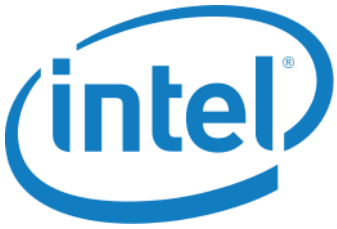
Intel Software Platform for Curie

iOS Developer's Guide

Version 1.0

December 2015

BETA DRAFT



Part Number 333551-001US

Contents

Chapter 3: Using the Wearable Platform Core APIs

Chapter 4: Using the Wearable Platform Body IQ APIs

Chapter 5: Using the Wearable Platform Social IQ APIs

Chapter 6: Using the Wearable Platform Time IQ APIs

Chapter 7: Using the Cloud Services SDK

Chapter 8: Third-Party License Information

List of Figures and Code Samples

Figures

Code Samples

Using the Wearable Platform Core APIs

Using the Cloud Services SDK

CHAPTER

1 About This Guide

This guide is your starting reference for developing iOS mobile applications for Curie-based wearable devices. The guide provides an architecture overview for Intel Software Platform for Curie, and detailed descriptions of supported functionality and APIs.

This chapter contains the following sections:

1.1 Who Should Read This Book

This guide is designed for iOS developers.

To use the Intel Curie Software Platform for Curie SDK, iOS developers should have proficiency in the Swift and Objective C programming languages, and experience developing mobile applications for the iOS mobile platform.

1.2 Terminology Used in This Guide

Table 1: Terms and Definitions

Term	Definition
BLE	Bluetooth Low Energy radio, a wireless personal area network technology
CRUD	Create, read, update, and delete operations
DAO	Data Access Object. Provides an abstract interface to a database or other persistence mechanism.
HTTP	Hypertext Transfer Protocol, an application protocol for hypermedia information systems.
HTTPS	Secure Hypertext Transfer Protocol, a protocol for secure communication over a computer network
iOS	Operating system used for Apple mobile devices
Jazzy	Program that generates HTML-based Swift API reference
JSON	JavaScript Object Notation, a lightweight data-interchange format
MOT	Mode of transport. In Time IQ, this can be walking, driving, or stationary.
PoI	Place of Interest Notable location of significance to the device wearer. This can be an internationally recognized location, or a location of importance to only the device wearer.
REST	Representational State Transfer, a software architectural style of the World Wide Web
TTL	Time IQ Time-To-Leave
SoC	Intel Quark SE System on a Chip
User State	Captures device wearer location and proprioception details. Examples of proprioception details are: device wearer is driving; device wearer is at home, work, or another PoI; device wearer is near a PoI.

Table 1: Terms and Definitions

Term	Definition
UUID	Universally Unique Identifier
Wearable Platform	Intel Software Platform for Curie The name is shortened in this guide for readability.

1.3 Additional Resources

The following documents are included in the Wearable Platform SDK to help you get started:

- *Wearable Platform API Reference for iOS*
- *Cloud Services Portal Administrator Guide*
- *Intel® Curie™ Platform Customer Reference Board (CRB) Hardware User Guide*
- *Intel® Curie™ Platform Hardware User's Guide*
- *Intel® Curie™ Platform Software User's Guide*

CHAPTER

2 Overview of Intel Software Platform for Curie

This chapter explains Intel Software Platform for Curie basic concepts. The chapter introduces Wearable Platform Core APIs, Intel® IQ Software kits, and Intel Cloud Services APIs. Together, these comprise the Intel Software Platform for Curie SDK.

Tip: In this guide, "Intel Software Platform for Curie" is sometimes called "Wearable Platform" for better readability.

This chapter contains the following sections:

2.1 About Intel Wearable Technology

Intel Wearable Technology is more than just smartwatches and step-counters. Today you can embed payment engines, medical health monitors, identity-based access systems –and so much more– into wearable devices. Wearable devices can be unobtrusive while delivering critical data and services to the device wearer.

Intel Wearable Technology houses the Intel Quark SE System on a Chip (SoC) which integrates the power of a full-sized computer into a single dime-sized chip. The hardware module includes a Bluetooth Low Energy radio (BLE), motion sensors, and battery-charging capabilities. The hardware runs on Intel Software Platform for Curie.

The small form factor Curie module combined with the Wearable Platform SDK is an attractive option for designers and developers who want to quickly turn innovative ideas into products. Intel Wearable Technology is designed to power both consumer wearable devices and industrial wearable devices.

2.1.1 Understanding the Major Components

Intel Wearable Technology encompasses the following:

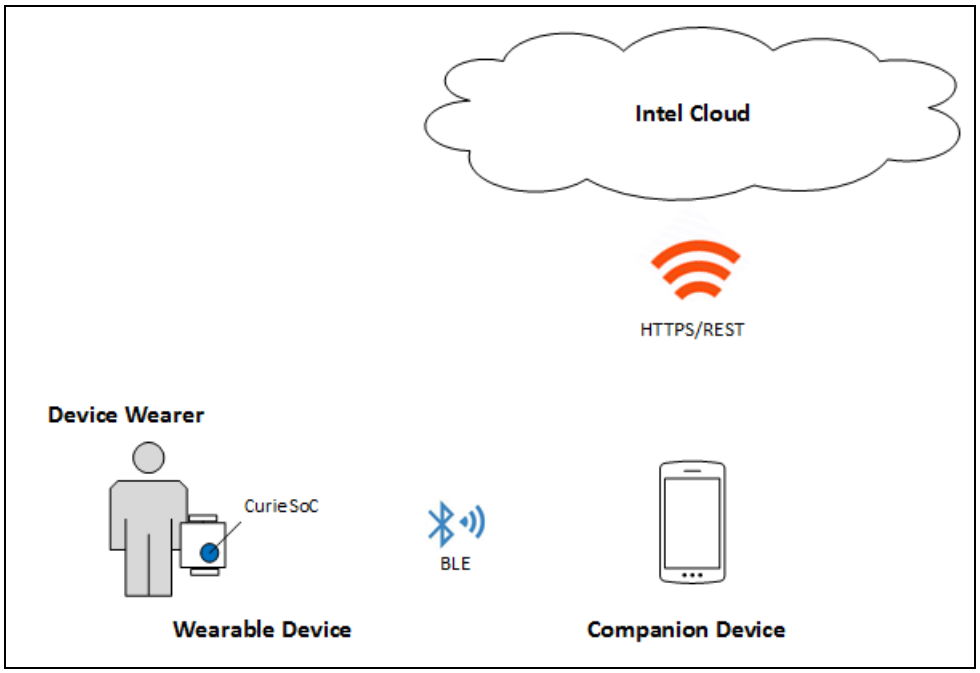
- A wearable device containing Intel Curie SoC
- A companion device running a mobile application created with Wearable Platform SDK
- Intel Cloud Services

The device wearer puts on a wearable device containing Intel Curie SoC. An application implemented with Wearable Platform SDK runs on a companion device, typically a mobile phone. The mobile application communicates with the wearable device over a BLE communication channel. The

companion device communicates with Intel Cloud Services using REST and HTTP(S) protocols.

The following image illustrates the communication flow among Intel Wearable Platform components.

Figure: Intel Wearable Platform Components



2.1.2 Intel Wearable Platform Architecture

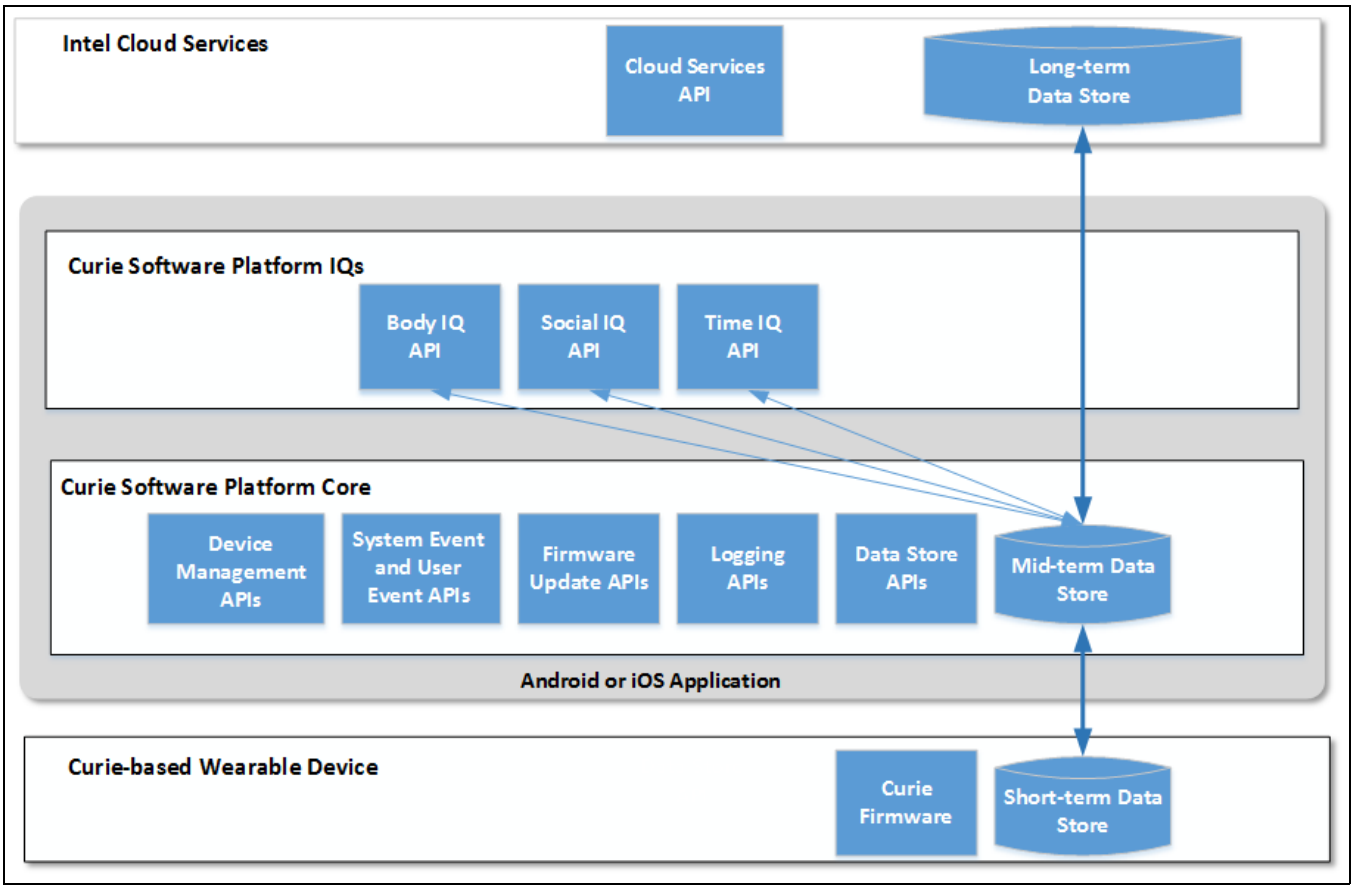
Intel Wearable Platform can be partitioned into three logical, binary components:

- Wearable Platform Core APIs
- IQ Software Kits
- Cloud Services APIs

You can innovate wearable device functionality by implementing one or more IQ Software Kits in your mobile application.

The following image illustrates the Wearable Platform SDK architecture:

Figure: Intel Wearable Platform Architecture



2.2 Wearable Platform Core APIs Overview

The following Wearable Platform Core APIs provide services useful to all mobile applications interacting with Curie-based wearable devices:

- Device Management APIs
- Firmware Update APIs
- System Event APIs
- User Event APIs

- Wearable Notification APIs
- Data Store APIs
- Logging APIs

2.2.1 Device Management APIs

The mobile application uses Device Management APIs to manage and communicate with the wearable device containing Intel Curie SoC. Device Management APIs provide the following operations:

- Scan and discover a Curie-based wearable device
- Pair with the Curie-based wearable device
- Unpair from the Curie-based wearable device
- Connect to the Curie-based wearable device
- Disconnect from the Curie-based wearable device
- Monitor the Curie-based wearable device connection status
- Monitor the Curie-based wearable device battery status
- Perform factory reset of the Curie-based wearable device
- Synchronize the clock between the Curie-based wearable device and the companion device

Note that clock synchronization between the Curie-based wearable device and the companion device is performed automatically each time the two devices connect, and is refreshed every 24 hours.

2.2.2 Firmware Update APIs

The mobile application uses Firmware Update APIs to programmatically check and update firmware on the

wearable device containing Intel Curie SoC. The Firmware Update APIs provide the following operations:

- Determine what version of firmware is currently installed on the Curie-based wearable device
- Install new firmware revision on the Curie-based wearable device
- Track progress of the firmware update on the Curie-based wearable device

2.2.3 System Event APIs

This set of APIs provides details of system events such as device boot, crash, shutdown, or low battery. The mobile application can monitor for any of these events after the Curie-based wearable device is successfully connected to the companion device.

Each system event is summarized as a four-tuple that includes the following:

- Universally unique identifier (UUID) of the wearable device issuing the event
- UUID of the wearable device user identity
- Type of the system event:
  - Main boot event
  - Boot event after the firmware update
  - Crash event
  - Low battery event
  - Shutdown event triggered by the user
  - Shutdown event triggered due to critically low battery
- Timestamp indicating when the event occurred

2.2.4 User Event APIs

The User Event APIs provide details of events issued by the user, such as a user tapping or activating a button on the wearable device. The mobile application can monitor for these events after the Curie-based wearable device is successfully connected to the companion device.

Each system event is summarized as a four-tuple that includes the following:

- UUID of the wearable device
- UUID of the wearable device user identity issuing the event.
- Type of the user event:
  - Double-tap
  - Triple tap
  - Single button press

Note: Triple button press events are not supported in this release of Intel Software Platform for Curie. The definition is reserved for future use.

2.2.5 Wearable Notification APIs

The Notification APIs support constructing and sending LED and vibration pattern notifications from the companion device to the Curie-based wearable device. The mobile application can specify LED pattern and/or vibration pattern as well as the time interval between the notifications.

Vibration Pattern specifications include:

- Vibration type
  Two types are currently supported: square and special effects
- Amplitude
  Ranges from 0 (no vibration) to 255 (full strength vibration)

- Duration pattern
  Specifies the ON/OFF duration
- Repetition count
  Specifies the number of times the pattern is repeated

LED Pattern specifications include:

- LED types are LEDBlink and LEDWave
- RGB color list specification
  Color code configuration for each notification
- Intensity specifies LED notification brightness.
  Ranges from 1 to 255.
- Duration pattern specifies the ON/OFF duration
- Repetition count specifies the number of times the pattern is repeated. Supports 1 to 255 repetitions.

2.2.6 Data Store APIs

Wearable Platform Core APIs provide a way to persist data in the local data store. Entities can be created, retrieved, updated, and deleted (CRUD). The APIs support CRUD operations on wearable device details and the user identity details.

Wearable device details data model includes:

- Unique identifier
- Bluetooth address
- Display name
- Manufacturer
- Model
- Serial number
- Firmware revision
- Software revision
- Hardware revision

User identity details data model includes:

- Unique identifier
- User ID
  Authenticated user ID provided by the Cloud Authentication services
- First name
- Last name
- Email address
- Phone number associated with the companion device

2.2.7 Logging APIs

Wearable Platform Core APIs provide logging capabilities supporting multiple log level granularities.

Log levels can be set as follows:

- ERROR
  Designates error events that might still allow the application to continue running.
- WARN
  Designates warnings and potentially harmful error events.
- INFO
  Designates informational messages that trace application at coarse-grained level.
- DEBUG
  Designates fine-grained information events that are the most useful in application debugging.
- VERBOSE
  Designates finer-grained informational events than the DEBUG level.
- ASSERT
  Used to check whether mandatory pre-conditions met.

The order in terms of verbosity, from least to most is ERROR, WARN, INFO, DEBUG, VERBOSE.

2.2.8 Error Code APIs

This API set provides error codes along with machine-readable and human-readable error descriptions other Wearable Platform Core APIs may return when an operation fails. Error codes include BLE errors, authentication errors, HTTP errors, and many others.

2.3 IQ Software Kits Overview

Intel® IQ Software kits include the embedded software that runs on the Intel® Curie™ module together with companion smartphone applications and associated cloud capabilities.

Each IQ Software Kit provides APIs with specialized functionality capabilities:

- Body IQ APIs
- Social IQ APIs
- Time IQ APIs

2.3.1 Body IQ APIs

Body IQ APIs provide mobile applications with capabilities related to detecting and measuring body activities recorded by the wearable device, including the following:

- Real-time motion activity detection and classification Activities can be categorized as walking, running or cycling.

- Measurement of step count for walking and running activities
- Measurement of distance covered while walking, running or biking
- Measurement of number of calories burned during the activity

2.3.2 Social IQ APIs

Social IQ APIs provide mobile applications with capabilities to send notifications to the wearable device, including the following:

- Configuring and sending LED color patterns in response to a social network event, an incoming phone call, SMS, etc.
- Configuring and sending haptic in the form of vibration patterns in response to a social network event an incoming phone call, SMS, etc.

These capabilities are provided by the Wearable Notification APIs on page 11.

2.3.3 Time IQ APIs

Time IQ APIs provide a diverse set of context-aware capabilities to help consumers with time management, multi-tasking, arriving on time to targeted destinations, and other daily routine optimization. Time IQ APIs include management of the following entities:

- Calendar events
  Meetings and appointments recorded in the device wearer calendar.
- Reminders
  Calendar reminders, Time-To-Leave (TTL) cues in

order to arrive to the next destination on time, task reminders, and so forth.

- Places of Interest (PoI)
  Notable location of significance to the device wearer. This can be an internationally recognized location, or a location of importance to only the device wearer.
- User state
  Captures device wearer location and proprioception details. Examples of proprioception details are: device wearer is driving; device wearer is at home, work, or another PoI; device wearer is near a PoI.

2.4 Intel Cloud Services APIs Overview

The Cloud Services APIs provide a communication layer between the companion device and the Intel Cloud Service. Functionality includes:

- User authentication, cloud request scheduling, cloud response processing and error handling functionality used by all of the other Cloud Services SDK modules.
- Storage and retrieval of custom JavaScript Object Notation (JSON) documents that contain details about the authenticated user.
- Upload, storage and retrieval capabilities for general JSON documents.
- Upload and download capabilities for BLOB data and software assets.

For more information about Intel Cloud Services SDK, see Using the Cloud Services SDK on page 95.

2.5 Benefits of Using Intel Software Platform for Curie

Intel Software Platform for Curie is designed to facilitate application development so you can quickly deploy an extendable and forward-compatible wearable product.

- The platform is comprehensive and supports all currently defined Curie IQ APIs.
- You can extend support to new Curie IQ APIs as they become available.
- The Wearable Platform Core APIs provide services common to all Curie IQ APIs, eliminating redundant implementations.

CHAPTER

3 Using the Wearable Platform Core APIs

The Wearable Platform Core APIs implement services essential to managing and communicating with all Curie-based wearable devices regardless of Curie IQ.

This chapter contains the following sections:

3.1 Adding Core APIs to Your Application

Before accessing any of the services Wearable Platform Core APIs provide, you must add `IQCore.framework` inside your project.

3.2 Using the Device Management APIs

Device Management APIs are part of the Wearable Platform Core SDK for iOS mobile platform.

Device Management APIs support operations for managing and communicating with all Curie-based wearable devices. For a summary of Management API capabilities, see Device Management APIs on page 9.

Tip: BLE is also known as Bluetooth Smart or Version 4.0+ of the Bluetooth specification.

The companion device (the mobile phone) communicates with the Curie-based wearable device using Bluetooth Low Energy (BLE) protocol. BLE provides significantly lower power consumption for both the wearable device and the companion mobile phone.

To establish a connection with the Curie-based wearable device, the mobile application must follow standardized development steps consisting of:

1. Scanning for nearby BLE-enabled devices
2. Selecting the device of interest
3. Pairing with and connecting to the wearable to the companion device

Device Management APIs further simplify this process.

This section contains the following examples which demonstrate how Device Management APIs can be used in your applications.

3.2.1 To Start a Wearable Device Scan

Perform the following steps to discover the Curie-based wearable device located in the close proximity:

Step 1. Import `IQCore.framework`.

Step 2. Create an instance of `WearableScannerType` with the `WearableScanner.scannerForProtocol` class method.

Step 3. Use the scanner `startScan` method to perform a scan. Use the scanner 3 closures to get wearable tokens, handle errors, and detect when scanning is finished.

Step 4. To customize how your application handles wearable device detection, the application should override two `of IWearableScannerListener` methods:

- `onWearableFound`
  This method is called each time a new BLE device is discovered. Overwrite this method to specify a set of actions to take when the BLE device is detected.
- `onScannerError`
  This method is called if an error has occurred during the scan. Overwrite this method to specify a set of actions to handle any errors that may occur during the course of device scanning.

You can include the following code sample in your application to start scanning for Curie-based devices. See the example on the next page.

Example: **Starting a Scan for a Wearable Device**

```
import IQCore

let scanner: WearableScannerType = WearableScanner.scannerForProtocol(.BLE)

scanner.startScan({ (tokenResult) in
            print("Found token \(tokenResult)")
            },
            error: { (error) in
}
```

In this example, when the new wearable device is discovered, the corresponding `WearableToken` is created and returned to the device wearer through token result closure. Depending upon your needs you can use this closure to populate table view data source, filter for specific token, and so forth.

3.2.2 To Stop a Wearable Device Scan

To abort wearable device discovery, call the `scanner.stopScan()` method, referencing the same scanner as in the example above.

3.2.3 About the Wearable Token

Intel Wearable Platform Core API defines the `WearableToken` class to uniquely identify any Intel wearable device and to track it from inception. When the wearable device is initially discovered during the scan, the scanner creates a new corresponding `WearableToken` to be associated with that wearable device. It contains a unique ID, address, and name of the wearable device. From the initial discovery onwards to pairing, connecting, or any other interaction with the wearable device, the `WearableToken` is used to reference it.

3.2.4 Obtaining the Wearable Controller

Once the wearable device is successfully discovered and is associated with an instance of the `WearableToken` class, your application can start communicating with that wearable device using an instance of a class adopting the `IWearableController` protocol. Intel Wearable Platform has defined the `IWearableController` protocol to assist applications with communicating and managing the wearable device, and providing support for a wide array of operations, including:

- `connect()`
  Establishes a connection with the wearable device
- `disconnect()`
  Disconnects from the wearable device
- `factoryReset()`
  Initiates a factory reset of the wearable device
- `getBatteryLevel(`
  Reads the current battery status, which is reported in the related method of supplied `IWearableControllerListener.`

Use the `WearableController.controllerForToken()` method to obtain an instance of the Wearable Controller corresponding to the wearable device detected during the scan.

Supply as the first parameter the `WearableToken` obtained during the scanning process. See the following example.

***Example:* Obtaining a Wearable Controller Instance**

```
import IQCore
...
    var wearableController: WearableControllerType?

    init(wearableToken: WearableToken) {
    wearableController = WearableController.controllerForToken(token)
    }
...
```

Attach the Wearable Controller to the wearable device before performing any operations with the wearable device. From that point on, use the Wearable Controller to control

the wearable device, including connectivity to a companion application running on the mobile device, clock synchronization, battery monitoring, message exchange, and so forth.

3.2.5 To Connect to the Wearable Device

To connect wearable device to the companion device, call the `wearableController.connect()` method.

Example: **Connecting Wearable and Companion Devices**

```
wearableController.connect({ [weak self] (wearableToken, error) -> () in
    if error == nil {
        print("Device connected with token \(wearableToken)")
    } else {
        print("Device failed to connect with error \(error)")
    }
})
```

3.2.6 To Disconnect the Wearable Device

To disconnect wearable device from the companion device, call the `wearableController.disconnect()`method.

3.2.7 To Monitor Battery Level Changes

To obtain the current measurement of the battery level on the wearable device, call the `wearableController.subscribeToBatteryLevelUpdates()` method. The following examples demonstrate how to connect, disconnect, and monitor connection status and battery level changes.

Example: **Obtaining Current Battery Level**

```
// Monitor battery level changes
wearableController.subscribeToBatteryLevelUpdates({ [weak self] (batteryLevel, error) -> ()
in print("Battery level updated to \(batteryLevel)")
})

// Stop monitoring battery level change
wearableController.unsubscribeToBatteryLevelUpdates()

// Monitor connection status changes
wearableController.addConnectionStatusObserver({ [weak self] (status) in
    print("Connection status changed to \(status)")
})
```

3.2.8 To Set the Wearable Device Name

To set the name for the wearable device, call the `wearableController.setWearableName("Wearable device name")` method.

3.2.9 To Perform a Factory Reset of a Wearable Device

To perform a factory reset on the wearable device, call the `wearableController.factoryReset()` method.

3.3 Using the Firmware Update APIs

Firmware Update APIs are part of Wearable Platform Core SDK for the iOS mobile platform.

Firmware Update APIs are provided with the `FirmwareControllerType` protocol. The APIs support operations for reading the latest available firmware version available of the wearable device, as well as updating it to a new revision.

3.3.1 Obtaining the Firmware Controller

Use the `FirmwareController.controllerForWearable()` method to obtain an instance of the `FirmwareControllerType` associated with the wearable device (after it has been connected to the companion device).

Example: **Obtaining the Firmware Controller**

```
import IQCore
...
    var wearableController: WearableControllerType
    var firmwareController: FirmwareControllerType
...
    firmwareController = FirmwareController.controllerForWearable(wearableController)
...
```

3.3.2 To Read the Wearable Device Firmware Version

To get a string representation of the latest version of firmware installed on the wearable device, call the `wearableController.WearableIdentity.firmwareRevision` property.

3.3.3 To Update the Wearable Device Firmware

To install firmware on the wearable device use the `installFirmware()` method on `FirmwareControllerType`.

Supply as the first parameter a string representing the file system path for the firmware version.

Supply as the second parameter callback implementing the `IWearableListener` protocol to monitor status of the firmware file transfer onto the wearable device as it progresses. The remaining parameters are closures for monitoring progress, and completion for the firmware `installationInstalling Firmware` on the wearable device.

Example: **Installing Firmware on the Wearable Device.**

```
firmwareController.installFirmware(pathToFirmwareFileString, started: {
           print(“Started firmware update")
            }, progress: { [weak self] bytesSent, bytesTotal in
                    let progress = (Float)(bytesSent)/(Float)(bytesTotal)
                    var percentComplete = (Int64)(progress * 100.0)

           print("Progress update \(percent complete)")
            }, completed: { [weak self] version in
                      print("Installation completed. Now at version: \(version)")
            }, failed: { error in
                      print("Installation failed: \(error.localizedDescription)")
        })
```

The firmware file containing a new revision of wearable device firmware can be obtained using Intel Cloud Services. See Using the Cloud Services SDK on page 95.

3.4 Using the System Event APIs

System Event APIs are provided in the `IQCore.framework` package of the Wearable Platform Core SDK for the iOS mobile platform.

The System Event APIs provide details of system events such as device boot, crash, shutdown or low battery.

After the Curie-based wearable device has been successfully connected to the companion device, the mobile application can monitor system events using `SystemEventsControllerType` class.

This section contains the following topics:

- To Obtain the System Event Controller
- To Subscribe to System Events
- To Unsubscribe from System Events
- Putting It All Together

3.4.1 To Obtain the System Event Controller

First get a `SystemEventsControllerType` so you can subscribe and unsubscribe to System Events.

Example: **Obtaining a System Event Controller**

```
import IQCore

    private var wearableController: WearableControllerType!
    private var systemEventsController: SystemEventsControllerType!

    self. SystemEventsControllerType =
                  SystemEventsController.controllerForWearable(wearableController!)
```

From this point on, your mobile application can subscribe to system events.

3.4.2 To Subscribe to System Events

To subscribe to receiving system events as they occur on the connected wearable device, the application calls the `subscribe(success: () -> Void, failure: (NSError) -> Void, receiver: SystemEventReceived)` method. The `SystemEventReceived` callback function is invoked as system events are received from the wearable device.

Example: **Subscribing to System Events**

```
self.systemEventsController?.subscribe({ [weak self] () -> Void In
          print("System Event subscribe completed")

   }, failure: { (error) -> Void In

          print("System Event subscribe failed \(error)")
   }, receiver: { [weak self] (event) -> () in

          print("System Event \(error)")

   })
```

3.4.3 To Unsubscribe from System Events

To unsubscribe from receiving system events from the connected wearable device, call the `unsubscribe()` method.

3.4.4 Putting It All Together

System events are `WearableSystemEvent` objects. All have a timestamp of type `NSDate`. Each has a subclass based on the type of system event:

- `WearableSystemBootEvent`
  The device was Rebooted. The `WearableSystemBootEvent` information can be retrieved from `event.bootEvent`.

- `WearableSystemShutdownEvent`
  The device shutdown. The `WearableSystemShutdownEvent` information can be retrieved from `event.shutdownEvent`.
- `WearableSystemCrashEvent`
  The device shutdown. The `WearableSystemShutdownEvent` information can be retrieved from `event.shutdownEvent`.
- `lowBatteryEvent`
  The device has a low battery. The `lowBatteryEvent` information can be retrieved from `event.lowBatteryEvent`.

The following example demonstrates how System Event APIs can be used in your applications. See the example on the next page.

Example: Using the System Event APIs

```
import IQCore

    private var systemEventsController: SystemEventsControllerType?

    func subscribeToSystemEvents () {

        if let = self.systemEventsController {

        } else {
            self.systemEventsController =
                   SystemEventsController.controllerForWearable(self.wearableController!)
        }

        if self.subscribedToSystemEvents {
            self.systemEventsController?.unsubscribe()
            self.subscribedToSystemEvents = false
        }
        else {
            self.systemEventsController?.subscribe({ [weak self] in
                   print("System Event subscribe completed")

               }, failure: { (error) in
                      print("System Event subscribe failed \(error)")
               }, receiver: { [weak self] (event) in

                    let formatter = NSDateFormatter()
                    formatter.dateStyle = .ShortStyle
                    formatter.timeStyle = .ShortStyle
                    print(event.timeStamp)
                    let time = formatter.stringFromDate(event.timeStamp)
                    var subText: String = "\(time): "
                    if event.bootEvent != nil {
                        subText += "Boot event"
                    } elseif event.shutdownEvent != nil {
                        subText += "Shutdown event"
                    } elseif event.crashEvent != nil {
                        subText += "Crash event"
                    } elseif event.lowBatteryEvent != nil {
                        subText += "Low battery event"
                    }

                   print ("New System Event: \( subText)")
            })
        }
    }

func unsubscribeToSystemEvents () {

     self.systemEventsController?.subscribe()

}
```

3.5 Using the User Event APIs

User Event APIs are provided in the `IQCore.framework`.

The User Event APIs provides details of user events such as the user double-tapping the wearable device, the user triple-tapping the wearable device, or the user pressing the wearable device button.

After the Curie-based wearable device has been successfully connected to the companion device, the mobile application can monitor user events using the `UserEventController` class.

This section includes the following topics:

- To Initialize the UserEvents Controller
- About Gesture and Button Events
- To Unsubscribe from User Events
- User Events Code Sample

3.5.1 To Initialize the UserEvents Controller

To set up the `UserEventsController`, use the token returned when connecting to the wearable device.

Example: **Initializing the UserEvents Controller**

```
import IQCore


    private var userEventController: UserEventsControllerType?
    self.userEventController = UserEventsController.controllerForWearable(controller)
```

3.5.2 About Gesture and Button Events

Each user event, as described using the `WearableUserEvent class`, contains details of each user event.

3.5.2.1 Gesture Events

If the event is a gesture event it will be a `WearableUserEventGesture` type.

Example: **Returning a Gesture Event**

```
if let gesture = event.gestureEvent {
}
```

Gesture events can be "Double Tap" and "Triple Tap".

Example: **Getting Gesture Type**

```
if let gesture = event.gestureEvent {

            var gestureType: String?
            switch gesture.type {
            case .DoubleTap:
                gestureType = "Double Tap"
            case .TripleTap:
                gestureType = "Triple Tap"
            default:
                gestureType = "Unknown"
            }
        }
```

3.5.2.2 Button Events

If the event is a button event it will be a `WearableUserEventButton` type

Example: **Returning a Button Event**

```
if let button = event.buttonEvent {
}
```

Button Events can be "Single Press", "Double Press", "Triple Press", or "Long Press".

Note: Triple-Press events are not supported in this release of Intel Software Platform for Curie. The definition is reserved for future use.

See the following example.

Example: **Getting Button Press Type**

```
        if let tap = event.buttonEvent {

            var tapType: String?
            switch tap.type {
            case .SinglePress:
                tapType = "Single Press"
            case .DoublePress:
                tapType = "Double Press"
            case .TriplePress:
                tapType = "Triple Press"
            case .LongPress:
                tapType = "Long Press"
            default:
                tapType = "Unknown"
            }
        }
```

3.5.3 To Subscribe to User Events

To subscribe to receiving user events on the connected wearable device as they occur, the application calls the `UserEventController.subscribe(IWearableUserEventListener listener)` method. The listener is the callback function, invoked as the user events are received from the wearable device.

Example: **Subscribing to User Events**

```
userEventController?.subscribe({ [weak self] () -> Void in print("subscribed")
            }, failure: { [weak self] (error) -> Void in print(error)
            }) { [weak self] (event) -> () in print("event Received")
        }
```

3.5.4 To Unsubscribe from User Events

To unsubscribe from receiving user events from the connected wearable device, call the `UserEventController.unsubscribe()` method.

3.5.5 User Events Code Sample

The following example demonstrates how User Event APIs can be used in your applications. See example on the next page.

Example: Using the User Event APIs

```
import IQCore

   //after wearable device is discovered and connected to
    private var userEventController: UserEventsControllerType?
    self.userEventController = UserEventsController.controllerForWearable(controller)


    func subscribe() {
        userEventController?.subscribe({ [weak self] () -> Void in print("subscribed")
            }, failure: { [weak self] (error) -> Void in print(error)
            }) { [weak self] (event) -> () in
           processEvent(event)
        }
    }
    func unsubscribe() {
        userEventController?.unsubscribe()
    }

    func processEvent(event : WearableUserEvent) {
        if let gesture = event.gestureEvent {

            var gestureType: String?
            switch gesture.type {
            case .DoubleTap:
                gestureType = "Double Tap"
            case .TripleTap:
                gestureType = "Triple Tap"
            default:
                gestureType = "Unknown"
            }
        }
        else if let tap = event.buttonEvent {

            var tapType: String?
            switch tap.type {
            case .SinglePress:
                tapType = "Single Press"
            case .DoublePress:
                tapType = "Double Press"
            case .TriplePress:
                tapType = "Triple Press"
            case .LongPress:
                tapType = "Long Press"

            default:
                tapType = "Unknown"
            }
        }
    }
```

3.6 Using the Wearable Notification APIs

Wearable Notification APIs are provided in the `IQCore.framework` package of the Wearable Platform Core SDK for the iOS mobile platform.

The Wearable Notification APIs support constructing and sending LED display pattern notifications and vibration pattern notifications from the companion device to the Curie-based wearable device.

3.6.1 Obtaining the Notification Controller

Use the `wearableController.getNotificationController()` method to obtain an instance of the `INotificationController` associated with the wearable device (after it has been connected to the companion device).

Example: **Obtaining the Notification Controller**

```
import IQCore

   private var wearableController: WearableControllerType!
   private var notificationController: NotificationControllerType!

   self.notificationController = NotificationController.
                                                 controllerForWearable(wearableController)
```

From this point on, your application can transmit notifications of choice by calling the `notificationController.sendNotification` method.

Example: **Transmitting Notifications**

```
notificationController.sendNotification(notification, success: { () -> Void In
                    Log.verbose?.message("Sent Notification: \(notification)")
                }, failure: { (error) -> Void In
                    Log.error?.message("Failed to send Notification: \(error)")
            })
```

3.6.2 WearableNotification Calls

There are three different calls to initialize a `WearableNotification` depending on whether you want to use LED, vibration, or both to notify the device wearer. Using both LED and vibration can mimic the other two calls. `NotificationVibrationPattern` and `NotificationLedPattern` are optional. The following summarize the three initialization calls:

- Notification for vibration
  This takes a `NotificationVibrationPattern` pattern which is described below.

Example: **Initializing Notification for Vibration**

```
let notification: WearableNotification =
    WearableNotification(vibrationPattern: vibra, delay: 0)
```

- Notification for LED
  This takes a `NotificationLedPattern` pattern which is described below.

Example: **Initializing Notification for LED**

```
let notification: WearableNotification = WearableNotification(ledPattern: leds, delay: 0)
```

- Notification for both
  This takes both a `NotificationLedPattern` pattern and a `NotificationVibrationPattern` which are described below.

Example: **Initializing Notification for both LED and Vibration**

```
let notification: WearableNotification =
    WearableNotification(ledPattern: leds?, vibrationPattern: vibra?, delay: 0)
```

3.6.3 Creating an LED Pattern

To transmit LED notifications to the wearable device, construct an object of class `NotificationLedPattern` and

supply it as a parameter to the `WearableNotification` class constructor.

The Wearable Notification APIs supply a set of pre-defined LED pattern types from which you can choose including the following:

- LED_BLINK
  You can also specify other attributes such as pattern color scheme, intensity, repetition count and duration.
- LED_WAVE
  You can also specify other attributes such as pattern color scheme, intensity, repetition count and duration.
- Intensity
  Specifies LED notification brightness which ranges from 0 - 255.
- Repetition count
  Specifies the number of times the pattern is repeated. Each repeat count processes all of the duration / color patterns in the LEDPattern.
- Duration / Color
  There can be multiple Duration / Colors in a pattern, each pattern can be the same or different.

A duration pattern includes ON DURATION and OFF DURATION. It is specified as in the following example, where

  – `durationOn` is the number of milliseconds the LED is active.
  – `durationOff` is the number of milliseconds the LED is inactive.

Example: **Specifying a Notification Duration Pattern**

```
NotificationDurationPattern(on: durationOn, off: durationOff)
```

Duration patterns are added to the `NotificationLedPattern usingdurationPatterns.append`.

Duration color is a `UIColor` object. There should be one `UIColor` for each `NotificationDurationPattern`. These are added to the `NotificationLedPattern` using `colors.append`.

The following is an example of adding durations and colors to the `LedPattern`.

Example: **Specifying LED Durations and Colors**

```
        let led = NotificationLedPattern(type: .LEDBlink)
        led.intensity = 130
        led.repititionCount = 3
        let duration = NotificationDurationPattern(
                on: NSTimeInterval(0.5), // NSTimeInterval is typically in seconds
                off: NSTimeInterval(0.5)
            )
        led.durationPatterns.append(duration)
        led.colors.append(UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0))
```

3.6.4 To Specify a Vibration Pattern

To transmit vibration notifications to the wearable device, construct an object of class `WearableNotification.VibrationPattern` and supply it as a parameter to the `WearableNotification` class constructor.

Two types of vibration patterns are currently supported: square and special effects. Each type can be created using a corresponding `NotificationVibrationPattern` class constructor.

The vibration type is specified as the parameter to the constructor of `NotificationVibrationPattern` class. The Wearable Notification API supplies a set of pre-defined vibration pattern types from which you can choose, including the following types:

- VIBRA_SQUARE

Example: **Specifying a Vibration Square Pattern**

```
let pattern = NotificationVibrationPattern(type: .Square)
```

- VIBRA_SPECIAL_EFFECTS

***Example:* Specifying a Vibration Special Effects Pattern**

```
let pattern = NotificationVibrationPattern(type: .SpecialEffects)
```

- Amplitude
  Ranges from 0 (no vibration) to 255 (full strength vibration)

***Example:* Specifying a Vibration Amplitude**

```
pattern.amplitude = 127
```

- Repetition count
  Specifies the number of times the pattern is repeated. Each repeat count processes all of the duration / color patterns in the VibrationPattern.

***Example:* Specifying a Vibration Repetition Count**

```
pattern.repititionCount = 3
```

- Duration
  There can be multiple Durations in a pattern, each pattern can be the same or different.

***Example:* Specifying a Vibration Duration**

```
let duration1 = NotificationDurationPattern(
                on: NSTimeInterval(0.5),
                off: NSTimeInterval(0.5)
            )
```

A duration pattern includes ON DURATION and OFF DURATION. It is specified as in the following call, where:

- `durationOn` is the number of milliseconds the LED is active
- `durationOff` is the number of milliseconds the LED is inactive.

Duration patterns are added to the `NotificationLedPattern usingdurationPatterns.append`. Duration pattern, ON/OFF duration, is specified using the `NotificationDurationPattern(on: durationOn, off: durationOff)` call.

Like with LED, there can be multiple durations in a vibration pattern. Each duration is added separately using the durationPatterns.append method. See the following example.

Example: **Specifying a Vibration Notification**

```
let pattern = NotificationVibrationPattern(type: .Square)
pattern.amplitude = 127
pattern.repititionCount = 3
let duration1 = NotificationDurationPattern(
    on: NSTimeInterval(0.5),
    off: NSTimeInterval(0.5))
pattern.durationPatterns.append(duration1)
```

3.6.5 Using LED and Vibration Patterns in One Notification

Once you have a `NotificationLedPattern` and/or a `NotificationVibrationPattern`, You can create a `WearableNotification`, which is sent to the `NotificationControllerType` to make the device light up and vibrate.

Example: **Using LED and Vibration in One Notification**

```
let vibra = vibraPattern()
let leds = ledPattern()
let notification: WearableNotification = WearableNotification(ledPattern: leds,
    vibrationPattern: vibra, delay: 0)

if notification.isValid {
    notificationController.sendNotification(notification, success: { () -> Void in
        print("Sent Notification: \(notification)")
    }, failure: { (error) -> Void in
        print("Failed to send Notification: \(error)")
    })
}
```

In the example, notice the `isValid` question. It will return `true` if the notification is set up correctly. `NotificationLedPattern` and `NotificationVibrationPattern` objects have `isValid` commands too, so you can test them separately.

3.6.6 Putting It All Together

The following example demonstrates how to set up both vibration and LED alerts in the same notification. See the example on the next page.

Example: Using the Notification APIs

```
import IQCore
    private var wearableController: WearableControllerType!
    private var notificationController: NotificationControllerType!
    self.notificationController =
                          NotificationController.controllerForWearable(wearableController)

    func vibraPattern() -> NotificationVibrationPattern? {

        let pattern = NotificationVibrationPattern(type: .Square)
        pattern.amplitude = 127
        pattern.repititionCount = 3
        let duration1 = NotificationDurationPattern(
                on: NSTimeInterval(0.5), // NSTimeInterval is typically in seconds
                off: NSTimeInterval(0.5)
            )
        pattern.durationPatterns.append(duration1)

        let duration2 = NotificationDurationPattern(
                on: NSTimeInterval(0.5),
                off: NSTimeInterval(0.5)
            )
        pattern.durationPatterns.append(duration2)

        return pattern
    }

    func ledPattern() -> NotificationLedPattern? {

        let led = NotificationLedPattern(type: .LEDBlink)
        led.intensity = 130
        led.repititionCount = 3
        let duration1 = NotificationDurationPattern(
                on: NSTimeInterval(0.5), // NSTimeInterval is typically in seconds
                off: NSTimeInterval(0.5)
            )
        led.durationPatterns.append(duration1)
        led.colors.append(UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0))

        let duration2 = NotificationDurationPattern(
                on: NSTimeInterval(0.5),
                off: NSTimeInterval(0.5)
            )
        led.durationPatterns.append(duration2)
        led.colors.append(UIColor(red: 0.0, green: 1.0, blue: 0.0, alpha: 1.0))

        return led
    }
```

Example is continued on next page.

Example: **(Continued from previous page)**

```
...
}

    func sendNotification() {

        let vibra = vibraPattern()
        let leds = ledPattern()
        let notification: WearableNotification = WearableNotification(ledPattern: leds,
              vibrationPattern: vibra, delay: 0)

        if notification.isValid {
            notificationController.sendNotification(notification, success: { () -> Void In
              print("Sent Notification: \(notification)")
                }, failure: { (error) -> Void In
                    print("Failed to send Notification: \(error)")
            })
        }
    }
```

3.7 Using the Data Store APIs

The Wearable Platform Core SDK provides data store for wearables and users. It is provided in the `IQCore.framework` package of the Wearable Platform Core SDK for the iOS mobile platform.

You will probably want to set up your own user information data store outside of `IQCore` because of the wide variety of information you may require.

3.7.1 About the UserIdentity Object

A `UserIdentity` can be created using a String that represents the identification of the user. This is usually obtained by logging into a cloud service, but it can be any unique string. The following example demonstrates the initialization of a `UserIdentity`.

Example: **Initializing a UserIdentity**

```
let user = UserIdentity("123123123")
```

Once the `UserIdentity` has been initialized, fields such as first and last name for the device wearer can be filled in. Available fields are:

`identifier`
External device wearer iID, usually obtained by logging into a cloud service

`uuid`
Unique ID for the device wearer. This is internally generated and managed.

`firstName`
Device wearer's first name

`lastName`
Device wearer 'ssurname

`email`
Device wearer's email address

`phoneNumber`
Device wearer's phone number

3.7.2 To Set the Current User

The following example sends an initialized `UserIdentity` object to the `IQCore` SDK.

Example: **Setting the Current User**

```
let user = UserIdentity("123123123")
IQCore.LocalDataStore.setCurrentUserIdentity(user)
```

3.7.3 To Get a UserIdentity Object

The `userIdentityForUUID()` method returns a `UserIdentity` object that matches either a UUID or a user identifier string. The returned `UserIdentity` object is optional. If a match to the UUID or identifier is not found, then the `UserIdentity` object will not be returned.

The `userIdentityForUUID()` method will match on the `uuid` field of the `UserIdentity` object.

Example: **Getting a UserIdentity Object Based on UUID**

```
Let user = IQCore.LocalDataStore.userIdentityForUUID ("123456")
```

The `userIdentityForIdentifier` method will match on the `identifier` field of the `UserIdentity` object.

Example: **Getting a UserIdentity Object Based on Identifier**

```
Let user = IQCore.LocalDataStore. userIdentityForIdentifier ("123123123")
```

3.7.4 About the WearableIdentity Object

A `WearableIdentity` object is created by `IQCore` based on information read from the wearable device. None of the fields can be modified because they are based on the device. There is no set wearable device method. Available fields are:

`displayName`
Display name for the wearable device

`uuid`
Unique id for the wearable device

`manufacturerName`
Manufacturer of the wearable device

`modelNumber`
Model of the wearable device

`serialNumber`
Unique serial number for the wearable device

`firmwareRevision`
Firmware revision of the wearable device

`softwareRevision`
Software revision of the wearable device

`hardwareRevision`
Hardware revision of the wearable device

3.7.5 To Get a WearableIdentity Object

You can get the WearableIdentity for the device using UUID, or using the wearable device serial number. See the following examples.

Example: **Getting a WearableIdentity Object Based on UUID**

```
let device = IQCore.LocalDataStore.WearableIdentityForUUID ("123456")
```

Example: **Getting a WearableIdentity Object Based on Serial Number**

```
let device = IQCore.LocalDataStore.WearableIdentityForSerialNumber ("device123123")
```

3.8 Using the Logging APIs

The Wearable Platform Core SDK provides a Logging object that is both simple and powerful. It is provided in the `IQCore.framework` package of the Wearable Platform Core SDK for the iOS mobile platform

Logging is provided for different urgency levels. Logging can go to the console, to a log file, or to both.

If logging goes to a file, the logger can be told how often to change log files, how large the files can get, and how many files are kept before deleting older log files.

3.8.1 About Logging Levels

Each log message is tagged with one of the following urgency levels:

- Error
  Indicates that something is wrong and is potentially fatal.
- Warning
  Indicates a problem that should be looked into.
- Info
  Indicates something of note. However, there is no problem.
- Debug
  Indicates something useful while debugging.
- Verbose
  Indicates detailed or frequently occurring information useful while debugging.

The log level can be set globally with the `logLevel` variable, and set separately for console logging and for file logging. The class checks against `logLevel` first, then checks the level set for files and console. If the log levels for console or file logging are set to a lower level than `logLevel`, then some messages could be lost.

3.8.2 Initializing the Logger

Logger initialization happens in the application `AppDelegate` object in the `didFinishLaunchingWithOptions` method. The log level is set using the `logLevel` variable. Setting a log level includes log levels above it. For example, if Error is set then only Error level messages will be displayed. If Verbose is set then all of the log level messages will be displayed.

Calling the `enableConsoleLogging()` method writes logging messages to the console.

Calling `enableFileLogging()` method writes logging messages to a log file.

Example: **Initializing the Logger**

```
    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
       [NSObject: AnyObject]?) -> Bool {

        Log.enableConsoleLogging()
        Log.enableFileLogging()
        Log.logLevel = .Debug

        return true
    }
```

3.8.3 To Log a Message

Use the `message()` method to log a message. It takes the message as a string. The log level for the message is part of an optional call. If that log level is active, then the option is set for it. The following is an example of a Debug level message.

Example: **Logging a Message**

```
Log.debug?.message("Initialize an object")
```

If debug is active, the message will be sent to the console or the logging file and the message will look like the following example.

Example: **Logging a message in Debug**

```
2015-12-07 18:05:55:337 testapp[2310:778573] <Debug> [MyViewController:31 init()]
    Initialize an object
```

The following are added to the message:

- Timestamp
- Application name
- Log level
- Location of the message call

3.8.4 To Enable Console Logging

Use the `Log.enableConsoleLogging()`method to enable the console logger. The console logger will use the logging level set in `logLevel`.

You can pass in a log level to `Log.enableConsoleLogging ()` to restrict the messages that logged.

Example: **Enabling the Console Logger**

```
Log.enableConsoleLogging(.Error)
```

3.8.5 Disable Console Logging

Use the `Log. disableConsoleLogging()` method to disable the console logger.

3.8.6 To Enable File Logging

Use the `Log.enableFileLogging()` method to enable file logging.

By default file logger sets a number of fields that control how large the log files can be and how many to keep. The following example demonstrates the full method for enabling file logging.

Example: **Enabling File Logging**

```
Log.enableFileLogging(logLevel level: LogLevel = .Verbose, maximumFileSize: UInt64 =
    1024 * 1024 * 1, rollingFrequency: NSTimeInterval = 60 * 60 * 24,
    maximumNumberOfLogFiles: UInt = 2)
```

In this example:

- `LogLevel .Verbose`
  Sets the log level

- `MaximumFileSize 1024 * 1024 * 1`
  Sets the maximum file size before rolling over to a new file
- `RollingFrequency 60 * 60 * 24`
  Creates a new file every 24 hours
- `MaximumNumberOfLogFiles 2`
  Keeps only 2 log files

3.8.7 To Disable File Logging

Use the Log. disableFileLogging () method to disable file logging.

3.8.8 To Disable All Logging

Use the `Log. disableAllLogging ()` method to disable both console logging and file logging.

3.8.9 Log File Location on the Companion Device

On the companion device such as a mobile phone, logs are placed in the directory `/AppData/Library/Caches/Logs` of the sandbox for the mobile application.

You can use two methods for getting log names and locations on the phone:

- `public class func logFileNames() -> [String]?`
- `public class func logFilePaths() -> [String]?`

Both return an array of strings.

The `logFileNames()` method returns the names of the log files.

Example: **Returning Log File Names**

```
["com.intel.wearable.coretest.CoreTestApp 2015-12-08 19-19.log",
    "com.intel.wearable.coretest.CoreTestApp 2015-12-07 18-09.log"]
```

The `logFilePaths()` method returns the paths of the log files.

Example: **Returning Paths to Log Files**

```
["/var/mobile/Containers/Data/Application/77121212-1234-1234-1234-77121212/Library/Caches/
    Logs/com.intel.wearable.coretest.CoreTestApp 2015-12-08 19-19.log",
    "/var/mobile/Containers/Data/Application/77121212-1234-1234-1234-77121212/Library/
    Caches/Logs/com.intel.wearable.coretest.CoreTestApp 2015-12-07 18-09.log"]
```

3.9 Using the Error Code APIs

The Wearable Platform Core SDK provides a an extension on the `NSError` class that defines failure conditions that may be encountered by the system. Each error is represented using an error code and a human readable error message containing error details.

The errors are reported in the `com.intel.wearable.core.error` domain.

The following errors are defined:

- Bluetooth Unavailable
- Wearable Not Found
- Wearable Not Ready
- Connection Timed Out
- Other wearable already connected
- Read Failed
- Subscription Failed
- Publish Failed
- Firmware Installation Failed

CHAPTER

4 Using the Wearable Platform Body IQ APIs

This chapter provides information to get you started using the Body IQ APIs.

This chapter contains the following sections:

- About the Body IQ APIs
- Using the Body IQ APIs

4.1 About the Body IQ APIs

The Body IQ APIs collect information about the physical activity of the device wearer. The Body IQ APIs provide information about type of activity the device wearer has performed or is currently performing. This information is provided in the form of a time series, and includes the following details for each data point:

- Type of an activity - walking, running or cycling
- Activity Start timestamp

- Activity End timestamp
- Step count for interim walking or running intervals

Based on time series data, Body IQ APIs can compute additional values for each data point such as activity duration, distance covered or number of calories burned. Time series data is persisted in the local data store on the companion device.

4.1.1 Body IQ Time Series Overview

To compute details of the device wearer's activity, such as distance covered or calories burned, the device wearer must enter the following minimal profile details:

- Height
- Weight
- Gender

The Body IQ APIs use this information, in conjunction with data observed by the wearable device sensor technology, to enhance each data point in the time series with details specific to the device wearer. The sensors on the wearable device continuously collect activity data about the device wearer. After Body IQ initialization, the application can subscribe to receiving user activity updates and record time series capturing user activity details.

4.1.2 How Body IQ Data is Stored

Time series observed with Body IQ follows the following Data Store storage policy:

Short-term storage: up to 3 days worth of activity data can be stored on the wearable device. If the wearable device is not connected to the companion device within three days of conducting activity, the oldest data points are purged to make room for new activity information.

Mid-term storage: Time series observed and collected by the Body IQ APIs on the companion application, are stored in the local data store on the companion device. They are stored for 30 days.

Long-term storage: The application can read time series data stored on the companion device in the mid-term data store, and transfer it to the cloud data store for long-term storage using Intel Cloud services.

4.2 Using the Body IQ APIs

All Body IQ APIs are provided in the `BodyIQ` module of the Wearable Platform SDK for the iOS mobile platform.

This section contains the following topics:

4.2.1 To Add BodyIQ to Your Application

Step 1. Add `BodyIQ` framework to your Xcode project or workspace.

Step 2. Import the `BodyIQ` module into source files.

Example: **Adding BodyIQ to Your Application**

```
import UIKit
import IQCore
import BodyIQ

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
    [NSObject: AnyObject]?) -> Bool {
        // Override point for customization after application launch.

        let exampleUserBodyStats = BodyIQUserBodyStatistics(weight_kg: 65, height_cm: 175,
                                    biologicalSex: .Female)
        BodyIQ.configureWithUserIdentifier("exampleBodyIQUser", bodyStats:
                                           exampleUserBodyStats)
        return true
    }
}
```

4.2.2 To Configure Body IQ

Before `BodyIQ` can be used for the first time you should create a user with body statistics of height, weight and biological sex for purposes of calorie computations and distance formulas.

Example: **Configuring Body IQ**

```
    let exampleUserBodyStats = BodyIQUserBodyStatistics
                   (weight_kg: 65, height_cm: 175, biologicalSex: .Female)
    BodyIQ.configureWithUserIdentifier("exampleBodyIQUser", bodyStats:
                   exampleUserBodyStats, dataRetentionDays: 45)
```

In `configureWithUserIdentifier` you can pass the user identifier of a user you have created with `IQCore`. If the `userIdentity` is not registered with `IQCore`, then the `configureWithUserIdentifier` will create it.

If the optional parameter `dataRetentionDays` is nil, then the default data retention policy is used by `BodyIQ`.

After configuring `BodyIQ`, you are able to observe a Wearable Device.

4.2.3 To Observe a Wearable Device

Observing a wearable device enables any accumulated body activities to be recorded to the BodyIQ time series data store.

***Example:* Observing a Wearable Device**

```
BodyIQ.beginObservingWearableController(wearableController,
            failure: { error in
                print("couldn't observe the wearable: \(error)")
            },

            success: {
                print("We are now recording body activity to the TimeSeries data store.")
            }
        )
```

4.2.4 To Subscribe to Activity Data Updates

After `beginObservingWearableController` has invoked the success block, you can `subscribe to activity` data by calling `addActivityListenerForWearableController(…)`

***Example:* Subscribing to Activity Data Updates**

```
BodyIQ.addActivityListenerForWearableController(wearableController,
                failure: {error in print("error: \(error)"),
                success: { print("success")},
                listener: "An object or string",
                callback: { observation, status, deviceID in
                          print("BodyIQ reported \(observation.type) status \(status) for
                          device \(deviceID)")

})
```

The callback block is invoked each time a body activity begins, finishes, or is ongoing after some period of time. The status parameter indicates whether the activity is started, finished or ongoing.

Note: If `beginObservingWearableController` has not been called prior to calling `addActivityListenerForWearableController`, then it will be called on your behalf.

4.2.5 To Query Time Series Activity Data

After some events have been recorded by `BodyIQ`, you can query the recorded events.

Example: **Querying Time Series Activity Data**

```
[weak self]
            error, timeSeriesArray in
            self?.devicesTimeSeries = timeSeriesArray
            if let error = error {
                print("Error: \(error)")
            }
            else if let timeseries = timeSeriesArray?.first {
                print("TimeSeries for device \(timeseries.info.deviceID)")
                for observation in timeseries.observations {
                    print("EventType: \(observation.eventType)")
                    if let value = observation.value as? BodyIQWalkingTimeSeriesValue {
                        print("Steps: \(value.stepCount)")
                    }
                }
            }
```

You can use `getCurrentActivityReport` to get the number of steps now in an ongoing activity.

Example: **Getting Current Step Count for Ongoing Activity**

```
        BodyIQ.getCurrentActivityReport(wearableController,  completion: { [weak self]
           (activity) -> () in
            dispatch_async(dispatch_get_main_queue(), { () -> Void in
                dispatch_async(dispatch_get_main_queue(), { () -> Void in
                    self?.showActivityAlert(activity)
                })
            })
            }, failure: { [weak self] (error) -> Void in
                dispatch_async(dispatch_get_main_queue(), { () -> Void in
                    self?.showNoActivityAlert()
                })
            })
```

The `setOngoingActivityGranularity()` method changes the granularity for reporting ongoing activity events from 5 minutes to something else in minutes.

Example: **Changing Granularity for Reporting Ongoing Activity**

```
BodyIQ.setOngoingActivityGranularity(self!.wearableController, granularity: intValue!)
```

4.2.6 To Convert Activity Data to a JSON Object

Use the `timeSeries.toJSON` method to convert the timeseries to a dictionary format suitable for JSON serialization.

4.2.7 To Delete Old Activity Data from the Data Store

Activity data is automatically purged from the local `BodyIQ` data store based on the data retention policy for `BodyIQ`. See How Body IQ Data is Stored on page 56.

CHAPTER

5 Using the Wearable Platform Social IQ APIs

Social IQ provides support to transfer LED and haptic notifications to the wearable device in response to an event. For example, after receiving a calendar reminder or a Time-to-Leave notification, the application can send an LED and/or vibration pattern to the wearable device to notify the device wearer of the event.

See Using the Wearable Notification APIs on page 37 for details on configuring and sending LED and vibration notifications to the wearable device.

CHAPTER

6 Using the Wearable Platform Time IQ APIs

The Curie Software Platform TimeIQ APIs implement scheduling services on Curie-based wearable devices.

This chapter contains the following sections:

6.1 Initializing TimeIQ In Your Application

Before accessing any of the scheduling services Curie Software Platform TimeIQ APIs provide, initialize `TimeIQApi` class in your application. Call the `TimeIQApi.init()` method on an instance of `TimeIQApi`, supplying two parameters:

- An instance of your authentication provider
- The URL for the Curie Platform Cloud Service

Example: **Initializing TimeIQ**

```
import TSO

class YourService : NSObject, IMessageListener {
    static let sharedInstance = YourService()
    let authCredentialsProvider = YourAuthCredentialsProvider.sharedInstance

    override func viewDidLoad() {
           self.mTimeIQApi = TimeIQApi()
           self.mTimeIQApi!.init__WithIAuthCredentialsProvider(self.authCredentialsProvider,
               withNSString:nil)
)
```

6.2 Handling Messages in the TimeIQ APIs

Message Handling APIs support operations for communicating information about events, reminders, Place-related notifications, and other types of alerts across the TimeIQ APIs. Message Handling APIs are provided with the `IMessageHandler` interface.

This section includes the following topics:

- About the Message Handler
- Registering a Listener with the Message Handler
- To register messages
- Initializing the Message Handler

- Listening to Messages from Message Handler

6.2.1 About the Message Handler

`IMessageHandler` manages a range of tasks including:

- `register()`
  Registers the listener to the messages sent by the Message Handler
- `init()`
  Initialization of Message Handler, which should be called after registering a listener
- `unregister()`
  Unregisters the specified listener from the Message Handler
- `dispose()`
  Clears the Message Handler and disposes any pending messages

Use the `getMessageHandler()` method to obtain an instance of the Message Handler to manage messages across the TimeIQ APIs.

Example: **TimeIQ Message Handling**

```
import TSO
class YourService : NSObject, IMessageListener {
    static let sharedInstance = YourService()
    override func viewDidLoad() {
       let messageHandler = self.timeIQApi?.getMessageHandler()
```

6.2.2 Registering a Listener with the Message Handler

You can listen for `TimeIQ` messages in your application. Each of the TimeIQ APIs sends messages to communicate when an event occurs or when a reminder or other type of notification is triggered. A good practice is to register to listen to messages using a shared instance. This ensures that

you will receive messages even if your application temporarily stops running.

6.2.3 To register messages

Step 1. First implement the `IMessageListener` interface in your class.

Step 2. Register to listen to messages using the `register()` method.

The following example demonstrates how to register to listen to messages using a service.

Example: **Registering to Listen to Messages Using a Service**

```
import TSO

class YourService : NSObject, IMessageListener {

    static let sharedInstance = YourService()

    override func viewDidLoad() {
       //register the current class to listen to messages
       messageHandler.register__WithIMessageListener(self)
```

6.2.4 Initializing the Message Handler

Initialization of the Message Handler should be called after registering at least one listener. All messages sent before `init()` will be accumulated. When `init()` is called, all of the accumulated messages will be received by the listener if a listener has been registered. If `init()` is called and no listener has been registered yet, the accumulated messages will be lost.

Example: **Initializing the Time IQ Message Handler**

```
import TSO


class YourService : NSObject, IMessageListener {

    static let sharedInstance = YourService()

    override func viewDidLoad() {

 // register the current class to listen to messages

    messageHandler.register__WithIMessageListener(self)

    let messageEngine =
       ClassFactory.getInstance().resolveInternalWithClass(IOSClass.classWithProtocol(
           IExternalMessageEngine)) as! ExternalEngine;

    messageEngine.init__();
```

6.2.5 Listening to Messages from Message Handler

`YourService`, or any other class that you registered to the `messageHandler`, should implement `IMessageListener`.

The `IMessageListener` interface contains one method that you should implement `void onReceive(IMessage message)`.

The message you receive has two methods: `getType` and `getData`.

Below is a list of the message types that may be returned when you call the `getType` method:

- ON_REMINDERS_TRIGGERED
  A reminder was triggered. For this `MessageType` the Message Data is `RemindersResult`.
- ON_EVENT_TRIGGERED
  TTL triggered for an event. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.
- ON_EVENT_START
  A defined event has started. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.
- ON_EVENT_END
  A defined event has ended. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.

6.3 Using the Reminders APIs

Reminders are a central feature within the TimeIQ SDK and can be used to complement functionality in the Events, Places, and User State Time IQ APIs. You can create some types of Reminders without reference to any event, place or user state. For example, you can include a functionality that creates a notification prompting the device wearer to complete a task at a particular time. This is a called a Do reminder.

This section contains the following topics:

6.3.1 About the Reminders Manager

The `IRemindersManager` interface provides a mechanism for creating, adding, storing and retrieving reminders. You can also use `IRemindersManager` to edit or remove existing reminders.

A reminder is always created with a trigger, which enables the developer to set the timing for the reminder to activate.

When it is time for the reminder to activate, you will get a message through the Message Handler, with a message with type ON_REMINDERS_TRIGGERED

6.3.2 Reminder and Trigger Types

The following are supported reminder types:

- `DoReminder`
  Reminds device wearer to perform an action specified in a user-defined string.
- CallReminder
  Reminds device wearer to call a specified contact.
- `NotifiationReminder`
  Reminds device wearer that a notification should be sent to specified contact.

The following are supported trigger types:

- `ChargeTrigger`
  Based on pre-defined charging conditions
- `MotTrigger`
  Based on pre-defined means of transport (MOT) conditions
- `PlaceTrigger`
  Based on pre-defined actions for a specific location
- `TimeTrigger`
  Specific pre-defined time

6.3.3 To Create a Reminder

Step 1. Create the trigger for the reminder.

Step 2. Create a reminder with the trigger.

The following example adds a reminder to turn on the car light. This reminder is triggered when the user starts driving.

Example: **Creating a Reminder**

```
import TSO

...
// trigger for mot to change to car started (will not be triggered now, if you are
//currently driving)

let trigger = MotTrigger_MotTriggerBuilder(motType: MotType.CAR(), withMotTransition:
    MotTransition.START()).build()

//create a reminder
    if (trigger != nil) {

reminder = DoReminder_DoReminderBuilder(ITrigger: trigger, withNSString:
    "Turn on the car lights").build()
     }
```

6.3.4 To Add a Reminder

Step 1. Get a reference to the `RemindersManager` object from an instance of the `TimeIQApi` object.

Step 2. Call the `addReminder` method to include the reminder you want to add as a parameter in the method call.

The output from the `addReminder` method is assigned to a `Result` object.

For more information see "The following table summarizes common result codes you can use." on page 93.

Step 3. Verify that the reminder was successfully added.

Check the `isSuccess()` method on the `result` object.

Example: **Adding a Reminder**

```
import TSO

...
  // add the reminder
  Result result =

YourService.sharedInstance.timeIQApi?.getRemindersManager().addReminderWithIReminder(reminder)
     let message: String?

     if (result?.isSuccess()) {
        message = "reminder_was_added" : // reminder added OK
     } else {
        // reminder was not added with the error
        message = "reminder_was_not_added" + result.getMessage();
     }
```

6.3.5 To Edit a Reminder

Editing a reminder requires that you create a new reminder with your changes and replace the old version with the updated version.

Step 1. Create a new reminder.

Save the details of the existing reminder.

Step 2. Remove the old version of the reminder.

Step 3. Add the new reminder reflecting the changes you made.

6.3.6 Adding Snooze Functionality for Reminders

Snooze functionality is commonly recognized as an alarm clock feature. You usually set an alarm clock to wake you from sleep, and then enable the snooze feature to allow you to “snooze” a few minutes past the alarm. Subsequent alarms sound at regular intervals until you are done snoozing, and disable all alarms associated with your wake time.

The Time IQ Reminders snooze functionality works in a similar way. The device wearer sets a time to be reminded of

an upcoming task or event, and then enables the snooze feature. When the first reminder notification is activated, the device wearer can ignore the alarm knowing that subsequent alarms will pester him or her at regular intervals. When the device wearer is ready to act on the reminder, he or she disables all alarms associated with the task or event.

6.3.7 To Snooze Reminders

To snooze a reminder, you will first need to get the snoozing options. See Supported Snooze Options and Supported Snooze TimeRange Types.

Step 1. Call the `getSnoozeOptions` method at the Reminders Manager, and pass the reminder ID.

You will get a `ResultData` containing a list of snooze options.

Step 2. Ask the device wearer to pick a `SnoozeOption`.

Step 3. Call the `snoozeReminder` method at the Reminders Manager.

Pass the reminder ID and the `SnoozeOption` that the user picked to snooze the reminder.

6.3.7.1 Supported Snooze Options

The snooze options are relevant to the specific reminder, and ordered by importance. They take into account the state of the user, the type of reminder and the type of the trigger. The following are supported snooze options:

- `WHEN_CHARGING`
  Next time the user will be charging his device
- `FROM_CAR`
  Next time the user will start driving
- `NEXT_DRIVE`
  Next time the user will start driving
- `DEFINE_HOME`
  Next time the user will arrive home, but home is not

yet defined. in this case, we recommend that the developer will prompt the user to define his home

Tip: For FROM_PLACE, the place can be obtained by casting the snoozeOption to PlaceSnoozeOption, and using the getPlaceId() method on the PlaceSnoozeOption instance. Then use the PlaceRepo to obtain the place itself

- `DEFINE_WORK`
  Next time the user will arrive to work, but work is not yet defined. In this case, you can prompt the user to define his work.
- `FROM_PLACE`
  Next time the user arrives to a specific place.
- `NEXT_TIME_AT_CURRENT_PLACE`
  Next time the user gets to the current place
- `LEAVE_CURRENT_PLACE`
  When the user leaves the current place

Tip: For IN_X_MIN, the delay offered can be obtained by casting the snoozeOption to TimeDelaySnoozeOption, and using the getDelayMinutes() method on the TimeDelaySnoozeOption instance.

- `IN_X_MIN`
  In a specific time delay
- `TIME_RANGE`
  In a specific time range.

For `TIME_RANGE`, the delay offered can be obtained by casting the `snoozeOption` to `TimeRangeSnoozeOption`. Use the `getTimeRange()` method on the `TimeDelaySnoozeOption` instance to get the `SnoozeTimeRange`.

6.3.7.2 Supported Snooze TimeRange Types

- `THIS_MORNING`
- `TODAY`
- `THIS_EVENING`
- `THIS_NIGHT`
- `TOMORROW_MORNING`

6.3.8 To End a Reminder

Once the user sees the reminder and acts upon it, you should mark the reminder as ended.

To end the reminder, call the `endReminder` method at the Reminders Manager, and pass the reminder ID along with the `ReminderEndReason` type.

The following are supported `ReminderEndReason` types:

- Dismiss
  Device wearer dismissed the reminder.
- Done
  Device wearer completed the reminder.

6.4 Using the Events APIs

The Events APIs are tailored to condition reminders upon a scheduled occurrence at a particular place or location. For example, a user may want to add an event to help plan a workout at the gym. The user can specify the gym's location, the time and the date of the workout.

This section contains the following topics:

6.4.1 About Time to Leave (TTL) Notification

One useful feature of the Events APIs is the Time-To-Leave (TTL) notification. A TTL notification is automatically generated upon the creation of an event. This notification sends the user an alert before the start time of the event to inform the device wearer that it is time to depart for the event. Although this can be used as standalone functionality, it is important to keep in mind that this

notification is automatically sent for each event that is added on `TimeIQ`.

6.4.2 About Calendar Integration

The Events APIs integrate with basic calendar functionality. The device wearer is able to define a set of read calendars from which the TimeIQ SDK will upload events and generate TTL notifications accordingly. See Using the Calendar Details APIs on page 92 for more information. The TTL notifications will be automatically generated for the calendar events whose location field was successfully resolved by the TIMEIQ SDK.

Upon creation of certain types of events, such as a `BeEvent`, you can add the event to the calendar. Use the `addToCalendar` in `BeEvent` builder upon event creation. The event will be added to the write calendar defined by the user through the Calendar Details APIs.

6.4.3 Using the Events Engine

Use the Time IQ Events Engine protocol to add and manage events. Defined events are ready for customization, and are associated with notifications to the device wearer at key time points relevant to a scheduled event.

In addition, the Events Engine protocol enables user interaction upon receiving the different notifications sent for each event. For example, upon receiving TTL notification for a doctor's appointment, the user can choose the time snooze option, and receive a subsequent notification at the specified time.

6.4.4 Using the EventBuilder

Use the `EventBuilder` function to create a new event. The `EventBuilder` is designed to give you flexibility to choose the properties or attributes required for any type of event you create for use in an application.

The `BaseEvent` class requires only two attributes:

- Location
- Start time

Additional optional parameters are documented in the API Reference. Some of the optional parameters have default values.

6.4.5 To Add an Event

Step 1. Get a reference to the application Event Engine by calling a method on the `TimeIQApi` object instance.

Step 2. Create the event by using the appropriate Event Builder and calling the `addEvent` method.
Place the event as an argument inside that method.

Step 3. Verify the successful completion of the operation by examining the outcome returned through the Result object.

See the following example.

Example: **Adding an Event**

```
let eventsEngine = YourService.sharedInstance.timeIQApi!.getEventsEngine();

let eventLocation = TSOPlace(double: 40.764367, withDouble: -73.981076,withNSString:
     "New York", withNSString: "New York");

 let doctorAppEvent = BeEvent_BeEventBuilder(TSOPlace: eventLocation, withLong:
     System.currentTimeMillis() +
        TEN_MINUTES).durationWithLong(DURATION_TWENTY_MINUTES)

 doctorAppEvent.addToCalendarWithBoolean(true)

 let res =  eventsEngine.addEventWithIEvent(doctorAppEvent.build());

 if (res.isSuccess())

 {



 }
```

6.5 Using the Places APIs

Locations can be used as triggers for reminders or for other Time IQ API events. Use the `TimeIQ` Places APIs to enable the device wearer to input custom locations.

The Places API enables the you to manage device wearer-defined places repository in the SDK. You can add places manually to the SDK or remove them. You can access the collection of all the known places, which were either added by the device wearer, or were automatically added by the SDK when resolving the location of calendar events detected by the SDK. The `TimeIQ` Places APIs also provide you access to the two detected places: the device wearer's home and workplace.

This section contains the following topics:

- About the TSOPlaces Engine
- About the TSOPlace Protocol

- Using IPlaceRepo
- To Add a New Place
- To Delete a Place
- To Retrieve All Places
- To Retrieve a Place by ID
- Managing Special Places: Home and Work

6.5.1 About the TSOPlaces Engine

Use `TSOPlaces` to create preferred places defined by the device wearer, and store them in the Places Repository with their corresponding semantic keys.

Important: You must start the TSOPlacesEngine upon the first use of the TimeIQ SDK in the application.

6.5.2 About the TSOPlace Protocol

The protocol `TSOPlace` represents a place in the `TimeIQ` SDK.

The `TSOPlace` object contains the following data items:

- `TSOCoordinate`
  Structure that holds the geographic Latitude/Longitude coordinates of the place.
  Example: 37.3865906,-121.9812071
- `Address`
  String that represents the street address of the place.
  Example: 3100 Lakeside Drive, Santa Clara, CA 95054, USA
- `Name`
  String that represents the name of the place.
  Example: The Plaza Suites Hotel

- `ManualPlaceSource`
  Holds an indication to the entity that generates the place.
- `ManualPlaceSource.USER`
  For places added explicitly by the application developer.
- `ManualPlaceSource.CALENDAR`
  For places added implicitly by the calendar resolver module within the SDK.

6.5.3 Using IPlaceRepo

The `IPlaceRepo` protocol is the main entry point for the Places Repository APIs. Obtain the `IPlaceRepo` protocol from the `TimeIQApi` object by calling its `getPlacesRepo()` method.

Example: **Obtaining the IPlaceRepo Interface**

```
let placesRepo = YourService.sharedInstance.timeIQApi!.getPlacesRepo();
```

Use this interface to add new places, delete existing ones and retrieve a collection of the existing places in the SDK.

These places can be used to create place-based triggers and reminders. For example, you can create a reminder to send a text to a friend when you leave your workplace and are headed to your meeting place.

6.5.4 To Add a New Place

Step 1. To create a new place, generate a Geographic coordinate object, address and name.

Step 2. To add a new place to the repository, create a new object called `TSOPlace`.

Example: **Creating a New Place**

```
let placeCoordinate = TSOCoordinate(double: 37.3865906,withDouble: -121.9812071)
let placeAddress = "3100 Lakeside Drive, Santa Clara, CA 95054, USA"
let placeName = "The Plaza Suites Hotel"
let newPlace = TSOPlace(NSString: placeName, withNSString: placeAddress,
                  withTSOCoordinate: placeCoordinate)
```

You will get a `ResultData` object with a `PlaceID` object, which is a place identifier generated by the SDK.

Example: **Obtaining a ResultData Object with a PlaceID**

```
let placeIDResultData = placesRepo.addPlace(newPlace)
        if (placeIDResultData.isSuccess())
        {
            // New place was added successfully
        }
```

6.5.5 To Delete a Place

Use the same `placeID` that was generated while adding the place to the repository. See the following example.

Example: **Deleting a Place**

```
let result = placesRepo.removePlaceWithPlaceID(placeId)
if (result.isSuccess())
{
   // The place was removed successfully
}
```

6.5.6 To Retrieve All Places

To get the list of all places in the Places repository, both user-defined and those resolved from calendar events, call the `getAllPlaces` method.

Note: Potential auto-detected home and work, will not be returned by this API method.

Example: **Retrieving All Places**

```
let myPlacesList = ArrayList()
let placesListResult = placesRepo.getAllPlaces();
if(placesListResult.isSuccess()) {
    myPlacesList  = placesListResult.getData();
}
```

6.5.7 To Retrieve a Place by ID

If you hold a `PlaceID` object, you can retrieve the relevant `TSOPlace` from the repository using the `getPlace()` method.

Example: **Retrieving a Place by ID**

```
let placeResultData = placesRepo.getPlaceWithPlaceID(placeId);
if(placeResultData != nil && placeResultData.isSuccess())
{
            let place = placeResultData.getData() as! TSOPlace;
}
```

6.5.8 Managing Special Places: Home and Work

You can add to the SDK two special places: home and work.

To add one of these places, use a specific API which enables adding a place with a argument called `SemanticKey`. You have two options:

- `SemanticKey_HOME_` for home
- `SemanticKey_WORK_` for work

Example: **Adding Home or Work Places**

```
let placeCoordinate = TSOCoordinate(double: 37.3866427,withDouble: -121.9886155)

     let placeAddress = "1287 Oakmead Pkwy, Sunnyvale, CA 94085, USA"
     let placeName = "Home sweet Home"
     let place = TSOPlace(NSString: placeName, withNSString: placeAddress,
                 withTSOCoordinate: placeCoordinate)

     if (placesRepo.addPlaceWithTSOPlace(place, withSemanticKey:
          SemanticKey_HOME_).isSuccess()) {
          // My Home was set successfully.
      }
```

You can get the special places by using this method.

Example: **Getting Special Places**

```
ResultData getPlaceIdWithSemanticKey(SemanticKey semanticKey);
```

Once you have the `PlaceID`, you can get the `TSOPlace` object as in the previous example.

In addition to these special places, the system might generate auto-detected home and auto-detected work, with following semantic keys:

- `SemanticKey_AUTODETECTED_HOME_`
- `SemanticKey_AUTODETECTED_WORK_`

You can get access to these auto-detected places (if they exist), using the same method. See the following example.

Example: **Generating Places Using Semantic Keys**

```
ResultData<PlaceID> getPlaceId(SemanticKey semanticKey);
```

6.6 Using the User State APIs

The User State APIs provide a means to get the current state of the device wearer, as well as setting a notification when the user state changes.

The user state includes the following data:

- Means of transport (MOT)
  CAR, WALK, or STATIONARY.
- Visited places
  A list of places that the user is currently visiting

This section contains the following topics:

6.6.1 About the UserState Object

The `UserState` object holds information about the user state as created by the SDK at a specific time. The SDK creates a new `UserState` object each time user state data changes. You usually will not create `UserState` objects by yourself, but instead obtain them from the `UserStateManager`.

You can get the creation time of the `UserState` by calling its `getTimeStamp()` method.

Example: **Getting the UserState Creation Time**

```
UserState state = obtainState()
let creationTime = state.getTimeStamp();
```

6.6.2 About UserStateData

Each data item that comprises the `UserState` object can be obtained in the form of `UserStateData`, and contain a specific data type.

To get the actual data from a `UserStateData`, object call its `getData()` method.

In addition to the actual data, this object also contains the timestamp in which the specific data was changed. **Note:** This is different from the timestamp in which the `UserState` object was changed.

Note also that the `UserState` object may return `null` when asked for a specific data if it has no information about it.

Example: **Getting Data from a UserStateData Object**

```
let motData = state.getMot();
      if (motData != nil) {
          let time = motData.getTimeStamp();
          let mot = motData.getData() as! MotType;
      }
```

6.6.3 Getting User State MOT Data

The means of transport (MOT) data contains the device wearer's means of transport and is represented by the `protocol.enums.MotType` enumeration.

The possible values of `MotType` are:

- `MotType.STATIONARY()`
- `MotType.WALK()`
- `MotType.CAR()`

Note: The value `MotType.PUBLIC_TRANSPORT` is not currently implemented. This MOT type may be implemented in a future release.

Example: **Getting Means of Transport (MOT) Data**

```
UserStateData<MotType> motData = state.getMot();
if (motData != null) {
    MotType mot = motData.getData();
    if (mot.equals(MotType.WALK()))
    // Look Mom, I'm walking!
   }
}
```

6.6.4 Getting Visited Places Data

The visited places data is a list of `PlaceIDs` that the device wearer is actively visiting. In the TimeIQ SDK, visiting means device wearer remains within his or her geographic bounds for a significant duration. This data is represented by the `userstate.VistedPlaces` class.

Example: **Getting Visited Places Data**

```
let visitsData = state.getVisits();
        if (visitsData != nil) {
            let visits = visitsData.getData() as! VisitedPlaces;
            for id:PlaceID in visits {
                if (id.getSemanticKey().isHome()) {
                    // Honey, I'm home!
                    break;
                                }
            }
        }
```

6.6.5 About the User State Manager

The `IUserStateManager` protocol is the main entry point for the user state APIs. Obtain this protocol from the `TimeIQApi` object by calling its `getUserStateManager()` method.

Example: **Obtaining the User State Manager**

```
let stateManager = mTimeIQApi.getUserStateManager();
```

Using this protocol, you can get the device wearer current state, register a listener to be notified when a state change occurs, and also unregister a listener to stop being notified.

6.6.6 Getting the Current User State

Obtain the current user state by calling the `IUserStateManager` method `getCurrentState()`. This method returns a `UserState` object which contains the most recently updated user state data.

Example: **Getting the Current User State**

```
let stateResult = stateManager.getCurrentState();
if (stateResult.isSuccess()) {
   let state = stateResult.getData() as! UserState
}
```

6.6.7 Registering a Listener to User State Changes

A new UserState object is created whenever some user state data is changed. To be notified when a new `UserState` object is created, call the `IUserStateManager` method `registerForStateChanges()` and pass an instance of `IUerStateChangeListener`. The `registerForStateChanges()` method returns the current `UserState` object. This is the same object that would be returned from a call to `getCurrentState`. Returning the current `UserState` object enables you to get the current state and to register for changes in one call.

***Example:* Registering a Listener to User State Changes**

```
class MyUserStateChangeListener: NSObject, IUserStateChangeListener {

    func onStateChangedWithUserState(oldState: UserState!, withUserState newState: UserState!,
        withUserStateChanges changes: UserStateChanges!) {

        if (changes.isChangedWithUserStateType(UserStateType.MOT())) {

             // Rangers citadel to team echo, detected MOT change

         }
    }
}

let stateResult = stateManager.registerForStateChanges(self);
if (stateResult.isSuccess()) {
           // Team echo to rangers citadel, state monitor installed successfully
}
```

The `onStateChanged` callback method parameters are:

- `oldState`
  The `UserState` object just before the change.
- `newState`
  The `UserState` object after the change.
- `changes`
  An instance of `com.intel.wearable.platform.timeiq.api.user state.UserStateChanges` that describes the data changes between the old and the new states.

6.6.8 Unregistering a Listener for User State Changes

To unregister any previously registered listener, use the `IUserStateManger` method `unregisterForStateChanges()`. Unregister a listener when listener notifications about user state changes are no longer necessary.

***Example:* Unregistering a Listener for User State Changes**

```
stateManager.unregisterForStateChanges(self);
```

6.7 Using the Route APIs

The Route APIs provide a structured way to get route information that can be used on its own, or in tandem with other TimeIQ APIs.

The `IRouteProvider` protocol specifies methods that can be used to initiate API calls to obtain routing information.

Tip: Though the RouteProvider uses a cache, it is a good practice to call the methods of IRouteProvider from an AsyncTask, since the call might take some time to complete execution.

You can call the following methods from RouteProvider:

- `getTTL()`
  Gets the TTL (Time to Leave) from a trip origin, in order to arrive at a certain time at a destination.
- `getETA()`
  Gets the ETA (Estimated Time of Arrival) from a trip origin to the destination for a given departure time.

The following example demonstrates how to get a TTL route from a trip origin to a destination for an event that will start in 2 hours. The example specifies the preferred MOT as driving.

Example: Using the Route APIs

```
import TSO


    func getTTL(origin: TSOCoordinate, destination: TSOCoordinate) ->
        TtlRouteData? {

         let routeProvider =
                YourService.sharedInstance.timeIQApi!.getRouteProvider();
                // get IRouteProvider

         let arrivalTime = System.currentTimeMillis() +
                TimeUnit.HOURS().toMillisWithLong(2);
                // arrival time is 2 hours from now

         let ttlRouteDataResultData =
                routeProvider.getTTLWithTSOCoordinate(origin,
                    withTSOCoordinate: destination,

    withLong: arrivalTime, withTransportType: TransportType.CAR());
           // gets the TTL route from origin to destination (preferably by car)

         return ttlRouteDataResultData.isSuccess() ?
                ttlRouteDataResultData.getData() as? TtlRouteData : nil;
           // returns the routeData or null if unsuccessful

     }
```

You can call the method with `preferredTransportType` as `null`. In this case the APIs will return the route with the most appropriate transport type for this route.

The method returns a `ResultData` which upon success, and contains a `TtlRouteData` or `EtaRouteData`, depending upon the specific method you called.

The `RouteData` contains, along other useful methods, the `getRouteSegments()` method, which returns all the route segments that constitute the route to the destination. The route to the destination is defined when the device wearer specifies the alert time.

The list of `RouteSegment` is ordered by time, from the time to leave until the arrival time.

The segments contains useful methods such as `getSegmentDuration()`, which returns the duration of the segment. The `getRouteInfo() method` which, if it is a travel

segment, contains the route information such as the `getTrafficIndication()` method.

Each segment has a type. Supported `SegmentTypes` are:

- `TimeToTTL`
  Time to Time to Leave (TTL)
- `OriginInDoor`
  `inDoor` navigation at trip origin
- `TimeToCar`
  Time to get to the car
- `Travel`
  Travel with desired means of transport (MOT)
- `Park`
  Park near destination + get from car to building
- `DestinationInDoor`
  `inDoor` navigation at destination

6.8 Using the Calendar Details APIs

The Calendar Details APIs enable all the calendar interaction required by the Time IQ SDK.

Through the Calendar Details APIs you can retrieve the list of available calendars on the wearable device, define the set of calendars to read events from, and write to the calendar for the Time IQ SDK to update.

To use the Calendar Details APIs, first get its reference by calling the `getCalendarsDetailsProvider()` method on the `TimeIQApi` object instance. Once received, it is ready for use and provides interaction with the available calendars.

Example: Initializing the Calendar Details APIs

```
let calendarDetailsProvider = YourService.mTimeIQApi.getCalendarsDetailsProvider();

        let resultData = calendarDetailsProvider.getAvailableCalendars();

        if (resultData.isSuccess()) {

            let availableCalendarsDetails = resultData.getData() as! ArrayList;

            // add usage of the available calendars

        }
```

The following table summarizes common result codes you can use.

Table 1: Useful Result Codes

Type	Codes
General	`ResultCode.SUCCESS` `ResultCode.GENERAL_ERROR`
Search	`ResultCode.SEARCH_NO_RESULTS` `ResultCode.SEARCH_TERM_NOT_SUPPORTED`
Location	`ResultCode.LOCATION_IS_NULL`
Time to Leave (TTL)	`ResultCode.TTL_IS_OVERDUE`

CHAPTER

7 Using the Cloud Services SDK

The Cloud Services SDK provides a communication layer between the Intel Software Platform for Curie and Intel Cloud Services. The following sections provide details and code samples for the Cloud Services SDK in an iOS environment.

7.1 Understanding the Cloud Services SDK

Features of the Cloud Services SDK include authentication, user and device profile storage, data storage, Binary Large Objects (BLOB) storage, native login, notification, time series data storage, error handling and logging. In addition to hiding the handling of complex protocols such as OAuth, Google Cloud Messaging and the Apple Push Notification service, the Cloud Services SDK handles header formatting, auto token refresh, background requests and queuing, device power management, and security review and scan. The Cloud Services SDK consists of the following modules.

- The `CDKCore` module provides user authentication, cloud request scheduling, cloud response processing and error handling functionality used by all of the other Cloud Services SDK modules. Authenticating to the Cloud and Handling Cloud Services SDK Errors contain additional information.
- The `CDKProfileStore` module provides storage and retrieval of a custom JavaScript Object Notation (JSON) documents that contain details about the authenticated user. Accessing Cloud User Profiles contains additional information.
- The `CDKDocStore` module provides the functionality to upload, store and retrieve general JSON documents. Accessing Cloud Document Stores contains additional information.
- The `CDKDeviceProfileStore` module provides storage and retrieval functionality for custom JSON documents that contain details about the companion device used to connect to the cloud. Accessing Cloud Device Profiles contains additional information.
- The `CDKBlobStore` module provides upload and download capabilities for BLOB data and software

assets. Accessing Cloud BLOB Data and Software Assets contains additional information.

- The `CDKTimeSeries` module provides storage, retrieval and analysis of time series data points. Accessing Time Series Data contains additional information.

7.2 Using the Cloud Services SDK

To use the Cloud Services SDK, you must incorporate the `CDKCore` module into your application either directly, or via an implicit dependency when using one of the other Cloud Services SDK modules listed in Understanding the Cloud Services SDK. The first step in using the SDK is to authenticate and log in the user by calling the appropriate authentication provider. (Details on calling the appropriate authentication provider are documented in Authenticating to the Cloud.) Once the user is logged in, the desired action can be taken.

The following sample code shows how to upload a file to the BLOB store in the cloud. Note that the `CDKCore` module is imported. The code illustrates the following actions.

1. The Google authentication provider `CloudGoogleAuthProvider` is instantiated and the user is logged in successfully.
2. The `CloudUserBlobStore` class is initialized.
3. The `MY_NAME` file is uploaded to the cloud.
4. Final responses (including progress messages and errors) are defined.

See example on the next page.

Example: Uploading a File to the BLOB Store

```
// Implement a ViewController and import the 'CDKCore' framework
#import "LoginViewController.h"
@import CDKCore;
...
@implementation LoginViewController {
- (IBAction) loginWasClicked:(id)sender {
    // Create an instance of an AuthProvider (In this case, we'll use Google):
    CloudAuthProvider *authProvider = [[CloudGoogleAuthProvider alloc]
                                       initWithBaseURL:BASEURL_UAA2
                                       ClientId:CLIENTID_FC
                                       ClientSecret:CLIENTSECRET_FC
                                       GoogleClientId:kGGClientId];
    // Call the login function of your chosen CloudAuthProvider
    [authProvider login:^(void) {
    // Success! let's upload our file...
    [CloudUserBlobStore init: authProvider
                              url: MY_URL
                              productId: MY_ID];
    [[CloudUserBlobStore getInstance] upload: MY_FILE_PATH
                                        tag: A_TAG
                                        fileName: MY_NAME
                                        // Do something for it starting...
                                        success: ^(CloudBlobStoreResponse *cloudResponse) {
                                                 }
                                        // Do something for it failing...
                                        failure: ^(CloudBlobStoreResponse *cloudResponse) {
                                                  }
                                        // Do something about progress...
                                        progress: ^(int64_t bytesWritten, int64_t
                                        totalBytesWritten,int64_t totalBytesExpectedToWrite)
                                            {
                                        }
                                        // Do something for it finishing...
                                        completed: ^(CloudBlobStoreResponse *cloudResponse)
                                               {
                                            }
                                        // Failure...
                                        } Failure:^(CDKError *error) {
                                            if (error) {
                                                UIAlertView *alert = [[UIAlertView alloc]
                                                           initWithTitle:@"Error"
                                                           message:error.message
                                                           delegate:self
                                                           cancelButtonTitle:@"OK"
                                                           otherButtonTitles:nil, nil, nil];
                                                [alert show];
                                                }
                                        }
    }];
```

7.3 Authenticating to the Cloud

The `CDKCore` module provides user authentication on iOS. To authenticate a user with the `CDKCore` module, the application must instantiate one of the provided authentication providers and call its respective `[login]` message. Once login is deemed to be successful, the application The application passes the provider instance to the SDK modules that need it (for example, the BLOB store). For details on how to use the Cloud Services SDK for authentication on iOS, see the appropriate procedures in the following sections.

- Using the Default Cloud Authentication Provider
- Using the Intel UAA Authentication Provider
- Using the Facebook Authentication Provider
- Using the Google Authentication Provider
- Using the Application Authentication Provider

Note: Each authentication provider inherits from the default authentication provider. Unless overridden, calling `[login]` without parameters for any provider will execute the default authentication provider.

7.3.1 Using the Default Cloud Authentication Provider

The default cloud authentication provider is the `CloudAuthProvider` class which, when called, presents a login/create-account page within a full-screen, borderless web browser window. The page will accept user credentials for any of the supported login mechanisms pre-configured on the cloud by the application administrator. To use the default cloud authentication provider, follow the steps

below. The sample implements a simple `ViewController` called `SignInViewController`.

Step 1. Implement the `ViewController`.

Example: **Implementing ViewController**

```
@import CDKCore;
...
@implementation SignInViewController
Inside viewDidLoad or similar lifecycle methods, instantiate CloudAuthProvider:
-(void)viewDidLoad {
    ...
    CloudAuthProvider *authProvider = [[CloudAuthProvider alloc]
                                initWithBaseUrl:@"https://authserver1.wearables.host.com"
                                clientId:@"ndg1"
                                clientSecret:@"ndg1SecRet"
                                onVC:self];
    ...
}
```

Step 2. Call the `[login]` function:

Example: **Calling the login() Function**

```
[cloudAuthProvider
  login: ^(void) { NSLog(@"Success"); }
  error: ^(CDKError *error) { NSLog(error.message); }];
```

After a successful authentication, the `authProvider` instance can be supplied to the other SDK modules as a response to a request for authentication.

To customize the look of the login/create-account page, inherit from `CloudAuthWebViewClient`, make the necessary customizations, and pass an instance of the inheriting class using `[authProvider setWebViewClient:instance]`.

7.3.2 Using the Intel UAA Authentication Provider

Intel User Account and Authentication (UAA) is the native Intel Cloud Services authentication provider. Users can create accounts and login to access Intel Cloud services. To

use the Intel UAA authentication provider, follow the steps below.

Step 1. Instantiate the `CloudUAAAuthProvider` authentication provider.

Example: **Instantiating a UAA Authentication Provider**

```
@import CDKCore;

    CloudUAAAuthProvider *authProvider = [[CloudUAAAuthProvider alloc]
                                  initWithBaseUrl:@"https://authserver1.wearables.host.com",
                                  clientId:@"ndg1"
                                  clientSecret:@"ndg1SecRet"];
```

Step 2. Create an account if necessary.

Example: **Creating a UAA User Account**

```
[authProvider createAccount:@"janedoe"
                   password:@"password_for_janedoe"
                   firstName:@"Jane"
                   lastName:@"Doe"
  success:^(void){...}
  failure:^(CDKError* error) {...}];
```

Step 3. Authenticate and log in the user.

Example: **Authenticating a UAA User**

```
[authProvider login:@"janedoe"
           password:@"password_for_janedoe"
  success:^(void) {...}
  failure:^(CDKError *error) {...}];
```

After a successful authentication, the `authProvider` instance can be supplied to the other SDK modules as a response to a request for authentication.

7.3.3 Using the Facebook Authentication Provider

The Facebook authentication provider enables a user to login using the credentials associated with a Facebook

account. The following procedure illustrates how to instantiate the Facebook authentication provider.

Step 1. Implement a `ViewController` to handle the login.

Example: **Implementing a ViewController**

```
@import CDKCore;

@implementation SignInViewController {
        CloudFacebookAuthProvider *_authProvider;

}
```

Step 2. Instantiate `CloudFacebookAuthProvider` and assign the instance to `mAuthProvider`.

Example: **Instantiating a Facebook Authentication Provider**

```
_authProvider = [[CloudFacebookAuthProvider alloc]
    initWithBaseUrl:@"https://authserver1.wearables.host.com"
    clientId:@"ndg1"
    clientSecret:@"ndg1SecRet"];

}

-(IBAction) loginWasClicked {

[mAuthProvider
  login:^(void) {...}
  failure:^(CDKError *error) {...}];
}
```

After a successful authentication, the `authProvider` instance can be supplied to the other SDK modules as a response to a request for authentication.

7.3.4 Using the Google Authentication Provider

The Google authentication provider enables a user to login using the credentials associated with a Google account. The

following procedure illustrates how to instantiate the Google authentication provider.

Step 1. Implement a `ViewController` to handle the login.

Example: **Implementing a ViewController**

```
@import CDKCore;
@implementation SignInViewController {
        CloudGoogleAuthProvider *_authProvider;
    }
```

Step 2. Instantiate `CloudGoogleAuthProvider` and assign the instance to `mAuthProvider`.

Example: **Instantiating a Google Authentication Provider**

```
_authProvider = [[CloudFacebookAuthProvider alloc]
    initWithBaseUrl:@"https://authserver1.wearables.host.com"
    clientId:@"ndg1"
    clientSecret:@"ndg1SecRet"]
    googleClientId:MY_GOOGLE_ID];
...
}

-(IBAction) loginWasClicked {
...
[mAuthProvider
    login:^(void) {...}
    failure:^(CDKError *error) {...}];
}
```

After a successful authentication, the `authProvider` instance can be supplied to the other SDK modules as a response to a request for authentication.

7.3.5 Using the Application Authentication Provider

The Application authentication provider authenticates the application to the cloud, not the user. It is used for services that do not need user-level access (for example, public

BLOB storage or to download new firmware). To use the Application authentication provider, follow the steps below.

Step 1. Instantiate the `CloudAppAuthProvider`.

Example: **Instantiating an Application Authentication Provider**

```
CloudAppAuthProvider *authProvider = [[CloudAppAuthProvider alloc] initWithBaseUrl: MY_URL];
```

Step 2. Call the `[login]` function.

Example: **Calling the login Function**

```
[authProvider login];
```

Note: This provider is used internally by the SDK and is not intended for use by the application.

7.4 Handling Cloud Services SDK Errors

Errors can be returned by the Cloud Services SDK in the following ways:

- Inside a `CloudResponse` object
- As return values
- As callback parameters.

In all of these cases, the error is captured by a `CDKError` object and the message inside can be accessed as illustrated in the `[login]` example below.

Example: **Accessing an Error Message**

```
[cloudAuthProvider
  login: ^(void) { NSLog(@"Success"); }
  error: ^(CDKError *error) { NSLog(error.message); }];
```

Some `CDKError` objects have a useful error code (for example, `CDKHttpError`). To get the error code, call for it as below.

```
NSInteger errorCode = error.code;
```

When the error is encapsulated in a `CloudResponse` object, you can access it as illustrated in the example below.

Example: **Accessing an Encapsulated Error Message**

```
CloudResponse *res;

if ([res hasErrorOccurred]) {
        CDKError *error = [res getError];
        NSLog(error.message);
}
```

7.5 Accessing Cloud User Profiles

The `CDKProfileStore` module provides a User Profile Store module to store and retrieve custom user profile information as JSON objects. To access a user profile, the user must be logged in. This entails that an `authProvider` instance exists and that `[authProvider login]` has successfully completed as documented in Authenticating to the Cloud. To use the API, follow the steps below.

Step 1. Create a `CloudProfileStore` instance.

Example: **Creating a CloudProfileStore Instance**

```
@import CDKCore;
@import CDKProfileStore;

    // create 'authProvider'

    [CloudEPS init:authProvider baseUrl:@"https://authServer1.wearables.host.com"];
    CloudEPS *profileStore = [CloudProfileStore getInstance];
```

Step 2. Create a `CloudProfileItem` user profile.

Example: **Creating a CloudProfileItem User Profile**

```
NSMutableDictionary *sampleProfile = [[NSMutableDictionary alloc] init];
sampleProfile[@"Name"]   = @"Superman";
sampleProfile[@"Age"]    = @"200";
sampleProfile[@"Height"] = @"6";
sampleProfile["Weight"]  = @"230";
CloudProfileItem *item = [[CloudProfileItem alloc] initWithProfileContent:sampleProfile];
```

Step 3. Upload the user profile to the cloud.

Example: **Uploading the Cloud User Profile to the Cloud**

```
[profileStore
  put:sampleProfile
  success:^(CloudResponse *response) { ... }
  failure:^(CloudResponse *failure {...}];
```

Step 4. Retrieve the user profile as needed.

Example: **Retrieving the Cloud User Profile**

```
[profileStore
  get:^(CloudResponse *response) { ... }
  failure:^(CloudResponse *response) { ... }];
```

7.6 Accessing Cloud Document Stores

The `CDKDocStore` module provides the functionality to upload, store and retrieve general JSON documents. Applications can save, retrieve, and delete documents based on a document ID. To access a document store, the user must be logged in. This entails that an `authProvider` instance exists and that `[authProvider login]` has

successfully completed as documented in Authenticating to the Cloud. To use the API, follow the steps below.

Step 1. Create a `CloudDocStore` instance.

Example: **Creating a CloudDocStore Instance**

```
@import CDKCore;
@import CDKDocStore;

    // create 'authProvider'

[CloudDataStore
    init:authProvider
    baseUrl:@"https://authserver1.wearables.host.com"];
    CloudDataStore *docStore = [CloudDataStore getInstance];
```

Step 2. Create a JSON document.

Example: **Creating a JSON Document**

```
NSMutableDictionary *sampleDocument = [[NSMutableDictionary alloc] init];
sampleDocument[@"test1"] = @"value1";
sampleDocument[@"test2"] = @"value2";
sampleDocument[@"test3"] = @"value3";
sampleDocument[@"test4"] = @"value4";
```

Step 3. Upload the document.

Example: **Uploading a JSON Document**

```
[docStore
  put:@"2222"
  document:sampleDocument
  success:^(CloudResponse *response) { ... }
  failure:^(CloudResponse *response) { ...}];
```

Step 4. Retrieve the document as needed.

Example: **Retrieving a JSON Document**

```
[docStore
  get:@"2222"
  success:^(CloudResponse *response) { ... }
  failure:^(CloudResponse *response) { ... }];
```

7.7 Accessing Cloud Device Profiles

The `CDKDeviceProfileStore` module stores and retrieves custom JSON documents that contain details about the companion device used to connect to the cloud. To access a device profile, the user must be logged in. This entails that an `authProvider` instance exists and that `[authProvider login]` has successfully completed as documented in Authenticating to the Cloud. To use the API, follow the steps below.

Step 1. Create a `CloudDeviceProfileStore` instance.

Example: **Creating a CloudDeviceProfileStore Instance**

```
@import CDKCore;
@import CDKDeviceProfileStore;
...
// see 'core' tutorial to learn how to create 'authProvider'.
    [CloudDeviceProfileStore init:identifier baseUrl:@"https://fc.wearables.intel.com"];
    CloudDeviceProfileStore *profileStore = [CloudDeviceProfileStore
                                             getInstance];
```

Step 2. Create a `CloudDeviceProfileItem` device profile.

Example: **Creating a CloudDeviceProfileItem Device Profile**

```
NSMutableDictionary *sampleProfile = [[NSMutableDictionary alloc] init];
sampleProfile[@"Name"] = @"Utility Belt";
sampleProfile[@"FirmwareVersion"] = @"1.2.0";
CloudDeviceProfileItem *item = [[CloudDeviceProfileItem alloc]
                                        initWithProfileContent:sampleProfile];
```

Step 3. Upload the profile to the cloud.

***Example:* Uploading a Device Profile to the Cloud**

```
[profileStore putWithDeviceType:"Wearable", deviceId: "my_device",
                      profile:sampleProfile
                      success:^(CloudResponse *response) {...}
                      failure:^(CloudResponse *failure {...}];
```

Step 4. Retrieve the profile from the cloud as needed.

***Example:* Retrieving the Device Profile from the Cloud**

```
[profileStore getWithDeviceId:"my_device",
                      success:^(CloudResponse *response) { ... }
                      failure:^(CloudResponse *response) { ... }];
```

7.8 Accessing Cloud BLOB Data and Software Assets

The `CDKBlobStore` module provides upload and download capabilities for BLOB data and software assets. The BLOB store API allow an application to upload and download arbitrary binaries to the cloud. Depending on the access restrictions imposed on the binary, one of three BLOB stores can be accessed. The following sections contain details.

- The `CloudAnonymousBlobStore` class allows for uploading of anonymous binary files. They are accessible by an administrator only. See Using CloudAnonymousBlobStore for details.
- The `CloudPublicBlobStore` class allows for downloading software assets like device firmware. See Using CloudPublicBlobStore for details.
- The `CloudBlobStore` class allows for uploading and downloading files belonging to a given user. Authentication is required for access. See Using CloudBlobStore for details.

7.8.1 Using CloudAnonymousBlobStore

The `CloudAnonymousBlobStore` class allows uploading anonymous binary files to the cloud. For example, an application administrator can use it to allow users to upload anonymous feedback about the application. Authentication is required although the files are stored anonymously. To use the anonymous BLOB store, follow the steps below.

Step 1. Create a `CloudAnonymousBlobStore` instance. The product ID argument is a string of your choice. When downloading, the administrator will be able to use calls like `[getList]` to retrieve available BLOBs by product ID and tag.

Example: **Creating a CloudAnonymousBlobStore Instance.**

```
@import CDKCore;
@import CDKBlobStore;
...
    [CloudAnonymousBlobStore init:@"https://authserver1.wearables.host.com"
                           productId:@"some_product_id"];
    CloudAnonymousBlobStore *anonymousBlobStore = [CloudAnonymousBlobStore getInstance];
```

Step 2. Upload some NSURL *f. The file tag argument can be any string.

Example: **Uploading some NSURL *f**

```
[anonymousBlobStore upload:f tag:@"some_tag"
                           success:^(CloudBlobResponse *response) {...}
                           failure:^(CloudBlobResponse *response) {...}
                           progress:^(int64_t bytesSent, int64_t totalBytesSent,
                               int64_t totalBytesExpectedToSend) {...}
                           complete:^(CloudBlobResponse *response) {...}];
```

To upload a file using a different name than the one associated with it (in this case, `f`), use the following API call.

Example: **Uploading a File Using a Different Name**

```
[anonymousBlobStore upload:file tag:@"some_tag" name:@"custom_name"
                   success:^(CloudBlobResponse *response) {...}
                   failure:^(CloudBlobResponse *response) {...}
                   progress:^(int64_t bytesSent, int64_t totalBytesSent,
                         int64_t totalBytesExpectedToSend) {...}
                   complete:^(CloudBlobResponse *response) {...}];
```

7.8.2 Using CloudPublicBlobStore

The `CloudPublicBlobStore` class allows downloading of binary files like firmware. These files are available to all users of the application thus no authentication is required. To use the public BLOB store, follow the steps below.

Step 1. Create a `PublicBlobStore` instance.
Make sure the value of `productId` is valid in that assets have been defined with one when uploaded using the administrative interface. See Using CloudBlobStore for details.

Example: **Create a PublicBlobStore Instance**

```
@import CDKCore;
@import CDKBlobStore;
...
    [CloudPublicBlobStore init:@"https://authserver1.wearables.host.com"
                          productId:@"some_product_id"];
    CloudPublicBlobStore *publicBlobStore = [CloudPublicBlobStore getInstance];
```

Step 2. Download a file named `my_asset`.
This call can be invoked multiple times. The first time results in a success callback and successive calls result in the progress or finished callback as applicable. The download call can therefore be placed, for example, in any lifecycle method.

Example: **Downloading the my_asset File**

```
```objc -(void) viewWillAppear { [super viewWillAppear];
CloudPublicBlobStore *publicBlobStore = [CloudPublicBlobStore getInstance];

// download to your application zone
[publicBlobStore download:@"my_asset"
  success:^(CloudBlobResponse *response) {...}
  failure:^(CloudBlobResponse *response) {...}
  progress:^(int64_t bytesSent, int64_t totalBytesSent, int64_t totalBytesExpectedToSend){...}
complete:^(CloudBlobResponse *response) {/* do stuff */ }];
}
```

Step 3. Use the following call to download a file and save it with a different name.

Example: **Saving the my_asset File with a Different Name**

```
[publicBlobStore download:@"my_asset" name:@"new_name"
                                success:^(CloudBlobResponse *response) {...}
                                failure:^(CloudBlobResponse *response) {...}
                                progress:^(int64_t bytesSent, int64_t totalBytesSent,
                                    int64_t totalBytesExpectedToSend) {...}
                                complete:^(CloudBlobResponse *response) {...}];
```

Important semantics of the download calls:

- The download file name (for example, `my_latest_voice_pack`) is appended to the root of your zone.
- The SDK does not download duplicates. This implies that if the destination path already exists, then the SDK will immediately callback finished.
- If a file with the given filename is already downloading, the SDK will callback progress.
- If the application is no longer in memory after the download call is made, the application must make the same download call again once its started. Without this call, the SDK will not have callbacks registered and so won't be able to inform the application when the download is finished. In this situation, the
```

SDK will log an error message saying that it could not call the application back.

- At this time, progress is not called periodically as the file is downloading.

To change the way downloads are handled by the SDK, set the `cellularAccess` parameter to true and the SDK will download on WiFi only.

```
[CloudPublicBlobStore init:/*...*/ cellularAccess:true);
```

7.8.3 Using CloudBlobStore

The `CloudBlobStore` class allows uploading and downloading binary files that belong to an authenticated user. The user must be logged in which entails that an `authProvider` instance exists and that `[authProvider login:]` has successfully completed as documented in Authenticating to the Cloud. To use the cloud BLOB store, follow the steps below.

Step 1. Create a `CloudBlobStore` instance. The product ID argument is a string of your choice. All uploads should define this parameter which will be useful in a future release.

Step 2. Upload a file. The file tag argument can be any string.

Step 3. Download the file just uploaded.
This call can be invoked multiple times. The first time results in the `onStarted()` callback while successive calls result in the `onProgress()` or `onFinished()` callback as necessary. For details on when each of the download callbacks are invoked, see the semantics documented in Using CloudPublicBlobStore.

Step 4. Use upload and download calls to upload or download using a custom file name.

7.9 Accessing Time Series Data

The `CDKTimeSeries` module stores, retrieves and analyzes time series data points. (This data typically consists of successive measurements made over a time interval; for example, GPS measurements.) Time series data is published and retrieved as a list of observations with one being a set of data points. To access time-series data for a user profile, the user must have a credential identifier as documented in Authenticating to the Cloud. To use the API, follow the steps below.

Step 1. Instantiate a `CloudTimeSeries` object (currently implemented as a Singleton).

Example: **Instantiating a CloudTimeSeries Object**

```
CloudTimeSeries.initWith(identifier: authProvider.getIdentifier(), baseUrl:"https://
wearables.intel.com")
let timeSeries = CloudTimeSeries.getInstance()
```

Step 2. Upload a set of observations as a JSON object.

Example: **Uploading a Set of Observations**

```
timeSeries.post(jsonObject:structuredObservations) {
    // do stuff on success
}
```

Step 3. Retrieve the observations in a window defined by a start date and an end date.

Example: **Retrieving Observations**

```
timeSeries.getWindowRawData(
  startDate: startDate,
  endDate: endDate,
  eventType: "my type",
  deviceId: "wearableId",
  filter: filter,
  success: {(response: CloudResponse) in
        // Do stuff on success
    },
  failure: {(response: CloudResponse) in
        // Do stuff on failure
    })
```

CHAPTER

8 Third-Party License Information

This chapter contains the following third-party software licenses:

- Alamofire Software Foundation License
- Apache License
- BSD 2-Clause License
- Domestic Cat Software License
- FMDB License
- GNU General Public License
- Magical Panda Software License
- MIT License
- PromiseKit License

8.1 Alamofire Software Foundation License

8.2 Apache License

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
* You must give any other recipients of the Work or Derivative Works a copy of this License; and
* You must cause any modified files to carry prominent notices stating that You changed the files; and
* You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
* If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or

any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Note: Other license terms may apply to certain, identified software files contained within or distributed with the accompanying software if such terms are included in the directory containing the accompanying software. Such other license terms will then apply in lieu of the terms of the software license above.

END OF TERMS AND CONDITIONS

8.3 BSD 2-Clause License

Copyright (c) 2015, BSD
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8.4 Domestic Cat Software License

Copyright (C) 2011 by Patrick Richards domesticcatsoftware.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8.5 FMDB License

If you are using FMDB in your project, I'd love to hear about it. Let Gus know by sending an email to gus@flyingmeat.com.

And if you happen to come across either Gus Mueller or Rob Ryan in a bar, you might consider purchasing a drink of their choosing if FMDB has been useful to you.

Finally, and shortly, this is the MIT License.

Copyright (c) 2008-2014 Flying Meat Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS ORIMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND

NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8.6 GNU General Public License

Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
Preamble
The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.
When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.
To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.
For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.
We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.
Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.
Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.
The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION
0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)
These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.
In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)
The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy

both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.
NO WARRANTY
11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS
How to Apply These Terms to Your New Programs
If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.
one line to give the program's name and an idea of what it does.
Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'. This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

8.7 Magical Panda Software License

8.8 MIT License

CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8.9 PromiseKit License

Copyright 2015, Max Howell; mxcl@me.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.