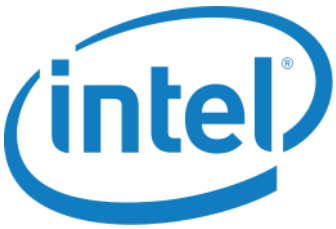
Intel Software Platform for Curie

Over-the-Air (OTA) Firmware Update Guide

Version 1.0

January 2016

BETA DRAFT



Part Number 333880-001US

Contents

List of Code Samples

Chapter 1: About This Guide

Chapter 2: Updating Firmware on Curie Wearable Devices

List of Code Samples

CHAPTER

1 About This Guide

This guide provides instructions for using Intel Curie Software Platform APIs and Intel Cloud Services to update firmware over-the-air (OTA) on Intel Curie-based wearable devices.

This chapter contains the following sections:

1.1 Who Should Read This Guide

This guide is designed for Android and iOS developers.

To use the Intel Curie Software Platform for Curie SDK, developers should have proficiency in Java or Swift, and experience developing mobile applications for the Android or iOS mobile platforms.

1.2 Terminology Used in This Guide

Table 1: Terms and Definitions

Term	Definition
Android	Android mobile operating system
BLE	Bluetooth Low Energy radio, a wireless personal area network technology
BLOB	Collection of binary data stored as a single entity
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	iOS mobile operating system
JSON	JavaScript Object Notation, a lightweight data-interchange format
OTA	Over-the-air transfer of files
REST	Representational State Transfer, a software architectural style of the World Wide Web
SoC	Intel Quark SE System on a Chip
UUID	Universally Unique Identifier
Wearable Platform	Intel Software Platform for Curie The name may be 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 Javadoc API Reference for Android*
- *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 Updating Firmware on Curie Wearable Devices

This chapter contains the information you need to complete the following steps:

Step 1: Get Access to the Firmware Image

- Download the Sources
- Build Environment Setup
- Creating an OTA Firmware Image

Step 2: Get the Firmware Image into Your Mobile Application Using Intel Cloud Services

- Registering your Mobile Application with Intel Cloud Services
- Generating Your Mobile Application Public Token
- Upload the Firmware Image to Intel Cloud Services
- Download the Firmware Image from Intel Cloud Services

Step 3: Get the Firmware Image into the Application Without Using Intel Cloud Services

Step 4: Upload Firmware to the Wearable Device

- Using the Firmware Update APIs on Android
- Using the Firmware Update APIs on iOS

2.1 Step 1: Get Access to the Firmware Image

You can flash the Curie Module with either a pre-built binary image found in Github, or you can build a custom firmware image described in the following sections.

- To flash the Curie Module with a pre-built binary, see the complete *Intel® Curie™ Module Software User Guide.*
- To build a custom over-the-air (OTA) firmware image, complete the steps described in the following sections:
  - Download the Sources
  - Build Environment Setup
  - Creating an OTA Firmware Image

These sections are excerpted from "Intel® Curie™ Module Software User Guide.

2.1.1 Download the Sources

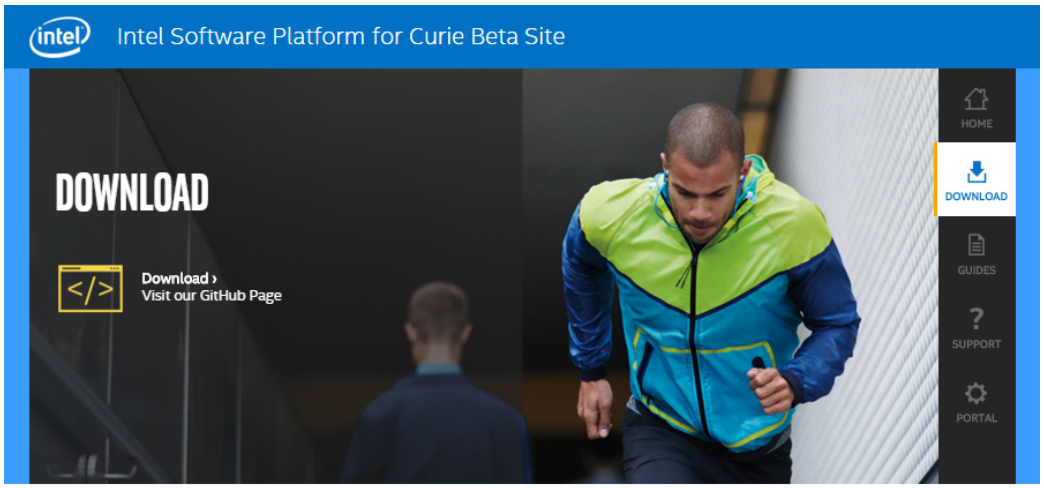
Step 1. Log in to Intel Software Platform for Curie Beta Site (`https://sandbox.iqsoftwarekit.intel.com`).

The Developer Portal landing page:

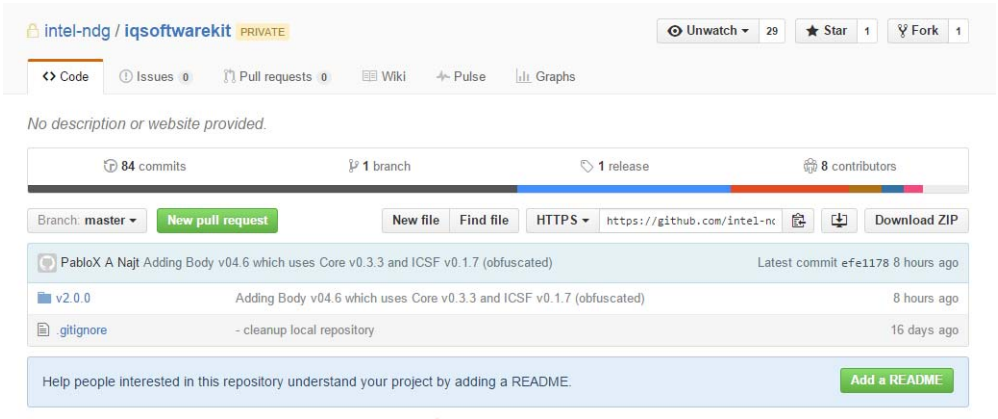
Step 2. Click on the "Sign In" icon and enter your registered e-mail and password to login.

Step 3. Once you log in, you can see sidebars on the home page. Click on the "Download" icon which will redirect you to the Intel® GitHub server.

Note: Please make sure you are logged in to GitHub prior to clicking on the "Visit our GitHub Page" icon.



Step 4. Click on the "Download Zip" icon to download the sources.



Step 5. Unzip the downloaded source files on your Linux machine:

```
$ unzip iqsoftwarekit-master.zip
```

Step 6. (Optional) As an alternative, you can download the source code using the following `git` command:

```
$ git clone https://github.com/intel-ndg/iqsoftwarekit.git
```

2.1.2 Build Environment Setup

Note: All the commands below to configure, create, and flash an Intel® Curie™ module-based project must be issued from the project directory using GNU make. All Intel® Curie™ module-based projects can be found in the folder `<intel_iq_sdk>/wearable_device_sw/projects/.`

The SDK build system allows a single project to produce firmware images differentiated along three dimensions:

- The PROJECT identifies the features.
- The BOARD identifies the hardware.
- The BUILDVARIANT is either debug or release.

Example: **Setting Up the Build Environment**

```
$ cd </path/to/project>
# For example;
$ cd wearable_device_sw/projects/curie_reference/
$ sudo make one_time_setup   //This command checks for all the prerequisites of the host
                             //and needs to be run only once

 //make setup BOARD=<xxx> BUILDVARIANT=<yyy>
$ make setup
```

Where the environment variables take the values as explained below, if nothing is entered after the `make setup` command, a set of default values for “BOARD=crb” and “BUILDVARIANT=release” are assigned.

1. xxx i.e., BOARD – crb (Intel® Curie™ Reference Board)

2. yyy i.e., BUILDVARIANT – release or debug

For more information, type:

```
$ make help
```

Or optionally view the project Makefile which can provide many useful details. The setup command involves SDK setup and project setup. The SDK setup will create the build output directory and prepare the environment:

- Compile host tools.

- Save setup parameters.

Several build environments can coexist under a project tree, all located under `<intel_iq_sdk>`. The naming convention for a specific build output directory is:

`./out/PROJECT_BOARD_BUILDVARIANT`

The current environment (the last one you did a setup for) is identified using a symbolic link.

Example:

```
├── curie_reference_crb_release
├── current -> /home/user/iqsoftwarekit/v2.0.0/device_software/
                intel_iq_sdk/wearable_device_sw/../out/curie_reference_crb_release
└── host_tools
```

2.1.3 Creating an OTA Firmware Image

Use the `make otapackage` command to create OTA firmware images. For example:

```
$ make otapackage
```

The resulting binaries are created in the folder `<intel_iq_sdk>/out/current/ota/pub/`.

The OTA firmware image is `package.ota.bin`.

Next Steps

Once you have a firmware image, you can either upload the image to Intel Cloud Services, or transfer the image directly to the wearable device.

- To upload the image to Intel Cloud Services, see Step 2: Get the Firmware Image into Your Mobile Application Using Intel Cloud Services on page 10.
- To transfer the image directly to the wearable, go to Step 3: Get the Firmware Image into the Application Without Using Intel Cloud Services on page 23.

2.2 Step 2: Get the Firmware Image into Your Mobile Application Using Intel Cloud Services

To complete this step, follow the instructions in the following sections:

- Registering your Mobile Application with Intel Cloud Services
- Generating Your Mobile Application Public Token
- Upload the Firmware Image to Intel Cloud Services
- Download the Firmware Image from Intel Cloud Services

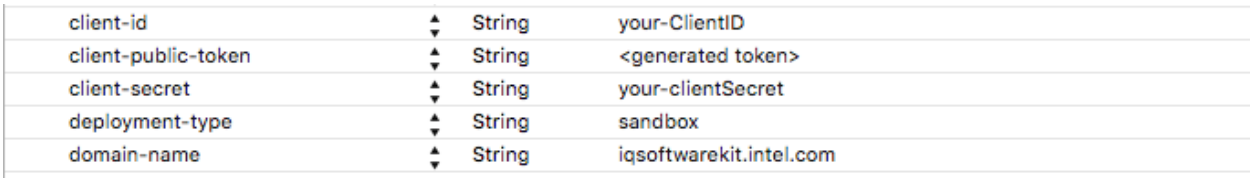
2.2.1 Registering your Mobile Application with Intel Cloud Services

This section is excerpted from the "Intel Curie Developer Portal Administrator's Guide."

When a new mobile application has been developed for use with the Curie Module, it must be registered as a client using the Developer Portal. When the application is added, it is assigned a Client ID and a Client Secret. The Client Secret must be integrated into the `AndroidManifest.xml` file of an Android application, or the `info.plist` file of an iOS application. The `AndroidManifest.xml` configuration will likely look like this.

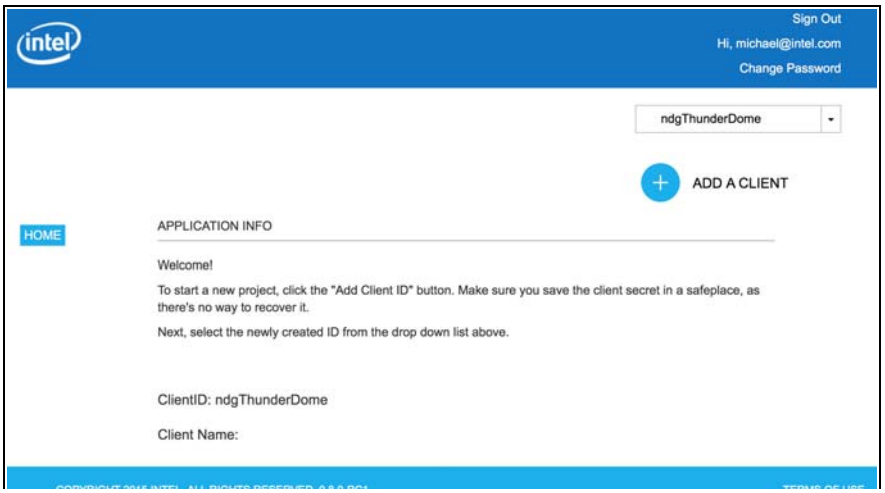
```
<meta-data android:name="deployment-type" android:value="sandbox" />
<meta-data android:name="domain-name" android:value="iqsoftwarekit.intel.com" />
<meta-data android:name="client-id" android:value="your-clientID" />
<meta-data android:name="client-secret" android:value="your-clientSecret" />
<meta-data android:name="client-public-token" android:value=<generated token> />
```

For iOS add the following keys as String types to the `info.plist` file.



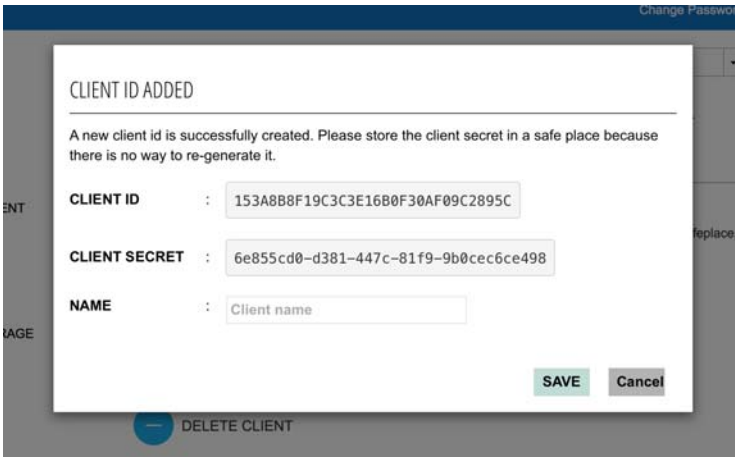
For information about the generated token, see Generating Your Mobile Application Public Token on page 12.

A developer can register a maximum of ten mobile applications. The following procedure assumes you have successfully logged into the Developer Portal, and the Home page is displayed.



Step 1. Click Add A Client to generate a new Client ID.

The Client ID Added page is displayed.



Step 2. Copy the Client ID and Client Secret to a text file.

There is no way to regenerate an assigned Client Secret so be sure to copy and paste the value before clicking Save. Both Client ID and Client Secret will be used in the `AndroidManifest. xml` and `info.plist` files.

Step 3. Enter a name for the client (for example, Wearable App) and click Save.

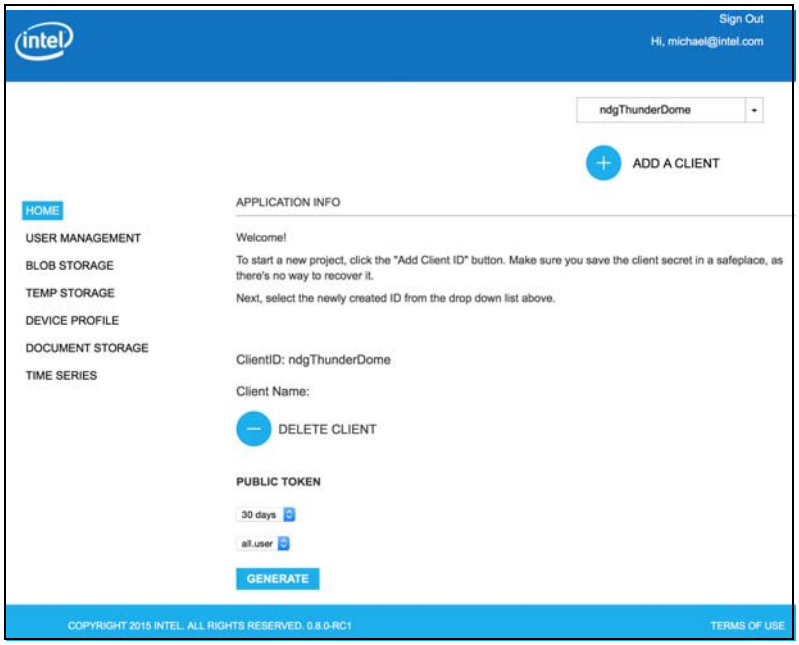
The Client ID of the new application will now be an option in the drop down list of registered applications on the top right of the Home page. When selected, its Client ID and Client Name are displayed on the Home page. At this point, you can manage the application's resources by clicking any of the links in the left side bar. This includes links for BLOB storage, user management and device profiles. To delete the client application, click Delete Client.

You must also generate a Public Token to authenticate the application. For details, see Generating Your Mobile Application Public Token on page 12.

2.2.2 Generating Your Mobile Application Public Token

Once a developer has logged into the portal, all applications registered to that email address are accessible from the drop down list at the top right of the Home page. When the Client ID assigned to an application is selected from the

drop down, the Client ID and Client Name are displayed on the Home Page.



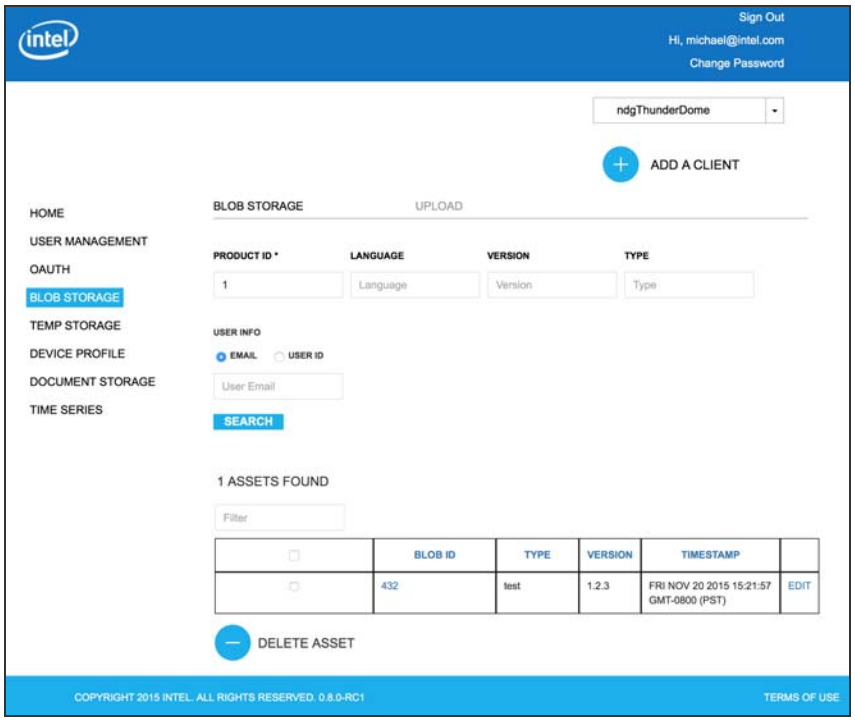
This Home Page is also where you generate a Public Token for the registered (and displayed) application. The Public Token is a long string used to authenticate the application for access to the Cloud Services public storage. The public token must be integrated into the `AndroidManifest.xml` file an Android app, or the `info.plist` file of an iOS app as are the Client ID and Client Secret. See Registering your Mobile Application with Intel Cloud Services on page 10 for examples.

To generate the Public Token, select the length of time for which the token will be valid and click Generate. The token can then be copied from the value field and incorporated into the application's code. For access to other storage (such as time series data), a user token is needed. A user email and password is exchanged for a user token when the user logs in from a companion device (smart phone). Tokens can be generated for periods of one day, 30 days, one year or ten years.

2.2.3 Upload the Firmware Image to Intel Cloud Services

For both Android and iOS, use the Intel Developer's Portal BLOB Storage page to upload the firmware image to Intel Cloud Services.

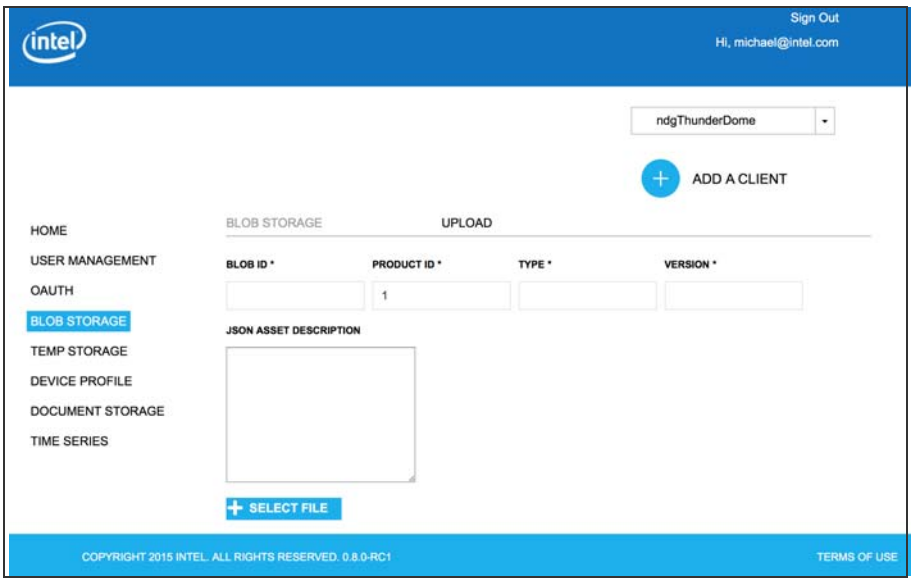
The BLOB Storage page is where the developer can upload Binary Large Object (BLOB) collections of firmware, software updates, voice packs, map data and other general purpose application data. BLOB data can be searched by Product ID, Langauge, Version or Type.



A user can upload files that the developer has deemed appropriate for the application. Files that a user has uploaded to public storage are accessible from this BLOB storage page. An email address and User ID can be used to find files mapped to a specific user.

Firmware updates for the Curie Module are also delivered using BLOB storage. To push firmware updates to the chip, the developer goes to the Intel Firmware Upload page, downloads the appropriate update and then uploads it to

this BLOB storage page - associating the BLOB with an application's ID.



Follow this procedure to upload a firmware file. This procedure assumes the update has been downloaded from an Intel server to your local drive.

Step 1. Click Upload to display the Upload page.

Step 2. Enter a BLOB ID, a Product ID, a Type, and a Version number.

Step 3. Enter the following information:

- BLOB ID
  Developer-defined identifier for the firmware file
- Product ID
  Developer-defined identifier associating the firmware with a product
- Type
  Enter the word "Firmware" (case-sensitive, no quotes).
- Version number
  Enter a positive integer using the form <Major>.<Minor>.<Patch>

Example: 2.0.1 is the first patch of the 2.0.0 Release.

- Description of the JSON asset
  Provide additional information associated with the firmware file in JSON format.

Step 4. Click Select File to display the drive search box and navigate to the firmware file.

Step 5. Click Upload to complete the process.

2.2.4 Download the Firmware Image from Intel Cloud Services

For Android. See the next sectionUsing CloudPublic-BlobStore on Android.

For iOS. See Using CloudPublicBlobStore on iOS on page 19.

2.2.4.1 Using CloudPublicBlobStore on Android

This section is excerpted from the "Intel Software Platform for Curie Android Developer's Guide."

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:

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 the code example on the next page.

Example: Create a PublicBlobStore Instance

```
/**
 * Download over WiFi only.
 *
 * Default is false
 */
final boolean wifiOnly = false;


/**
 * specifies whether download should be shown in the Android Downloads UI or not.
 *
 * Default is true.
 */
final boolean showInDownloadApp = true;

/**
 * Since the download file appears before it is complete, we will place all the in-progress
 * downloads in a temporary directory (place holder) and move it to the destination directory
 * once the download completes
 *
 * Default value is "temporaryAssetDownloads"
 */
final String temporaryDir = null;

mCloudPublicBlobStore = new CloudPublicBlobStore(PRODUCT_ID, new CloudDownloadPolicy(wifiOnly,
showInDownloadApp, temporaryDir),
        Application.getContext());
```

Step 2. Check for a new firmware version, and download the file.

See the code example on the next page.

Example: Downloading the Firmware File

```
/** Gets name of firmware file asset in the cloud and also checks to see if the version in the
cloud is higher than the version on the device(in which case it should be updated).
The code in this sample uses the Semantic Versioning library here:

https://github.com/zafarkhaja/jsemver/blob/master/src/main/java/com/github/zafarkhaja/
semver/Version.java

which allows for comparing version numbers. */

mCloudPublicBlobStore.getLatestDownloadInfo("Firmware", new ICloudResponseCb() {
    @Override
    public void onSuccess(CloudResponse cloudResponse) {
        try {
            JSONObject jsonObject = cloudResponse.getPayload();
            if (jsonObject != null) {
                JSONObject objAsset = jsonObject.getJSONObject("swAssetInfo");
                if (objAsset != null) {
                    my_asset = objAsset.getString("assetId");
                    final String version = objAsset.getString("assetVersion");
                    Logger.i("Found version " + version);
                    Version cloudFirmwareVersion = Version.valueOf(version);

                    if (firmwareController != null) {
                     deviceFirmwareVersion = (firmwareController.getWearableFirmwareVersion()
!= null && firmwareController.getWearableFirmwareVersion().length() > 0)
                                ?
Version.valueOf(firmwareController.getWearableFirmwareVersion()) : null;
                    }
                      if (deviceFirmwareVersion == null ||
                             cloudFirmwareVersion.greaterThan(deviceFirmwareVersion)) {
                      // if this flag is true, allow downloading the new version
                        updateAvailable = true;

                    } else {
                        updateAvailable = false;
                    }
                }
            }
        } catch (final JSONException e) {
            Logger.e(e.getMessage());
        }
    }

    @Override
    public void onFailed(final CloudResponse cloudResponse) {
        Logger.e(cloudResponse.getError().toString());
    }
});
```

This code sample is continued on the next page.

Example: **Downloading the Firmware File (Continued)**

```
//             … … … {If updateAvailable is true} … … …

mCloudPublicBlobStore.download(my_asset, saveInFolder, saveAsFilename, new ICloudDownloadCb()
{
    @Override
    public void onStarted(CloudDownloadResponse cloudDownloadResponse) {
       Logger.i("Download started.");
    }

    @Override
    public void onFailed(final CloudDownloadResponse cloudDownloadResponse) {
       Logger.e("Download failed.");
    }

    @Override
    public void onFinished(CloudDownloadResponse cloudDownloadResponse) {
       String absPath =
Application.getContext().getExternalFilesDir(null).getAbsoluteFile().getAbsolutePath() +
File.separator + fileName;
        firmwareFileForUploadToWearable = new File(absPath);
        Logger.i("firmwareFile.exists(): " + firmwareFileForUploadToWearable.exists());
        Logger.i("firmwareFile.getAbsolutePath(): " +
firmwareFileForUploadToWearable.getAbsolutePath());
    }
```

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

Example: **Setting the wifiOnly Value**

```
/**
 * Download over WiFi only.
 *
 * Default is false
 */
final boolean wifiOnly = false;
```

2.2.4.2 Using CloudPublicBlobStore on iOS

This section is excerpted from the "Intel Software Platform for Curie iOS Developer's Guide."

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. 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.

Example: **Create a PublicBlobStore Instance**

```
import IQCore
import CDKCore
import CDKBlobStore
...
let ProductId = /* Your product id */
let authProvider = CloudAppAuthProvider.init()
authProvider.login()
var blobStore: CloudPublicBlobStore?

do {
        blobStore = try CloudPublicBlobStore(productId: ProductId, doAllowCellularAccess: true)
} catch {
        Log.warning?.message("Problem with public token in info.plist")
}
```

Step 2. Check for a new firmware version, and download the file.

See the code sample on the next page.

Example: Downloading the Firmware File

```
// Hold on to firmware info
struct CloudFirmwareInfo {
    var assetId: String
    var productId: String
    var version: String
    var path: String?

    init(assetId: String, productId: String, version: String) {
        self.assetId = assetId
        self.productId = productId
        self.version = version
    }
}
...

// Get the latest firmware version information from the cloud blob store
var updateAvailable = false
blobStore?.getLatestDownloadInfo("Firmware", success: { cloudResponse in
       let JSONPayload = cloudResponse.getJsonPayload()
       guard let assetInfo = JSONPayload["swAssetInfo"] as? NSDictionary else {
              let message = "Couldn't get JSON payload"
              Log.warning?.message(message)
              return
       }
       guard let assetId = assetInfo["assetId"] as? String,
              let productId = assetInfo["productId"] as? String,
              let version = assetInfo["assetVersion"] as? String else {
                     let message = "No Firmware Found"
                     Log.warning?.message(message)
                     return
       }

       let firmwareInfo = CloudFirmwareInfo(assetId: assetId, productId: productId, version:
version)
       self.latestCloudFirmwareInfo = firmwareInfo

       //The firmware version is in semantic version format. So it's possible to parse and do
//a greater than comparison. However, we're just checking for inequality here

       if self.deviceFirmwareRevision != self.latestCloudFirmwareInfo?.version {
              dispatch_async(dispatch_get_main_queue()) { [weak self] in
                     self?.updateAvailable = true
                     Log.info?.message("Firmware available for download...")
              }
       }
}, failure: { response in
       let title = "Latest Firmware Check Failed"
       let message = "\(response.getCloudError())"
       self.presentDefaultAlert(title, subText: message, dismissText: "OK")
       Log.warning?.message(title + ": " + message)
})
```

This code sample is continued on the next page.

Example: **Downloading the Firmware File (Continued)**

```
// ... ... ... {If updateAvailable is true} ... ... ...
guard let firmwareAssetId = latestCloudFirmwareInfo?.assetId else {
        let message = "No firmware asset id"
        Log.warning?.message(message)
        return
}

blobStore?.download(firmwareAssetId, saveAsFile: firmwareAssetId, success: { cloudResponse in
        dispatch_async(dispatch_get_main_queue()) { [weak self] in
                Log.info?.message("Download started")
        }
        }, failure: { cloudResponse in
                Log.warning?.message("Failed to download:  \(cloudResponse.getCloudError())")
        }, progress: { [weak self] bytesWritten, totalBytesWritten, totalBytesExpectedToWrite in
                dispatch_async(dispatch_get_main_queue()) {
                        let progress = (Float)(totalBytesWritten)/
(Float)(totalBytesExpectedToWrite)
                        let percentComplete = (Int64)(progress * 100.0)
                        Log.info?.message("Percent complete: \(percentComplete)%")
                }
        }, completed: { [weak self] cloudResponse in
                dispatch_async(dispatch_get_main_queue()) {
                        let docDirPath = NSSearchPathForDirectoriesInDomains(.DocumentDirectory,
.UserDomainMask, true).first!
                        let docPath = docDirPath + "/" + firmwareAssetId
                        self?.latestCloudFirmwareInfo?.path = docPath
                        Log.info?.message("Completed download: \(docPath)")
                }
})
```

To change the way downloads are handled by the SDK, set the `doAllowCellularAccess` parameter to true when initializing the `CloudPublicBlobStore`, and the SDK will download over Wifi and cellular data.

Example: **Setting the cellularAccess Value**

```
CloudPublicBlobStore(_, doAllowCellularAccess: true)
```

2.3 Step 3: Get the Firmware Image into the Application Without Using Intel Cloud Services

Simply include your firmware image in the application bundle.Then you can upload the firmware image directly to the wearable device. See Step 4: Upload Firmware to the Wearable Device.

2.4 Step 4: Upload Firmware to the Wearable Device

Once your firmware file is included in your application file, you can use the Firmware Update APIs to upload the firmware directly to the wearable device.

For Android. See Using the Firmware Update APIs on Android on page 23.

For iOS. See Using the Firmware Update APIs on iOS on page 26.

2.4.1 Using the Firmware Update APIs on Android

This section is excerpted from the "Intel Software Platform for Curie Android Developer's Guide."

Firmware Update APIs are provided in the `com.intel.wearable.platform.core.firmware` package of the Wearable Platform Core SDK for Android mobile platform.

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

2.4.1.1 Obtaining the Firmware Controller

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

Example: **Obtaining the Firmware Controller**

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.firmware.IFirmwareController;
...
private IWearableController wearableController; private IFirmwareController
firmwareController;
 if (wearableController != null) { if (wearableController.getFirmwareController() != null) {
 firmwareController = wearableController.getFirmwareController(); } else {
 throw new NullPointerException(
 "This device does not support firmware upgrades."); }
... }
```

Below are a few examples demonstrating how this API can be used in your applications.

2.4.1.2 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 `firmwareController.getWearableFirmwareVersion()` method.

2.4.1.3 To Update the Wearable Device Firmware

To install firmware on the wearable device, call the `firmwareController.installFirmware(File firmwareFile, IFirmwareInstallListener listener)` method.

Supply as the first parameter a reference to the file containing the new firmware revision.

Example: **Updating the Wearable Device Firmware**

```
private File firmwareFile = new File("absolute path to firmware file");
...
firmwareController.installFirmware(firmwareFile, new FirmwareInstallEventManager());
```

Supply as the second parameter a callback implementing the `IWearableListener` interface to monitor status of the firmware file transfer onto the wearable device as it progresses.

Example: **Monitoring Status of the Firmware File Transfer**

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.firmware.IFirmwareInstallListener;
public class FirmwareInstallEventManager implements IFirmwareInstallListener {
	@Override
	public void onFirmwareInstallStarted(IWearableController controller) {
		//firmware file transfer onto wearable device has started
	}
	@Override
	public void onFirmwareInstallProgress(IWearableController controller,
			int progress, int total) {
		// [progress] bytes of firmware file [total] bytes
		// have been transfered onto wearable device
	}
	@Override
	public void onFirmwareInstallComplete(IWearableController controller) {
		// firmware file transfer onto wearable device has completed successfully
	}
	@Override
	public void onFirmwareInstallError(IWearableController controller,
	com.intel.wearable.platform.core.error.Error error) {
		// error occurred during the first transfer
	}
}
```

The following occur in the firmware update process:

1. The wearable device reboots to device firmware update mode.
2. The firmware update file is transferred to the wearable.
3. The wearable device reboots to apply the new firmware.

The firmware file containing a new revision of wearable device firmware can be obtained using Intel Cloud Services.

When initializing your application, after calling the `Core.init(context)` method, your application can create a new instance of the `CloudPublicBlobStore` class.

2.4.2 Using the Firmware Update APIs on iOS

This section is excerpted from the "Intel Software Platform for Curie iOS Developer's Guide."

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.

2.4.2.1 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)
...
```

2.4.2.2 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.

2.4.2.3 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 file.

Supply as the second parameter a closure to be called when the firmware file transfer onto the wearable device has started.

The remaining parameters are closures for monitoring progress, and completion for the firmware installation on the wearable device.

The following occur in the firmware update process:

1. The wearable device reboots to device firmware update mode.
2. The firmware update file is transferred to the wearable.
3. The wearable device reboots to apply the new firmware.

The progress closure is called periodically to update the progress of the firmware file transfer.

The completed closure is called after the final reboot and firmware installation. It is normal for a few minutes to elapse between the last call to the progress closure and the completed closure.

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

```
firmwareController.installFirmware(pathToFirmwareFileString, started: {
        Log.info?.message(â€œStarted firmware update")
}, progress: { [weak self] bytesSent, bytesTotal in
        let progress = (Float)(bytesSent)/(Float)(bytesTotal)
        var percentComplete = (Int64)(progress * 100.0)
        Log.info?.message("Progress update \(percent complete)")
}, completed: { [weak self] version in
        Log.info?.message("Installation completed.")
}, failed: { error in
        Log.warning?.message("Installation failed: \(error.localizedDescription)")
})
```