Cordova Plugin to enable communication between a Cordova iOS app and an Apple WatchKit extension.

This plugin enables the addition of a companion Apple WatchKit extension to a Cordova iOS app. It provides a Cordova plugin interface and a Watchkit framework that provides iOS/watchOS app communication.

The following points should be noted.

- An existing Cordova iOS app can be extended to include a companion WatchKit extension by opening the app project in Xcode and adding a "Watch App for iOS App" target.
- The companion WatchKit app target must be coded in Swift, and use the WatchKit App Delegate lifecycle.
- After the companion WatchKit app target has been added, **you cannot use any functions of the Cordova CLI on the project** since this will damage the project and make it unusable. All further additions or modifications to the project must be made via Xcode. Therefore, all required Cordova plugins must be installed before the WatchKit app target is added, and any changes to the HTML and Javascript code of the app (including plugins) must be applied directly to the *Staging* directory of the Xcode project.
- As an alternative to making all Cordova iOS additions or modifications to the project via Xcode, a baseline project representing only the Cordova iOS app can be maintained, and updates to the Cordova iOS app can be applied to a separate project that contains the complete app (iOS and watchOS targets).
- This plugin and companion Watchkit framework has been designed and tested for the latest versions of Cordova (10.0.0), Cordova iOS (6.2.0), iOS (14), watchOS (7), Xcode(12) and Swift (5). Compatibility with earlier versions of any of these components is not assured and will not be addressed.
- This plugin includes a fully functional test app that illustrates the use of the watchLink messaging framework.

Design objectives

At minimum, this plugin allows an Apple Watch to be easily used as as a user interface device for a Cordova iOS app.

- All iOS application logic can be provided via Javascript in the Cordova iOS app. No Swift code is required on the iOS side since all required communication functionality is provided through the plugin interface.

- User interface logic must be coded for the Watch using Swift, with the watchLink messaging framework providing incoming and outgoing message communication with the iOS app.
- Comprehensive logging is available which enables the iOS app and the watchOS app to be monitored from the Javascript console as well as the Xcode consoles for each.
- The framework Javascript and Swift code guards against error conditions that would cause an app crash and abandons the requested operation after issuing console error messages.
- There are no third-party software dependencies. The plugin and WatchOS framework are completely self-contained and rely only upon established Cordova iOS and WatchOS APIs. Of course, as much logic as desired can be included in the watchOS app. It is not limited to user interface functions and any of the WatchKit APIs can be accessed. Similarly, the iOS app may contain custom Swift modules to support the operation of the Cordova iOS app.

Communication framework

The Cordova plugin and Watchkit framework are built on the Apple iOS/watchOS WCSession class and WCSessionDelegate protocol.

All of the features of WCSession and WCSessionDelegate are supported with the exception of file transfer.

Table of Contents

- **Dictionary message transfer**. How messages consisting of dictionary payloads are transmitted and received in background if necessary.
- **Data message passing**. How messages consisting of untyped data payloads are transmitted and received.
- **User information transfers**. How user information payloads are transmitted, received and managed.
- **Application context transfers**. How application context payloads are transmitted, received and managed.
- **Complication data transfers**. How complication payloads are transmitted and managed.
- **Scheduled local notifications**. How local notifcations can be scheduled and managed.
- **Console log management**. How iOS and Watch app log output can be filtered by severity level.
- **watchLink test app**. How to install and use the test app included with the plugin.

Installation and Xcode project setup

The plugin creates the global `window.cordova.plugins.watchLink` which is abbreviated as `watchLink` in this document.

Latest published version on npm

```
cordova plugin add cordova-plugin-apple-watch-link
```

Latest version from GitHub

```
cordova plugin add
  https://github.com/nickredding/cordova-plugin-apple-watch-link.git
```

Xcode project setup

Add the watchLink plugin (and any other required plugins) to the Cordova project before creating the WatchKit app target.

After adding the WatchKit app target, add the file `WatchLinkExtensionDelegate.swift` to the watchOS app and modify the standard file `ExtensionDelegate.swift` as follows:

```
class ExtensionDelegate: WatchLinkExtensionDelegate {
```

```
    override func applicationDidFinishLaunching() {
        // Perform any final initialization of your application.
        super.applicationDidFinishLaunching()
        printAppLog("ExtensionDelegate.applicationDidFinishLaunching")

        // app-specific code ...
    }

    override func applicationDidBecomeActive() {
        super.applicationDidBecomeActive()

        // app-specific code ...
    }

    override func applicationWillResignActive() {
        super.applicationWillResignActive()

        // app-specific code ...
    }

    // app-specific code ...
```

The file `WatchLinkExtensionDelegate.swift` provides the framework for communication with the plugin, and the changes to `ExtensionDelegate.swift` enable the framework to process incoming messages from the plugin.

The following modifications to the Xcode project are necessary after creating the WatchKit app target:

- Remove the bridging header reference under build settings for the watchOS app extension (but **not** for the watchOS app).
- Set the bundle version and bundle version (short) for both the watchOS app and extension to match the iOS app.
- Set the Swift Language Version to 5 under Build Settings for the iOS app (ignore the spurious warning message about the project containing Swift Version 3 code)
- Accept Xcode recommendations to update project settings. Now add the watchOS app and extension files.

Migrating Watch target files from another Xcode project

You can drag Swift files and .xcassets graphic assets from the old Xcode project target to the new project. However, project settings should be set directly in the new project, and .storyboard files should be copied to the new project using the "File/Add Files to ..." Xcode command.

Initiate testing via Xcode

Use either the Xcode simulator or iPhone and Watch devices (preferred).

To launch using the Xcode simulator, select the Watch target and a Watch simulator, and run.

- The simulator will load both a Watch simulator and an iPhone simulator.
- The Xcode console will show only the Watch console (the iOS Xcode console is not available).
- You can open a Web Inspector window from Safari to inspect the iOS Cordova app and view the console, which will show watchLink log messages from both the iOS app and the Watch.

To launch using iPhone and Watch devices, connect your iPhone to your Mac and run the watchOS app. Then run the iOS app.

- This will result in both Xcode consoles being available (selected from the console selector).
- You can open a Web Inspector window from Safari to inspect the iOS Cordova app and view the console, which will show watchLink log messages from both the iOS app and the Watch.

Although the iOS app is installed and launched when you run the watchOS app, installing the iOS app again makes its Xcode console available.

iOS platform and plugin maintenance

As an alternative to making all Cordova iOS additions or modifications to the project via Xcode, a baseline project representing only the the Cordova iOS app can be maintained, and updates to the Cordova iOS app can be applied to a separate project that contains the complete app (iOS and watchOS targets).

The plugin includes two shell scripts that will facilitate this. These scripts should be run from a terminal window with the current directory set to the baseline project directory.

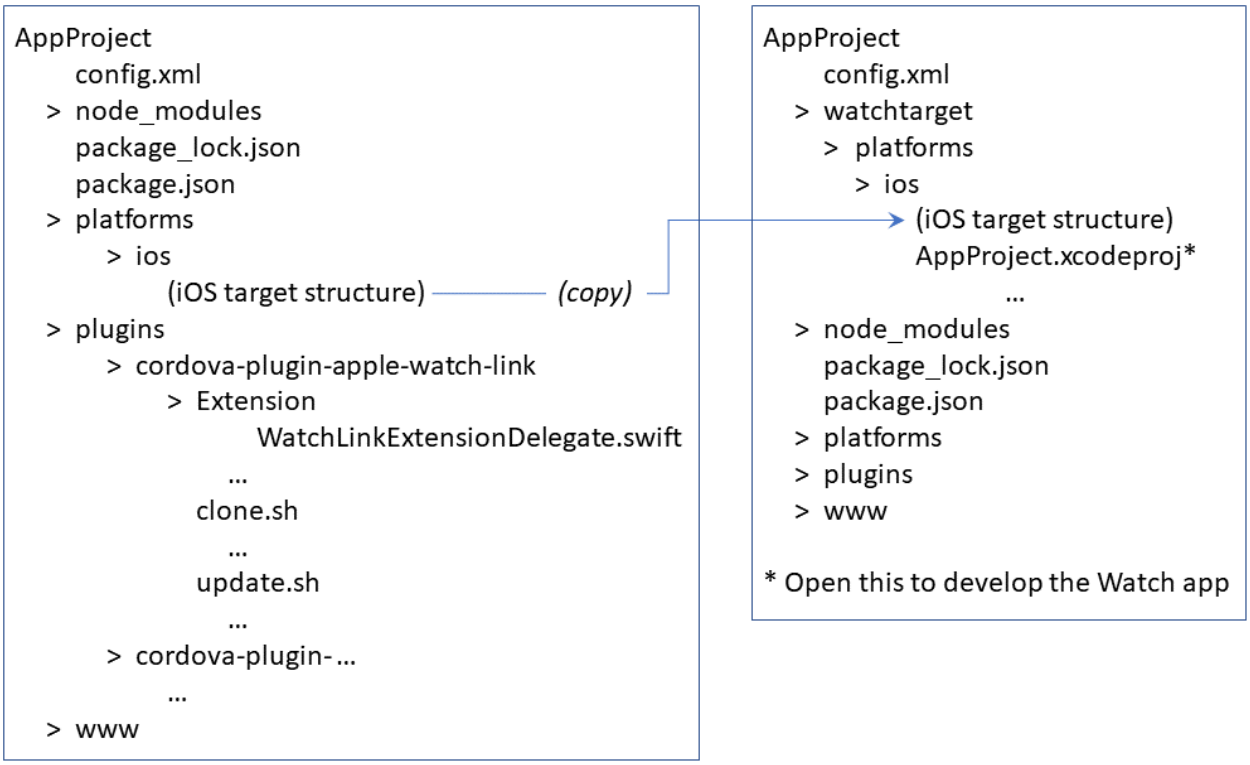
- The script `clone.sh` will perform `cordova prepare ios`, create a folder `watchtarget` and copy the iOS platform code to it.
- The script `update.sh` will perform `cordova prepare ios` and update the iOS platform code in the `watchtarget` directory. The watchOS target development should take place in the `watchtarget` iOS platform directory. Subsequent changes to the Cordova iOS platform, plugins and application code can be made to the baseline project and applied to the watchtarget iOS platform via `update.sh`.

Inital project setup

After setting up the Cordova iOS project `MyAppProject` under the `Documents` directory, the plugin can be installed and the `watchtarget` directory can be created as illustrated by the following Terminal session.

```
$cd Documents/AppProject
cordova plugin add cordova-plugin-apple-watch-link
$cp plugins/cordova-plugin-apple-watch-link/*.sh .
$chmod 777 *.sh
$./clone.sh
$
```

This creates the `watchtarget` directory structure as illustrated below.



The watchOS target development can take place in the `watchtarget` iOS platform directory. Open this project from `AppProject.xcodeproj` in the `watchtarget/platforms/ios` directory.

Note: XCode will show the `config.xml` and `www` items in the project root as missing. This is not an error. These represent items in the baseline project which should be updated there as necessary.

After the Watch targets have been created, the file `WatchLinkExtensionDelegate.swift` should be added to the Extension and the standard file `ExtensionDelegate.swift` modified as specified above.

Cordova iOS updates

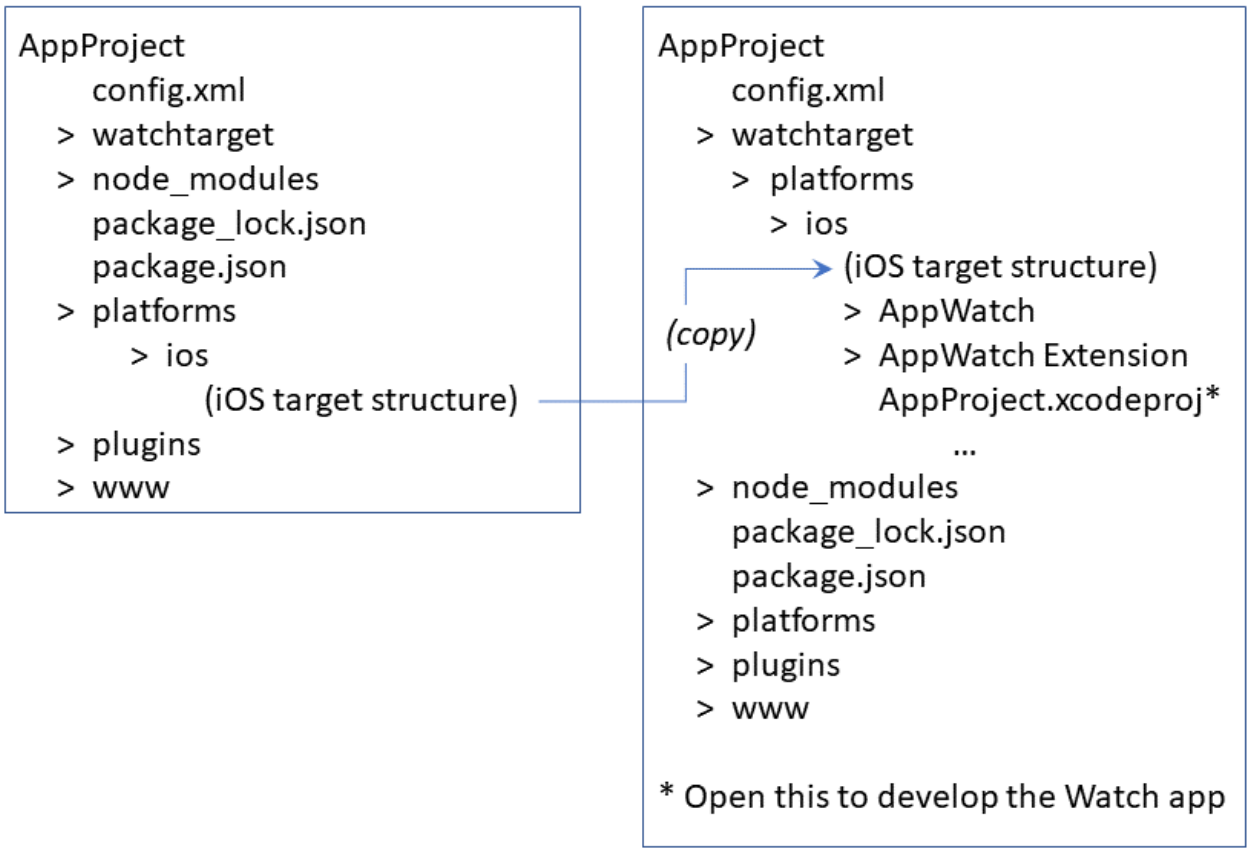
Changes to the Cordova iOS app code (platform, plugins and application logic) can be applied to the baseline project and then applied to the `watchtarget` iOS platform directory as illustrated by the following Terminal session (where `AppIOS` is the name of the iOS app target).

```
$cd Documents/AppProject
$.update.sh AppProject AppIOS

        Updating AppProject in watchtarget/platforms/ios

$
```

This updates the `watchtarget` directory structure as illustrated below.



Note: The update script does **not** copy any non-code resources to the `watchtarget/platforms/ios`. Graphic assets, storyboards and plist files (which should be developed from Xcode within the `watchtarget/platforms/ios` project) are not affected.

Watch session initialization

The Watch session at the Swift level might not have completed initialization by the time the plugin completes initialization. Therefore, the iOS and watchOS apps must ensure the Watch session is ready before attempting to query its status or send messages to the counterpart.

Watch session initialization status (iOS)

The variable `watchLink.initialized` is true if the Watch session has completed initialization. `watchLink.ready` adds a function to a list of app functions to invoke when the Watch session completes initialization. The function will be invoked immediately if the Watch session initialization has already been completed at the time `watchLink.ready` is invoked.

iOS Cordova app initialization can therefore be organized as follows:

```
document.addEventListener('deviceready', onDeviceReady, false);

function onDeviceReady() {
        // binding handlers and dealing with notifications can
        // proceed immediately
        window.cordova.watchLink.bindMessageHandler(...);

        // communication must wait until Watch session initialization is complete
        window.cordova.watchLink.ready(
                function() {
                        // initiate Watch communication
                        window.cordova.watchLink.sendMessage(...)
                });
```

Watch session initialization status (watchOS)

The global `watchInitialized` is true if the Watch session has completed initialization, and the completion of initialization can be signaled to the watchOS app.

```
watchReady(_ f: (() -> Void))
//      sets a function f to invoke when the Watch session
//      completes initialization.
```

The function will be invoked immediately if the Watch session initialization has already been completed at the time `watchReady` is invoked.

Session status

The status of the Watch can be queried by the iOS app, which can also bind handlers to be invoked when the status of the Watch changes.

Watch availability (iOS)

The **availability** of the watchOS app reflects whether the companion watchOS app is available, which is required for communication.

```
watchLink availability states
-----------------------------

watchLink.watchAvailable = true;
watchLink.watchUnavailable = false;
watchLink.watchNotPaired = "NOTPAIRED";
watchLink.watchNotInstalled = "NOTINSTALLED";

watchLink.available is the availability state.

        null: yet to be initialized
        watchLink.watchAvailable: Watch session is available, Watch is paired
                and Watch companion app installed
        watchLink.watchUnavailable: Watch session is not available
        watchLink.watchNotPaired: Watch session is available but Watch is
                not paired to phone
        watchLink.watchNotInstalled: Watch session is available but
                Watch companion app has not been installed

Note: Use the === operator to test availability states since all states except
        watchLink.watchUnavailable would evaluate as true in a conditonal test.

watchLink.availability updates and returns the current availability state
of the Watch session. It is not normally required to call this since
watchLink.available is kept up to date as soon as the Watch session
has completed initialization.

Using the traditional Cordova callback method:

watchLink.availability(callback, error)
//      success = function(availability)
//              availability is the current availability state
//                      of the Watch session

//      error = function(errorString)
//              If the Watch session has not completed initialization
//                      errorString will be "uninitialized"


Using the Promise construct:
```

```
watchLink.availability().then(callback).catch(error)

watchLink.availabilityChanged(callback) registers a callback to invoke when the
availability state changes. The callback parameter represents the new
availability state.

Supply null to deregister a previously set callback. Otherwise,
this call will overwrite a previously set callback with a different callback.
```

Watch reachability (iOS)

The **reachability** of the watchOS app reflects whether the app is in the foreground (i.e., active) and able to process messages. The watchOS app can only be reachable if it is available.

```
watchLink.reachable is the reachability state.

null: yet to be initialized
true: watchOS app is reachable
false: watchOS app is not reachable

watchLink.reachability updates and returns the current reachability state
of the Watch session. It is not normally required to call this since
watchLink.reachable is kept up to date as soon as the Watch session
has completed initialization.

Using the traditional Cordova callback method:

watchLink.reachability(callback, error)
//      success = function(reachability)
//              reachability is the current reachability state of the
//                      Watch session

//      error = function(errorString)
//              If the Watch session has not completed initialization
//                      errorString will be "uninitialized"


Using the Promise construct:

watchLink.reachability().then(callback).catch(error)

watchLink.reachabilityChanged(callback) registers a callback to invoke
when the reachability state changes The callback parameter represents
the new reachability state.

Supply null to deregister a previously set callback. Otherwise, this
call will overwrite a previously set callback with a different callback.
```

Watch application state (iOS)

The **application state** of the watchOS app reflects whether the companion watchOS app is active, inactive or in background.

```
watchLink application states

watchLink.watchApplicationActive = "ACTIVE";
watchLink.watchApplicationInactive = "INACTIVE";
watchLink.watchAapplicationBackground = "BACKGROUND";

watchLink.applicationState is the full application state.

        null: yet to be initialized, otherwise
        watchLink.applicationState =
                { state: <state>, complication: <Boolean>, isPaired: <Boolean>,
                        isAppInstalled: <Boolean>, directoryURL: <string> }
        watchLink.applicationState.state
                false: Watch is not available, or the application is
                        not running or suspended
                watchLink.watchApplicationActive: The watchOS app is
                        running in foreground and responding to events
                watchLink.watchApplicationInactive: The watchOS app is
                running in foreground but not yet responding to events
                watchLink.watchApplicationBackground: The watchOS app is
                        running in the background
        watchLink.applicationState.complication
                true: complication is enabled
                false: complication is not enabled
        watchLink.applicationState.isPaired
                true: Watch is paired
                false: Watch is not paired
        watchLink.applicationState.isAppInstalled
                true: watchOS app is installed
                false: watchOS app is not installed
        watchLink.applicationState.directoryURL
                The URL of a directory for storing information specific to the
                        currently paired and active Watch

Note that watchLink.applicationState.state is false if watchLink.available
is not true.

watchLink.watchApplicationState updates and returns the current application
state of the watchOS app. It is not normally required to call this since
watchLink.applicationState is kept up to date as soon as the Watch session
has completed initialization.

Using the traditional Cordova callback method:

watchLink.watchApplicationState(callback, error)
```

```
//      success = function(state)
//              state is the current application state

//      error = function(errorString)
//              If the Watch session has not completed initialization
//                      errorString will be "uninitialized"

Using the Promise construct:

watchLink.watchApplicationState().then(callback).catch(error)

watchLink.applicationStateChanged registers a callback to invoke
when the application state changes. The callback parameter
represents the new application state.

Supply null to deregister a previously set callback. Otherwise, this call
will overwrite a previously set callback with a different callback.
```

Session status (watchOS)

To query the availability status of the session, examine the global variable `phoneAvailable` which maintains the availability state of the iPhone companion app.

To query the reachability status of the session, examine the global variable `phoneReachable` which maintains the reachability state of the iPhone companion app.

You can also examine the session directly. For example,

```
let watchSession = WCSession.default
watchSession.isReachable // is the iOS app reachable
        ...
```

You can bind handlers to be notified of changes in companion app availability and reachability.

```
func bindAvailabilityHandler(_ handler: @escaping ((Bool) -> Void))

func bindReachabilityHandler(_ handler: @escaping ((Bool) -> Void))
```

See WCSession class for details.

iPhone companion app status (watchOS)

To query the running status of the iPhone companion app, examine the global variable `phoneRunning` which maintains the running state.

If the iPhone companion app is running (in foreground or background) the running state is `true`.

You can bind a handler to be notified of changes in companion app running state.

```
func bindRunningStateHandler(_ handler: @escaping ((Bool) -> Void))
```

iPhone companion app state considerations

If your watch extension does not support running without the iOS companion app installation (as specified in the `Info.plist` file), the watchOS session status will always be avaialable since the watch app will be uninstalled if the iPhone app becomes unavailable.

If the watchOS session status is available it will always be reachable. Sending a message to the iOS companion app when it is not running will cause iOS to launch the app in background to process the message.

The global variable `phoneRunning` can be used to detect that the iPhone companion app is not currently running. If it is not desired to send messages to the iPhone companion app when it is not running, the global variable `requirePhoneRunning` can be set to `true` and any attempts to send messages, or transfer user information or application contexts, will be ignored if the iPhone companion app is not running. In this case, the error handler (if supplied) will be invoked with error message "NOTRUNNING".

watchLink message session management

Cordova plugins are initialized at the Objective-C/Swift level upon initial launch of a Cordova app. However, a Cordova app can restart at the Javascript level (e.g., via `window.reload`) and this does not cause plugins at the Objective-C/Swift level to restart.

For this reason, watchLink uses a unique **session ID** to identify communication in the current messaging session. This key is created when the Javascript layer starts initially or restarts, and any previous session IDs become obsolete. Communications arriving with an obsolete session ID are discarded, and when a session starts any pending communications awaiting transmission with an obsolete session ID are also discarded.

The function `watchLink.resetSession` starts a new session, discarding any pending communications with an obsolete session ID and sending a message to the watchOS app providing the new session ID. Upon receipt of the reset message, the watchOS app flushes any pending communications with an obsolete key, and will also discard any communications arriving with an obsolete session ID.

```
watchLink.resetSession(completion)
//      completion = function(msg) is an optional callback to invoke when the
//                          session reset is complete (the reset message has been
acknowledged
//                                  by the watchOS framework).
//              msg = true if successful, or "uninitialized" if the Watch
//                          session initialization is not complete
```

It is not normally necessary for the iOS app to invoke `watchLink.resetSession` since this is done by watchLink.js when it initializes or reinitializes. However, this function can be invoked to invalidate the current session and start a new one if the app needs to do that absent a restart at the Javascript level.

When the watchOS app initializes, and prior to receiving a reset message from the iOS app, communications are dispatched with session ID zero which is accepted by the iOS watchLink framework. Upon receipt of a reset message the watchOS app updates the session ID of pending communications that have a zero session ID to the new session ID.

Watch session reset callback (watchOS)

It may be useful for a reset issued by the iOS app to be signaled to the watchOS app. This can be accomplished by binding a reset handler.

```
watchReset(_ f: (() -> Void))
//      sets the function f to invoke when the Watch session is reset.
```

Communication methods

There are six methods of communication provided:

- **Dictionary message passing**: messages may be exchanged containing a dictionary of values. Messages will be queued until the companion app is reachable.
- **Data message passing**: messages may be exchanged containing an untyped data object. Messages will be queued until the companion app is reachable.
- **User information transfers**: a dictionary of values representing user information may be transmitted in either direction. These transfers can occur in background (when the companion app is available but not reachable).
- **Application context transfers**: a dictionary of values representing application context may be transmitted in either direction. These transfers can occur in background (when the companion app is available but not reachable).

- **Complication data transfers**: a dictionary of values representing complication user information may be transmitted from the iOS app to the watchOS app. These transfers can occur in background (when the watchOS app is not reachable, i.e. the watchOS app is not in foreground).
- **Scheduled local notifications**: you can schedule a local notification to be presented at a specific time. The notification will be shown on the iOS device or Watch, depending on which one is active.

Dictionary format

A `Dictionary` is defined as a `property list dictionary,` which is a dictionary of `property list values.`

A `property list value` is defined as follows:

```
<property-list-value> = <Boolean (Bool)> | <Number (Int64)> | String (String)> |
                                <property-list-array> |
                                <property-list-dictionary>

<property-list-array> = [ <property-list-value>, ... ]
                                ([ <property-list-value> ])
<property-list-dictionary> = { <String> : <property-list-value>, ... }
                                ([ <String> : <property-list-value> ])
```

In this definition, the types refer to Javascript types, with the corresponding Swift type in parentheses.

Untyped data object format

An `untyped data object` may be created via Javascript as an ArrayBuffer.

```
var buffer = new ArrayBuffer(<length>)
```

where `<length>` is the size of the data object in bytes. An ArrayBuffer can be accessed for reading and writing by creating a view in the desired format. For example,

```
// byteView is an array of 8 bit unsigned integers
var byteView = new Uint8Array(buffer);

// wordView is an array of 32 bit integers
var wordView = new Int32Array(buffer);
```

In Swift, a data object is declared as type `Data` and then cast to an appropriate type for access.

Dictionary message passing

Dictionary messages are accompanied by a **message type** which is used to determine how to route the message at the receiving end. Both the iOS app and watchOS app can bind a message handler function to a message type, and messages with that message type will be delivered to the handler that is bound.

Dictionary messages cannot be sent unless the companion app is reachable. If the companion app is available but not reachable, dictionary messages are queued until the companion app becomes reachable.

The following message types are reserved and may not be used:

```
ACK, DATA, SESSION, RESET, SETLOGLEVEL, SETPRINTLOGLEVEL, UPDATEDCONTEXT,
UPDATEDUSERINFO, WATCHLOG, WATCHERRORLOG, WATCHAPPLOG, WCSESSION, IOSINITIALIZED,
IOSTERMINATED
```

The iOS app and watchOS app can bind a default message handler to receive messages with message types that are not bound.

Dictionary message transmission (iOS)

Dictionary messages are sent from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.sendMessage(msgType, msgBody, success, error)
//      Upon return, msgBody.TIMESTAMP contains the unique numeric
//              timestamp that can be used to refer to the message.
//      msgType = <String>
//      msgBody = <property-list-dictionary>
//      success = function(timestamp)
//              Invoked when the message has been delivered and acknowledged
//                      timestamp is the string representation of
//                              msgBody.TIMESTAMP
//      error = function(errorString0
//              Invoked when an error occurred
//                      errorString = string describing the error followed
//                              by ":<timestamp>"
//                      <timestamp> is the value of msgBody.TIMESTAMP
//              If the Watch is unavailable errorString will
//                      be "unavailable:<timestamp>"
```

```
//              If the session was reset errorString will
//                      be "sessionreset:<timestamp>"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
//              Errors are logged regardless of whether an error
//                      function is provided
```

Note: the msgBody sent to the Swift layer is a clone of the object submitted to the sendMessage function. Therefore, changes to msgBody made prior to transmission will not be reflected in the message payload.

Dictionary messages can also be sent from the iOS app using a Promise, as in

```
watchLink.sendMessage(msgType, msgBody).then(success).catch(error)
//      Upon return, msgBody.TIMESTAMP contains the unique numeric
//      timestamp that can be used to refer to the message.
```

Dictionary messages are normally acknowledged by the receiving end, at which point the success function is invoked. However, if `null` is provided for the success parameter, the message will be sent without acknowledgement. For this behavior you must use the traditional Cordova plugin call.

A dictionary message that is sent with acknowledgement will block subsequent dictionary messages (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

Note that regardless of acknowledgement, messages are sent and delivered in the order in which they are dispatched.

Dictionary message transmission (watchOS)

Dictionary messages are sent from the watchOS app using a call to the function `messageToPhone.`

```
func messageToPhone(msgType: String, msg: [String: Any], ack: Bool = false,
                        ackHandler: (Int64) -> Void)= nullHandler,
                        errHandler: ((String, Int64) -> Void)= nullHandler)
        -> Int64
```

Setting the `ack` parameter to true will send the message with acknowledgement. A dictionary message that is sent with acknowledgement will block subsequent dictionary messages (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

The `errHandler` function (if provided) will be invoked in the case of an error arising. If the session was reset errorString will be "sessionreset:". Errors are logged regardless of whether an error handler is provided.

This function returns the unique timestamp assigned to the message, and this value is provided to the `ackHandler` and `errHandler` functions.

Dictionary message flushing

Any outstanding messages can be flushed from the outgoing message queue using `watchLink.flushMessages()` on the iOS side and `flushMessages()` on the watchOS app side. The error handlers for flushed messages are **not** invoked. Note that any acknowledgements that arrive for flushed messages are ignored (the success handlers are **not** invoked).

Dictionary message receipt (iOS)

Handlers can be bound to specific message types, and are invoked to process incoming messages with the corresponding type.

```
watchLink.bindMessageHandler(expr, handler)
//      expr = <String> | <RegExp>
//      handler = function(msgType, msgBody)
//              Return false to terminate message processing
//              Otherwise processing will continue with additional
//                      matching handlers
```

To handle an incoming message, the message type is extracted and each handler with a matching expression is invoked with the message type and message body as parameters. The order of invocation is the order in which the handlers were bound.

- Processing continues until all matching expressions have been checked, or until a handler returns false to halt processing.
- The expression can be a string (requires an exact match) or a regular expression.
- Supplying null for the handler will unbind a previously bound handler.
- Supplying a handler for an existing match expression will overwrite the existing handler for that expression.

- Supplying null for the match expression will set the default handler, unless it is null also in which case the default handler will be unbound.

Dictionary message receipt (watchOS)

Message handlers in the watchOS app are bound using `bindMessageHandler` and `bindDefaultMessageHandler.`

```
bindMessageHandler(msgType: String,
        handler: @escaping ((String, [String: Any]) -> Bool))

bindMessageHandler(msgRegex: String,
        handler: @escaping ((String, [String: Any]) -> Bool))

bindDefaultMessageHandler(handler: @escaping ((String, [String: Any]) -> Void))
```

Message processing in the watchOS app is handled in the same way as on the iOS app.

Direct dictionary message handling

It is possible to use the `WCSession` interface directly for sending messages to the counterpart. This will bypass the watchLink framework and these messages will be detected as direct and delivered to the default message handler (if configured) with `msgType = "WCSESSION"`.

Note that if you access the `WCSession` interface directly your code is responsible for error handling and ensuring that the counterpart app is reachable before sending messages.

Dictionary messsages can be sent directly (bypassing the watchLink framework) from the iOS app using `watchLink.wcSessionCommand`. The Watch app must be reachable (otherwise the error callback will be invoked).

```
watchLink.wcSessionCommand('sendMessage', payload, error)
//      payload = <property-list-dictionary>
//      error = function(msg)
//              Invoked in the case of an error
```

Dictionary message transfer

Dictionary messages cannot be sent unless the companion app is reachable. However, the iOS message transfer function will send a message immediately if the watch companion app is reachable, but use user information transfer in background if the companion app is not reachable.

This enables the watch companion app to receive, acknowledge and process messages while in background.

Messages received via message transfer are processed by the watch companion app in the same way (using the same handlers) as messages receieved via message transmission. The fact that the mesage was sent via user information transfer is transparent to the watch companion app.

Dictionary message transfer (iOS)

Dictionary messages are transferred from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.transferMessage(msgType, msgBody, success, error)
//      Upon return, msgBody.TIMESTAMP contains the unique numeric
//              timestamp that can be used to refer to the message.
//      msgType = <String>
//      msgBody = <property-list-dictionary>
//      success = function(timestamp)
//              Invoked when the message has been delivered and acknowledged
//                      timestamp is the string representation of
//                              msgBody.TIMESTAMP
//      error = function(errorString0
//              Invoked when an error occurred
//                      errorString = string describing the error followed
//                              by ":<timestamp>"
//                      <timestamp> is the value of msgBody.TIMESTAMP
//              If the Watch is unavailable errorString will
//                      be "unavailable:<timestamp>"
//              If the session was reset errorString will
//                      be "sessionreset:<timestamp>"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
//              Errors are logged regardless of whether an error
//                      function is provided
```

Note: the msgBody sent to the Swift layer is a clone of the object submitted to the sendMessage function. Therefore, changes to msgBody made prior to transmission will not be reflected in the message payload.

Dictionary messages can also be sent from the iOS app using a Promise, as in

```
watchLink.transferMessage(msgType, msgBody).then(success).catch(error)
//      Upon return, msgBody.TIMESTAMP contains the unique numeric
//      timestamp that can be used to refer to the message.
```

Dictionary messages are normally acknowledged by the receiving end, at which point the success function is invoked. However, if `null` is provided for the success parameter, the message will be sent without acknowledgement. For this behavior you must use the traditional Cordova plugin call.

A dictionary message that is sent with acknowledgement will block subsequent dictionary messages (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

Note that regardless of acknowledgement, messages are sent and delivered in the order in which they are dispatched.

Data message passing

Data messages are actually sent using dictionary messages. Both the iOS app and watchOS app can bind a message handler function to handle incoming data messages.

Data messages cannot be sent unless the watchOS app is reachable. If the Watch is available but not reachable, data messages are queued until the Watch becomes reachable.

Data message transmission (iOS)

Data messages are sent from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.sendDataMessage(msgData, success, error);
//      msgData = <ArrayBuffer>
//      success = function(msgData)
//              Invoked when the message has been delivered and acknowledged
//                      msgData is the message payload
//      error = function(errorString)
//              Invoked when an error occurred
//                      errorString = string describing the error
//              If the Watch is unavailable errorString will be "unavailable"
//              If the session was reset errorString will be "sessionreset"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

```
//              Errors are logged regardless of whether an error
//                      function is provided
```

Note: the msgData sent to the Swift layer and returned to the success function is a clone of the object submitted to the sendDataMessage function. Therefore, changes to msgData made prior to transmission will not be reflected in the message payload. Data messages can also be sent from the iOS app using a Promise, as in

```
watchLink.sendDataMessage(msgType, msgBody).then(success).catch(error)
```

Data messages are normally acknowledged by the receiving end, at which point the success function is invoked. However, if `null` is provided for the success parameter, the message will be sent without acknowledgement. For this behavior you must use the traditional Cordova plugin call.

A data message that is sent with acknowledgement will block subsequent data messages (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

Note that regardless of acknowledgement, messages are sent and delivered in the order in which they are dispatched.

Data message transmission (watchOS)

Data messages are sent from the watchOS app using a call to the function `dataMessageToPhone`

```
func dataMessageToPhone(msg: Data, ack: Bool = false,
                        ackHandler: ((Int64) -> Void)= nullHandler,
                        errHandler: ((String) -> Void)= nullHandler) -> Int64
```

This function returns the unique timestamp assigned to the data message.

Setting the `ack` parameter to true will send the message with acknowledgement. A data message that is sent with acknowledgement will block subsequent data messages (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

The `ackHandler` function (if provided) receives the timestamp returned by `dataMessageToPhone`.

The `errHandler` function (if provided) will be invoked in the case of an error arising. If the message is cancelled errorString will be "sessionreset:". Errors are logged regardless of whether an error handler is provided.

Data message flushing (iOS)

Any outstanding data messages can be flushed from the outgoing message queue using `watchLink.flushDataMessages()` on the iOS side and `flushDataMessages()` on the watchOS app side. The error handlers for flushed messages are **not** invoked. Note that any acknowledgements that arrive for flushed messages are ignored (the success handlers are **not** invoked).

Data message receipt (iOS)

A single handler can be bound to process incoming data messages.

```
watchLink.bindDataMessageHandler(handler)
//      handler = function(msgData)
```

- Supplying null for the handler will unbind a previously bound handler.
- Binding a handler will overwrite any existing handler (only one handler can be invoked for an incoming data message).

Data message receipt (watchOS)

A data message handler in the watchOS app is bound using `bindDataMessageHandler.`

```
bindDataMessageHandler(handler: @escaping ((Data) -> Bool))
```

Direct data message handling

It is possible to use the `WCSession` interface directly for sending data messages to the counterpart. This will bypass the watchLink framework and these messages will be detected as direct and delivered to the data message handler (if configured).

Note that if you access the `WCSession` interface directly your code is responsible for error handling and ensuring that the counterpart app is reachable before sending messages.

Dictionary messsages can be sent directly (bypassing the watchLink framework) from the iOS app using `watchLink.wcSessionCommand`. The Watch app must be reachable (otherwise the error callback will be invoked).

```
watchLink.wcSessionCommand('sendDataMessage', payload, error)
//      payload = <ArrayBuffer>
//      error = function(msg)
//              Invoked in the case of an error
```

User information transfers

A dictionary of values representing user information may be transmitted in either direction. These transfers can occur in background (when the companion app is not reachable).

User information transfer (iOS)

User information transfers are sent from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.sendUserInfo(userInfo, success, error)
//      Upon return, userInfo.TIMESTAMP contains the unique
//              numeric timestamp that can be used to refer to the transfer.
//      userInfo = <property-list-dictionary>
//      success = function(timestamp)
//              Invoked when the message has been delivered and acknowledged
//                      timestamp is the string representation of
//                              userInfo.TIMESTAMP
//      error = function(errorString)
//              Invoked when an error occurred
//                      errorString = string describing the error
//                              followed by ":<timestamp>"
//                      <timestamp> is the value of userInfo.TIMESTAMP
//              If the session was reset errorString will be "sessionreset"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
//              Errors are logged regardless of whether an
//                      error function is provided
```

Note: the userInfo sent to the Swift layer is a clone of the object submitted to the sendUserInfo function.

User information transfers can also be sent from the iOS app using a Promise, as in

```
watchLink.sendUserInfo(userInfo).then(success).catch(error)
//      Upon return, userInfo.TIMESTAMP contains the unique numeric
//      timestamp that can be used to refer to the transfer.
```

User information transfers are normally acknowledged by the receiving end, at which point the success function is invoked. However, if `null` is provided for the success parameter, the transfer will be sent without acknowledgement. For this behavior you must use the traditional Cordova plugin call.

A transfer that is sent with acknowledgement will block subsequent transfers (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

Note that regardless of acknowledgement, transfers are sent and delivered in the order in which they are dispatched.

watchLink.sendUserInfo adds the key `TIMESTAMP` to the userInfo object which contains a unique timestamp representing the transfer. Note that if userInfo contains an existing key `TIMESTAMP` it will be overwritten with the timestamp. This value can be used to query the status of the transfer, and cancel the transfer if desired. The value of this key can be retrieved from the supplied userInfo object immediately upon return from the function invocation.

The userInfo object received by the counterpart app will include keys `TIMESTAMP`, `SESSION` and `ACK`. If the original userInfo object contains any of these keys they will be overwritten.

User information transfer (watchOS)

User information transfers are sent from the watchOS app using a call to the function `updateUserInfoToPhone` which returns the Watch equivalent of `TIMESTAMP.`

```
func updateUserInfoToPhone(userInfo: [String: Any],
                           ackHandler: (([String: Any]) -> Void)? = nil,
                           errHandler: ((String, [String: Any]) -> Void)? = nil)
        -> Int64
```

Setting the `ack` parameter to true will send the information with acknowledgement. A transfer that is sent with acknowledgement will block subsequent transfers (acknowledged or not) until it is acknowledged or flushed due to an error or session reset.

The `errHandler` function (if provided) will be invoked in the case of an error arising. If the transfer is cancelled errorString will be "sessionreset". Errors are logged regardless of whether an error function is provided

User information transfer status and flushing (iOS)

A user information transfer can be queried and/or cancelled using the `TIMESTAMP` set by `watchLink.sendUserInfo` or `updateUserInfoToPhone.`

A user information transfer can be queried via `watchLink.queryUserInfo` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.queryUserInfo(timestamp, success, error)
//      success = function(transferInfo)
//              transferInfo = { timestamp: <Number>,
                                 isComplication: <Boolean>,
//                               transmitComplete: <Boolean>,
//                               userInfo: <property-list-dictionary>
//              transferInfo = false if the transfer status is no
//                      longer available
//      error = function(errorString)
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The in-progress user information transfers can also be accessed from the iOS app using a Promise, as in

```
watchLink.queryUserInfo(timestamp).then(success).catch(error)
```

The status of a user information transfer will remain available until the transfer is complete and a subsequent user information transfer is initiated.

A user information transfer can be cancelled via `watchLink.cancelUserInfo` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.cancelUserInfo(timestamp, success, error)
//      success = function(cancelled
//              cancelled = <Boolean>, true if the transfer was cancelled,
//                      false otherwise
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

An information transfer can also be cancelled from the iOS app using a Promise, as in

```
watchLink.cancelUserInfo(timestamp).then(success).catch(error)
```

All outstanding user information transfers can be cancelled and flushed from the outgoing transfer queue using `watchLink.flushUserInfoTransfers()`. The error handlers for cancelled or flushed transfers are **not** invoked. Note that any acknowledgements that arrive for flushed transfers are ignored (the success handlers are **not** invoked).

User information transfer status and flushing (watchOS)

User information transfers from the watchOS app can be managed as follows:

```
func queryUserInfo(timestamp: Int64) ->
                (timestamp: Int64,
                isComplication: Bool,
                transmitComplete: Bool,
                userInfo: [String: Any])

func cancelUserInfo(timestamp: Int64) -> Bool

func flushUserInfoTransfers()
```

The error handlers for cancelled or flushed transfers are **not** invoked. Note that any acknowledgements that arrive for flushed transfers are ignored (the success handlers are **not** invoked).

User information transfers outstanding (iOS)

The in-progress user information transfers can be accessed via `watchLink.outstandingUserInfoTransfers` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.outstandingUserInfoTransfers(success, error)
//      success = function(outstandingUserInfo
//              outstandingUserInfo = [
//                      { userInfoID: <Number>,
//                      isComplication: <Boolean>,
//                      transmitComplete: <Boolean>,
//                      userInfo: <property-list-dictionary> } ]
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The in-progress user information transfers can also be accessed from the iOS app using a Promise, as in

```
watchLink.outstandingUserInfoTransfers().then(success).catch(error)
```

User information transfers outstanding (watchOS)

The in-progress user information transfers can be accessed from the watchOS app as follows:

```
func outstandingUserInfoTransfers() -> [ [
//              timestamp: Int64,
//              isComplication: Bool,
//              transmitComplete: Bool,
//              userInfo: [String: Any] ] ]
```

User information transfer receipt (iOS)

A single handler can be bound to process incoming user information transfers.

```
watchLink.bindUserInfoHandler(handler)
//      handler = function(userInfo)
```

Supplying null for the handler will unbind a previously bound handler.

Binding a handler will overwrite any existing handler (only one handler can be invoked for an incoming transfer).

User information transfer receipt (watchOS)

User information transfer handlers in the watchOS app are bound using `bindUserInfoHandler.`

```
bindUserInfoHandler(handler: @escaping (([String: Any]) -> Void))
```

Note: The dictionary passed to the handler will contain the key `ISCOMPLICATION` with value `true` if this is a complication update.

Direct user information transfer handling

It is possible to use the `WCSession` interface directly for transfering user information to the counterpart. This will bypass the watchLink framework and these transfers will be detected as direct and delivered to the user information handler (if configured).

Note that if you access the `WCSession` interface directly your code is responsible for error handling and ensuring that he counterpart app is available before sending user information transfers.

User information transfers can be sent directly (bypassing the watchLink framework) from the iOS app using `watchLink.wcSessionCommand`. The Watch app must be reachable. There is no provision for an error callback.

```
watchLink.wcSessionCommand('updateUserInfo', payload)
//      payload = <property-list-dictionary>
```

Application context transfers

A dictionary of values representing application context information may be transmitted in either direction. These transfers can occur in background (when the watchOS app is not reachable, i.e. the watchOS app is not in foreground).

Application context transmission (iOS)

Application context transfers are sent from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.sendContext(context, success, error);
//      Upon return, context.TIMESTAMP contains the unique
//              numeric timestamp that can be used to refer to the transfer
//      context = <property-list-dictionary>
//      success = function(timestamp)
//              Invoked when the message has been delivered and acknowledged
//              Context is the application context payload
//      error = function(errorString)
//              Invoked when an error occurred
//                      errorString = string describing the error followed
//                              by ":<timestamp>"
//                      <timestamp> is the value of userInfo.TIMESTAMP
//              If the transfer is superceded errorString
//                      will be "reset:<timestamp>"
//              If the transfer is flushed errorString
//                      will be "sessionreset:<timestamp>"
//              If the Watch session has not completed initialization
//                      errorString will be "uninitialized"
//              Errors are logged regardless of whether an error
//                      function is provided
```

Note: the context sent to the Swift layer is a clone of the object submitted to the sendContext function.

Application context transfers can also be sent from the iOS app using a Promise, as in

```
watchLink.sendContext(context).then(success).catch(error)
//      Upon return, context.TIMESTAMP contains the unique
//      numeric timestamp that can be used to refer to the transfer
```

Application context transfers are normally acknowledged by the receiving end, at which point the success function is invoked. However, if `null` is provided for the success parameter, the transfer will be sent without acknowledgement. For this behavior you must use the traditional Cordova plugin call.

A transfer that is sent with acknowledgement will **not** block subsequent transfers (acknowledged or not). However, transfers requested before the completion of a previously requested transfer will overwrite that transfer. In this case, the error handler (if configured) for the previous transfer will be invoked.

The context object received by the counterpart app will include keys `TIMESTAMP`, `SESSION` and `ACK`. If the original userInfo object contains any of these keys they will be overwritten.

Application context transmission (watchOS)

Application context transfers are sent from the watchOS app using a call to the function `updateUserInfoToPhone.`

```
func updateContextToPhone(context: [String: Any], ack: Bool = false,
                          ackHandler: ((Int64) -> Void)? = nil,
                          errHandler: ((String) -> Void)? = nil) -> Int64
```

Setting the `ack` parameter to true will send the information with acknowledgement. A transfer that is sent with acknowledgement will **not** block subsequent transfers (acknowledged or not). However, transfers requested before the completion of a previously requested transfer will overwrite that transfer. In this case, the error handler (if configured) for the previous transfer will be invoked.

The `errHandler` function (if provided) will be invoked in the case of an error arising. If the transfer is cancelled errorString will be "sessionreset". Errors are logged regardless of whether an error function is provided.

Application context transfer status (iOS)

Unlike user information transfers, the status of application context transfers cannot be queried.

However, the contents of the most recently transmitted and received application context transfers can be queried from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.latestContextSent(success, error)
//      success = function(context
//              context = <property-list-dictionary>
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"

watchLink.latestContextReceived(success, error
//      success = function(context)
//              context = <property-list-dictionary>
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The contents of the most recently transmitted and received application context transfers can also be accessed from the iOS app using a Promise, as in

```
watchLink.latestContextSent().then(success).catch(error)

watchLink.latestContextReceived().then(success).catch(error)
```

Application context transfer status (watchOS)

The contents of the most recently transmitted and received application context transfers can be accessed from the watchOS app as follows:

```
func latestContextSent() -> [String: Any]

func latestContextReceived() -> [String: Any]
```

Application context flushing (iOS)

All outstanding application context transfers can be flushed from the outgoing transfer queue using `watchLink.flushContextTransfers()`. The error handlers for flushed transfers are **not** invoked. Note that any acknowledgements that arrive for flushed transfers are ignored (the success handlers are **not** invoked).

Application context flushing (watchOS)

All outstanding application context transfers can be flushed from the outgoing transfer queue using `flushContextTransfers()`. The error handlers for flushed transfers are **not** invoked. Note that any acknowledgements that arrive for flushed transfers are ignored (the success handlers are **not** invoked).

Application context transfer receipt (iOS)

A single handler can be bound to process incoming application context transfers.

```
watchLink.bindContextHandler(handler)
//      handler = function(context)
```

Supplying null for the handler will unbind a previously bound handler.

Binding a handler will overwrite any existing handler (only one handler can be invoked for an incoming transfer).

Application context transfer receipt (watchOS)

Application context transfer handlers in the watchOS app are bound using `bindUserInfoHandler.`

```
bindContextHandler(handler: @escaping (([String: Any]) -> Void))
```

Direct application context transfer handling

It is possible to use the `WCSession` interface directly for transfering application contexts to the counterpart. This will bypass the watchLink framework and these transfers will be detected as direct and delivered to the user information handler (if configured).

Note that if you access the `WCSession` interface directly your code is responsible for error handling and ensuring that he counterpart app is available before sending context updates.

Dictionary messsages can be sent directly (bypassing the watchLink framework) from the iOS app using `watchLink.wcSessionCommand`. The Watch app must be reachable (otherwise the error callback will be invoked).

```
watchLink.wcSessionCommand('updateContext', payload, error)
//      payload = <property-list-dictionary>
//      error = function(msg)
//              Invoked in the case of an error
```

Complication data transfers

A dictionary of values representing complication data may be transmitted from the iOS to the watchOS app. These transfers can occur in background (when the watchOS app is not reachable, i.e. the watchOS app is not in foreground).

Complication data transfers are sent as high priority user information transfers. They are delivered to the watchOS app via the user information transfer handler that has been bound.

Complication data transmission (iOS)

Complication data transfers are sent from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.sendComplicationInfo(complicationInfo);
//      Upon return, context.TIMESTAMP contains the unique
//      numeric timestamp that can be used to refer to the transfer
//      complicationInfo = <property-list-dictionary>
//      returns: true if transmission was initiated, false if Watch
//              is unavailable
```

watchLink.sendComplicationInfo adds the keys `TIMESTAMP` and `ISCOMPLICATION` to the complicationInfo object. `TIMESTAMP` contains a unique timestamp representing the transfer. Note that if complicationInfo contains existing keys `TIMESTAMP` or `ISCOMPLICATION` they will be overwritten. The `TIMESTAMP` value can be used to query the status of the transfer, and cancel the transfer if desired. The value of this key can be retrieved from the supplied complicationInfo object immediately upon return from the function invocation.

Note: the complicationInfo sent to the Swift layer is a clone of the object submitted to the sendComplicationInfo function.

Complication data transfer status and flushing (iOS)

A complication data transfer can be queried and/or cancelled using the `TIMESTAMP` set by `watchLink.sendUserInfo` or `updateUserInfoToPhone.`

A complication data transfer can be queried via `watchLink.queryComplication` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.queryComplication(timestamp, success, error)
//      success = function(transferInfo
//              transferInfo = { timestamp: <Number>,
                                 isComplication: <Boolean>,
                                 transmitComplete: <Boolean>,
//                               complicationInfo: <property-list-dictionary>
//              transferInfo = null if the transfer status is no longer available
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The in-progress complication data transfers can also be accessed from the iOS app using a Promise, as in

```
watchLink.queryComplication(userInfoID).then(success).catch(error)
```

A complication data transfer can be cancelled via `watchLink.cancelComplication` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.cancelComplication(userInfoID, success, error)
//      success = function(cancelled
//              cancelled = <Boolean>, true if the transfer was cancelled,
                        false otherwise
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

A information transfer can also be cancelled from the iOS app using a Promise, as in

```
watchLink.cancelComplication(userInfoID).then(success).catch(error)
```

The status of a complication data transfer will remain available until the transfer is complete and a subsequent complication data transfer is initiated.

The number of daily complications remaining in quota can be retrieved from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.queryComplicationQuota(success, error)
//      success = function(number
//              number = the number of complication transfers remaining
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The number of daily complications remaining can also be queried from the iOS app using a Promise, as in

```
watchLink.queryComplicationQuota().then(success).catch(error)
```

Complication data transfers outstanding (iOS)

The in-progress complication data transfers can be accessed via `watchLink.outstandingComplicationTransfers` from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.outstandingComplicationTransfers(success, error)
//      success = function(outstandingUserInfo
//              outstandingUserInfo = [
//                      { userInfoID: <Number>,
//                      isComplication: <Boolean>,
//                      transmitComplete: <Boolean>,
//                      complicationInfo: <property-list-dictionary> } ]
//      error = function(errorString
//              If the Watch is unavailable errorString will be "unavailable"
//              If the Watch session has not completed initialization errorString
//                      will be "uninitialized"
```

The in-progress complication data transfers can also be accessed from the iOS app using a Promise, as in

```
watchLink.outstandingComplicationTransfers().then(success).catch(error)
```

All outstanding complication transfers can be cancelled using `watchLink.flushComplicationTransfers().`

Scheduled local notifications

Local notifications can be scheduled by the iOS app and will be displayed on the iOS device and/or Watch, depending on which one is active. The following table describes the various possibilities.

iOS app	watchOS app	Notification type
Foreground, screen on	Foreground	iOS delegate
Background/off, screen on	Foreground	iOS alert
Foreground, screen off	Foreground	iOS banner + watchOS delegate
Background/off, screen off	Foreground	iOS banner + watchOS delegate
Foreground, screen on	Background	iOS delegate
Background/off, screen on	Background	iOS alert
Foreground, screen off	Background	iOS banner/watchOS alert
Background/off, screen off	Background	iOS banner/watchOS alert

The following notification types may be invoked:

- **iOS delegate:** Normally, this type of notification will not be shown because the iOS app is in the foreground. However, a notification delegate action can be specified which is invoked when the notification arises. As well, the delegated notifications can be set to be displayed (independent of whether a notification delegate action has been set).
- **iOS alert:** This type of notification is shown on the screen and in the notification center, with optional accompanying alert sound.
- **iOS banner:** This type of notification is shown in the notification center. There is no accompanying alert sound.
- **watchOS delegate:** Normally, this type of notification will not be shown because the watchOS app is in the foreground. However, a notification delegate action can be specified which is invoked when the notification arises. As well, the delegated notifications can be set to be displayed (independent of whether a notification delegate action has been set).

- **watchOS alert**: This type of notification is shown on the Watch face with the short look and long look.

Request notification permission

You app must request permission (typically at initialization) to post local notifications.

```
watchLink.requestNotificationPermission(allowSound, success, error)
//      allowSound = <Boolean>, whether to use sound with notifications
//      success = function(), invoked is the user gave permission
//      error = function(errorString), invoked if failed or the user refused
//              errorString describes the failure
```

Add a notification (iOS)

Notifications are scheduled from the iOS app using a traditional Cordova plugin call, as in

```
watchLink.scheduleNotification(trigger, payload, success, error)
//      trigger = <Number> |
//              { year: <Number>, month: <Number>, day: <Number>,
//                      hour: <Number>, minute: <Number>,
//                      second: <Number> }
//              If trigger is a number, the notification will occur
//                      in that number of seconds after the current time.
//              Otherwise, the date and time are used as the
//                      notification trigger.
//              The year, month, and day are optional and default to
//                      the current year, month
//                      and day if omitted
//              The hour, minute and second are optional and default to
//                      the current time if omitted
//      payload = { title: <String>, subtitle: <String>, body: <String>,
//                                      userInfo: <property-list-dictionary>
//              subtitle, body and userInfo are optional
//              userInfo is returned to notification delgate and/or
//                      notification handler to identify the notification
//      success = function(timestamp)
//              Invoked when the notification has been scheduled.
//              timestamp is the unique timestamp (as a string) created
//                      to identify the notification
//              The timestamp represents the Unix epoch (number of milliseconds
//                      since 1970/01/01) corresponding to the trigger
//      error = function(errorString)
//              Invoked when an error occurred
//                      errorString = string describing the error
//                      errorString will be "duplicate" in the case of a
//                              notificaton that already pending at the same
//                              time as the notification requested
```

Notifications can also be scheduled from the iOS app using a Promise, as in

```
watchLink.scheduleNotification(trigger, payload).then(success).catch(error)
```

Retrieve outstanding notifications (iOS)

Outstanding notifcations can be retrieved by the iOS app using a traditional Cordova plugin call, as in

```
watchLink.retrieveNotifications(success, error)
//      success = function(<notification-list>
//              <notification-list> = [ <property-list-dictionary> ]
//              Each element is the userInfo associated with a notification, with
//                      userInfo.TIMESTAMP set to the timestamp ID
//                              of the notification
//      error = function(errorString
//              Invoked when an error occurred
//                      errorString = string describing the error
```

Notifications can also be retrieved from the iOS app using a Promise, as in

```
watchLink.retrieveNotifications().then(success).catch(error)
```

Cancel a notification (iOS)

Notifications can be cancelled from the iOS app as follows:

```
watchLink.cancelNotification(timestamp)
//      timestamp is the value returned by scheduleNotification
```

Cancel all notifications (iOS)

All pending notifications can be cancelled from the iOS app as follows:

```
watchLink.cancelAllNotifications()
```

Define notification handler action (iOS)

A notification handler action can be defined to be invoked to process notifications that the user taps on.

```
watchLink.bindNotificationHandler(handler)
//      handler = function(userInfo)
//              Invoked with the userInfo associated with the notification
//              userInfo.TIMESTAMP is the timestamp ID of the notification
//              Use handler = null to cancel a previously bound handler
//              Otherwise handler will overwrite any previously bound handler
```

Define notification handler action (watchOS)

A notification handler action can be defined to be invoked to process notifications that the user taps on.

```
bindNotificationHandler(handler: (@escaping (String) -> Void)?)
//      handler = func (userInfo: [String: Any])
//              Invoked with the userInfo associated with the notification
//              userInfo["TIMESTAMP"] is the timestamp ID of the notification
//              Use nil to cancel a previously bound handler
//              Otherwise handler will overwrite any previously bound handler
```

Define notification delegate action (iOS)

A notification delegate action can be defined to be invoked to process notifications that arise when the app is in foreground and would normally not be shown.

```
watchLink.bindNotificationDelegate(handler)
//      handler = function(userInfo)
//              Invoked with the userInfo associated with the notification
//              userInfo.TIMESTAMP is the timestamp ID of the notification
//              Use handler = null to cancel a previously bound handler
//              Otherwise, handler will overwrite any previously bound handler
```

Define notification handler action (watchOS)

A notification delegate action can be defined to be invoked to process notifications that arise when the app is in foreground and would normally not be shown.

```
bindNotificationHandler(handler: (@escaping (String) -> Void)?)
//      handler = func (userInfo: [String: Any])
//              Invoked with the userInfo associated with the notification
//              userInfo["TIMESTAMP"] is the timestamp ID of the notification
```

```
//              Use nil to cancel a previously bound handler
//              Otherwise, handler will overwrite any previously bound handler
```

Display delegated notifications (iOS)

Delegated notifications will be shown or not (regardless of whether a notification delegate action has been defined). By default, they will not be shown.

```
watchLink.showDelegatedNotification(show)
//      show = <boolean>, set to true if delegated notifications should be shown
```

Display delegated notifications (watchOS)

Delegated notifications will be shown or not (regardless of whether a notification delegate action has been defined). By default, they will not be shown.

```
func showDelegatedNotification(show: Bool)
//      show = <Bool>, set to true if delegated notifications should be shown
```

Console log management

The plugin logging functions enable logs from the Swift layers of the iOS app and watchOS app to be shown on the Javascript console for the iOS app, as well as the Xcode console for both the iOS and watchOS app.

Log levels

There are four levels of log display:

- All logs displayed (level 3 or "all")
- Only application and error logs displayed (level 2 or "app")
- Only error logs displayed (level 1 or "error")
- No logs displayed (level 0 or "none") The level of log display can be controlled from the `config.xml` file (defaulting to 3 if unspecified) as well as dynamically from the iOS app.

```
config.xml
<preference name="watchLinkAppLogLevel" value="all">
<preference name="watchLinkWatchLogLevel" value="all">
<preference name="watchLinkWatchPrintLogLevel" value="all">
```

```
watchLink.appLogLevel(n)
//      set the log level for iOS Swift logs
//      n = 0, 1, 2, 3 or "none", "error", "app", "all"

watchLink.watchLogLevel(n)
//      set the log level for watchOS Swift logs to the Javascript console
//      n = 0, 1, 2, 3 or "none", "error", "app", "all"

watchLink.watchPrintLevel(n)
//      set the log level for watchOS Swift logs to the Xcode console
//      n = 0, 1, 2, 3 or "none", "error", "app", "all"

watchLink.JSlogLevel(n)
//      set the log level for Javascript logs to the Javascript console
//      n = 0, 1, 2, 3 or "none", "error", "app", "all"
```

Log timestamps

Log timestamps are generated locally (on the Watch, from iOS Swift or from iOS Javascript). There might be small differences in timestamps generated at exactly the same moment due to absence of clock synchronization, and therefore timestamps from different sources that seem close should be regarded as unreliable indicators of order of events.

Mute Javascript logs

Log display on the Javascript console can be muted and unmuted.

```
watchLink.muteLog()   // mute logs

watchLink.unmuteLog() // unmute logs
```

Issue Swift log messages (iOS)

Swift log messages are issued by the plugin communication framework. Other iOS Swift application code can access the logging capabilities as follows:

```
func swiftLog(_ msg: String)
//      Send the message msg to the iOS Xcode and Javascript consoles
//      Will appear as "[hh:mm:dd]>> msg"
//      Will be ignored if log level is "none", "app" or "error"

func swiftAppLog(_ msg: String)
//      Send the message msg to the iOS Xcode and Javascript consoles
//      Will appear as "[hh:mm:dd]App>> msg"
//      Will be ignored if log level is "none" or "error"
```

```
swiftErrorLog(_ msg: String)
//      Send the message msg to the iOS Xcode and Javascript consoles
//      Will appear as "[hh:mm:dd]Error>> msg"
//      Will be ignored if log level is "none"
```

Issue Javascript log messsages (iOS)

Log messages can be issued from Javascript, subject to the same filtering as Swift logs (as determined by `watchLink.JSlogLevel`). Logs are issued by watchlink.js using these functions.

```
// issue "[hh:mm:ss.mm] msg" to the iOS Xcode and Javascript consoles
watchLink.log(msg)

// issue "[hh:mm:ss.mm]App: msg" to the iOS Xcode and Javascript consoles
watchLink.AppLog(msg)

// issue "[hh:mm:ss.mm]Error: msg" to the iOS Xcode and Javascript consoles
watchLink.ErrorLog(msg)
```

Issue Swift log messages (watchOS)

Swift log messages can be issued from the watchOS Swift layer as follows:

```
func printLog(_ msg: String)
//      Send the message msg to the Watch Xcode console
//      Will appear as "Print[hh:mm:dd]>> msg"
//      Will be ignored if Watch print log level is "none", "app" or "error"

func printAppLog(_ msg: String)
//      Send the message msg to the Watch Xcode console
//      Will appear as "Print[hh:mm:dd]App>> msg"
//      Will be ignored if Watch print log level is "none" or "error"

func printErrorLog(_ msg: String)
//      Send the message msg to the Watch Xcode console
//      Will appear as "Print[hh:mm:dd]Error>> msg"
//      Will be ignored if Watch print log level is "none"

func watchLog(_ msg: String)
//      Send the message msg to the Watch Xcode and Javascript consoles
//      Will appear as "<<WATCH [hh:mm:dd]>> msg":
//      Will be ignored if Watch log level is "none", "app" or "error"

func watchAppLog(_ msg: String)
//      Send the message msg to the Watch Xcode and Javascript consoles
//      Will appear as "<<WATCH [hh:mm:dd]App>> msg"
```

```
//      Will be ignored if Watch log level is "none" or "error"

func watchErrorLog(_ msg: String)
//      Send the message msg to the Watch Xcode and Javascript consoles
//      Will appear as "<<WATCH [hh:mm:dd]Error>> msg"
//      Will be ignored if Watch log level is "none"
```

watchLink test app

The plugin includes a fully functional test app that illustrates the use of the watchLink messaging framework.

To run the app, copy and decompress the file `TestWatchLink.zip` and open the

`watchtarget/platforms/ios/TestWatchLink.xcodeproj`

Xcode file. Set the *Team* in each of the targets to your team identifier. The app is now ready to run.

Xcode limitations and quirks

There are some limitations using the Xcode simulator with iOS and watchOS companion apps.

- The simulator does not support user information transfers.
- The simulator allows a complication to be added to the watch screen but the iOS app will not see the complication as being available and updates to the complication will not be processed.

If you run the watchOS app from Xcode and then quit Xcode (disconnecting it from the watch) the watchOS app will not function properly for background processing. To restore correct operation you must quit the iOS app (swipe it up from the iPhone window display) and launch it again.

Test app iPhone screen

The iPhone screen provides buttons to exercise the following communication functions:

- Send a dictionary message to the Watch
- Send a data message to the watch
- Send a user information update to the watch
- Send an application context update to the watch
- Enable or disable acknowledgement for the above functions

- Use direct access to WCSession for the above functions
- Send a complication update to the watch
- Schedule a local notification
- Cancel a scheduled notification
- Cancel all scheduled notification
- List pending notifications
- Initiate a message session reset
- Clear the console The area below the buttons is a console where the test app posts results of communication actions and received communications.

The iPhone screen shot shows the button menu and the result of sending an acknowledged message to the Watch and receiving a message.

Test app Watch screen

The Watch screen provides a console where the test app posts results of communication actions and received communications. There are additional screens that provide buttons to exercise the following communication functions:

- Send a dictionary message to the Watch
- Send a data message to the watch
- Send a user information update to the watch
- Send an application context update to the watch
- Show the last received user information and aplication context transfers. Acknowledgement can be enabled or disbled acknowledgement for each of functions from their respective pages.

The Watch screen shots show the console screen reporting the receipt of the iPhone message and the message screen showing the transmission of an acknowledged message to the iPhone.

Testing via Javascript

The test app can be observed and exercised from the Javascript console. After launching the app, open the Safari Web Inspector console to view the logs generated by the test app activity.

You can also invoke functions to generate message traffic and notifications from the console. Acknowledgements will be used based on the checkbox setting.

```
testmsg(msg)
        // msg is an object (<property-list-dictionary>)

datamsg(msg)
        // msg is an array of 8-bit unsigned integers
        // which will be converted to an ArrayBuffer

testuserinfo(userinfo)
        // userinfo is an object (<property-list-dictionary>)

testcontext(context)
        // context is an object (<property-list-dictionary>)

testcomplication(userinfo)
        // userinfo is an object (<property-list-dictionary>)

notify(delay, title, subtitle, body, userInfo)
//      delay is the trigger
//      { title, subtitle, body, userInfo } is the payload
```

Annotated log example (iOS test app sends message to Watch)

The following logs appear in the Javascript console.

```
// test app calls watchLink.sendMsg
[11:59:52.636]App: msg {"a":223,"b":253,"TIMESTAMP":1613764792634}

// WatchLink.swift add message to outgoing queue
```

```
[11:59:52.639]>> Adding 1613764518401.1613764792634 ack:true
        TEST: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634]
        callbackID=WatchLink1676863945

// WatchLink.swift signals Watch becomes reachable
[12:00:00.077]>> Watch reachable

// WatchLink.swift services outgoing message queue
[12:00:00.078]>> Sending message 1613764518401.1613764792634 ack:true
        TEST: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634]

// WatchLink.swift indicates waiting for ack
[12:00:00.083]>> processQueue processing Messages:
        [(timestamp: 1613764792634, session: 1613764518401,
        ack: true, callbackId: "WatchLink1676863945",
        msgType: "TEST", msg: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634])]

// WatchLink.swift signals Watch app state changed to active
[12:00:00.145]>> Received message ["msgType": WATCHAPPACTIVE, "msgBody": { },
        "timestamp": 1613764799942, "session": 1613764518401]
[12:00:00.148]>> Watch Application State: ACTIVE

// WatchLinkExtensionDelegate.swift indicates message received
WATCH [12:00:00.224]>> Received message
        TEST: ["TIMESTAMP": 1613764792634, "b": 253, "a": 223]

// WatchLink.swift indicates message ack received
[12:00:00.248]>> Acknowledged 1613764792634 queue=[]

// test app indicates message acknowledged via .then())
[12:00:00.254]App: ACK msg 1613764792634

// WatchLinkExtensionDelegate.swift signalas to WatchLink.swift that
// Watch is moving to background (no longer reachable)
[12:00:16.196]>> Received message ["session": 1613764518401, "msgBody": { },
        "msgType": WATCHAPPBACKGROUND, "timestamp": 1613764816147]
[12:00:16.197]>> Watch Application State: BACKGROUND
[12:00:16.284]>> Watch NOT reachable
```

The following logs appear in the iOS Xcode console. Note that all of the Javascript console logs appear here as well.

```
// WatchLink.swift add message to outgoing queue
[11:59:52.639]>> Adding 1613764518401.1613764792634 ack:true
        TEST: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634]
        callbackID=WatchLink1676863945

// test app calls watchLink.sendMsg
2021-02-19 11:59:52.640177-0800 TestWatchLink[13900:7421264]
```

```
        [11:59:52.636]App: msg {"a":223,"b":253,"TIMESTAMP":1613764792634

// WatchLink.swift signals Watch becomes reachable
[12:00:00.077]>> Watch reachable

// WatchLink.swift services outgoing message queue
[12:00:00.078]>> Sending message 1613764518401.1613764792634 ack:true
        TEST: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634]

// WatchLink.swift indicates waiting for ack
[12:00:00.083]>> processQueue processing Messages:
        [(timestamp: 1613764792634, session: 1613764518401,
        ack: true, callbackId: "WatchLink1676863945",
        msgType: "TEST", msg: ["b": 253, "a": 223, "TIMESTAMP": 1613764792634])]

// WatchLink.swift signals Watch app state changed to active
        "timestamp": 1613764799942, "session": 1613764518401]
[12:00:00.148]>> Watch Application State: ACTIVE

// WatchLinkExtensionDelegate.swift indicates message received
WATCH [12:00:00.224]>> Received message
        TEST: ["TIMESTAMP": 1613764792634, "b": 253, "a": 223]

// WatchLink.swift indicates message ack received
[12:00:00.248]>> Acknowledged 1613764792634 queue=[]

// test app indicates message acknowledged via .then())
2021-02-19 12:00:00.256017-0800 TestWatchLink[13900:7421264]
        [12:00:00.254]App: ACK msg 1613764792634

// WatchLinkExtensionDelegate.swift signalas to WatchLink.swift that
// Watch is moving to background (no longer reachable)
[12:00:16.196]>> Received message ["session": 1613764518401,
        "msgBody": {}, "msgType": WATCHAPPBACKGROUND,
        "timestamp": 1613764816147]
[12:00:16.197]>> Watch Application State: BACKGROUND
[12:00:16.284]>> Watch NOT reachable
```

The following logs appear in the watchOS Xcode console.

```
// WatchLinkExtensionDelegate.swift signalas to WatchLink.swift that
// Watch is moving to foreground (reachable)
Print[11:59:59.942]>> Watch App Active
Print[11:59:59.943]>> Added message to queue timestamp: 1613764799942 ack:false
        WATCHAPPACTIVE: [:]
Print[11:59:59.943]>> Sending message
        ["timestamp": 1613764799942, "ack": false,
        "msgType": "WATCHAPPACTIVE", "msgBody": [:]]
```

```
// WatchLinkExtensionDelegate.swift sends log to WatchLink.swift
Print[12:00:00.212]>> Added message to queue timestamp: 1613764800211 ack:false
        WATCHLOG: ["msg": "Received message
        TEST: [\"TIMESTAMP\": 1613764792634, \"b\": 253, \"a\": 223]"]
Print[12:00:00.215]>> Sending message ["ack": false,
        "msgBody": ["msg": "Received message
        TEST: [\"TIMESTAMP\": 1613764792634, \"b\": 253, \"a\": 223]"],
                "timestamp": 1613764800211, "msgType": "WATCHLOG"]

// WatchLinkExtensionDelegate.swift prints log message received
Print[12:00:00.218]>> Received message TEST:
        ["TIMESTAMP": 1613764792634, "b": 253, "a": 223]

// watch app logs received message
Print[12:00:00.217]App>> Received TEST:
        ["TIMESTAMP": 1613764792634, "b": 253, "a": 223]

// WatchLinkExtensionDelegate.swift indicates message received
Print[12:00:00.218]>> Received message ["timestamp": 1613764792634,
        "msgBody": { TIMESTAMP = 1613764792634; a = 223; b = 253;},
        "session": 1613764518401, "msgType": TEST]

// WatchLinkExtensionDelegate.swift signals to WatchLink.swift that
// Watch is moving to background (no longer reachable)
Print[12:00:16.146]>> Watch App Background
Print[12:00:16.147]>> Added message to queue timestamp: 1613764816147
        ack:false WATCHAPPBACKGROUND: [:]
Print[12:00:16.147]>> Sending message ["ack": false,
        "msgBody": [:], "timestamp": 1613764816147,
        "msgType": "WATCHAPPBACKGROUND"]
```

Annotated log example (watchOS test app sends message to iPhone)

The following logs appear in the Javascript console.

```
// WatchLink.swift indicates message received
[12:01:17.501]>> Received message ["timestamp": 1613764877428, "msgType": TEST,
        "msgBody": {  A = 249; B = 42; }, "session": 1613764518401]

// test app indicates message received
[12:01:17.510]App: RCV msg {"B":42,"A":249}
```

The following logs appear in the iOS Xcode console. Note that all of the Javascript console logs appear here as well.

```
// WatchLink.swift indicates message received
[12:01:17.501]>> Received message ["timestamp": 1613764877428, "msgType": TEST,
```

```
        "msgBody": { A = 249; B = 42; }, "session": 1613764518401]

// test app indicates message received
2021-02-19 12:01:17.512309-0800 TestWatchLink[13900:7421264]
        [12:01:17.510]App: RCV msg {"B":42,"A":249}
```

The following logs appear in the watchOS Xcode console.

```
// WatchLinkExtensionDelegate.swift adds message to outgoing queue
Print[12:01:17.429]>> Added message to queue timestamp: 1613764877428 ack:true
        TEST: ["B": 42, "A": 249]

// WatchLinkExtensionDelegate.swift services outgoing message queue
Print[12:01:17.429]>> Sending message ["msgType": "TEST",
        "timestamp": 1613764877428,
        "ack": true, "msgBody": ["B": 42, "A": 249]]

// WatchLinkExtensionDelegate.swift indicates waiting for ack
Print[12:01:17.429]>> processQueue--processing Messages

// test app indicates message sent
Print[12:01:17.431]App>> msg timestamp=1613764877428

// WatchLinkExtensionDelegate.swift indicates ack received
Print[12:01:17.653]>> Acknowledged 1613764877428

// test app indicates ack received
Print[12:01:17.656]App>> ACK msg 1613764877428
```