject to the
    ///     leeway and other
considerations described above. The
deadline is based on
    ///     `gettimeofday(3)`.
    /// - parameter repeating: the repeat
interval for the timer in secondss, or
`.infinity` if the timer
    /// should fire only once.
    /// - parameter leeway: the leeway
for the timer.
    ///
    @available(swift 4)
    public func schedule(wallDeadline:
DispatchWallTime, repeating interval:
Double, leeway: DispatchTimeInterval
= .nanoseconds(0))
}

extension DispatchSourceFileSystemObject
{
```

```
    public var handle: Int32 { get }

    public var data:
DispatchSource.FileSystemEvent { get }

    public var mask:
DispatchSource.FileSystemEvent { get }
}

extension DispatchSourceUserDataAdd {

    /// @function add
    ///
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_ADD
    /// and submits its event handler
block to its target queue.
    ///
    /// @param data
    /// The value to add to the current
pending data. A value of zero has no
effect
    /// and will not result in the
submission of the event handler block.
    public func add(data: UInt)
}

extension DispatchSourceUserDataOr {

    /// @function or
    ///
```

```
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_OR and
    /// submits its event handler block
to its target queue.
    ///
    /// @param data
    /// The value to OR into the current
pending data. A value of zero has no
effect
    /// and will not result in the
submission of the event handler block.
    public func or(data: UInt)
}

extension DispatchSourceUserDataReplace {

    /// @function replace
    ///
    /// @abstract
    /// Merges data into a dispatch
source of type
DISPATCH_SOURCE_TYPE_DATA_REPLACE
    /// and submits its event handler
block to its target queue.
    ///
    /// @param data
    /// The value that will replace the
current pending data. A value of zero
will be stored
    /// but will not result in the
submission of the event handler block.
```

```
    public func replace(data: UInt)
}

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

    /// The scheduler time type used by
the dispatch queue.
    public struct SchedulerTimeType :
Strideable, Codable, Hashable, Sendable {

        /// The dispatch time represented
by this type.
        public var dispatchTime:
DispatchTime

        /// Creates a dispatch queue time
type instance.
        ///
        /// - Parameter time: The
dispatch time to represent.
        public init(_ time: DispatchTime)

        /// 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

        /// 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

        /// Returns the distance to
another dispatch queue time.
        ///
        /// - Parameter other: Another
dispatch queue time.
        /// - Returns: The time interval
between this time and the provided time.
        public func distance(to other:
DispatchQueue.SchedulerTimeType) ->
DispatchQueue.SchedulerTimeType.Stride

        /// Returns a dispatch queue
```

```
scheduler time calculated by advancing
this instance’s time by the given
interval.
        ///
        /// - Parameter n: A time
interval to advance.
        /// - Returns: A dispatch queue
time advanced by the given interval from
this instance’s time.
        public func advanced(by n:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType

        /// 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 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:
DispatchQueue.SchedulerTimeType, rhs:
DispatchQueue.SchedulerTimeType) -> Bool

        /// A type that represents the
```

```
distance between two values.
        public struct Stride :
SchedulerTimeIntervalConvertible,
Comparable, SignedNumeric,
ExpressibleByFloatLiteral, Hashable,
Codable {

            /// If created via floating
point literal, the value is converted to
nanoseconds via multiplication.
            public typealias
FloatLiteralType = Double

            /// Nanoseconds, same as
DispatchTimeInterval.
            public typealias
IntegerLiteralType = Int

            /// A type that can represent
the absolute value of any possible value
of the
            /// conforming type.
            public typealias Magnitude =
Int

            /// The value of this time
interval in nanoseconds.
            public var magnitude: Int

            /// A `DispatchTimeInterval`
created with the value of this type in
nanoseconds.
            public var timeInterval:
```

```
DispatchTimeInterval { get }

            /// Creates a dispatch queue
time interval from the given dispatch
time interval.
            ///
            /// - Parameter timeInterval:
A dispatch time interval.
            public init(_ timeInterval:
DispatchTimeInterval)

            /// Creates a dispatch queue
time interval from a floating-point
seconds value.
            ///
            /// - Parameter value: The
number of seconds, as a `Double`.
            public init(floatLiteral
value: Double)

            /// Creates a dispatch queue
time interval from an integer seconds
value.
            ///
            /// - Parameter value: The
number of seconds, as an `Int`.
            public init(integerLiteral
value: Int)

            /// Creates a dispatch queue
time interval from a binary integer type
representing a number of seconds.
            ///
```

```
            /// If `source` cannot be
exactly represented, the resulting time
interval is `nil`.
            /// - Parameter source: A
binary integer representing a time
interval.
            public init?<T>(exactly
source: T) where T : BinaryInteger

            /// 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:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> Bool
```

```
            /// Multiplies two values and
produces their product.
            ///
            /// The multiplication
operator (`*`) calculates the product of
its two
            /// arguments. For example:
            ///
            ///     2 * 3
// 6
            ///     100 * 21
// 2100
            ///     -10 * 15
// -150
            ///     3.5 * 2.25
// 7.875
            ///
            /// You cannot use `*` with
arguments of different types. To multiply
values
            /// of different types,
convert one of the values to the other
value's type.
            ///
            ///     let x: Int8 = 21
            ///     let y: Int = 1000000
            ///     Int(x) * y
// 21000000
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
```

```
            ///   - rhs: The second value
to multiply.
            public static func * (lhs:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Adds two values and
produces their sum.
            ///
            /// The addition operator
(`+`) calculates the sum of its two
arguments. For
            /// example:
            ///
            ///     1 + 2
// 3
            ///     -10 + 15
// 5
            ///     -15 + -5
// -20
            ///     21.5 + 3.25
// 24.75
            ///
            /// You cannot use `+` with
arguments of different types. To add
values of
            /// different types, convert
one of the values to the other value's
type.
            ///
            ///     let x: Int8 = 21
```

```
            ///     let y: Int = 1000000
            ///     Int(x) + y
// 1000021
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func + (lhs:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Subtracts one value from
another and produces their difference.
            ///
            /// The subtraction operator
(`-`) calculates the difference of its
two
            /// arguments. For example:
            ///
            ///     8 - 3
// 5
            ///     -10 - 5
// -15
            ///     100 - -5
// 105
            ///     10.5 - 100.0
// -89.5
            ///
            /// You cannot use `-` with
```

```
arguments of different types. To subtract
values
            /// of different types,
convert one of the values to the other
value's type.
            ///
            ///     let x: UInt8 = 21
            ///     let y: UInt = 1000000
            ///     y - UInt(x)
// 999979
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func - (lhs:
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
-> DispatchQueue.SchedulerTimeType.Stride

            /// Subtracts the second
value from the first and stores the
difference in the
            /// left-hand-side variable.
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func -= (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
```

```
rhs:
DispatchQueue.SchedulerTimeType.Stride)

            /// Multiplies two values and
stores the result in the left-hand-side
            /// variable.
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
            ///   - rhs: The second value
to multiply.
            public static func *= (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)

            /// Adds two values and
stores the result in the left-hand-side
variable.
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func += (lhs:
inout
DispatchQueue.SchedulerTimeType.Stride,
rhs:
DispatchQueue.SchedulerTimeType.Stride)
```

```
            /// Converts the specified
number of seconds, as a floating-point
value, into an instance of this scheduler
time type.
            public static func seconds(_
s: Double) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of seconds into an instance of
this scheduler time type.
            public static func seconds(_
s: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of milliseconds into an instance
of this scheduler time type.
            public static func
milliseconds(_ ms: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of microseconds into an instance
of this scheduler time type.
            public static func
microseconds(_ us: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// Converts the specified
number of nanoseconds into an instance of
this scheduler time type.
            public static func
```

```
nanoseconds(_ ns: Int) ->
DispatchQueue.SchedulerTimeType.Stride

            /// 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 == (a:
DispatchQueue.SchedulerTimeType.Stride,
b:
DispatchQueue.SchedulerTimeType.Stride)
-> Bool

            /// 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

            /// 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 }

            /// 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
        }

        /// 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 }
    }

    /// Options that affect the operation
of the dispatch queue scheduler.
```

```
    public struct SchedulerOptions :
Sendable {

        /// The dispatch queue quality of
service.
        public var qos: DispatchQoS

        /// The dispatch queue work item
flags.
        public var flags:
DispatchWorkItemFlags

        /// The dispatch group, if any,
that should be used for performing
actions.
        public var group: DispatchGroup?

        public init(qos: DispatchQoS
= .unspecified, flags:
DispatchWorkItemFlags = [], group:
DispatchGroup? = nil)
    }

    /// The minimum tolerance allowed by
the scheduler.
    public var minimumTolerance:
DispatchQueue.SchedulerTimeType.Stride {
get }

    /// This scheduler's definition of
the current moment in time.
    public var now:
DispatchQueue.SchedulerTimeType { get }
```

```
    /// Performs the action at the next
possible opportunity.
    public func schedule(options:
DispatchQueue.SchedulerOptions?, _
action: @escaping () -> Void)

    /// Performs the action at some time
after the specified date.
    public func schedule(after date:
DispatchQueue.SchedulerTimeType,
tolerance:
DispatchQueue.SchedulerTimeType.Stride,
options: DispatchQueue.SchedulerOptions?,
_ action: @escaping () -> Void)

    /// Performs the action at some time
after the specified date, at the
specified frequency, optionally taking
into account tolerance if possible.
    public func schedule(after date:
DispatchQueue.SchedulerTimeType,
interval:
DispatchQueue.SchedulerTimeType.Stride,
tolerance:
DispatchQueue.SchedulerTimeType.Stride,
options: DispatchQueue.SchedulerOptions?,
_ action: @escaping () -> Void) -> any
Cancellable
}

extension DispatchIO {
```

```
    public enum StreamType : UInt,
Sendable {

        case stream

        case random

        /// 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: UInt)
```

```
        /// 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 = UInt

        /// 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: UInt { get }
    }

    public struct CloseFlags : OptionSet,
RawRepresentable, Sendable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from the given raw value.
        ///
        /// This initializer always succeeds, even if the value passed as `rawValue`
        /// exceeds the static properties declared as part of the option set. This
        /// example creates an instance of `ShippingOptions` with a raw value beyond
        /// the highest element, with a bit mask that effectively contains all the
        /// declared static members.
        ///
        ///     let extraOptions = ShippingOptions(rawValue: 255)
        ///
        /// print(extraOptions.isStrictSuperset(of: .all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw value of the option set to create. Each bit
        ///   of `rawValue` potentially represents an element of the option set,
        ///   though raw values may include bits that are not defined as
```

```
distinct
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public static let stop:
DispatchIO.CloseFlags

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchIO.CloseFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchIO.CloseFlags

        /// 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 = UInt
    }

    public struct IntervalFlags :
OptionSet, RawRepresentable, Sendable {

        /// 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 let rawValue: UInt

        /// Creates a new option set from
the given raw value.
        ///
        /// This initializer always
succeeds, even if the value passed as
`rawValue`
        /// exceeds the static properties
declared as part of the option set. This
        /// example creates an instance
of `ShippingOptions` with a raw value
beyond
        /// the highest element, with a
bit mask that effectively contains all
the
        /// declared static members.
        ///
        ///     let extraOptions =
ShippingOptions(rawValue: 255)
        ///
print(extraOptions.isStrictSuperset(of: .
all))
        ///     // Prints "true"
        ///
        /// - Parameter rawValue: The raw
value of the option set to create. Each
bit
        ///   of `rawValue` potentially
represents an element of the option set,
        ///   though raw values may
include bits that are not defined as
distinct
```

```
        ///   values of the `OptionSet`
type.
        public init(rawValue: UInt)

        public init(nilLiteral: ())

        public static let strictInterval:
DispatchIO.IntervalFlags

        /// The type of the elements of
an array literal.
        public typealias
ArrayLiteralElement =
DispatchIO.IntervalFlags

        /// The element type of the
option set.
        ///
        /// To inherit all the default
implementations from the `OptionSet`
protocol,
        /// the `Element` type must be
`Self`, the default.
        public typealias Element =
DispatchIO.IntervalFlags

        /// 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 = UInt
    }

    public class func
read(fromFileDescriptor: Int32,
maxLength: Int, runningHandlerOn queue:
DispatchQueue, handler: @escaping (_
data: DispatchData, _ error: Int32) ->
Void)

    public class func
write(toFileDescriptor: Int32, data:
DispatchData, runningHandlerOn queue:
DispatchQueue, handler: @escaping (_
data: DispatchData?, _ error: Int32) ->
Void)

    public convenience init(type:
DispatchIO.StreamType, fileDescriptor:
Int32, queue: DispatchQueue,
cleanupHandler: @escaping (_ error:
Int32) -> Void)

    @available(swift 4)
    public convenience init?(type:
DispatchIO.StreamType, path:
UnsafePointer<Int8>, oflag: Int32, mode:
mode_t, queue: DispatchQueue,
```

```
cleanupHandler: @escaping (_ error:
Int32) -> Void)

    public convenience init(type:
DispatchIO.StreamType, io: DispatchIO,
queue: DispatchQueue, cleanupHandler:
@escaping (_ error: Int32) -> Void)

    public func read(offset: off_t,
length: Int, queue: DispatchQueue,
ioHandler: @escaping (_ done: Bool, _
data: DispatchData?, _ error: Int32) ->
Void)

    public func write(offset: off_t,
data: DispatchData, queue: DispatchQueue,
ioHandler: @escaping (_ done: Bool, _
data: DispatchData?, _ error: Int32) ->
Void)

    public func setInterval(interval:
DispatchTimeInterval, flags:
DispatchIO.IntervalFlags = [])

    public func close(flags:
DispatchIO.CloseFlags = [])
}

/// dispatch_group
extension DispatchGroup {

    public func notify(qos: DispatchQoS =
.unspecified, flags:
```

```
DispatchWorkItemFlags = [], queue:
DispatchQueue, execute work: @escaping
@convention(block) () -> Void)

    @available(macOS 10.10, iOS 8.0, *)
    public func notify(queue:
DispatchQueue, work: DispatchWorkItem)

    public func wait()

    public func wait(timeout:
DispatchTime) -> DispatchTimeoutResult

    public func wait(wallTimeout timeout:
DispatchWallTime) ->
DispatchTimeoutResult
}

/// dispatch_semaphore
extension DispatchSemaphore {

    @discardableResult
    public func signal() -> Int

    public func wait()

    public func wait(timeout:
DispatchTime) -> DispatchTimeoutResult

    public func wait(wallTimeout:
DispatchWallTime) ->
DispatchTimeoutResult
}
```