Digi Embedded for Android

Internal documentation - Do not distribute

Table of Contents

Digi Embedded for Android

Android™ is one of the most popular and easy-to-use operating systems for mobiles devices and embedded systems, and every day thousands of new devices use this OS.

In this section you will find all the information you need to work with your ConnectCore device 6 and Android, from straightforward getting started steps to detailed API documentation.

Digi Embedded for Android in a nutshell
Learn a bit more about Android and the value Digi adds to it.

Get started
Program Android into your ConnectCore 6 device, boot it for the first time and create your first application.

Application development
Create Android applications from scratch or by importing a Digi sample for your ConnectCore 6 device.

System development
Download Android and take the first steps with the ConnectCore 6 platform to create and build your own Android image.

FAQ [Android]
Get answers to the most common and frequently asked questions related to the ConnectCore 6 platform and Android.

Digi Embedded for Android in a nutshell

Android is now integrated into multiple embedded systems that take it well beyond its traditional use in the consumer electronics market.

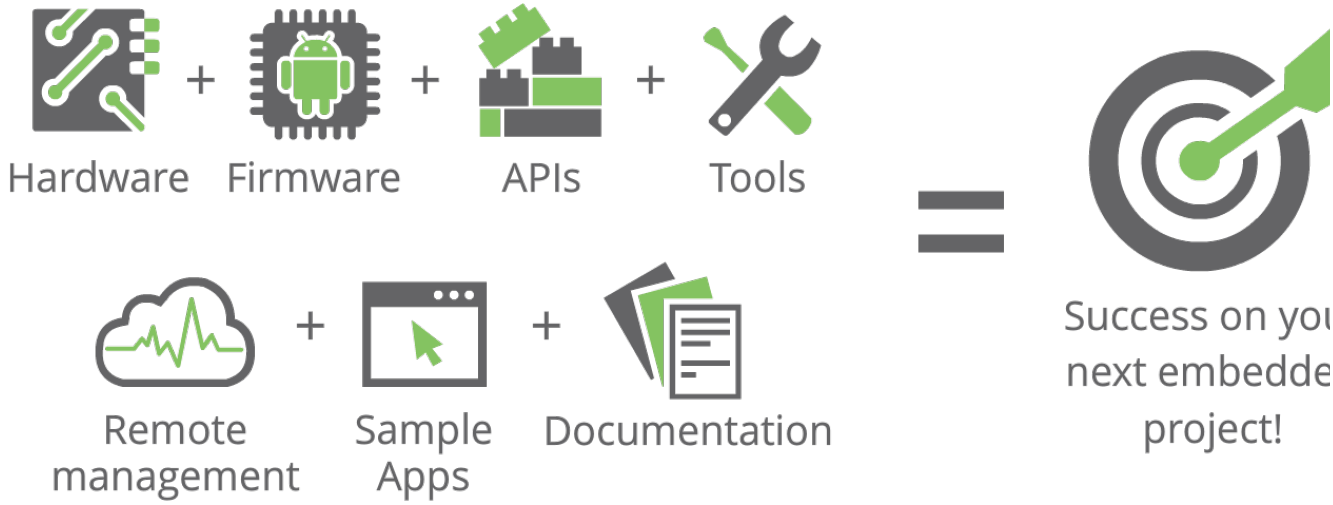
Android is becoming an excellent choice for embedded systems that have a rich, multimedia-heavy user interface. Additionally, the powerful communication capabilities of Android (cellular, Wi-Fi, Bluetooth, and more) make it the perfect OS to build smart and connected devices for the IoT era.

But the use of Android as an embedded OS still presents several challenges. The standard Android doesn't provide APIs to access many peripherals and interfaces present in most embedded systems, such as GPIO, I2C, SPI or CAN. Therefore, the firmware and software developers have to implement all the code to access these peripherals, which is not trivial. This task requires solid knowledge about the internal architecture of Linux and Android and avoids focusing on the specific application that the embedded system runs.

These embedded systems are not oriented to the consumer electronics market, and in most cases the standard Android distributions do not fit customer requirements. Adapting the standard Android platform to embedded systems requires several modifications and optimizations. Additionally, access to low-level hardware interfaces such as GPIO, I2C, or SPI is very common in the embedded universe, and Android does not provide a standard way to do it.

What does Digi provide?

Digi Embedded for Android provides the required hardware and software to effectively design embedded products with Android. Digi's unique product offering combines all the necessary ingredients needed for the success of your next embedded product, built with Android.



- **Hardware**. ConnectCore 6 is an ultra-compact and highly integrated system-on-module solution based on the NXP i.MX6 Cortex-A9 processor family. The ConnectCore 6 family provides great scalability and flexibility through the different variants of the module to meet the needs of your project.
- **Firmware** . Digi provides a customized Android BSP for ConnectCore 6. The software includes the following components:
  - A **U-Boot** based bootloader to boot the Android system. U-Boot is easily configurable at boot time and integrates a useful set of commands to program your own scripts. Digi provides source code that you can tailor to fit the needs of your design and integration within the build system. This way, when you build the Android image, the U-Boot image is also built with it.
  - A **Linux Kernel** based on the 3.14 version fully adapted for ConnectCore 6, with the required Android modifications to work with the ConnectCore 6 hardware and all its interfaces. The firmware includes the kernel source code, which you can easily customize to the needs of your

embedded system. For example, you can modify the peripheral configuration, and add new drivers and modules. Like U-Boot, the kernel is also integrated in the build system so it is built with the Android image.

- The **Android** system is customized and optimized to run on the ConnectCore 6 module. It includes all the new Digi APIX to talk with the hardware as well as other extensions such as Wi-Fi or Ethernet. The firmware includes the sources of the Android image, along with the required instructions to build it.

> Digi Embedded for Android includes pre-built images to accelerate your development process.

- **APIX**. The API extensions provide an abstraction layer to access embedded peripherals and interfaces that are not available as part of the standard Android framework.
- **Tools**. Digi also provides extension tools for Android Studio to help you work with Digi Sample applications and access the documentation.
- **Remote management** . Remotely monitor and analyze the device, access the Android file system, manage the configuration of some interfaces, reboot the module, or update an Android application or the entire system, through the integrated Device Cloud support.
- **Sample Apps**. A set of sample applications demonstrates how to use each of the APIs as well as other common Android functionality.
- **Documentation**. Digi Embedded for Android comes with complete and comprehensive documentation covering all the development and technical aspects of the ConnectCore 6 device that you can access online at any time.

Get started

This section guides you through your first steps with Android and the ConnectCore 6.

You will learn how to connect your hardware, install Android, and create and execute an application that switches an LED on and off for ConnectCore 6 SBC.

1. Requirements

To start working with Android and ConnectCore 6 SBC, you need the following items:

- A JumpStart Development Kit.
  - Verify your kit contains the following components...

Qty.	Part	Image
1	ConnectCore 6 SBC	
1	Power supply with universal adapters	
1	Console cable adapter	
1	Antenna cable	

1	Antenna	

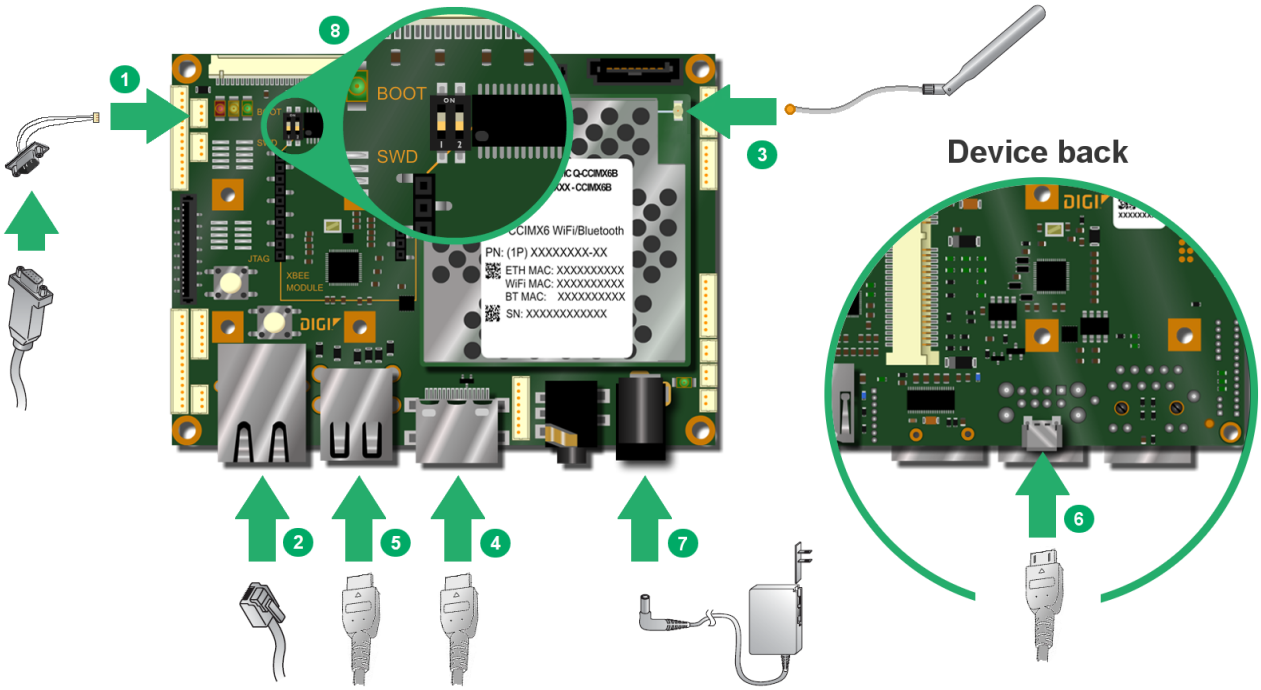
- A Windows or Linux machine able to run Android Studio. Find the Android Studio system requirements at http://developer.android.com/sdk/index.html#Requirements.

- Some additional components not included in the kit:
  - HDMI monitor
  - HDMI cable
  - USB mouse
  - Micro USB cable
  - Computer with serial port interface
  - Null-modem serial cable
  - A 2 GB FAT-formatted micro SD card

See supported software to verify that your hardware is compatible with Android Lollipop.

2. Hardware setup

Follow these steps to set up the hardware components of your ConnectCore 6 JumpStart Development Kit.



1. Connect the serial adapter cable to the console port [CONS]. Connect a serial cable from the adapter to the development computer.
2. (Optional) Connect the Ethernet cable to the Ethernet port.
3. Connect the antenna to the UFL connector.
4. Connect an HDMI cable to the HDMI connector, and then to an HDMI compatible display.
5. Connect a USB mouse to the USB connector.
6. Connect a micro USB cable to the development computer.
7. Add the appropriate plug and connect the power supply.
8. Ensure the BOOT micro-switches are both OFF (down position).

3. Program the Android firmware

The ConnectCore 6 SBC ships without an operating system loaded. Only U-Boot is programmed in the device. The following instructions demonstrate how to install Android in your ConnectCore 6 SBC.

1. Establish a serial connection with your device

Before you start programming the firmware, you must open a serial connection with the device. You can use any serial terminal program such as Tera Term, PuTTY, Minicom, CoolTerm, or HyperTerminal.

Open a serial connection with the following settings:

- **Port**: Serial port where ConnectCore 6 SBC is connected
- **Baud rate**: 115200
- **Data Bits**: 8
- **Parity**: None
- **Stop Bits**: 1
- **Flow control**: None

2. Program the firmware

Once you have established the serial connection with your device, you can start with the firmware update process

1. Download the firmware images from <TODO: insert link here to the zip with images (uboot.imx, boot.img, system.img) + script (gs-android.scr) + readme for Android>
2. Decompress the <insert_name_here>.zip.
3. Place the decompressed files in the root of a FAT formatted micro SD card and insert it in the micro SD socket of the ConnectCore 6 SBC.
4. Reset the device by pressing the **Reset** button on the board, and immediately press a key in the serial terminal to stop the auto-boot process. You will be stopped at the U-Boot bootloader prompt:

```
U-Boot dub-2015.04-r3.1 (Mar 14 2016 - 17:06:20), Build:
jenkins-uboot_release-45

CPU:   Freescale i.MX6Q rev1.5 1200 MHz (running at 792 MHz)
CPU:   Extended Commercial temperature grade (-20C to 105C) at 47C
Reset cause: POR
I2C:   ready
DRAM:  1 GiB
MMC:   FSL_SDHC: 0 (eMMC), FSL_SDHC: 1
In:    serial
Out:   serial
Err:   serial
Board: ConnectCore 6 SBC v1
Variant: 0x02 - Consumer quad-core 1.2GHz, 4GB eMMC, 1GB DDR3,
-20/+70C, Wireless, Bluetooth, Kinetis
Boot device: MMC4
PMIC:  DA9063, Device: 0x61, Variant: 0x50, Customer: 0x00, Config:
0x56
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
=>
```

5. Install the firmware into the internal eMMC, by executing the following commands:

```
=> fatload mmc 1 $loadaddr install_android_kit_fw.scr
=> source $loadaddr
```

During installation, error messages appear when the installation process attemps to override MAC addresses. These messages are expected.

Error messages:

```
## Resetting to default environment
oldval: 00:40:9D:7D:17:9A  defval: 00:04:f3:ff:ff:fa
## Error: Can't overwrite "ethaddr"
himport_r: can't insert "ethaddr=00:04:f3:ff:ff:fa" into
hash table
oldval: 00:40:9D:7D:17:9B  defval: 00:04:f3:ff:ff:fb
## Error: Can't overwrite "wlanaddr"
himport_r: can't insert "wlanaddr=00:04:f3:ff:ff:fb" into
hash table
oldval: 00:40:9D:7F:4A:C1  defval: 00:04:f3:ff:ff:fc
## Error: Can't overwrite "btaddr"
himport_r: can't insert "btaddr=00:04:f3:ff:ff:fc" into
hash table
```

Once the firmware is installed, the device automatically boots.

The first Android boot takes several minutes due to the system deployment.

6. Verify Android has started and unlock the welcome screen.
Work with your ConnectCore 6 SBC as with any standard Android device. Use the **All Apps** button to navigate and explore the applications or configure your ConnectCore 6 SBC in the Settings application.

4. Create your first application

This tutorial demonstrates how to develop and launch your first Android application in the ConnectCore 6 SBC. The application allows you to blink the ConnectCore 6 SBC LED on and off.

Before creating your project, make sure your device is correctly connected, powered and running. Once your device is ready for development, follow these steps:

4.1. Install the software

You must have the following software components in order to start developing Android applications for your ConnectCore 6:

1. Java SE Development Kit 7 (JDK 7)
2. Android Studio
3. Digi Extensions for Android Studio
4. SDK Add-on for ConnectCore 6
5. Google USB driver

If you already have Android Studio 2.0 or later in your computer, you can skip the first two installation steps.

1. Java SE Development Kit 7 (JDK 7)

Before you set up Android Studio, be sure you have installed **JDK 7** or later. The Java Runtime Environment (JRE) alone is not sufficient.

For Linux users:

We recommend the Java **OpenJDK-7** package. Depending on your distribution, you can install this package in two ways:

- For Debian and Ubuntu distributions, issue this command:

```
$ sudo apt-get install openjdk-7-jdk
```

- For Fedora, Oracle Linux or Red Hat Enterprise Linux, issue this command:

```
$ su -c "yum install java-1.7.0-openjdk-devel"
```

For Windows users:

1. Check if you already have a JDK 7 or later installed on your computer. If you do, you can skip this step.
   How to check if you have a JDK installed and which version
   a. Open a terminal and type the following:

```
~> javac -version
```

   b. The version should be 7 or later. Java SE version strings have the form:
      - `1.x`
      - `1.x.0`
      - `1.x.0_u`

      In these examples, **x** is the product version number and u is the update version number. For example, an installed JDK 7 update 60 (JDK 7u60) returns:

```
~> javac -version
javac version "1.7.0_60"
```

2. If the JDK is not installed or the version is not 7 or later:

a. Download the Java SE Development Kit 7 at http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
b. Once the download is complete, run the executable file and follow the on-screen instructions to finish the installation process.

2. Android Studio

Android Studio provides everything you need to start developing apps for Android. Digi Embedded for Android has been validated with Android Studio 2.0, although newer versions should also work.

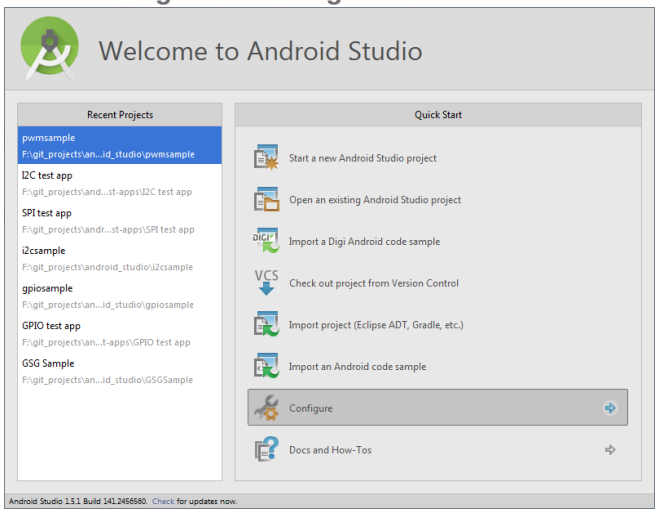
To download and install Android Studio on your computer:

1. Download Android Studio at http://tools.android.com/download/studio/stable.
2. Follow the steps for installing Android Studio at Installing Android Studio.

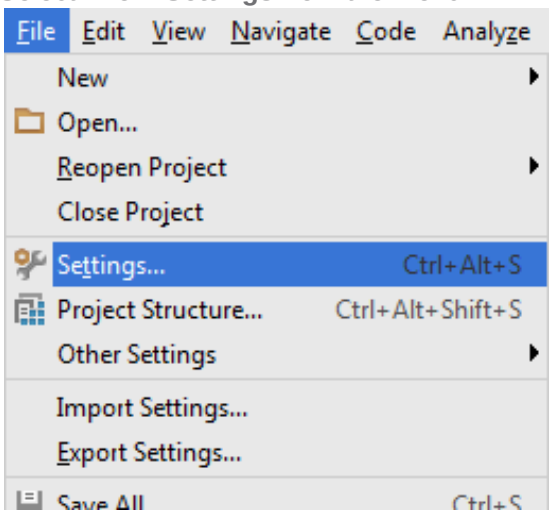
3. Digi Extensions for Android Studio

To download and install the Digi Extensions on your Android Studio:

1. Open Android Studio.
2. Open the **Android Studio Settings** dialog from by doing one of the following:
   - Select **Configure > Settings** on the Welcome screen.



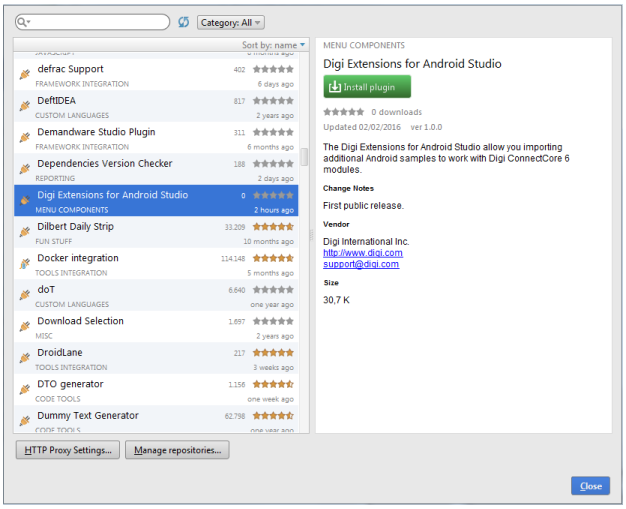
   - Select **File > Settings** from the menu.



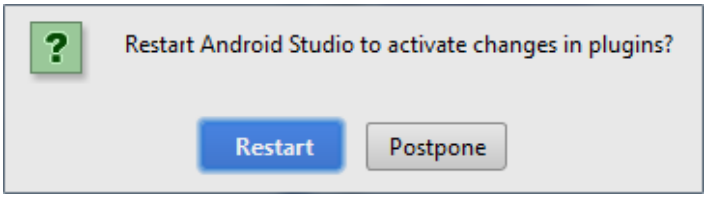
3. Select **Plugins** from the tree on the left of the **Settings** dialog.

> You can also access to the **Plugins** dialog from the Welcome screen. Click **Configure** on the Quick Start panel and then click **Plugins**.

4. Click **Browse repositories** at the bottom of the page.
5. Inside the **Browse Repositories** dialog, select **Digi Extensions for Android Studio** from the list on the left. You can use the search box to look for the plugin.



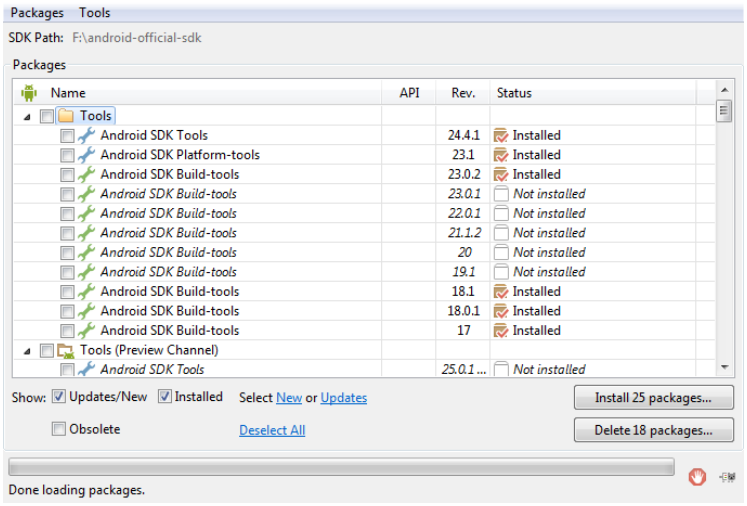
6. Click **Install plugin** on the right panel.
7. A dialog is displayed asking whether you would like to download and install the plugin **Digi Extensions for Android Studio**. Click **Yes**.
   A dialog displays the progress of the download process.
8. Once the download process finishes, click **Close** on the **Browse Repositories** dialog.
9. Click **OK** to close the **Plugins** dialog.
10. When a message appears asking you to restart, click **Restart** to activate the plugin.



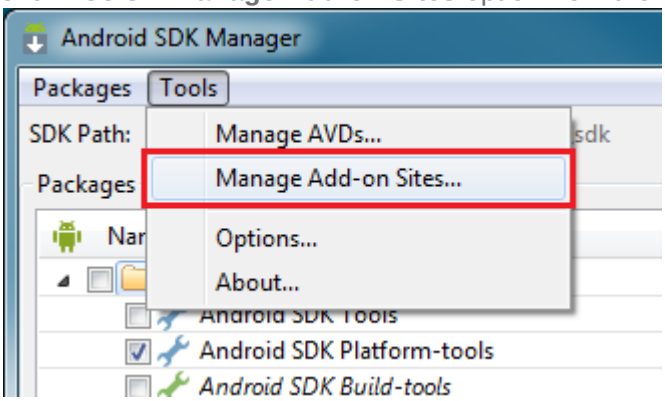
4. SDK Add-on for ConnectCore 6

You are now ready to start developing Andoid applications. The Get started section provides an initial tutorial, while the Application development section gives you everything you need to start using the Digi APIX for Android extension to create more advanced apps..

1. On the Android Studio Welcome screen, click **Configure** on the Quick Start panel and then click **SDK Manager**.
2. Click the **Launch Standalone SDK Manager** link at the bottom of the list. The **SDK Manager** dialog opens.

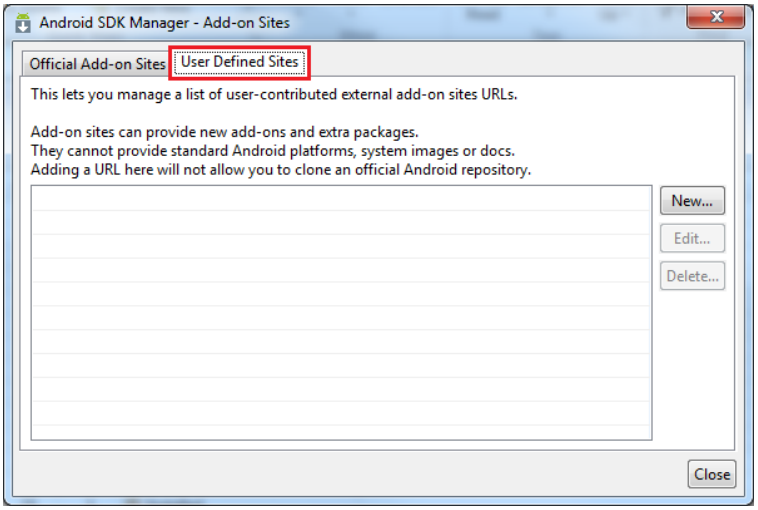


3. Click **Deselect All** to uncheck any previously selected package from the list.
4. Click **Tools** > **Manage Add-on Sites** option from the main menu of the application.

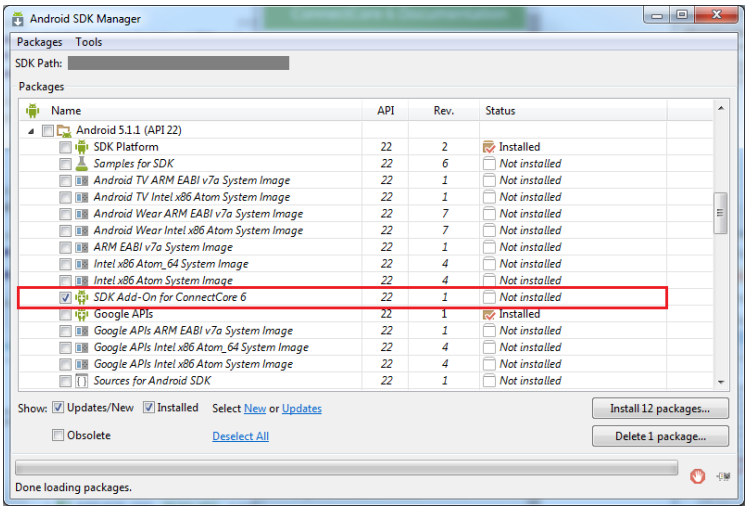


A new configuration window opens

5. Select the **User Defined Sites** tab and click on **New** button to add a new add-on site.



6. Enter the following URL for the new add-on Site: *<TODO file:///X:/Engineering/products/Android Studio Plugins/builds/v1.0.0_sprint7/1/addons/addon.xml>*
7. Click **OK** and then close the **Add-on Sites configuration** dialog. A new package named **SDK Add-On for ConnectCore 6** appears under **Android 5.1.1 (API 22)**.



8. Select the new package and click **Install 1 package**.
9. Accept the License Agreement and click **Install**.
10. Once installation is complete, close the SDK Manager.
11. Restart Android Studio.

You are now ready to develop Android applications using the custom Digi APIX for Android.

5. Google USB driver

You need the Google USB Driver for Windows if you want to perform adb (Android Debug Bridge) debugging with your ConnectCore device.

1. Download the Google USB driver at http://developer.android.com/sdk/win-usb.html.
2. Once you have downloaded your USB driver, perform the following instructions to install it, based on your Windows version:

Windows 7 and higher

To install the Google USB driver on Windows 7 or later for the first time:

1. Unzip the downloaded file.
2. Connect your Android-powered device to your computer's USB port.
3. Right-click on **Computer** from your desktop or Windows Explorer, and select **Manage**.
4. Select **Device Manager** in the left pane of the **Computer Management** window.
5. Locate and expand **Other device** in the right pane.
6. Right-click the device name and select **Update Driver Software** to launch the **Update Driver Software** wizard.
7. Select **Browse my computer for driver software** and click **Next**.
8. Click **Browse** and locate the USB driver folder, called `usb_driver`, wich appears after you decompress the downloaded file.
9. Click **Next** to install the driver.

Windows Vista

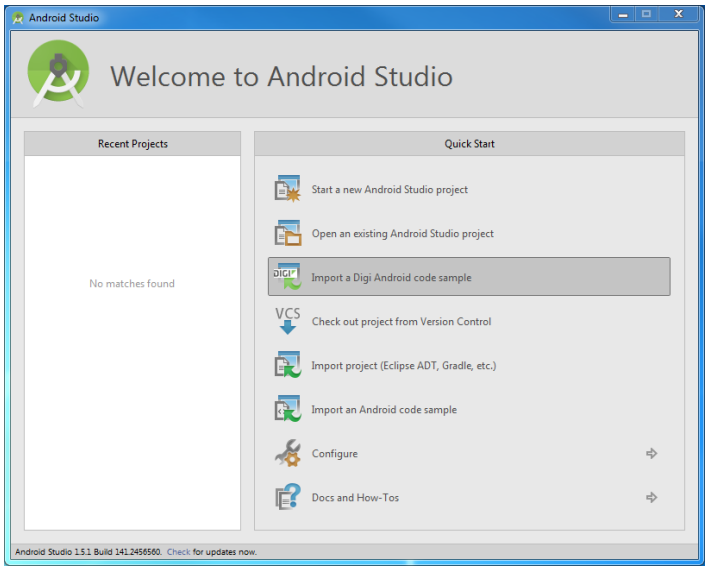
To install the Google USB driver on Windows Vista for the first time:

1. Unzip the downloaded file.
2. Connect your Android-powered device to your computer's USB port. Windows detects the device and launches the **Found New Hardware** wizard.
3. Select **Locate and install driver software**.
4. Select **Don't search online**.
5. Select **I don't have the disk. Show me other options**.
6. Select **Browse my computer for driver software**.
7. Click **Browse** and locate the USB driver folder, called `usb_driver`, which appears after you decompress the downloaded file.
8. Click **Next**. Vista may prompt you to confirm the privilege elevation required for driver installation. Confirm it.

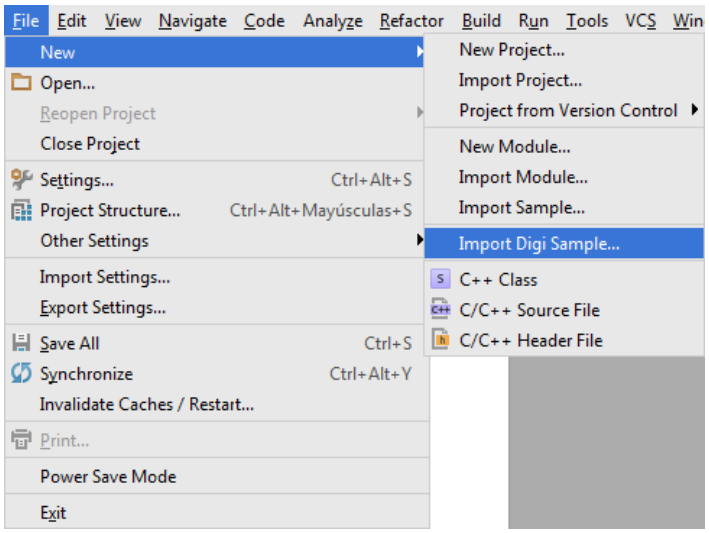
9. When Vista asks if you'd like to install the Google ADB Interface device, click **Install** to install the driver.

4.2. Create the Android application

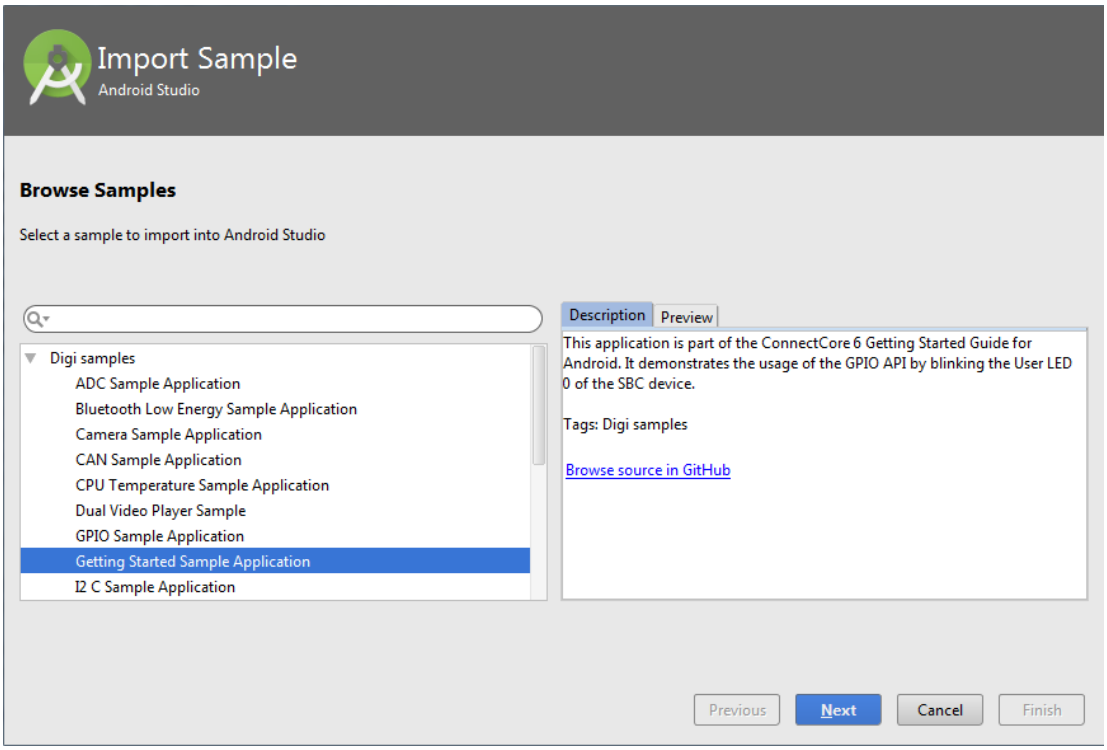
1. Open Android Studio.
2. Open the **Import Digi Sample** wizard by doing one of the following:
   - Select **Import a Digi Android code sample** on the Welcome screen.



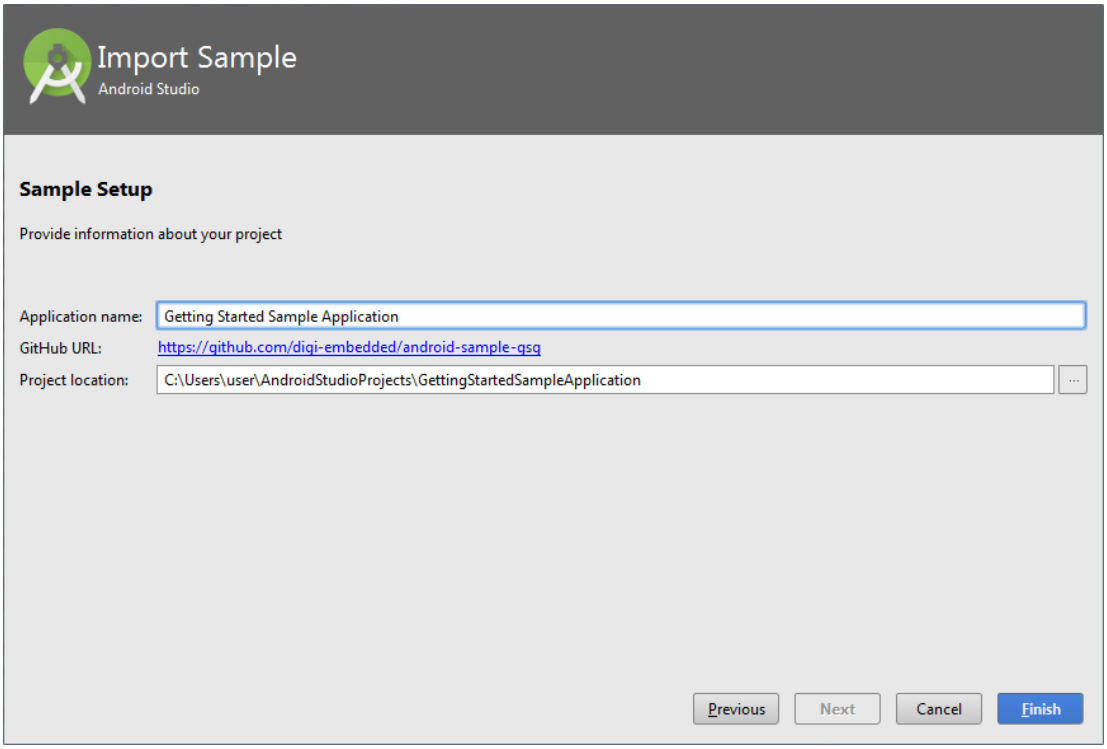
   - Select **File > New > Import Digi Sample** from the menu.



3. On the **Browse Samples** page of the wizard, select the **Getting Started Sample Application** under **Digi Samples**. Then click **Next**.

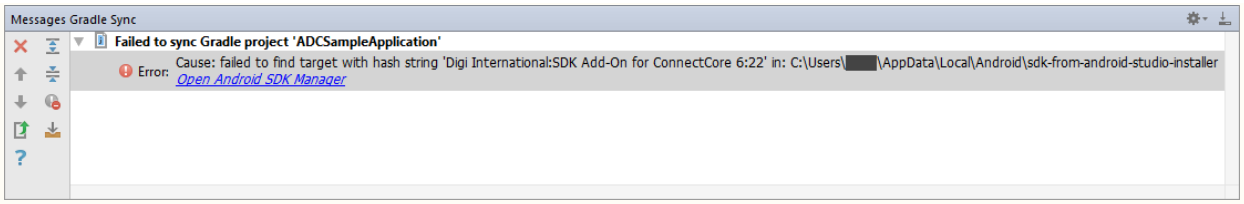


4. In the **Project location** field of the **Sample Setup** page of the wizard, specify the path where you want to download the sources of the application.



5. Click **Finish** to create the sample application. A new Android Studio window opens with the Getting Started Sample Application project.

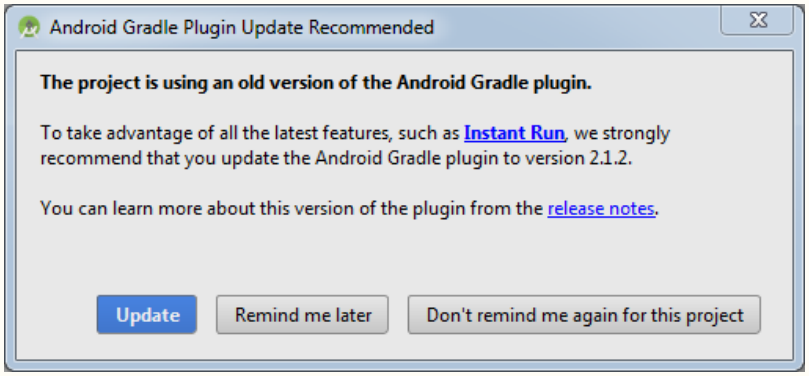
Due to an Android Studio issue, if the following Gradle synchronization error appears:



Follow these steps to fix the issue:

1. Go to **File > Project Structure**.
2. Select the item inside **Modules** section in the list on the left.
3. In the **Compile Sdk Version** drop-down , select **API 22: Android 5.1 (Lollipop)**.
4. Click **OK**. Android Studio starts to index the selected SDK.
5. Once the indexing process finishes, go to **File > Project Structure**.
6. Select the item inside **Modules** section in the list on the left.
7. In the **Compile Sdk Version** drop-down, select **SDK Add-On for ConnectCore 6. Android 22 (API 22)**.
8. Click **OK**.

The Android Gradle plugin is in ongoing development, but the Digi samples are configured with a specific Gradle version. For that reason, it's likely that Android Studio recommends you to update it when you import any example.



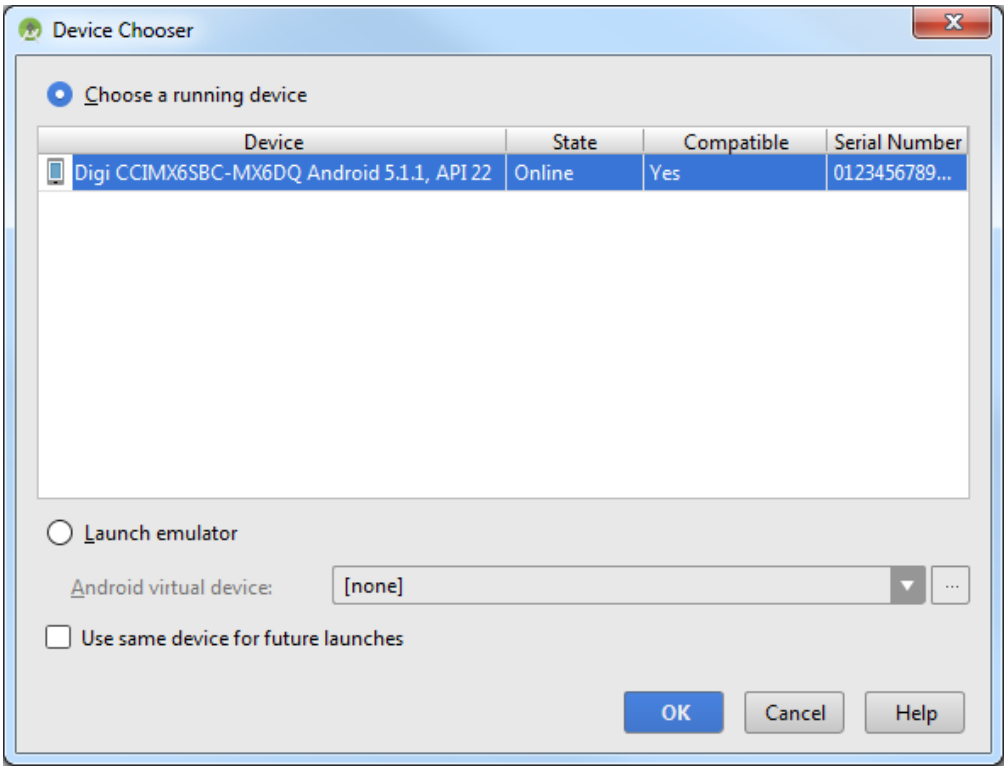
To do so, click **Update** and Android Studio will take care of the update.

4.3. Launch the Android application

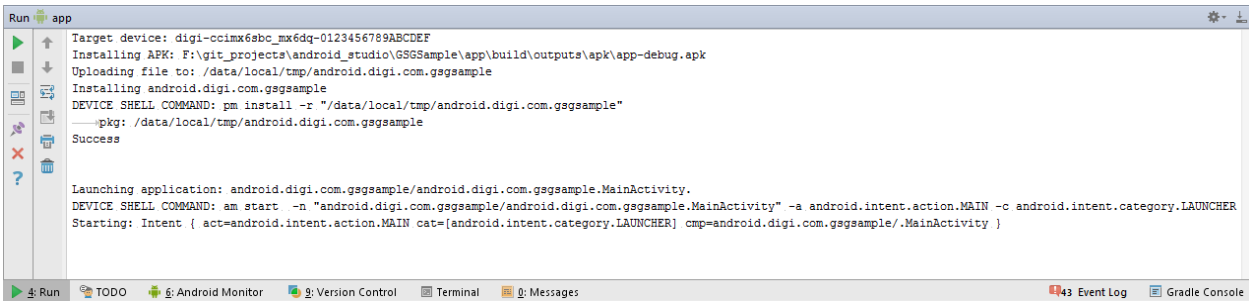
Once the project is created, you can launch it in your Android device:

1. Click **Run > Run 'app'**.
2. Select your device from the **Device Chooser** dialog and click **OK**.

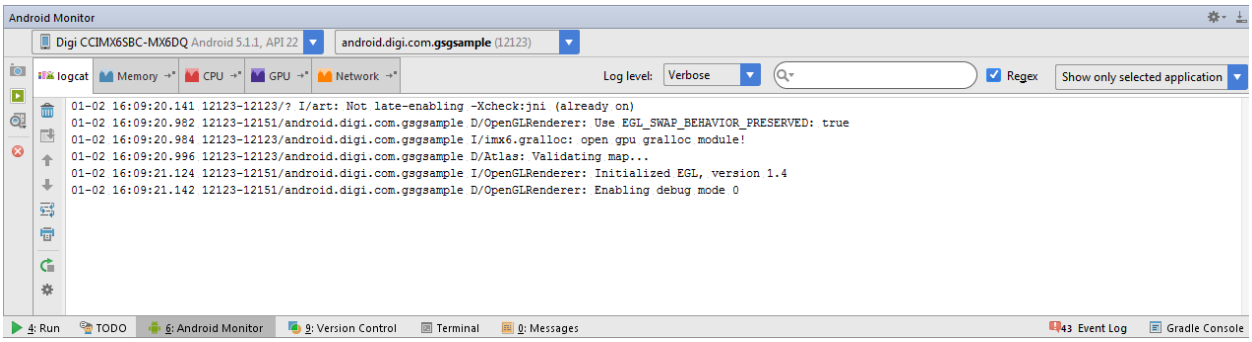
> If your device appears as **Unauthorized** in the **Device Chooser** dialog, go to your Android device. A dialog shows up on your device, presenting an RSA key for your computer and asking for confirmation to accept the connection.
>
> Select **Always allow from this computer** option and click **OK**.
>
> After accepting your computer RSA key, the device state changes to **Online** on the **Device Chooser** dialog.



The application automatically transfers and launches in the Android device through the USB connection. The progress of the launch process is displayed in the Run view:

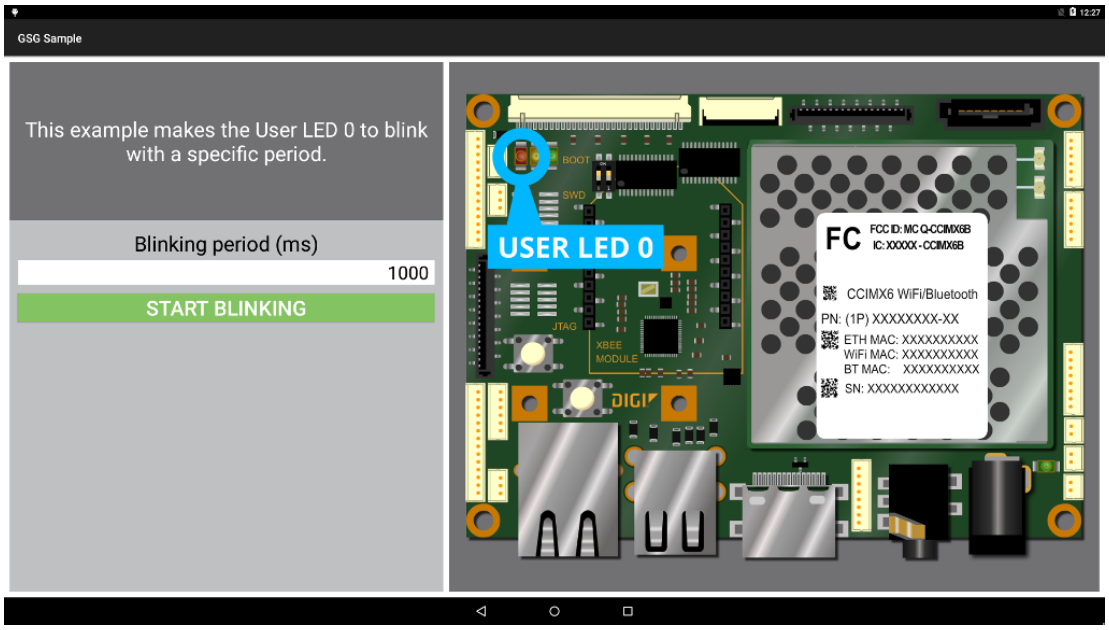


3. The Android Monitor view is automatically open. You can review the adb and device log messages from the logcat tab:



4.4. Test the Android application

Testing the Getting Started Sample application in the Android device is very simple. You can blink the User LED on and off and configure its period.



You have successfully created and executed an Android application in your ConnectCore 6 SBC. As you can see, the process is the same as with any other standard Android device.

Now you can develop more applications with Android Studio using Digi APIs to access the hardware interfaces available in your ConnectCore 6 SBC, such as the SPI, I2C, ADC, and CAN.

To meet all your project requirements, Digi also lets you customize Android to create your own firmware images from the source code.

5. Next steps

Now that you have finished the Get started tutorial, it is time to learn more about all the other possibilities that the kit offers.

Application development

Create an Android application and launch it in your ConnectCore 6.
Use Digi Android APIs to easily manage from Java the device hardware interfaces, such as ADC, CAN, GPIOs, I2C, etc.

System development

Learn how to build the Digi Embedded Android sources and create your own images to fit your project requirements.

FAQ

Visit the FAQ topic for more information about Digi Embedded Android and your ConnectCore 6.

Application development [Android]

Digi Embedded for Android includes a set of sample applications ready to be compiled and executed in your ConnectCore 6 device. You can use these sample applications as a reference to create your own Android application or start developing one from scratch. In addition, you can use the Android API Extensions, Digi APIX, to access and manage all the ConnectCore 6 platform interfaces in an easy manner.

- Create an Android application
- Digi APIX for Android

Create an Android application

The way you create and develop an Android application for the ConnectCore 6 is similar to how you create an application for Android devices, such as a tablet or mobile phone: You select the appropriate SDK for the platform, which allows your application to access the functional extensions that Digi uses to control the hardware not supported by the standard Android.

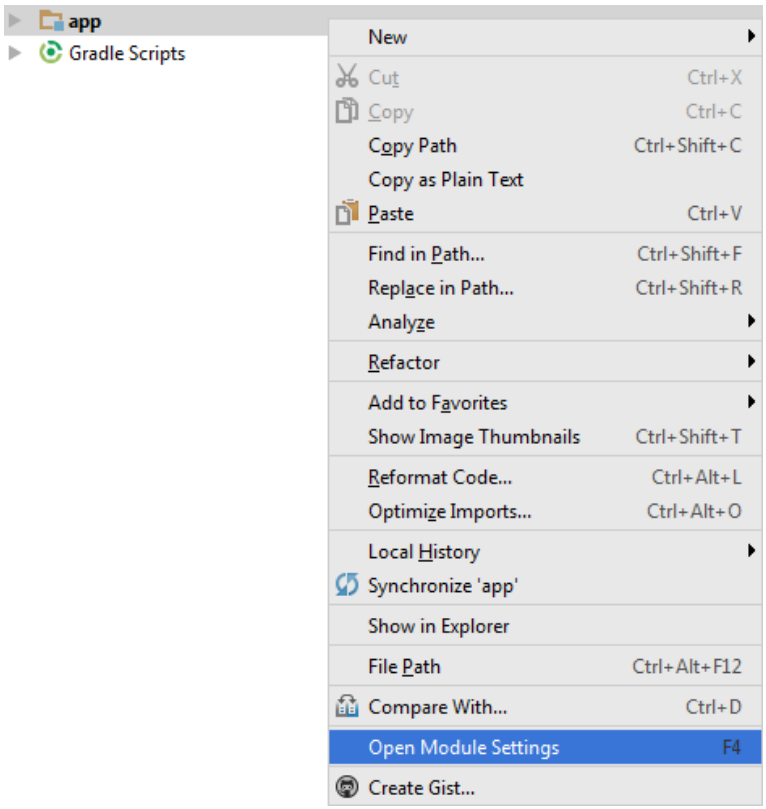
For this tutorial you will use Android Studio, which is the official IDE to create, build and debug applications for Android devices. There are two ways to work with an Android application:

- Create an Android application from scratch
- Import a Digi Sample Application

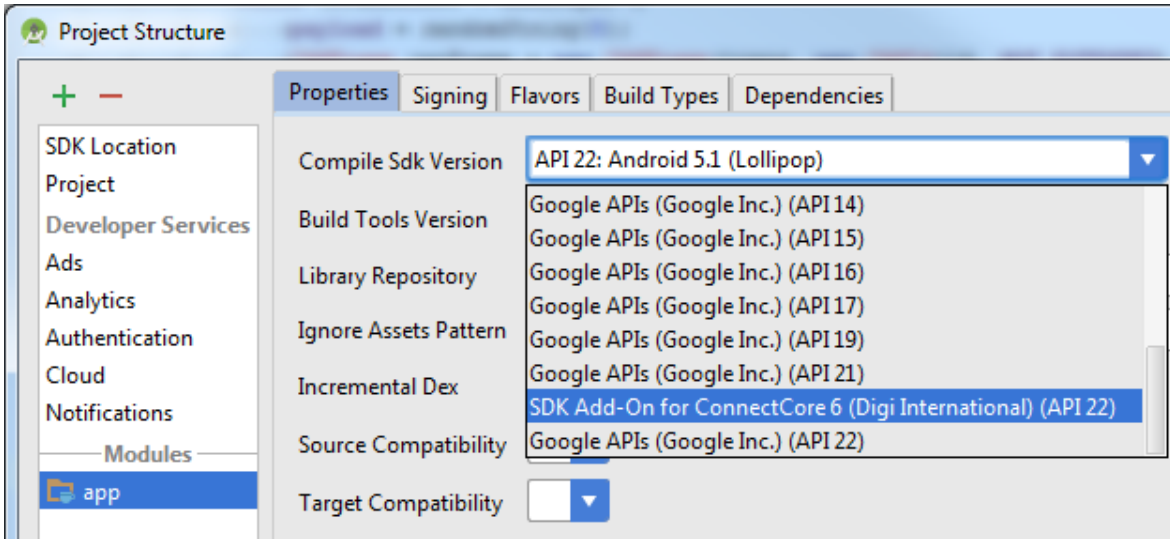
> The Digi API extensions don't work on Android Virtual Devices, so you need a physical device with access to all the hardware interfaces if you want to launch or debug your application.

Create an Android application from scratch

1. Create a new application with Android Studio. For instructions, see the Android Developers Guide.
2. Ensure that the Digi API extensions are installed in your Android SDK. To learn how to install them, see the instructions in 4.1. Install the software.
3. Configure your project to use the Digi API extensions instead of the standard Android SDK.
   a. Right-click on your Android project and select **Open Module Settings**. The **Project Structure** dialog appears.



   b. In the **Compile SDK Version** box, select **SDK Add-On for ConnectCore 6 (Digi International) (API 22)**.



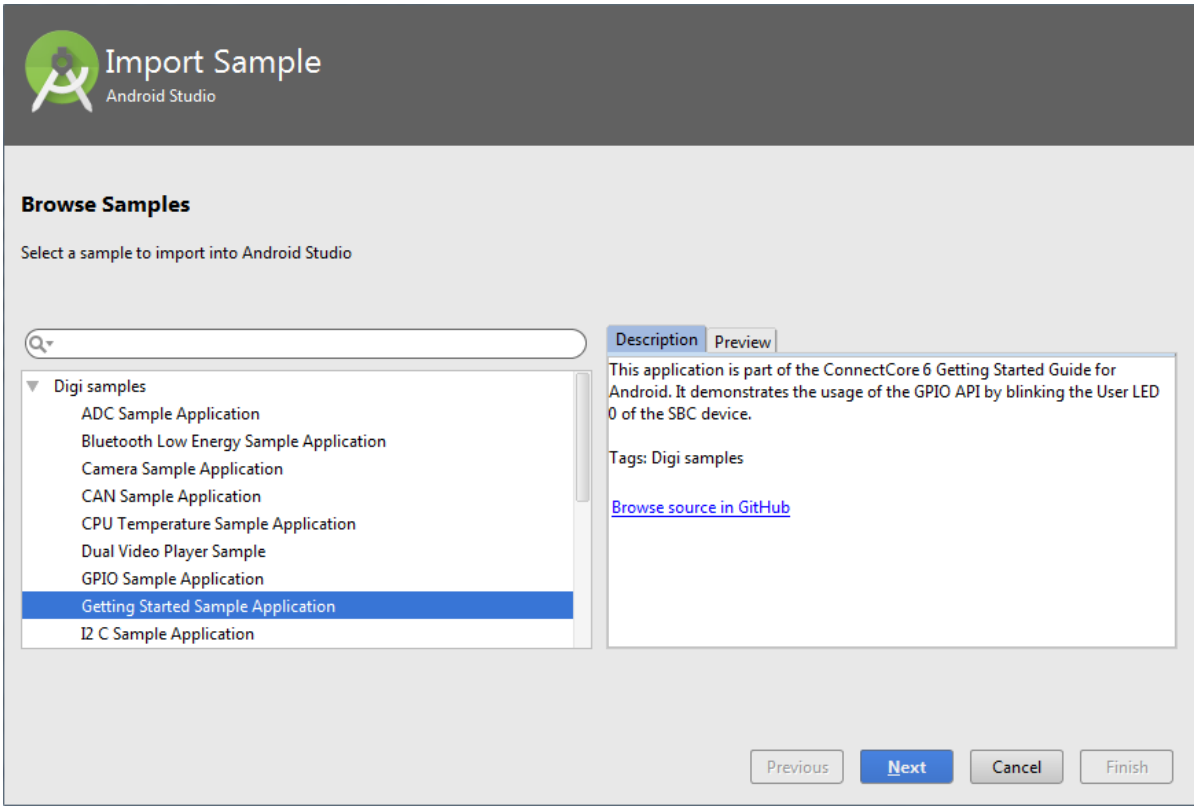
Import a Digi Sample Application

Digi provides sample projects that demonstrate how to use the API extensions to access hardware interfaces available in your device.

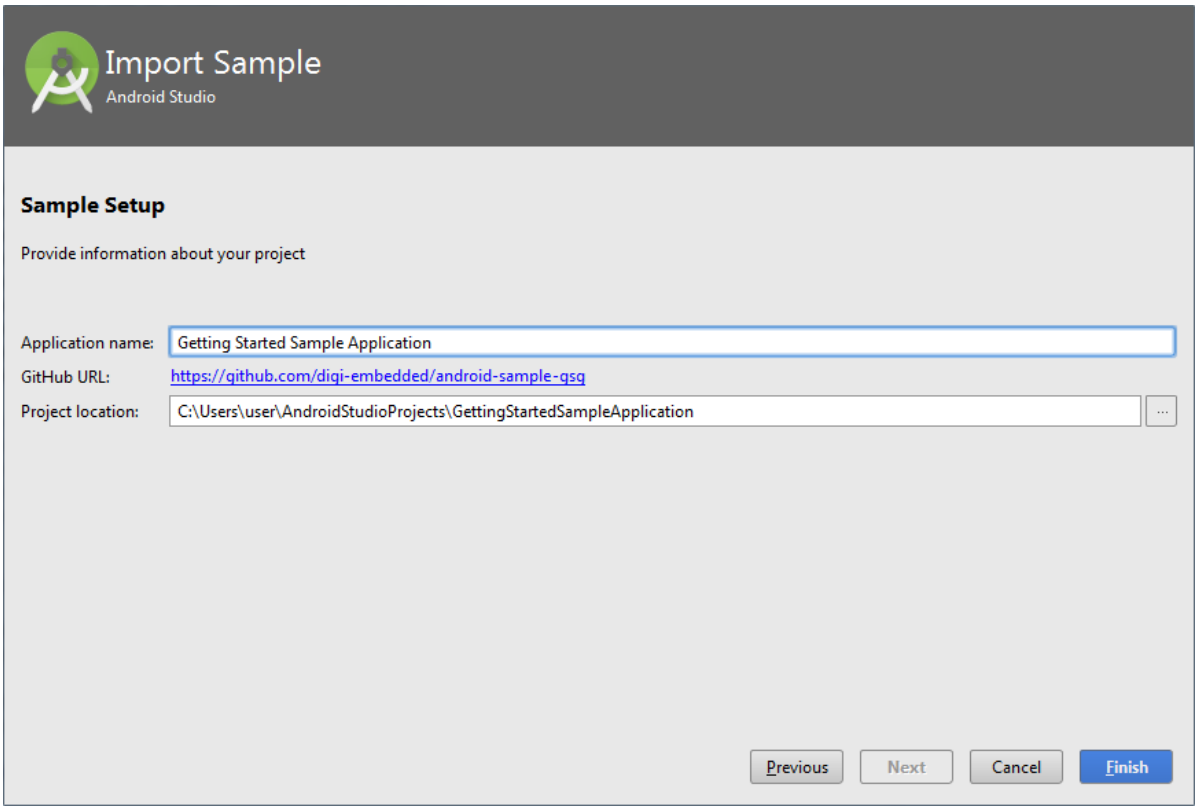
> Ensure that the Digi plugin is installed. To learn how to install it, see the instructions for "Digi Extensions for Android Studio" in 4.1. Install the software.

To import a Digi Sample Application:

1. Open the **Import Digi Sample** wizard by doing one of the following:
   - Select **Import a Digi Android code sample** on the Welcome screen.
   - Select **File > New > Import Digi Sample** from the menu.
2. On the **Browse Samples** page of the wizard, select one of the available Digi samples and click **Next**.



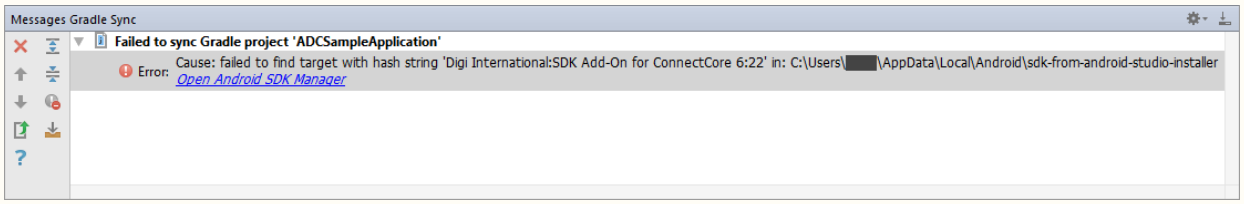
3. On the **Sample Setup** page of the wizard, configure the **Application name** and the **Project location**.



4. Click **Finish** to import the example. It is opened in a new Android Studio window.
5. Wait a few seconds while the compiler is working and click the **Run** button

   to launch the application on the device.

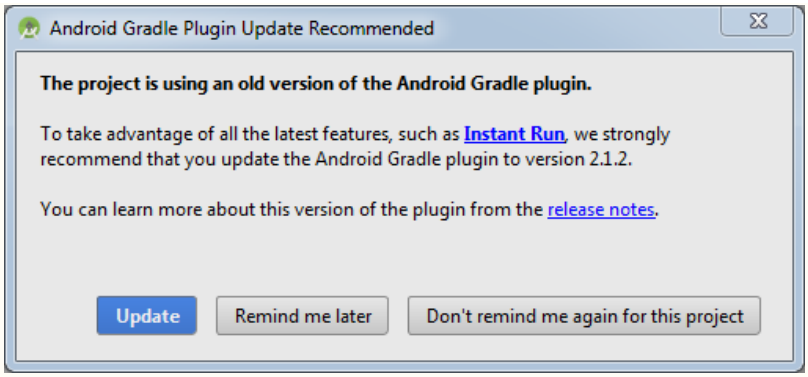
Due to an Android Studio issue, if the following Gradle synchronization error appears:



Follow these steps to fix the issue:

1. Go to **File > Project Structure**.
2. Select the item inside **Modules** section in the list on the left.
3. In the **Compile Sdk Version** drop-down , select **API 22: Android 5.1 (Lollipop)**.
4. Click **OK**. Android Studio starts to index the selected SDK.
5. Once the indexing process finishes, go to **File > Project Structure**.
6. Select the item inside **Modules** section in the list on the left.
7. In the **Compile Sdk Version** drop-down, select **SDK Add-On for ConnectCore 6. Android 22 (API 22)**.
8. Click **OK**.

The Android Gradle plugin is in ongoing development, but the Digi samples are configured with a specific Gradle version. For that reason, it's likely that Android Studio recommends you to update it when you import any example.



To do so, click **Update** and Android Studio will take care of the update.

Digi APIX for Android

Android comes with a very rich set of APIs (Application Programming Interface) for several peripherals, but it lacks APIs to access common interfaces presented in many embedded devices. Digi APIX for Android extends the Android sources, providing extra functionality to manage additional hardware interfaces.

Digi provides some examples to teach you how to work with the different interfaces in an easy way. For more information on Digi APIs, see the Javadoc documentation. You can import all the examples using the Digi Extension for Android Studio plugin. For more details, see Import a Digi Sample Application.

- ADC
- CAN
- Cloud Connector
- CPU management
- Ethernet
- Firmware update (API)
- GPIO
- GPU management
- I2C
- Memory
- PWM
- Serial port
- SPI
- Watchdog

ADC

An **Analog to Digital Converter** (**ADC**) is a device that translates an analog voltage to a digital value that a microprocessor can understand. There are several channels available depending on module version and technology; however, these channels all work the same way.
The ConnectCore 6 has several ADC interfaces to measure analog signals and transform them into digital values. In the ConnectCore 6 Hardware Reference Manual you can find information about the available ADC channels.

Digi adds to Android an API to manage these ADC channels. You can configure them, read values, and program periodic sampling among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all ADC API methods require the **`com.digi.android.permission.ADC`** permission.

> If your application does not have the `com.digi.android.permission.ADC` permission it will not have access to any ADC service feature.

First of all, a new `ADCManager` object must be instantiated by passing the Android Application Context.

Instantiating the ADCManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.adc.ADC;
import com.digi.android.adc.ADCManager;

public class ADCSampleActivity extends Activity {

 ADCManager adcManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the ADC manager object.
        adcManager= new ADCManager(this);

  [...]
 }

 [...]
}
```

Instantiate an ADC channel

The `ADCManager` allows you to create an `ADC` object for a specific ADC channel and get its current value.

Getting an ADC channel

```
import com.digi.android.adc.ADC;
import com.digi.android.adc.ADCManager;

[...]

ADCManager adcManager = ...;

// Create ADC object for channel 1.
ADC adc = adcManager.createADC(1);

// Get the current value of channel 1.
System.out.format("Channel %d: %d%n", adc.getChannel(), adc.getValue());

[...]
```

Monitor ADC channels

You can monitor a specific ADC channel if you subscribe an `IADCListener` to the ADC object. Use the `registerListener(IADCListener)` method to register for a particular ADC channel updates. The listener starts receiving ADC samples as soon as the `startSampling(int)` method is called specifying the interval between samples in milliseconds.

ADC channel updates registration

```
import com.digi.android.adc.ADC;
import com.digi.android.adc.ADCManager;

[...]

ADCManager adcManager = ...;

// Create ADC object for channel 1.
ADC adc = adcManager.createADC(1);
// Create the ADC listener.
MyADCListener myADCListener = ...;

// Register the ADC listener.
adc.registerListener(myADCListener);

// Start monitoring ADC channel every 5 seconds.
adc.startSampling(5000);

[...]
```

The registered listener class, `MyADCListener`, must implement the `IADCListener` interface. This interface defines the `sampleReceived(ADCSample)` method, that is called when a new ADC sample for the channel is received, in the example above every 5 seconds.

The `sampleReceived(ADCSample)` method receives a new `ADCSample` object with the value of the ADC channel (`getValue()`) and the time when the sample was taken (`getTimestamp()`).

IADCListener implementation example, MyADCListener

```
import com.digi.android.adc.IADCListener;

public class MyADCListener implements IADCListener {
    @Override
    public void sampleReceived(ADCSample sample) {
        // This code will be executed every time a new ADC sample is
received.
  System.out.format("ADC value (%l): %d%n", sample.getTimestamp(),
sample.getValue());
    }
}
```

You can have more than one `IADCListener` waiting for updates in the same channel. To stop the ADC channel notifications, use the `stopSampling()` method. If you no longer wish to receive ADC channel samples in a determined listener, use the `unregisterListener(IADCListener)` method to unsubscribe an already registered listener.

ADC updates unregistration

```
[...]

ADC adc = ...;
MyADCListener myADCListener = ...;

adc.registerListener(myADCListener);

adc.startSampling(5000);

[...]

// Stop ADC notifications.
adc.stopSampling();
// Remove the ADC listener.
adc.unregisterListener(myADCListener);

[...]
```

ADC Example

The **ADC Sample Application** demonstrates the usage of the ADC API. In this example, you can select one of the ADC channels available in your ConnectCore 6 and configure the sample rate of that interface.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

CAN

A **Controller Area Network** (**CAN Bus**) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer. It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles, but is also used in many other contexts.
The ConnectCore 6 has several CAN ports to communicate with other devices using this protocol. In the Javadoc documentation you can find information about the available CAN channels.

Digi adds to Android an API to manage these CAN interfaces. You can configure the CAN settings, write, read, and add listeners among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all CAN API methods require the **`com.digi.android.permission.CAN`** permission.

> If your application does not have the `com.digi.android.permission.CAN` permission it will not have access to any CAN service feature.

First of all, a new `CANManager` object must be instantiated by passing the Android Application Context.

Instantiating the CANManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.can.CAN;
import com.digi.android.can.CANManager;

public class CANSampleActivity extends Activity {

 CANManager canManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the CAN manager object.
        canManager= new CANManager(this);

  [...]
 }

 [...]
}
```

Open/close a CAN interface

The `CANManager` allows you to create a `CAN` object for a specific CAN interface.

Getting a CAN object

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANManager;

[...]

CANManager canManager = ...;

// Create CAN object for interface number 0.
CAN can0 = canManager.createCAN(0);
```

To work with a CAN interface you must open it so you can read and write data. To manage the interfaces use the following `CAN` class methods:

Method	Description
`open()`	Opens the CAN interface
`close()`	Closes the CAN interface
`isInterfaceOpen()`	Returns the CAN interface status

`open()` and `close()` methods may fail for the following reasons:

- If the interface number represented by the CAN object does not exist, the `open()` method throws a `NoSuchInterfaceException`.
- If there is any error while opening or closing the CAN interface an `IOException` is thrown.

Opening/closing a CAN interface

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANManager;

[...]

CANManager canManager = ...;

// Create CAN object for interface number 0.
CAN can0 = canManager.createCAN(0);

// Check if the CAN 0 is already opened. If not, open it.
if (!can0.isInterfaceOpen())
 can0.open();

[...]

// Close CAN 0.
can0.close();
```

Configure a CAN interface

Before sending and receiving data, the CAN interface bitrate must be configured. By default, it is set to 500 kbps.

Use the `setBitrate(int)` method to change the bitrate of a CAN interface. This method may fail if:

- The CAN interface does not support bitrate changes. In this case a `UnsupportedOperationException` is thrown.
- Any error occurs while setting the new bitrate, then an `IOException` is thrown.

Setting CAN bitrate

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANManager;

[...]

CANManager canManager = ...;
CAN can0 = ...;

[...]

// Set the new bitrate.
can0.setBitrate(125000);

[...]
```

Send data

Once the bitrate is properly configured, you can send data through the CAN interface. The CAN API includes a class called `CANFrame` to represent the data over the CAN bus. To instantiate a CAN frame, you need to provide the following parameters:

- A `CAN` object to send the frame.
- A `CANId` object, the identifier of the frame.
- The data to send.

Creating a CANFrame

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANFrame;

[...]

CAN can0 = ...;
CANId id = new CANId(50, false);
byte[] dataToSend = new byte[]{'D', 'a', 't', 'a', ' ', 't', 'o', ' ',
's', 'e', 'n', 'd'};

// Instantiate a new frame to send.
CANFrame frame = new CANFrame(can0, id, dataToSend);

[...]
```

To send the data, use the `write(CANFrame)` method with the `CANFrame` object that contains the data to be transmitted. This method may fail if the interface is closed or if any error occurs while transmitting the data throwing an `IOException`.

Sending a CANFrame

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANFrame;

[...]

CAN can0 = ...;
CANId id = ...;
byte[] dataToSend = ...;

if (!can0.isInterfaceOpen())
 can0.open();

[...]

CANFrame frame = new CANFrame(can0, id, dataToSend);
// Transmit the data.
can0.write(frame);

[...]
```

Remember that when you are done with the CAN interface you need to close it calling the `close()` method.

Receive data

To receive data you must:

1. Register a listener to receive frames
2. Start the reading process with the proper filters

Once you received all the required data you can stop the reading process and unregister the listener.

Register for frame notifications

To receive data you must subscribe an `ICANListener` to the `CAN` object that represents the interface where you are going to read data. Use the `registerListener(ICANListener)` method to register for the notification of new CAN frames notifications.

ICANListener registration

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANFrame;

[...]

CAN can0 = ...;

// Create the CAN listener.
MyCANListener myCANListener = ...;

// Register the CAN listener.
can0.registerListener(myCANListener);

[...]
```

The registered listener class, `MyCANListener`, must implement the `ICANListener` interface. This interface defines the `frameReceived(CANFrame)` method that is called every time a CAN frame is received.

ICANListener implementation example, MyCANListener

```
import com.digi.android.can.CANFrame;
import com.digi.android.can.ICANListener;

public class MyCANListener implements ICANListener {
 @Override
 public void frameReceived(CANFrame frame) {
  System.out.println("Received on CAN " +
frame.getCAN().getInterfaceNumber()
       + ": " + new String(frame.getData()));
 }
}
```

Start the data reception

The listener starts receiving frames as soon as the `startRead(List<CANFilter>)` method is called with the list of filters specified. This allows you to only listen to specific CAN identifiers and masks.

A received frame is examined to decide if it is relevant using the acceptance filter id and mask value to filter out unwanted messages. The filter mask is used to determine which bits in the identifier of the received frame are compared with the filter:

- If a mask bit is set to a zero, the corresponding ID bit will automatically be accepted, regardless of the value of the filter bit.
- If a mask bit is set to a one, the corresponding ID bit will be compared with the value of the filter bit; if they match it is accepted otherwise the frame is rejected.

Starting reading

```
import com.digi.android.can.CAN;
import com.digi.android.can.CANId;
import com.digi.android.can.CANFilter;

[...]

CAN can0 = ...;
MyCANListener myCANListener = ...;

[...]

can0.registerListener(myCANListener);

// Define filters.
ArrayList<CANFilter> filters = new ArrayList<CANFilter>();
CANId id = new CANId(50, false);
CANFilter f = new CANFilter(id, 00000000); // Only accepts messages with
ID = 50.
filters.add(f);
// Start reading for frames on CAN 0.
can0.startRead(filters);

[...]
```

The `startRead(List<CANFilter>)` method throws an `IOException` if the CAN interface is closed.

Stop the data reception

You can have more than one `ICANListener` waiting for frames in the same CAN interface. To stop all CAN interface notifications, use the `stopRead()` method.

If you no longer wish to receive CAN frames in a determined listener, use the `unregisterListener(ICANListener)` method to unsubscribe an already registered listener.

Stopping reading

```
[...]

CAN can0 = ...;
MyCANListener myCANListener = ...;
ArrayList<CANFilter> filters = ...;

[...]

can0.registerListener(myCANListener);

can0.startRead(filters);

[...]

// Stop reading from CAN 0.
if (can0.isReading())
 can0.stopRead();

// Remove the CAN listener.
can0.unregisterListener(myCANListener);

[...]
```

The `stopRead()` method throws an `IOException` if the CAN interface is closed.

> Remember that when you are done with the CAN interface you need to close it calling the `close()` method.

CAN Example

The **CAN Sample Application** demonstrates the usage of the CAN API. In this example, you can configure all the settings of the CAN interface, write, and read data from the port.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Cloud Connector

Cloud Connector is a new Android service used to communicate with Device Cloud, a Digi's proprietary platform to manage devices remotely. This service is disabled by default as some configuration steps are required before running it. You can enable and configure it in the **Device Cloud** section of the Android settings. Refer to the Remote management topic for more information about Device Cloud and the ConnectCore 6 specific features.

Cloud Connector includes a rich API that can be used in your applications to communicate with Device Cloud, receive remote commands and save data in the cloud. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all the Cloud Connector API methods require the **`com.digi.android.permission.CLOUD_CONNECTOR`** permission.

> If your application does not have the **`com.digi.android.permission.CLOUD_CONNECTOR`** permission it will not have access to any Cloud Connector service feature.

To work with this API and communicate with Device Cloud, the first step is to instantiate a `CloudConnectorManager` object. To do so, you need to provide the Android application Context.

Instantiating the CloudConnectorManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.cloudconnector.CloudConnectorManager;

public class CloudConnectorSampleActivity extends Activity {

    CloudConnectorManager connectorManager;

    [...]

    @Override
 public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceStace);

        // Instantiate the CloudConnector manager object.
        connectorManager = new CloudConnectorManager(this);

        [...]
    }

    [...]
}
```

Get the device ID

Every device registered in Device Cloud has a unique identifier that is used by the platform to refer and communicate with the device. The device ID is calculated automatically by the service using the MAC address of the Ethernet interface. It can be obtained in the Device Cloud configuration within the Android settings or by using a specific method of the Device Cloud API.

Method	Description
`getDeviceID()`	Returns the Cloud Connector Device ID

> You don't need to be connected to Device Cloud in order to get the device ID.

Getting Device ID

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Get the device ID.
String deviceID = connectorManager.getDeviceID();
```

Configure the Vendor ID

Before connecting to Device Cloud it is necessary to configure a Vendor ID for the device.

The Vendor ID is a 4-byte value indicating the manufacturer of the selected device. This value will be used to determine the manufacturer the device belongs to, usually tied to the Device Cloud account.

A Vendor ID can be obtained by logging in to your Device Cloud account and navigating to **Admin > Account Settings**. This will display the My Account page; within the Vendor Information section of the page you can request a Vendor ID (if you haven't already) or view your assigned Vendor ID. If you do not properly configure this setting or if it is left empty, an error will be shown while trying to connect.

The format of this setting is:

`0xXXXXXXXX`
You can configure it in the **Device Cloud** section of the Android Settings or by using the Cloud Connector API.

Method	Description
`getVendorID()`	Returns the device Vendor ID
`setVendorID(String)`	Sets the Cloud Connector Vendor ID setting

The `setVendorID(String)` method may fail for the following reasons:

- The specified vendor ID is `null` throwing a `NullPointerException`.
- The specified vendor ID does not follow the correct format throwing an `IllegalArgumentException`.

Configuring Vendor ID

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Configure the Vendor ID (remember you must get it from your account
page in Device Cloud).
connectorManager.setVendorID("0x12345678");
```

Configured settings are persistent between boots. There is no need to re-configure Cloud Connector service each time device is powered on.

Connect/Disconnect from Device Cloud

Once the Vendor ID is configured, the next step is to open the connection with the platform. Using the Device Cloud API, you can connect or disconnect from Device Cloud at any time. You can manage the connection to Device Cloud using the following methods.

Method	Description
`connect()`	Opens the Cloud Connector connection
`disconnect()`	Closes the Cloud Connector connection
`isConnected()`	Returns the Cloud Connector connection status

Managing Device Cloud connection

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Check if the device is already connected to Device Cloud. If not,
connect.
if (!connectorManager.isConnected())
    connectorManager.connect();

[...]

// Disconnect from Device Cloud.
connectorManager.disconnect();
```

Depending on the value of the **auto-connect** setting of the Device Cloud service, the device may

try to connect to Device Cloud automatically as soon as the system is initialized.

The connect and disconnect processes generate events indicating when the device has connected and disconnected from Device Cloud. Refer to the Capture Cloud Connector events topic to learn how to listen for Cloud Connector events.

Make sure your device is registered within your Device Cloud account before attempting to connect. For more information on registering your device in Device Cloud read the Add your ConnectCore 6 to Device Cloud topic.

Enable/Disable the System Monitoring feature

System Monitoring is a feature that comes embedded within the Cloud Connector service and is used to monitor some parameters of the device remotely. Refer to the Monitor the system topic for more information about it.

This feature can be enabled and configured in the **Device Cloud** section of the Android settings, but it can be also managed using the following Cloud Connector API methods:

Method	Description
`enableSystemMonitor(boolean)`	Enables/disables the system monitor feature
`isSystemMonitorEnabled()`	Returns the current state of the system monitor

Managing System Monitoring feature

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Check if the System Monitoring is already enabled. If not, enable it.
if (!connectorManager.isSystemMonitorEnabled())
    connectorManager.enableSystemMonitor(true);

[...]

// Disable the System Monitor.
connectorManager.enableSystemMonitor(false);
```

The Cloud Connector API includes other features that allow you to perform the following actions:

- Capture Cloud Connector events
- Configure Cloud Connector service
- Receive data from Device Cloud
- Send data to Device Cloud

Cloud Connector Example

The **Cloud Connector Sample Application** demonstrates the usage of the Cloud Connector API. In this example, you can enable and disable the service, configure some parameters, send data to Device Cloud and receive commands remotely.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the <TODO verify link GitHub repository >.

Capture Cloud Connector events

Some processes performed by the Cloud Connector service generate events that you might want to be aware of in order to execute a specific action. The Cloud Connector API allows you to register a listener to be notified when some of these events occur such as:

- Connection events
- Data points events

The first thing you need to do in order to listen for Cloud Connector events is to create your events listener. This listener must implement the `ICloudConnectorEventListener` interface.

Cloud Connector events listener

```
import com.digi.android.cloudconnector.ICloudConnectorEventListener;

public class MyEventsListener implements ICloudConnectorEventListener {
    @Override
    public void connected() {
    // TODO
    }

    @Override
    public void disconnected() {
    // TODO
    }

    @Override
    public void connectionError(String s) {
    // TODO
    }

    @Override
    public void sendDataPointsSuccess() {
    // TODO
    }

    @Override
    public void sendDataPointsError(String s) {
    // TODO
    }
}
```

The `ICloudConnectorEventListener` interface implements the following methods to inform you about different events.

Method	Description
`connected()`	Notifies that the Cloud Connector service has connected. The connect method returns immediately, but the connection process may take some seconds to complete.
`disconnected()`	Notifies that the Cloud Connector service has disconnected

`connectionError(String)`	Notifies that there was an error connecting and provides the error message
`sendDataPointsSuccess()`	Notifies that the data points have been sent successfully. Refer to the Send data to Device Cloud topic for more information about sending data points.
`sendDataPointsError(String)`	Notifies that there was an error sending the data points and provides the error message

You need to write the code you want to be executed inside those event callbacks you are interested on. You can leave in blank those callbacks you don't want to execute.

The next step is to register your events listener in the `CloudConnectorManager` instance. If any event is generated after registering your listener, Cloud Connector will execute the code contained in the corresponding callback. If you want to stop receiving events just unregister your events listener.

Method	Description
`registerEventListener(ICloudConnectorEventListener)`	Registers the given listener to receive Cloud Connector events
`unregisterEventListener(ICloudConnectorEventListener)`	Removes the given listener from the event listeners list

The previous methods may fail if the `ICloudConnectorEventListener` to register or unregister is `null` throwing a `NullPointerException`.

Registering and unregistering an event listener

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Register an instance of your events listener class to start receiving
events.
MyEventsListener myEventsListener = new MyEventsListener();
connectorManager.registerEventListener(myEventsListener);

[...]

// Unregister your events listener to stop receiving events.
connectorManager.unregisterEventListener(myEventsListener);
```

Configure Cloud Connector service

The Cloud Connector service comes already configured with some default values. You can check the configuration and modify it in the **Device Cloud** section of the Android settings, or you can use the Cloud Connector API to do so.

> It does not matter if you modify a setting using the Android settings menu or the Cloud Connector API. Settings values are saved in the Android system and their value will prevail after a reboot.

The first step to read or modify Cloud Connector settings using the Cloud Connector API is to get the `CloudConnectorPreferencesManager` from the `CloudConnectorManager`.

Getting the CloudConnectorPreferencesManager

```
import com.digi.android.cloudconnector.CloudConnectorManager;
import com.digi.android.cloudconnector.CloudConnectorPreferencesManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Get the CloudConnectorPreferencesManager object.
CloudConnectorPreferencesManager preferencesManager =
connectorManager.getPreferencesManager();
```

All the settings within the `CloudConnectorPreferencesManager` are set and get the same way, but they can be categorized in the following groups for a better understanding of them.

- Device Information
- Connection
- System Monitor

Device Information

Settings of this group allow you to give some descriptive information about your device in Device Cloud. This information can be accessed through the platform and may help you to identify your devices there in an easy way.

Device ID

Device ID is used to identify the device in Device Cloud. A Device ID is a 16-octet number that is unique to the device and does not change over its lifetime. The Device ID of a ConnectCore 6 is derived from the MAC address of the Ethernet interface.

The canonical method for writing Device IDs is as four groups of eight hexadecimal digits separated by a dash, as follows:

`XXXXXXXX-XXXXXXXX-XXXXXXXX-XXXXXXXX`

The device ID is automatically generated by the Cloud Connector service, so it cannot be set. It is a read-only setting of the `CloudConnectorPreferencesManager`

Method	Description
`getDeviceID()`	Returns the Device ID

Vendor ID

The Vendor ID is a 4-byte value indicating the manufacturer of the selected device. This value will be used to determine the manufacturer the device belongs to, usually tied to the Device Cloud account.

A Vendor ID can be obtained by logging in to your Device Cloud account and navigating to **Admin > Account Settings**. This will display the My Account page; within the Vendor Information section of the page you can request a Vendor ID (if you haven't already) or view your assigned Vendor ID. If you do not properly configure this setting or if it is left empty, an error will be shown while trying to connect.

The format of this setting is:

`0xXXXXXXXX`

> Before connecting your device to Device Cloud it is very important to get and configure a Vendor ID

Method	Description
`getVendorID()`	Returns the device Vendor ID
`setVendorID(String)`	Sets the Cloud Connector Vendor ID setting

The `setVendorID(String)` method may fail for the following reasons:

- The specified vendor ID is `null` throwing a `NullPointerException`.
- The specified vendor ID does not follow the correct format throwing an `IllegalArgumentExcepti on`.

Device name

This value is the device's model information. It is a string that uniquely identifies this model of a device in the Device Cloud server. When the server finds two devices with the same device name, it infers that they are the same product and product-scoped data may be shared among all devices of the same device type. A device's type cannot be an empty string, nor contain only white-spaces.

The device name along with the firmware version and Vendor ID setting are used by Device Cloud to cache the RCI device descriptor. Changing any of these values will cause Device Cloud to ask the device for a new RCI device descriptor.

Method	Description
`getDeviceName()`	Returns the configured device name preference
`setDeviceName(String)`	Sets the device name preference

The `setDeviceName(String)` method may fail for the following reasons:

- The specified device name is `null` throwing a `NullPointerException`.
- The length of the device name is 0 or greater than `CloudConnectorPreferencesManager.DEVI CE_NAME_MAXIMUM_LENGTH` throwing an `IllegalArgumentException`.

Device description

This setting is a description of the device running Cloud Connector service. It allows descriptive device information to be assigned to each of your devices.

Method	Description
`getDeviceDescription()`	Returns the configured device description preference

`setDeviceDescription(String)`	Sets the device description preference

The `setDeviceDescription(String)` method may fail for the following reasons:

- The specified description is `null` throwing a `NullPointerException`.
- The length of the description is greater than `CloudConnectorPreferencesManager.DESCRIPTION_MAXIMUM_LENGTH` throwing an `IllegalArgumentException`.

Contact information

Contact information of the maintainer of this device.

Method	Description
`getDeviceContactInformation()`	Returns the configured device contact information preference
`setDeviceContactInformation(String)`	Sets the device contact information preference

The `setDeviceContactInformation(String)` method may fail for the following reasons:

- The specified contact information is `null` throwing a `NullPointerException`.
- The length of the contact information is greater than `CloudConnectorPreferencesManager.CONTACT_MAXIMUM_LENGTH` throwing an `IllegalArgumentException`.

Connection

The `CloudConnectorPreferencesManager` allows you to configure some parameters related to the connection established by the ConnectCore 6 and Device Cloud.

Connection URL

The URL of Device Cloud is not intended to change, but it is possible to modify it if you want to use a different Device Cloud server such as a test one.

Method	Description
`getURL()`	Returns the configured Cloud Connector URL preference
`setURL(String)`	Sets the Cloud Connector URL preference

The `setURL(String)` method may fail if the specified URL is `null` throwing a `NullPointerException`.

> The value of this setting should be always set to **devicecloud.digi.com**

Auto-connect

This setting is used to indicate the Cloud Connector service that it should try to connect to Device Cloud just after booting or if the Internet connection is lost.

Method	Description
`isAutoConnectEnabled()`	Returns whether auto-connect setting is enabled
`setAutoConnectEnabled(boolean)`	Sets the new value of the auto-connect setting

> This setting is disabled by default.

Secure connection

This setting allows you to connect to Device Cloud using the secure protocol (SSL) or to establish standard TCP connection. When the setting is enabled, Cloud Connector will use the secure protocol (SSL) to connect to Device Cloud. If it is disabled, Cloud Connector will establish a standard TCP connection.

Method	Description
`isSecureConnectionEnabled()`	Returns whether secure connection is enabled
`setSecureConnectionEnabled(boolean)`	Sets the new value for the use secure connection setting

> This setting is disabled by default. Changes applied to this setting only take effect after a re-connection.

Message compression

This setting can be used to reduce the amount of network traffic. If the setting is enabled, Cloud Connector will use ZLib compression while sending/receiving data from Device Cloud.

Method	Description
`isCompressionEnabled()`	Returns whether messages compression is enabled
`setCompressionEnabled(boolean)`	Sets the new value of the compress messages setting

> This setting is disabled by default. Changes applied to this setting only take effect after a re-connection.

System Monitor

System monitoring feature of Cloud Connector allows you to control some parameters of the device remotely. Refer to Monitor the system topic for more information about it.

At the moment System monitoring can get sample values of the System memory, CPU load and CPU temperature. From the `CloudConnectorPreferencesManager` you can enable or disable this feature and configure some advanced settings such as the parameters to monitor.

Method	Description
`isSystemMonitorEnabled()`	Returns the current state of the system monitor
`enableSystemMonitor(boolean)`	Sets the new value of the enable system monitor setting

> This setting is disabled by default.

Sample rate

This is the time in seconds at which samples of each parameter are captured. Samples are captured and saved until they reach the `getSystemMonitorUploadSamplesSize()` value. Then, they are uploaded to Device Cloud.

Method	Description
`getSystemMonitorSampleRate()`	Returns the system monitor sample rate

`setSystemMonitorSampleRate(int)`	Sets the system monitor sample rate

The `setSystemMonitorSampleRate(int)` method may fail if the sample rate to set is lower than 5 throwing an `IllegalArgumentException`.

Samples before uploading

This System monitoring setting establishes the number of samples of each parameter to capture before uploading them to Device Cloud.

Method	Description
`getSystemMonitorUploadSamplesSize()`	Returns the system monitor number of samples to store for each channel before uploading to Device Cloud
`setSystemMonitorUploadSamplesSize(int)`	Sets the system monitor number of samples to store for each channel before uploading to Device Cloud.

The `setSystemMonitorUploadSamplesSize(int)` method may fail if the samples size value to set is lower than 1 or if it is greater than the maximum upload samples size (`CloudConnectorPreferencesManager.MAXIMUM_UPLOAD_SAMPLES_SIZE`) throwing an `IllegalArgumentException`.

Monitor system memory

This setting enables or disables the system memory monitoring. It is the total RAM memory used by the device in MB.

Method	Description
`isSystemMonitorMemorySamplingEnabled()`	Returns whether the system monitor memory sampling is enabled
`enableSystemMonitorMemorySampling(boolean)`	Sets the new value of the enable system monitor memory sampling

Monitor CPU load

With this setting you can enable or disable the CPU load parameter in the System monitoring feature. The CPU load is measured in %.

Method	Description
`isSystemMonitorCPULoadSamplingEnabled()`	Returns whether the system monitor CPU load sampling is enabled
`enableSystemMonitorCPULoadSampling(boolean)`	Sets the new value of the enable system monitor CPU load sampling

Monitor CPU temperature

This setting allows you to enable or disable the CPU temperature parameter in the System monitoring feature. The value of the temperature is taken and saved in Device Cloud in Celsius.

Method	Description

`isSystemMonitorCPUTemperatureSamplingEnabled()`	Returns whether the system monitor CPU temperature sampling is enabled
`enableSystemMonitorCPUTemperatureSampling(boolean)`	Sets the new value of the enable system monitor CPU temperature sampling

This is a usage example of the `CloudConnectorPreferencesManager` object:

Configuring settings using the CloudConnectorPreferencesManager

```
import com.digi.android.cloudconnector.CloudConnectorManager;
import com.digi.android.cloudconnector.CloudConnectorPreferencesManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Get the CloudConnectorPreferencesManager object.
CloudConnectorPreferencesManager preferencesManager =
connectorManager.getPreferencesManager();

// Configure device information parameters.
preferencesManager.setVendorID("0x12345678");
preferencesManager.setDeviceDescription("My device description");
preferencesManager.setContactInformation("my.name@company.com");

// Configure some connection parameters.
if (!preferencesManager.isAutoConnectEnabled())
    preferencesManager.setAutoConnectEnabled(true);
if (!preferencesManager.isSecureConnectionEnabled()) {
    preferencesManager.setSecureConnectionEnabled(true);
    // This setting requires a re-connection in order to apply it.
    connectorManager.disconnect();
    connectorManager.connect();
}

// Configure the System monitoring and enable it. With this
configuration
// the System monitoring will upload a total of 10 samples of each
parameter
// every 150 seconds.
preferencesManager.setSystemMonitorSampleRate(15);
preferencesManager.setSystemMonitorUploadSamplesSize(10);
if (!preferencesManager.isSystemMonitorEnabled())
    preferencesManager.enableSystemMonitor(true);
```

Receive data from Device Cloud

Another feature provided by the Cloud Connector service is the possibility to receive data from Device Cloud, also known as receiving **Device Requests**. Transfers are initiated from a Web Services client connected to Device Cloud, which hosts the ConnectCore 6 device. This transfer is used to send data to the device and the device may send a response back.

In order to make your application aware of Device Requests sent to the device, the first thing you need to do is to create your own Device Requests listener. This listener class must implement the `IDeviceRequestListener` interface.

Cloud Connector Device Request listener

```
import com.digi.android.cloudconnector.IDeviceRequestListener;

class MyDeviceRequestListener implements IDeviceRequestListener {
    @Override
        public String handleDeviceRequest(String target, byte[] data) {
        return null;
    }

    @Override
        public String handleDeviceRequest(String target, String data) {
        return null;
    }
}
```

The `IDeviceRequestListener` interface implements the following methods to handle the data sent through the Cloud.

Method	Description
`handleDeviceRequest(String, byte[])`	Handles a binary device request for the given target with the given data
`handleDeviceRequest(String, String)`	Handles a plain device request for the given target with the given data

Previous methods are pretty similar. The first parameter corresponds to the target name, which is the identifier of the Device Request. The second parameter is the data of the Device Request and, depending on the callback executed, it is provided as a String or as a byte array. The methods optionally return a string which will be sent to Device Cloud as response of the device request. The maximum data size of a Device Request is 2MB.

> The `handleDeviceRequest(String, byte[])` callback is only executed when the data format of the request sent to the device is `base64`.

Write inside each callback the code you want to be executed. You can use the first parameter of the callbacks (target) to identify the ID if the Device Request received.

The next step is to register your Device Requests listener in the `CloudConnectorManager` instance for a specific target. Whenever a Device Request for the registered target is received by the device after registering your listener, Cloud Connector will execute the code contained in the corresponding callback. If you want to stop listening for Device Requests just unregister your listener.

Method	Description
`registerDeviceRequestListener(String, IDeviceRequestListener)`	Registers the given listener to receive Device Requests from the Cloud Connector for the given target
`unregisterDeviceRequestListener(IDeviceRequestListener)`	Removes the given listener from the Device Request listeners list

The previous methods may fail if the `IDeviceRequestListener` to register or unregister is `null` throwing a `NullPointerException`.

Registering and unregistering an events listener

```
import com.digi.android.cloudconnector.CloudConnectorManager;

[...]

// Instantiate the CloudConnector manager object.
CloudConnectorManager connectorManager = new
CloudConnectorManager(this);

// Register an instance of your Device Request listener class to start
receiving Device Requests
// with target "myTarget".
MyDeviceRequestListener myDeviceRequestListener = new
MyDeviceRequestListener();
connectorManager.registerDeviceRequestListener("myTarget",
myDeviceRequestListener);

[...]

// Unregister your Device Request listener to stop listening for Device
Requests.
connectorManager.unregisterDeviceRequestListener(myDeviceRequestListener
);
```

Use the API Explorer to send Device Requests

A Device Request can be sent via Web Services within the Device Cloud platform. If you have implemented a Device Request listener and you want to test it, follow these steps:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to **Documentation > API Explorer**.
3. Select **Examples > SCI > Data Service > Send Request**.
   Device Cloud automatically creates the necessary code
4. Replace the "device id" value with the ID of your device.
5. Enter your target name and data for the device request.

Device Cloud - API Explorer

```
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <requests>
      <device_request target_name="myTarget">
        This is a Device Request sample
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

6. Click **Send**

Send data to Device Cloud

The Cloud Connector service allows you to store time-series data in Device Cloud that can be retrieved later by an external application using WEB services. This feature is also known within Device Cloud as **Dat a streams**.

Virtually any type of data can be stored and you can create real-time charts to monitor your data streams. Data streams are fully searchable and the data within a stream can be rolled up into time interval summaries. Data is stored and replicated in multiple secure, commercial-grade storage systems to ensure complete data protection.

Time series data involves two concepts:

- **Data points** are the individual values which are stored in Data streams with a unique time stamp.
- **Data streams** are containers of data points. Data streams hold metadata about the data points held within them. Data streams and the data points they hold are addressed using hierarchical paths (much like folders).

Data stream

Before sending a data point to Device Cloud, you need to define the data stream that will hold it. Remember that a data stream can store multiple data points of the same type. A `DataStream` object (`com.digi.android.cloudconnector.DataStream`) represents the destination of data points in Device Cloud. For example, a `DataStream` object with name "temperature" will create a stream in Device Cloud with the path `<DeviceID>/temperature`, where `<DeviceID>` is the ID of your device.

A data stream can be instantiated providing just the name in the constructor and configuring the rest of parameters later, or providing all the settings directly in the constructor. Once instantiated you can manage its parameters with the following methods.

Method	Description
`getName()`	Returns the name of this data stream
`getForwardTo()`	Returns an array of data stream names to replicate data points to*
`setForwardTo(String[])`	Sets the list of streams to replicate data points to
`getUnits()`	Returns the units for this data stream
`setUnits(Stream)`	Sets the data stream units

> * The list of data stream names contains the names of those data streams to replicate data points to. So, as soon as a data point is stored in the data stream, it will be also copied or replicated in each data stream whose name is contained in that list.

DataStream constructor 1

```
import com.digi.android.cloudconnector.DataStream;

[...]

// Create a new DataStream with "temperature" as name and "C" as units.
DataStream temperatureDataStream = new DataStream("temperature");
temperatureDataStream.setUnits("C")
```

DataStream constructor 2

```
import com.digi.android.cloudconnector.DataStream;

[...]

// Create a new DataStream with "temperature" as name and "C" as units.
DataStream temperatureDataStream = new DataStream("temperature", "C",
new String[0]);
```

Data point

A data point is defined by the `DataPoint` class (`com.digi.android.cloudconnector.DataPoint`) and it represents a single value that is stored at a specific time in a data stream in Device Cloud.

The data point value and the stream destination (a `DataStream` object) are required to create a `DataPoint`. You need to use the correct constructor depending on the value type of the data point. Android system time is used as a time stamp for the data point when it is instantiated. Other attributes such description, quality and location, can be specified using the corresponding set methods.

Method	Description
`getData()`	Returns the data point data
`getDataStream()`	Returns the data stream destination of the data point
`getTimestamp()`	Returns the data point time stamp
`getDescription()`	Returns the data point description
`setDescription(String)`	Sets the data point description
`getLocation()`	Returns the data point location
`setLocation(android.location.Location)`	Sets the data point location
`getQuality()`	Returns the data point quality
`setQuality(int)`	Sets the data point quality

DataPoint constructors

```
import com.digi.android.cloudconnector.DataPoint;
import com.digi.android.cloudconnector.DataStream;

[...]

DataStream temperatureDataStream = new DataStream("temperature", "C",
new String[0]);
DataStream inValveDataStream = new DataStream("inValve", "", new
String[0]);
DataStream levelDataStream = new DataStream("level", "%", new
String[0]);

DataPoint temperatureDataPoint = new DataPoint(20.7f,
temperatureDataStream);
temperatureDataPoint.setDescription("Tank temperature");

DataPoint inValveDataPoint = new DataPoint(true, inValveDataStream);
inValveDataPoint.setDescription("In valve status");

DataPoint levelDataPoint = new DataPoint(64, levelDataStream);
levelDataPoint.setDescription("Tank level percentage");
```

Once all data points are created, send them to Device Cloud using one of the following methods within the `CloudConnectorManager` object.

Method	Description
`sendDataPoint(DataPoint)`	Sends the given `DataPoint` to Device Cloud
`sendDataPoints(List<DataPoint>)`	Sends the given list of `DataPoint` to Device Cloud

The `sendDataPoint(DataPoint)` method may fail for the following reasons:

- The specified `DataPoint` is `null` throwing a `NullPointerException`.
- Cloud Connector is disconnected from Device Cloud throwing an `UnsupportedOperationException`.

The `sendDataPoints(List<DataPoint>)` method may fail for the following reasons:

- The specified `DataPoint` list is `null` throwing a `NullPointerException`.
- Cloud Connector is disconnected from Device Cloud throwing an `UnsupportedOperationException`.
- Size of data points list is < `CloudConnectorManager.MINIMUM_DATA_POINTS` or size of data points list is > `CloudConnectorManager.MAXIMUM_DATA_POINTS` throwing an `IllegalArgumentException`.

Sending a single data point

```
import com.digi.android.cloudconnector.CloudConnectorManager;
import com.digi.android.cloudconnector.DataPoint;
import com.digi.android.cloudconnector.DataStream;

[...]

CloudConnectorManager connectorManager = new
CloudConnectorManager(this);
connectorManager.connect();

DataStream myDataStream = new DataStream("temperature", "C", new
String[0]);

DataPoint dataPoint1 = new DataPoint(20.7f, myDataStream);
DataPoint dataPoint2 = new DataPoint(21.2f, myDataStream);
DataPoint dataPoint3 = new DataPoint(21.5f, myDataStream);

// Upload data points to Device Cloud one by one.
connectorManager.sendDataPoint(dataPoint1);
connectorManager.sendDataPoint(dataPoint2);
connectorManager.sendDataPoint(dataPoint3);
```

Sending a list of data points

```
import com.digi.android.cloudconnector.CloudConnectorManager;
import com.digi.android.cloudconnector.DataPoint;
import com.digi.android.cloudconnector.DataStream;

[...]

CloudConnectorManager connectorManager = new
CloudConnectorManager(this);
connectorManager.connect();

DataStream myDataStream = new DataStream("temperature", "C", new
String[0]);

ArrayList<DataPoint> datapoints = new ArrayList<DataPoint>();
dataPoints.add(new DataPoint(20.7f, myDataStream));
dataPoints.add(new DataPoint(21.2f, myDataStream));
dataPoints.add(new DataPoint(21.5f, myDataStream));

// Upload a list of data points to Device Cloud.
connectorManager.sendDataPoints(datapoints);
```

The process that uploads data points to Device Cloud generates events that can be captured by your application. Send methods return immediately so if you want to track the result of the operation you need to use Cloud Connector events listeners. Refer to the Capture Cloud Connector events topic for more information about events.

Binary data point

If you need to minimize the data traffic, use the **binary concise alternative format** to send data points to Device Cloud. It satisfies the requirement of being concise and less verbose than the `sendDataPoint` met hod: you can specify a simple binary value, when you push this value to Device Cloud it will be stored as is, in the exact binary format you provided.

To use this format, binary data points have to be defined. They are represented by the class `BinaryDataP oint` (`com.digi.android.cloudconnector.BinaryDataPoint`) and contain a single value that it is stored at a specific time in a data stream in Device Cloud, similar to the `DataPoint` objects with the following differences:

- Their data is **always a byte array**.
- They **do not have other attributes** such as description, quality or location.
- The **timestamp will be always set by Device Cloud** when the value is received.
- The data stream containing binary data points will not have units and it is not possible to use a list of other data streams names to replicate data points. This is, the `units` and `forwardTo` attributes of the associated `DataStream`object will not be used.

The constructor needs the value of the binary data point (which is an array of bytes) and the DataStream object as parameters. Once instantiated you can get some of its parameters using the following methods.

Method	Description
`getData()`	Returns a copy of the data point data
`getDataStream()`	Returns the data stream destination of the data point.

BinaryDataPoint constructor

```
import com.digi.android.cloudconnector.BinaryDataPoint;
import com.digi.android.cloudconnector.DataStream;

[...]

DataStream binaryDataStream = new DataStream("binaryData");
BinaryDataPoint binaryDataPoint = new BinaryDataPoint(new byte[16],
binaryDataStream);
```

The binary concise mechanism has the following limitations:

- **Only a single binary data point** can be uploaded at the same time.
- The data of the `BinaryDataPoint` object must be **smaller than 64KB**.

Once you have your binary data point instantiated you can upload it to Device Cloud using the following method of the `CloudConnectorManager` object.

Method	Description
`sendBinaryDataPoint(BinaryDataPoint)`	Sends the given `BinaryDataPoint` to Device Cloud.

The `sendBinaryDataPoint(BinaryDataPoint)` method may fail for the following reasons:

- The specified `BinaryDataPoint` is `null` throwing a `NullPointerException`.
- Cloud Connector is disconnected from Device Cloud throwing an `UnsupportedOperationExcept ion`.
- The binary value to be sent is > `CloudConnectorManager.MAX_SIZE_OF_BINARY_DATA_POIN TS_KB` throwing an `IllegalArgumentException`.

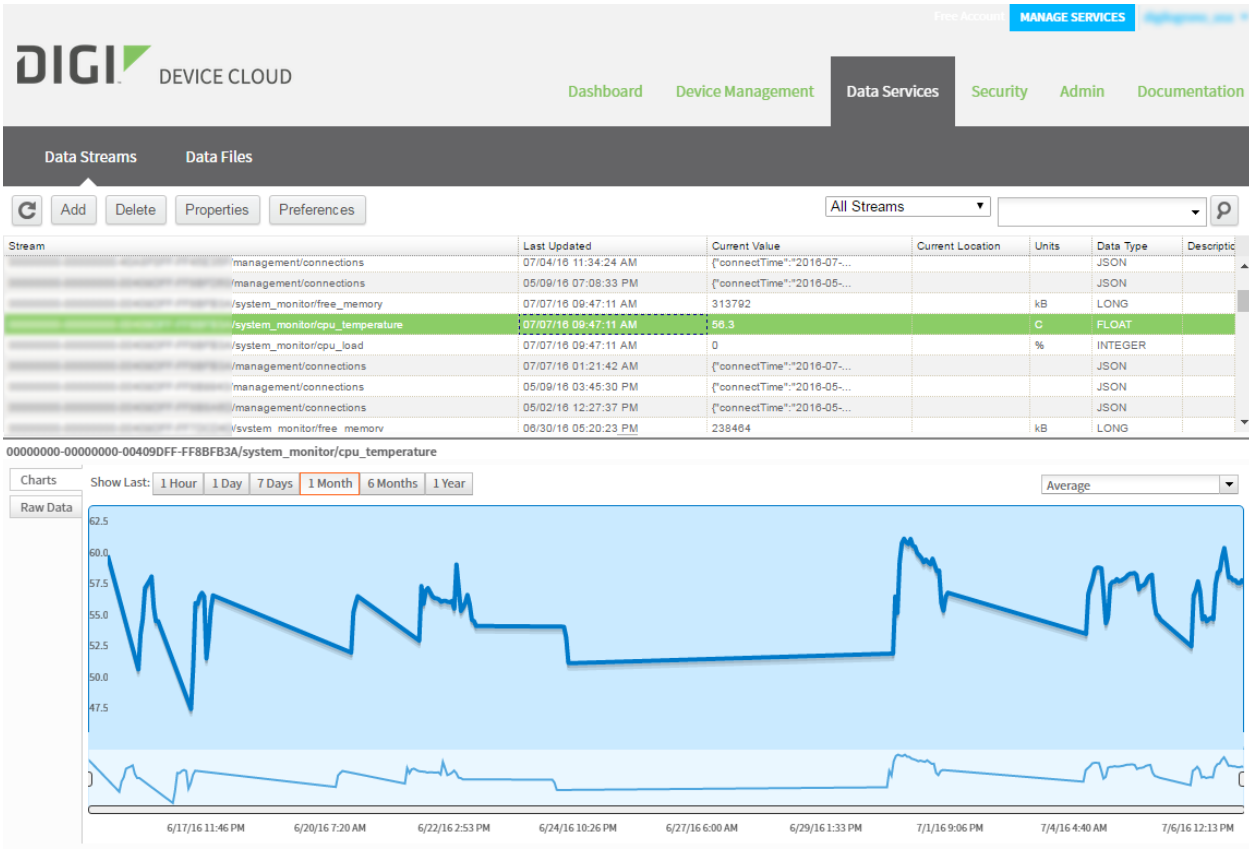
Sending a binary data point

```
import com.digi.android.cloudconnector.BinaryDataPoint;
import com.digi.android.cloudconnector.CloudConnectorManager;
import com.digi.android.cloudconnector.DataStream;

[...]

CloudConnectorManager connectorManager = new
CloudConnectorManager(this);
connectorManager.connect();

DataStream binaryDataStream = new DataStream("binaryData");

BinaryDataPoint binaryDataPoint = new BinaryDataPoint(new byte[16],
binaryDataStream);

// Upload the binary data point to Device Cloud.
connectorManager.sendBinaryDataPoint(binaryDataPoint);
```

Reviewing data stream series in Device Cloud

You can view the data points stored in the cloud (in charts and in text format) from the Device Cloud platform. To do so follow these steps:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Data Services** tab and select **Data Streams**.
3. Select the data stream you want to analyze from the list of streams.
4. In the box below, select the format to represent the values (**Charts** or **Raw Data**). Data points contained in the data stream are displayed in a chart or in a list.



CPU management

The device CPU can scale their operating frequency (changing a voltage power supply input) according to the system/user needs. This way, when the entire processor resources are not needed, the system can greatly reduce the overall power consumption, lowering temperatures.

This API allows you to enable and disable the available CPU cores, change their frequencies and establish the governor on the fly. It also provides access to the current CPU temperature and trip points. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all CPU API methods require the **`com.digi.android.permission.CPU`** permission.

> If your application does not have the `com.digi.android.permission.CPU` permission it will not have access to any CPU service feature.

To work with this API and manage your device CPU, the first step is to instantiate a `CPUManager` object. To do this, you need to provide the application `Context`.

Instantiating the CPUManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.system.cpu.CPUManager;

public class CPUSampleActivity extends Activity {

    CPUManger cpuManager;

    [...]

    @Override
 public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceStace);

        // Instantiate the CPU manager object.
        cpuManager = new CPUManager(this);

        [...]
    }

    [...]
}
```

The `CPUManager` allows you to:

- Control CPU cores
- Configure governors
- Manage CPU temperature

Control CPU cores

The `CPUManager` includes the ability to control the CPU cores of your device by enabling and disabling them, increase and decrease the CPU frequencies, and display the usage of each core as well as the overall CPU usage.

Control CPU cores

There are different methods to enable and disable the different cores of your device and verify if any of them is online or offline.

Method	Description
`enableCore(int)`	Specifies the index of the core to be enabled
`disableCore(int)`	Specifies the index of the core to be disabled
`isCoreEnabled(int)`	Specifies the index of the core to check whether is enabled

These methods may fail for the following reasons:

- The specified core does not exists throwing a `NoSuchCoreException`.
- If there is an error during the configuration or while checking the core status throwing a `CPUException`.

To know the number of available cores in the device, use the method `getNumberOfCores()` of the `CPUManager` class.

> It is not possible to disable CPU0 on your device. CPU0 is the default core and must be always running.

Controlling CPU cores

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = new CPUManager(context);

// Get the number of available cores.
int numberOfCores = cpuManager.getNumberOfCores();
System.out.println("Number of cores: " + numberOfCores);

// Check enabled cores.
for (int i = 1; i < numberOfCores; i++) {
    boolean coreEnabled = cpuManager.isCoreEnabled(i);
    System.out.println("Core " + i + ": " + coreEnabled);

    // Enable disabled cores.
    if (!coreEnabled) {
        System.out.println("Enabling core " + i);
        cpuManager.enableCore(i);
    }
}

[...]
```

Monitor CPU usage

This API lets you monitor the CPU utilization. It calculates the percentage of total CPU time (including all cores) and the percentage per each core over a period of time.

Method	Description
• `getCPUUsage()` • `getCPUUsage(long)`	Returns the total CPU usage in percentage. This method blocks during 300 milliseconds or the specified time.
• `getCoreUsage(int)` • `getCoreUsage(int, long)`	Returns the usage of the provided core in percentage. This method blocks during 300 milliseconds or the specified time.
• `getUsage()` • `getUsage(long)`	Returns a list with all the CPU usage percentages. Overall CPU usage is at index 0, it is followed by the core usages for the available cores. This method blocks during 300 milliseconds or the specified time.

> Getting CPU usage is a blocking operation. Any of these methods waits until the default interval (300 milliseconds) or the specified interval is reached.

These methods may fail for the following reasons:

- The specified core does not exists throwing a `NoSuchCoreException`.
- If there is an error during while calculating the usage throwing a `CPUException`.

Monitoring CPU usage

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = ...;

// Get CPU usage in the last minute.
System.out.println("Overall CPU usage: " + cpuManager.getCPUUsage(60 *
1000) + " %");

// Get usage per core.
ArrayList<Float> usages = cpuManager.getUsage();
for (int i = 1; i < usages.size(); i++) {
    System.out.println("CPU " + i + " usage: " + usages.get(i));
}

[...]
```

Configure CPU frequencies

You can obtain the list of available frequencies that your device can scale. There are also methods to read the current CPU speed and the frequency limits, some of them can also be changed.

Method	Description
`getAvailableFrequencies()`	Lists the available frequencies supported by the device in kHz
• `getMinFrequency()` • `getMaxFrequency()`	Return the minimum and maximum operating frequency the CPU supports in kHz
`getTransitionLatency()`	Returns the time in nanoseconds it takes on the CPU to switch between two frequencies
• `getMinScalingFrequency()` • `getMaxScalingFrequency()`	Shows the current "policy limits". The minimum and maximum frequency (kHz) the device may scale the CPU to
• `setMinScalingFrequency(int)` • `setMaxScalingFrequency(int)`	Change the current "policy limits". When setting these frequency limits you need to first set the maximum value, then the minimum. These values must be between the maximum and minimum frequencies returned by `getMinFrequency()` and `getMaxFrequency()`.
`getFrequency()`	Returns the current frequency of the CPU as obtained from the hardware, in KHz. This is the frequency the CPU actually runs at

These methods may fail if there were any error while reading or changing the values throwing a `CPUException`.

CPU frequencies

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = ...;

// List supported frequencies.
System.out.println("Available frequencies:");
ArrayList<Integer> availableFreqs =
cpuManager.getAvailableFrequencies();
for (Integer f: availableFreqs) {
    System.out.println("    " + f);
}

// Min and max supported frequencies.
System.out.println("Min freq: " + cpuManager.getMinFrequency());
System.out.println("Max freq: " + cpuManager.getMaxFrequency());

// Set min and max frequency limits.
cpuManager.setMinScalingFrequency(availableFreqs[0]);
cpuManager.setMaxScalingFrequency(availableFreqs[availableFreqs.size() -
1]);

System.out.println("Min scaling freq: " +
cpuManager.getMinScalingFrequency());
System.out.println("Max scaling freq: " +
cpuManager.getMaxScalingFrequency());

// Get current CPU speed.
System.out.println("Current freq: " + cpuManager.getFrequency());

[...]
```

Besides all of this you can perform additional actions depending on the governor you have selected:

- Userspace governor: The method `setFrequency(int)` allows the change of the CPU operating frequency to a specific value in kHz, but only within the limits defined by `getMinScalingFrequency()` and `getMaxScalingFrequency()`.
- Interactive governor: The method `boostPulse()` immediately boosts the speed of all CPUs to `hispeed_freq` (`getHiSpeedFreq()` of `InteractiveGovernor` class) for the period of time specified by the `boost_pulse_duration` property (`getBoostPulseDuration()` of `InteractiveGovernor` class).

For more information about governors go to Configure governors.

CPU Management Example

The **CPU Management Sample Application** demonstrates the usage of the CPU API. In this example, you can enable and disable the CPU cores, configure frequencies, select and set up the governor while monitoring the total CPU usage as well as the usage of each core.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the <TODO verify link GitHub repository >.

Configure governors

A CPU governor controls how the CPU raises and lowers its frequency in response to the demands the user is placing on their device. Governors have a large impact on performance and power save.

There are some important things to look out for before selecting a governor:

- **Performance**. Usually having lots of speed equates to lower power save, so it is best to balance this out.
- **Power save**. Being very battery friendly usually means less speed (or sometimes smoothness).
- **Stability**. Some governors are plain unstable and some are rock solid.
- **Smoothness (or Fluidity)**. This is not the same as speed, a governor can be fast but it doesn't mean it is smooth. A way to test this is to scroll down/up pages or open and close apps. Of course, more smoothness = awesome experience.

Governor types

There are several governor types to configure the CPU, all of them are contained in a enumerator called `GovernorType`. Each `GovernorType` is also represented by a class in the API.

Governor	Type	Class	Description
Performance	`GovernorType.PERFORMANCE`	`GovernorPerformance`	Sets the CPU statically to the highest frequency
Powersave	`GovernorType.POWERSAVE`	`GovernorPowerSave`	Sets the CPU statically to the lowest frequency
Userspace	`GovernorType.USERSPACE`	`GovernorUserSpace`	Sets the CPU statically to the user specified frequencies
Ondemand	`GovernorType.ONDEMAND`	`GovernorOnDemand`	Sets the CPU depending on the current usage. It jumps to max speed when there is load on the CPU. If the CPU load abates, it slowly steps back down through the kernel's frequency steppings until it settles at the lowest frequency.
Conservative	`GovernorType.CONSERVATIVE`	`GovernorConservative`	Sets the CPU depending on the current usage. It gracefully increases and decreases the CPU speed rather than jumping to the maximum the moment there is any load on the CPU.
Interactive	`GovernorType.INTERACTIVE`	`GovernorInteractive`	Sets the CPU depending on the current usage. This governor is more aggressive about scaling the CPU speed up in response to CPU-intensive activity.

To obtain the list of available governor types supported by the CPU use the `getAvaliableGovernorTypes()` method of `CPUManager`.

Getting available governor types

```
import com.digi.android.system.cpu.CPUManager;
import com.digi.android.system.cpu.GovernorType;

[...]

// Get the CPU manager.
CPUManager cpuManager = new CPUManager(context);

// Get the available governor types.
ArrayList<GovernorType> governorTypes =
cpuManager.getAvailableGovernorTypes();
for (GovernorType type: governorTypes) {
    System.out.println(type.getID() + ": " + type.getDescription());
}

[...]
```

Get and set the CPU governor

You can get and set the current CPU governor using the following methods:

Method	Description
`getGovernorType()`	Returns the governor type the CPU is configured with
`getGovernor()`	Returns a `Governor` object corresponding to the governor that the CPU is configured with
`setGovernorType(GovernorType)`	Configures the CPU with the provided governor type and returns the corresponding `Governor` object

These methods may fail throwing a `CPUException`, if there is any error during the reading or configuration process.

Getting and setting current governor type

```
import com.digi.android.system.cpu.CPUManager;
import com.digi.android.system.cpu.GovernorType;

[...]

CPUManager cpuManager = ...;

// Get the CPU governor type.
GovernorType governorType = cpuManager.getGovernorType();
System.out.println("Current CPU governor: " + governorType.getID());

// Configure the CPU to use the Powersave governor.
Governor governor = cpuManager.setGovernorType(GovernorType.POWERSAVE);
System.out.println("Configured new CPU governor: " +
governor.getGovernorType());

[...]
```

The `getGovernor()` and `setGovernorType(GovernorType)` methods return a `Governor` object. You can cast this object to the corresponding governor class to access its specific features.

Governor classes

```
import com.digi.android.system.cpu.CPUManager;
import com.digi.android.system.cpu.Governor;
import com.digi.android.system.cpu.GovernorType;

[...]

CPUManager cpuManager = ...;

// Get the CPU governor type.
Governor governor = cpuManager.getGovernor();

// Cast governor depending on its type.
switch (governor.getGovernorType()) {
    case GovernorType.PERFORMANCE:
        PerformanceGovernor performanceGovernor =
(PerformanceGovernor)governor;
        // TODO Custom implementation.
        break;
    case GovernorType.POWERSAVE:
        PowerSaveGovernor powerSaveGovernor =
(PowerSaveGovernor)governor;
        // TODO Custom implementation.
        break;
    case GovernorType.USERSPACE:
        UserSpaceGovernor userSpaceGovernor =
(UserSpaceGovernor)governor;
        // TODO Custom implementation.
        break;
    case GovernorType.ONDEMAND:
        OnDemandGovernor onDemandGovernor = (OnDemandGovernor)governor;
        // TODO Custom implementation.
        break;
    case GovernorType.CONSERVATIVE:
        ConservativeGovernor conservativeGovernor =
(ConservativeGovernor)governor;
        // TODO Custom implementation.
        break;
    case GovernorType.INTERACTIVE:
        InteractiveGovernor interactiveGovernor =
(InteractiveGovernor)governor;
        // TODO Custom implementation.
        break;
    default:
        // TODO Custom implementation.
        break;
}

[...]
```

Configure governor parameters

Some governors are configurable and specify different parameters that can be read and set but not others.

Performance, powersave, and userspace governors do not define any parameter. They just configure the CPU statically to the maximum, minimum, or user-specified frequency respectively when they are selected.

The configurable governors are:

- Ondemand
- Conservative
- Interactive

All of them define some common and specific accessible parameters:

- Ondemand governor parameters

Parameter	Method	Description
`sampling_rate`	• `getSamplingRate()` • `setSamplingRate(long)`	Represents in microseconds how often the kernel will look at the CPU usage and make decisions on what to do about the frequency
`sampling_rate_min`	• `getMinSamplingRate()`	Minimum sampling rate
`up_threshold`	• `getUpThreshold()` • `setUpThreshold(int)`	Defines what the average CPU usage between the samplings of `sampling_rate` needs to be for the kernel to make a decision on whether it should increase the frequency
`ignore_nice_load`	• `isIgnoreNiceLoadEnabled()` • `enableIgnoreNiceLoad()` • `disableIgnoreNiceLoad()`	When disabled (its default), all processes are counted towards the 'CPU utilization' value. When enabled, the processes that are run with a 'nice' value will not count (and thus be ignored) in the overall usage calculation
`sampling_down_factor`	• `getSamplingDownFactor()` • `setSamplingDownFactor(int)`	Controls the rate at which the kernel makes a decision on when to decrease the frequency while running at top speed

- Conservative governor parameters

Parameter	Method	Description

`sampling_rate`	• `getSamplingRate()` • `setSamplingRate(long)`	Represents in microseconds how often the kernel will look at the CPU usage and make decisions on what to do about the frequency
`sampling_rate_min`	• `getMinSamplingRate()`	Minimum sampling rate
`up_threshold`	• `getUpThreshold()` • `setUpThreshold(int)`	Defines what the average CPU usage between the samplings of `sampling_rate` needs to be for the kernel to make a decision on whether it should increase the frequency
`ignore_nice_load`	• `isIgnoreNiceLoadEnabled()` • `enableIgnoreNiceLoad()` • `disableIgnoreNiceLoad()`	When disabled (its default), all processes are counted towards the 'CPU utilization' value. When enabled, the processes that are run with a 'nice' value will not count (and thus be ignored) in the overall usage calculation
`sampling_down_factor`	• `getSamplingDownFactor()` • `setSamplingDownFactor(int)`	Controls the rate at which the kernel makes a decision on when to decrease the frequency while running at top speed
`down_threshold`	• `getDownThreshold()` • `setDownThreshold(int)`	Defines what the average CPU usage between the samplings of `sampling_rate` needs to be for the kernel to make a decision on whether it should decrease the frequency
`freq_step`	• `getFreqStep()` • `setFreqStep(int)`	Describes what percentage steps the CPU freq should be increased and decreased smoothly by.

- Interactive governor parameters

Parameter	Method	

min_sample_time	• getMinSampleTime() • setMinSampleTime(long)	Represents the minimum amount of microseconds to spend at the current frequency before ramping down
hispeed_freq	• getHiSpeedFreq() • setHiSpeedFreq(int)	An intermediate "high speed" at which to initially ramp when CPU load hits the value specified in go_hispeed_load
go_hispeed_freq	• getGoHiSpeedLoad() • setGoHiSpeedLoad(int)	The CPU load at which to ramp to the intermediate "high speed" specified in hispeed_freq
above_hispeed_delay	• getAboveHiSpeedDelay() • setAboveHiSpeedDelay(long)	Once speed is set to hispeed_freq, time in microseconds ti wait before bumping speed higher in response to continued high load
timer_rate	• getTimerRate() • setTimerRate(long)	Sample rate in microseconds for reevaluating CPU load when the system is not idle
timer_slack	• getTimerSlack() • setTimerSlack(long)	Maximum additional time in microseconds to defer handling the governor sampling timer beyond timer_rate when running at speeds above the minimum
boost	• isBoostEnabled() • enableBoost() • disableBoost()	When boost is enabled, immediately boosts the speed of all CPUs to at least hispeed_freq until boost is disabled again. When disabled, allows CPU speeds to drop below hispeed_freq to load as usual

`bootspulse_duration`	• `getBoostPulseDuration()` • `setBoostPulseDuration(long)`	Amount of microseconds to hold CPU speed at `hispeed_freq` on a boost pulse, before allowing speed to drop according to load as usual

Configuring governor parameters

```
import com.digi.android.system.cpu.CPUManager;
import com.digi.android.system.cpu.GovernorType;
import com.digi.android.system.cpu.InteractiveGovernor;

[...]

CPUManager cpuManager = ...;

// Set the interactive governor.
InteractiveGovernor interactiveGovernor =
(InteractiveGovernor)cpuManager.setGovernorType(GovernorType.INTERACTIVE
);

// Configure the governor.
interactiveGovernor.setMinSampleTime(80000);
interactiveGovernor.setTimerRate(20000);

[...]

// Read some governor parameters.
System.out.println("High speed frequency: " +
interactiveGovernor.getHiSpeedFreq());
System.out.println("Boost enabled: " + interactiveGovernor.getBoost());

[...]
```

CPU Management Example

The **CPU Management Sample Application** demonstrates the usage of the CPU API. In this example, you can enable and disable the CPU cores, configure frequencies, select and set up the governor while monitoring the total CPU usage as well as the usage of each core.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the <TODO verify link GitHub repository >.

Manage CPU temperature

The CPU manager addresses three concepts related to temperature:

- **Current temperature**. The current CPU temperature.
- **Hot temperature**. The temperature limit at which system will reduce CPU and GPU frequency to avoid system overheating.
- **Critical temperature**. The temperature limit at which system will halt to avoid system damage caused by overheating. This always occurs after reaching hot temperature.

Get CPU temperature

There are different methods to obtain the different temperature values:

Parameter	Method
Current temperature	`getCurrentTemperature()`
Hot temperature	`getHotTemperature()`
Critical temperature	`getCriticalTemperature()`

All these methods returns a `float` value and may throw a `CPUTemperatureException` if there is an error reading the corresponding temperature value.

Getting temperature values

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = new CPUManager(context);

// Read the temperature values.
System.out.println("Current temperature: " +
cpuManager.getCurrentTemperature());
System.out.println("Hot temperature: " +
cpuManager.getHotTemperature());
System.out.println("Critical temperature: " +
cpuManager.getCriticalTemperature());

[...]
```

Monitor CPU temperature

You can receive notifications about CPU temperature if you subscribe a **`ICPUTemperatureListener`** to the `CPUManager`. Use the **`registerListener(ICPUTemperatureListener, long)`** method to register for temperature updates specifying the interval between updates, in milliseconds.

Temperature updates registration

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = new CPUManager(context);

// Create the temperature listener.
MyCPUTemperatureListener myTempListener = ...;

// Register the temperature listener to be notified every 5 seconds.
cpuManager.registerListener(myTempListener, 5000);

[...]
```

The registered listener class, `MyCPUTemperatureListener`, must implement the `ICPUTemperatureListener` interface. This interface defines the `onTemperatureUpdate(float)` method, that is called when the temperature is updated.

ICPUTemperatureListener implementation example, MyCPUTemperatureListener

```
import com.digi.android.system.cpu.ICPUTemperatureListener;

public class MyCPUTemperatureListener implements ICPUTemperatureListener
{
    @Override
    public void onTemperatureUpdate(float temperature) {
        // This code will be executed every 5 seconds, the interval
configured in the registerListener method.
        System.out.println("New temperature value " + temperature);
    }
}
```

To stop the temperature notifications, use the `unregisterListener(ICPUTemperatureListener)` method to unsubscribe the already registered listener.

Temperature updates unregistration

```
[...]

CPUManager cpuManager = ...;
MyCPUTemperatureListener myTempListener = ...;

cpuManager.registerListener(myTempListener, 5000);

[...]

// Remove the temperature listener.
cpuManager.unregisterListener(myTempListener);

[...]
```

Set CPU temperature trip points

The hot and critical temperature values can be configured using the following methods:

Parameter	Method
Hot temperature	`setHotTemperature(float)`
Critical temperature	`setCriticalTemperature(float)`

Both require the new value of the trip point to configure as a `float` and return the configured value.

The configuration of any of these trip points may fail for the following reasons:

- The hot temperature is equal or greater than the critical temperature, throwing an `IllegalArgumentException`.
- If any error occurs while setting the new trip point value, throwing a `CPUTemperatureException`.

For a ConnectCore 6 the maximum value for the trip points are:

- Maximum hot temperature value: 85C
- Maximum critical temperature value: 100C

Setting temperature trip points

```
import com.digi.android.system.cpu.CPUManager;

[...]

// Get the CPU manager.
CPUManager cpuManager = new CPUManager(context);

// Set the new values of the trip points.
float configuredHotTemp = cpuManager.setHotTemperature(75);
float configuredCriticalTemp = cpuManager.setCriticalTemperature(95);

[...]
```

CPU Temperature Example

The **CPU Temperature Sample Application** demonstrates how to manage the CPU temperature. In this example you can read the trip points and configure the sampling rating of the current temperature in ConnectCore 6.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Ethernet

Ethernet is a family of computer networking technologies commonly used in local area networks (LANs) and metropolitan area networks (MANs). It describes how networked devices can format data for transmission to other network devices, and how to put that data out on the network connection.

Ethernet is one of the most important interfaces of the ConnectCore 6 SBC as it plays a critical role accessing Internet and communicating with other devices.
The ConnectCore 6 SBC provides one Gigabit Ethernet interface for networking communication. You can find more information about it in the ConnectCore 6 Hardware Reference Manual.

Digi adds a new API to Android that allows you to manage this Ethernet interface. You can reset the interface or read and change its configuration. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all API methods require the **`android.permission.ACCESS_NETWORK_STATE`** and/or **`android.permission.CHANGE_NETWORK_STATE`** permissions, depending on whether you are accessing or changing the Ethernet configuration.

> If your application does not have the `android.permission.ACCESS_NETWORK_STATE` and/or a `ndroid.permission.CHANGE_NETWORK_STATE` permissions, it will not have access to any Ethernet service feature.

First of all, you have to instantiate the `EthernetManager` object by passing the Android Application Context.

Instantiating the EthernetManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.ethernet.EthernetManager;

public class EthernetSampleActivity extends Activity {

 EthernetManager ethernetManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the Ethernet manager object.
        ethernetManager = new EthernetManager(this);

  [...]
 }

 [...]
}
```

Read the current configuration

The `EthernetManager` class allows you to read the current interface configuration using the following

methods:

Method	Description
`getInterfaceName()`	Returns the Ethernet interface name
`getConnectionMode()`	Returns the configured connection mode of the interface: `EthernetConnectionMode.DHCP`, `EthernetConnectionMode.STATIC`, or `EthernetConnectionMode.UNKNOWN`
`getIp()`	Returns the configured IP address of the interface
`getNetmask()`	Returns the configured netmask address of the interface
`getGateway()`	Returns the configured gateway address of the interface
`getDns1()`	Returns the configured DNS1 address of the interface
`getDns2()`	Returns the configured DNS2 address of the interface
`getMacAddress()`	Returns the MAC address of the interface
`isConnected()`	Returns whether the interface is connected or not
`isEnabled()`	Returns whether the interface is enabled or not

The `getMacAddress()` method may fail if the configured interface is null, throwing a `NullPointerException`.

Reading the interface configuration

```
import com.digi.android.ethernet.EthernetManager;

[...]

EthernetManager ethernetManager = ...;

[...]

if (ethernetManager.isEnabled && ethernetManager.isConnected) {
 System.out.println("Name: " + ethernetManager.getInterfaceName());
 System.out.println("MAC Address: " + ethernetManager.getMacAddress());
 System.out.println("Mode: " + ethernetManager.getConnectionMode());
 System.out.println("IP: " + ethernetManager.getIp());
 System.out.println("Netmask: " + ethernetManager.getNetmask());
 System.out.println("Gateway: " + ethernetManager.getGateway());
 System.out.println("DNS1: " + ethernetManager.getDns1());
 System.out.println("DNS2: " + ethernetManager.getDns2());
}

[...]
```

All these operations require the `android.permission.ACCESS_NETWORK_STATE` permission.

Configure the interface

It is also possible to configure and enable or disable the Ethernet interface using this API. You can use these methods to achieve it:

Method	Description
`configureInterface(EthernetConfiguration)`	Configures the interface with the given configuration. It requires an object of type `EthernetConfiguration`. Note that this method resets the interface in order to apply the changes
`setEnabled(boolean)`	Enables or disables the interface
`resetInterface()`	Resets the interface

The `configureInterface(EthernetConfiguration)` method may fail for the following reasons:

- If the given configuration is null, throwing a `NullPointerException`.
- If the given IP address cannot be resolved, throwing an `UnknownHostException`.

The `EthernetConfiguration` class offers a series of methods to fill and read the configuration object:

Method	Description
`setConnectionMode(EthernetConnectionMode)`	Sets the Ethernet connection mode
`getConnectionMode()`	Returns the Ethernet connection mode
`setInterfaceName(String)`	Sets the interface name
`getInterfaceName()`	Returns the interface name
`setIpAddress(InetAddress)`	Sets the IP address
`getIpAddress()`	Returns the IP address
`setGateway(InetAddress)`	Sets the gateway address
`getGateway()`	Returns the gateway address
`setNetMask(InetAddress)`	Sets the net mask address
`getNetMask()`	Returns the net mask address
`setDns1Addr(InetAddress)`	Sets the DNS 1 address
`getDns1Addr()`	Returns the DNS 1 address
`setDns2Addr(InetAddress)`	Sets the DNS 2 address
`getDns2Addr()`	Returns the DNS 2 address

Changing the interface configuration

```
import com.digi.android.ethernet.EthernetManager;
import com.digi.android.ethernet.EthernetConfiguration;
import com.digi.android.ethernet.EthernetConnectionMode;

import java.net.InetAddress;

[...]

EthernetManager ethernetManager = ...;

[...]

// Enable the interface if it isn't.
if (!ethernetManager.isEnabled())
 ethernetManager.setEnabled(true);

// Set a static configuration.
EthernetConfiguration config = new EthernetConfiguration();
config.setInterfaceName(ethernetManager.getInterfaceName());
config.setConnectionMode(EthernetConnectionMode.STATIC);
config.setIp(InetAddress.getByName("192.168.1.150"));
config.setNetmask(InetAddress.getByName("255.255.255.0"));
config.setGateway(InetAddress.getByName("192.168.1.1"));
config.setDns1(InetAddress.getByName("8.8.8.8"));

ethernetManager.configureInterface(config);

[...]
```

All these operations require the `android.permission.CHANGE_NETWORK_STATE` permission.

Firmware update (API)

Latest version of the Android system provided by Digi includes the possibility to update the firmware of the ConnectCore 6 device from the Android system itself. Refer to the Update the Android firmware topic to learn how to perform the firmware update through the Android settings. If you prefer, you can add the firmware update functionality to an Android application using the firmware update service provided by Digi.

With the Firmware update API you can perform a firmware update of the system from an Android application, install new applications and wipe the user and data partitions. In the Javadoc documentation y ou can find a complete list of the available methods in this API.

Unless noted, all Firmware update API methods require the **`com.digi.android.permission.FIRMWARE_UPDATE`** permission.

> If your application does not have the `com.digi.android.permission.FIRMWARE_UPDATE` per mission it will not have access to any Firmware update service feature.

First of all, a new `FirmwareUpdateManager` object must be instantiated by passing the Android Application Context.

Instantiating the FirmwareUpdateManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.firmwareupdate.FirmwareUpdateManager;

public class FirmwareUpdateSampleActivity extends Activity {

 FirmwareUpdateManager firmwareUpdateManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the FirmwareUpdateManager manager object.
        firmwareUpdateManager = new FirmwareUpdateManager(this);

  [...]
 }

 [...]
}
```

Listen for firmware update events

The firmware update process generates some events that can be get and processed in your application. In order to make your application aware of these events, the first thing you need to do is to create your own Firmware update listener. This listener class must implement the `IFirmwareUpdateListener` interface.

Firmware update listener

```
import com.digi.android.firmwareupdate.IFirmwareUpdateListener;

class MyFirmwareUpdateListener implements IFirmwareUpdateListener {
    @Override
    void updatePackageCopyStarted() {
    }

    @Override
    void updatePackageCopyFinished() {
    }

    @Override
    void verifyStarted() {
    }

    @Override
    void verifyProgress(int progress) {
    }

    @Override
    void verifyFinished() {
    }

    @Override
    void updateStarted() {
    }

    @verride
    void onError(String error) {
    }
}
```

The `IFirmwareUpdateListener` interface implements the following methods to handle the events generated during a firmware update process.

Method	Description
`updatePackageCopyStarted()`	Notifies that update package copy operation started
`updatePackageCopyFinished()`	Notifies that update package copy operation finished
`verifyStarted()`	Notifies that update package verification started
`verifyProgress(int)`	Notifies about update package verification progress and provides the progress percentage
`verifyFinished()`	Notifies that package verification finished
`updateStarted()`	Notifies that the update process started and device is about to reboot
`onError(String)`	Notifies about firmware update error and provides the error message

Depending on the parameters provided in the `installUpdatePackage` command of the API and the location of the update package, some events won't be generated. For example, the firmware update is always performed from the `cache` partition of the device. If the update package file is not there it is copied during the process. If the update package file is already located in the `cache` partition it won't be copied there and the `updatePackageCopyStarted` and `updatePackageCopyFinished` callbacks won't be fired. Also, all the verify callbacks will be called only if the verify option is set to true when calling the `installUpdatePackage` command.

This listener is a mandatory parameter of the `installUpdatePackage` command of the Firmware update API. It is the command used to perform a firmware update of the Android system. Refer to the Install an update package section for more information about it.

Install an application

One of the features provided by the Firmware update service is to install an Android application. This is done calling the corresponding install application method and providing the full path of the APK to be installed as parameter. If you want to reboot the module after installing the application you have to indicate it with the second parameter corresponding to the reboot (boolean) option.

Method,	Description
`installApplication(String, boolean)`	Installs the given application

The `installApplication(String)` method may fail for the following reasons:

- The provided application path is `null` throwing a `NullPointerException`.
- There is an error installing the application throwing an `IOException`.

Installing an application

```
import com.digi.android.firmwareupdate.FirmwareUpdateManager;

[...]

FirmwareUpdateManager firmwareUpdateManager = [...]

[...]

// Install the application located at
'/storage/emulated/0/MyApplication.apk'
// - Do not restart the device after application installation.
firmwareUpdateManager.installApplication("/storage/emulated/0/MyApplicat
ion.apk", false);
```

> Notice that this method will not ask for any permission validation of the application to install as the installation process is performed by the Firmware update Android service.

Install an update package

The main purpose of this Firmware update service is to install an update package to update the Android system. This operation is performed calling the installUpdatePackage and providing the following parameters:

- **updatePackagePath**: Full path of the update package to install
- **listener**: An object implementing the IFirmwareUpdateListener interface to receive firmware update progress information. Refer to the Listen for firmware update events section for more information

about it.
- **verify**: Boolean parameter indicating if the update package signature should be verified.

  > Package signature will be verified using the provided certificates zip file if any. If not, the default system certificates file will be used.

- **deviceCertsZipFile**: Full path to the zip file of certificates whose public keys will be accepted. Verification succeeds if the package is signed by the private key corresponding to any public key in this file. May be null to use the system default file, currently `"/system/etc/security/otacerts.zip"`. This parameter only applies if `verify` is `true`.

Method	Description
`installUpdatePackage(String, listener, boolean, String)`	Installs the given update package

This `installUpdatePackage(String, listener, boolean, String)` may fail if the provided update package path or listener are null throwing a `NullPointerException`.

Installing an update package

```
import com.digi.android.firmwareupdate.FirmwareUpdateManager;

[...]

FirmwareUpdateManager firmwareUpdateManager = [...]

[...]

// Install the update package located at ''.
// - Provide the listener we declared previously.
// - Do not verify the image.
// - Do not provide any certifications file since image won't be
verified
firmwareUpdateManager.installUpdatePackage("/storage/udisk1/ota_update_p
ackage.zip", myFirmwareUpdateListener, false, null);
```

> The module will be restarted during the firmware update process. Remember that events produced during the process can be get with the `IFirmwareUpdateListener` provided in the method.

Wipe partitions

The latest feature you can find inside the Firmware update service is the possibility to wipe some of the Android system partitions. These are the `data` and `cache` partitions, where you have full access and can read/write files. This wipe process can be done using the following methods.

Method	Description
`wipeCache()`	Wipes the cache (/cache) partition
`wipeCache(String)`	Wipes the cache (/cache) partition providing a 'reason' message
`wipeUserData()`	Wipes the user data (/data) and cache (/cache)

`wipeUserData(String)`	Wipes the user data (/data) and cache (/cache) partitions providing a 'reason' message
`wipeUserData(String, boolean)`	Wipes the user data (/data) and cache (/cache) partitions providing a 'reason' message. It also allows for a shutdown instead of reset

All these methods may fail if there is any error preparing for the wipe throwing an `IOException`.

Wiping cache partition

```
import com.digi.android.firmwareupdate.FirmwareUpdateManager;

[...]

FirmwareUpdateManager firmwareUpdateManager = [...]

[...]

// Wipe the cache partition and provide a reason message.
firmwareUpdateManager.wipeCache("I'm sorry for your cache");
```

Wiping cache and data partitions

```
import com.digi.android.firmwareupdate.FirmwareUpdateManager;

[...]

FirmwareUpdateManager firmwareUpdateManager = [...]

[...]

// Wipe the user data and cache partitions and provide a reason message.
firmwareUpdateManager.wipeData("I'm extremely sorry for your data and
cache");
```

The module will be restarted in order to perform the wipe operation. In the case of the `wipeUserData(String, boolean)` method, if the boolean `shutdown` parameter is `true`, the module will be shut down instead of reset.

GPIO

A **general-purpose input/output** (**GPIO**) is a generic pin on an integrated circuit whose behavior—including whether it is an input or output pin—is controllable by the user at run time.

A standard GPIO can be used for the following purposes:

- Reading from switches.
- Reading from sensors such as IR, liquid level.
- Writing output to LEDs for status, relays, etc.

The ConnectCore 6 has several GPIO interfaces; you can find more information in the ConnectCore 6 Hardware Reference Manual.

Digi adds to Android an API to manage these GPIO interfaces. You can configure them, read and set values, and listen for state changes. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all GPIO API methods require the **`com.digi.android.permission.GPIO`** permission.

> If your application does not have the `com.digi.android.permission.GPIO` permission it will not have access to any GPIO service feature.

First of all, a new `GPIOManager` object must be instantiated by passing the Android Application Context.

Instantiating the GPIOManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.gpio.GPIOManager;

public class GPIOSampleActivity extends Activity {

 GPIOManager gpioManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the GPIO manager object.
        gpioManager = new GPIOManager(this);

  [...]
 }

 [...]
}
```

Instantiate a GPIO

The `GPIOManager` allows you to create a `GPIO` object for a specific GPIO interface. You can create it with one of the following methods:

Method	Description
`createGPIO(int, GPIOMode)`	Creates and returns a GPIO object with the given kernel number, configuring it with the given mode.
`createGPIO(int, int, GPIOMode)`	Creates and returns a GPIO object with the given port index and pin index, configuring it with the given mode.

Both methods may fail for the following reasons:

- If there is an error creating the GPIO, throwing a `GPIOException`.
- If kernel number, port index or pin index is less than 0, throwing an `IllegalArgumentException`.
- If mode is null, throwing a `NullPointerException`.

The `GPIOMode` enumeration contains all the possible modes a GPIO can be configured:

- `INPUT`: the GPIO is used as input.
- `OUTPUT_HIGH`: the GPIO is used as output and the output level is set high.
- `OUTPUT_LOW`: the GPIO is used as output and the output level is set low.
- `INTERRUPT_EDGE_RISING`: the GPIO is used as an interrupt on rising edge.
- `INTERRUPT_EDGE_FALLING`: the GPIO is used as an interrupt on falling edge.
- `INTERRUPT_EDGE_BOTH`: the GPIO is used as an interrupt on both rising and falling edges.

Once you have created the GPIO, you can read or set its mode using the `getMode()` and `setMode(GPIOMode)` methods.

Creating a GPIO

```
import com.digi.android.gpio.GPIO;
import com.digi.android.gpio.GPIOManager;
import com.digi.android.gpio.GPIOMode;

[...]

GPIOManager gpioManager = ...;

// Create a GPIO object for GPIO with kernel number 34 as output low.
GPIO gpio = gpioManager.createGPIO(34, GPIOMode.OUTPUT_LOW);

[...]
```

Read and set the GPIO value

The `GPIO` class offers two methods to read and set the GPIO value:

Method	Description
`getValue()`	Retrieves the value of the GPIO. Only has effect if GPIO mode is `INPUT`, `INTERRUPT_EDGE_RISING`, `INTERRUPT_EDGE_FALLING` or `INTERRUPT_EDGE_BOTH`.
`setValue(GPIOValue)`	Sets the GPIO value. Only has effect if GPIO mode is `OUTPUT_HIGH` or `OUTPUT_LOW`.

The `getValue()` method may fail if there is an error reading the GPIO value, throwing a `GPIOException`.

The `setValue(GPIOValue)` method may fail for the following reasons:

- If there is an error setting the value, throwing a `GPIOException`.
- If the value is null, throwing a `NullPointerException`.

The `GPIOValue` enumeration contains the following elements:

- `HIGH`: the GPIO value is high.
- `LOW`: the GPIO value is low.

Reading and setting the GPIO value

```
import com.digi.android.gpio.GPIO;
import com.digi.android.gpio.GPIOManager;
import com.digi.android.gpio.GPIOMode;
import com.digi.android.gpio.GPIOValue;

[...]

GPIOManager manager = ...;

GPIO inputGPIO = manager.createGPIO(37, GPIOMode.INPUT);
GPIO outputGPIO = manager.createGPIO(34, GPIOMode.OUTPUT_HIGH);

// Read the value of the GPIO configured as input.
GPIOValue value = inputGPIO.getValue();
System.out.println("GPIO value is: " + value.getDescription());

// Set to LOW the GPIO configured as output.
outputGPIO.setValue(GPIOValue.LOW);

[...]
```

Detect GPIO interruptions

When a GPIO is configured as `INTERRUPT_EDGE_RISING`, `INTERRUPT_EDGE_FALLING` or `INTERRUPT_EDGE_BOTH`, you can register a listener and be notified when its value changes. To do so, you have to subscribe a `IGPIOListener` to the `GPIO` object by using the `registerListener(IGPIOListener)` method.

Registering the GPIO listener

```
import com.digi.android.gpio.GPIO;
import com.digi.android.gpio.GPIOManager;
import com.digi.android.gpio.GPIOMode;

[...]

GPIOManager gpioManager = ...;

// Create GPIO object for kernel number 37.
GPIO gpio = gpioManager.createGPIO(37, GPIOMode.INTERRUPT_EDGE_BOTH);

// Create the listener.
MyGPIOListener listener = ...;

// Register the GPIO listener.
gpio.registerListener(listener);

[...]
```

The registered listener class, `MyGPIOListener`, must implement the `IGPIOListener` interface. This interface defines the `valueChanged(GPIOValue)` method, which is called whenever the value of the GPIO changes.

IGPIOListener implementation example, MyGPIOListener

```
import com.digi.android.gpio.IGPIOListener;

public class MyGPIOListener implements IGPIOListener {
    @Override
    public void valueChanged(GPIOValue value) {
        // This code will be executed every time the GPIO value changes.
        System.out.println("New GPIO value: " + value);
    }
}
```

Note that it is possible to have more than one `IGPIOListener` waiting for updates in the same GPIO. If you no longer wish to receive GPIO value updates in a determined listener, use the `unregisterListener(IGPIOListener)` method to unsubscribe an already registered listener.

You can also configure the polling rate, that is, the amount of milliseconds between checks for changes in the GPIO value. By default is configured to 100 milliseconds, but you can change it by using the `setPollingRate(int)` method or read it with the `getPollingRate()` method.

GPIO Example

The **GPIO Sample Application** demonstrates the usage of the GPIO API. In this example, one GPIO is configured as input and another as output. You can press the virtual button to switch on and off the User 0 LED corresponding to the output GPIO.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

GPU management

The GPU service API allows you to manage the GPU frequency using a multiplier factor. In some scenarios the 3D performance of an application is not as critical as the temperature or the energy the module is consuming.

With this API it is possible to scale the frequency that the GPU is working at. The frequency is always divided by 64 and then multiplied by a value that can go from 1 to 64. You can control this multiplier factor, this way you can have a better control on the temperature and power consumption of the module.

Apart from managing the GPU frequency, this API allows you to configure the minimum frequency multiplier that will be applied when module's temperature reaches the hot trip point. The hot trip point is and Android security mechanism that consists on reducing CPU and GPU frequencies when the temperature configured in this trip point is reached. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all GPU API methods require the **`com.digi.android.permission.GPU`** permission.

> If your application does not have the `com.digi.android.permission.GPU` permission it will not have access to any GPU service feature.

In order to manage the GPU frequency multiplier, you need to instantiate a `GPUManager` object. To do so, you have to provide the application `Context`.

Instantiating the GPUManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.system.gpu.GPUManager;

public class GPUSampleActivity extends Activity {

    GPUManager gpuManager;

    [...]

    @Override
 public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceStace);

        // Instantiate the GPU manager object.
        gpuManager = new GPUManager(this);

        [...]
    }

    [...]
}
```

Once the `GPUManager` object is instantiated you will be able to:

- Manage the GPU frequency multiplier
- Change the minimum GPU frequency multiplier

Manage the GPU frequency multiplier

One of the main features of the `GPUManager` is to control the GPU frequency using a multiplier factor that can go from 1 to 64. A value of 64 means that the GPU frequency is configured at its maximum, while 1 means that the frequency is configured at its minimum. The GPU frequency is directly associated to the temperature and power consumption of the module. The lower the frequency is, the lower the temperature and consumption will be.

There are two methods that allow you to get and set the GPU frequency multiplier.

Method	Description
`setMultiplier(int)`	Sets the frequency multiplier. It can go from 1 to 64
`getMultiplier()`	Gets the frequency multiplier

> Notice that in some variants of the module, the minimum frequency multiplier allowed is greater than 1. So, it is convenient to get and check the returned value of the method, which is the actual value that could be set.

These methods may fail for the following reasons:

- The specified multiplier is out of range throwing an `IllegalArgumentException`.
- If there is an error setting or getting the multiplier throwing an `IOException`.

Getting and setting the frequency multiplier

```
import com.digi.android.system.gpu.GPUManager;

[...]

// Get the GPU manager.
GPUManager gpuManager = new GPUManager(context);

// Get the current frequency multiplier.
int currentMultiplier = gpuManager.getMultiplier();

// Set the the frequency multiplier to 48.
gpuManager.setMultiplier(48);

[...]
```

Change the minimum GPU frequency multiplier

When the temperature of the module reaches the hot trip point, Android sets the CPU and GPU frequencies to their minimums. When the temperature is 10 degrees below the hot trip point, Android configures the CPU and GPU frequencies to their previous values.

> The hot trip point temperature value can be get and set using the `getHotTemperature()` and `setHotTemperature(int)` methods of the CPU service.

The minimum frequency multiplier value of the GPU can be configured with the GPU service using these methods:

Method	Description
`setMinMultiplier(int)`	Sets the minimum frequency multiplier. It can go from 1 to 64
`getMinMultiplier()`	Gets the minimum frequency multiplier

> Notice that in some variants of the module, the minimum frequency multiplier allowed is greater than 1. So, it is convenient to get and check the returned value of the method, which is the actual value that could be set.

These methods may fail for the following reasons:

- The specified multiplier is out of range throwing an `IllegalArgumentException`.
- If there is an error setting or getting the multiplier throwing an `IOException`.

Getting and setting the minimum frequency multiplier

```
import com.digi.android.system.gpu.GPUManager;

[...]

// Get the GPU manager.
GPUManager gpuManager = new GPUManager(context);

// Get the minimum frequency multiplier.
int minimumMultiplier = gpuManager.getMinMultiplier();

// Set the the minimum frequency multiplier to 2.
gpuManager.setMinMultiplier(2);

[...]
```

GPU Management Example

The **GPU Management Sample Application** demonstrates the usage of the GPU API. In this example, you can modify the GPU frequency multiplier value with a slice bar and see how it impacts in the GPU performance and temperature of the module.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the <TODO verify link GitHub repository>.

I2C

The **Inter-Integrated Circuit**, I²C or two-wire interface is a multi-master synchronous serial data link standard, invented by Phillips. I²C uses only two bidirectional open-drain lines:

- **SCL:** serial clock (output from master)
- **SDA:** serial data line (bidirectional).

The I²C bus can operate with a single master device and with up to 256 devices, as long as they all have different device addresses. The API provides functions to change the target.
The ConnectCore 6 has several I2C (Inter Integrated Circuits) interfaces to communicate with other I2C devices using this protocol. In the ConnectCore 6 Hardware Reference Manual you can find information about the available I2C interfaces.

Digi adds to Android an API to manage these I2C interfaces. You can configure the slave address, write, and read among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all I2C API methods require the **`com.digi.android.permission.I2C`** permission.

> If your application does not have the `com.digi.android.permission.I2C` permission it will not have access to any I2C service feature.

First of all, a new `I2CManager` object must be instantiated by passing the Android Application Context.

Instantiating the I2CManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.i2c.I2CManager;

public class I2CSampleActivity extends Activity {

 I2CManager i2cManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the I2C manager object.
        i2cManager= new I2CManager(this);

  [...]
 }

 [...]
}
```

Instantiate an I2C Interface

The `I2CManager` allows you to create an `I2C` object representing a physical module's I2C interface. With this object you can write to the interface and read from it. If you don't know how many interfaces are

available in the device you can list them.

Method	Description
`createI2C(int)`	Creates and returns an I2C object with the given interface number
`listInterfaces()`	Lists all available I2C interface numbers in the device

The `createI2C(int)` method may fail if the provided index is 0 or lower than 0 throwing an `IllegalArgumentException`.

Getting I2C interfaces

```
import com.digi.android.i2c.I2C;
import com.digi.android.i2c.I2CManager;

[...]

I2CManager i2cManager = ...;

// Create an I2C object for each available I2C interface.
int[] nInterfaces = i2cManager.listInterfaces();
I2C[] interfaces = new I2C[nInterfaces.length];

for (int i = 0; i < nInterfaces.length; i++)
 interfaces[i] = i2cManager.createI2C(nInterfaces[i]);

[...]
```

Manage the I2C interface

The next step is to open the I2C interface. To do so, you just need to call the `open()` method. Prior to open the interface you can get its interface number and check its current status (open or closed). When you are done with the interface, you must close it. All this management can be performed using the following methods.

Method	Description
`getInterfaceNumber()`	Retrieves the I2C interface number
`isInterfaceOpen()`	Retrieves whether interface is open or not
`open()`	Opens the I2C interface
`close()`	Attempts to close the I2C interface

The `open()` method may fail if the interface number to open does not exists throwing a `NoSuchInterfaceException` or if there is an IO issue accessing the interface throwing an `IOException`.

The `close()` method may fail if there is an issue accessing the interface throwing an `IOException`.

Getting I2C interfaces

```
import com.digi.android.i2c.I2C;
import com.digi.android.i2c.I2CManager;

[...]

I2CManager i2cManager = ...;

I2C i2cInterface = ...;

// Print interface number.
System.out.println("Created I2C interface " +
i2cInterface.getInterfaceNumber());

[...]

// Check if the interface is open, if not, open it.
if (!i2cInterface.isInterfaceOpen())
    i2cInterface.open();

[...]

// Close the I2C interface.
i2cInterface.close();
```

Communicate with the I2C interface

Once the I2C interface has been successfully opened, you can perform the following actions:

- Read data
- Write data
- Transfer data

Read data

One of the most common things you can do with an I2C interface is to read one or more bytes from a specific device slave address. In all the read methods, the first parameter corresponds to this slave address.

Method	Description
`read(int, int)`	Reads the given amount of bytes from the given slave device address
`readByte(int)`	Reads one byte from the given slave device address

All the previous methods may fail for the following reasons:

- The provided slave address lower than 0 throwing an `IllegalArgumentException`.
- The interface is closed or an error occurs while accessing the interface throwing an `IOException`.

In addition, the `readByte(int, int)` method may fail if the number of bytes to read is 0 or lower than 0 throwing an `IllegalArgumentException`.

Reading data from the I2C interface

```
import com.digi.android.i2c.I2C;
import com.digi.android.i2c.I2CManager;

[...]

I2CManager i2cManager = ...;

I2C i2cInterface = ...;

[...]

byte[] readData = new byte[8];

// Read 8 bytes (byte by byte) from address 0.
for (int i = 0; i < readData.length; i++)
    readData[i] = i2cInterface.readByte(0);

// Read 8 bytes from address 0.
readData = i2cInterface.read(0, 8);

[...]
```

Remember that when you are done with the I2C interface you need to close it calling the `close()` method.

Write data

The I2C API allows you also to write data in the slave address of the I2C device. In this case you can write data byte by byte or write an array of bytes.

Method	Description
`write(int, byte)`	Writes the given byte in the given slave device address
`write(int, byte[])`	Writes the given byte array in the given slave device address

All the previous methods may fail for the following reasons:

- The provided slave address is lower than 0 throwing an `IllegalArgumentException`.
- The interface is closed or an error occurs while accessing the interface throwing an `IOException`.

In addition, the `write(int, byte[])` method may fail if the data to write is `null` throwing a `NullPointerException`.

Writing data to the I2C interface

```
import com.digi.android.i2c.I2C;
import com.digi.android.i2c.I2CManager;
[...]

I2CManager i2cManager = ...;

I2C i2cInterface = ...;

byte[] data = "abcdefgh";

[...]

// Write bytes contained in data (byte by byte).
for (int i = 0; i < data.length; i++)
    i2cInterface.write(0, data[i]);

// Write bytes contained in data.
i2cInterface.write(0, data);

[...]
```

Remember that when you are done with the I2C interface you need to close it calling the `close()` method.

Transfer data

Finally, the I2C interface gives you the possibility to read and write data simultaneously in one operation. Like in previous methods, the first parameter corresponds to the slave address of the I2C device.

Method	Description
`transfer(int, byte[], int)`	Simultaneously writes the given byte array in the given slave device address and reads the given amount of bytes
`transfer(int, byte, int)`	Simultaneously writes the given byte in the given slave device address and reads the given amount of bytes

All these methods may fail for the following reasons:

- The provided slave address is lower than 0 or if the number of bytes to transfer is lower than 1 throwing an `IllegalArgumentException`.
- The interface is closed or an error occurs while accessing the interface throwing an `IOException`.

In addition, the `transfer(int, byte[], int)` method may fail if the data to transfer is `null` throwing a `NullPointerException`.

Transferring data to the I2C interface

```
import com.digi.android.i2c.I2C;
import com.digi.android.i2c.I2CManager;
[...]

I2CManager i2cManager = ...;

I2C i2cInterface = ...;

byte[] data = "abcdefgh";
byte[] readData;

[...]

// Write bytes contained in data and read the same number of bytes (8)
in the slave address 0.
readData = i2cInterface.transfer(0, data, data.length);

// Write one byte ('a') and read 8 bytes in the slave address 0.
readData = i2cInterface.transfer(0, (byte)'a', 8);

[...]
```

Remember that when you are done with the I2C interface you need to close it calling the `close()` method.

I2C Example

The **I2C Sample Application** demonstrates the usage of the I2C API. In this example you can access and control an external I2C EEPROM memory. Application can perform read, write and erase actions displaying results in an hexadecimal list view.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Memory

The random access memory, RAM, is a type of computer memory that can be accessed randomly; that is, any byte of memory can be accessed without touching the preceding bytes. It is the most common type of memory found in computers, smartphones and embedded devices.

Digi adds to Android an API to obtain different memory values of the ConnectCore 6 modules. You can get the total, free, cached and available memory values. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all GPU API methods require the **`com.digi.android.permission.MEMORY`** permission.

> If your application does not have the `com.digi.android.permission.MEMORY` permission it will not have access to any GPU service feature.

In order to obtain the memory values, you need to instantiate a `MemoryManager` object. To do so, you have to provide the application `Context`.

Instantiating the MemoryManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.system.memory.MemoryManager;

public class MemorySampleActivity extends Activity {

    MemoryManager memoryManager;

    [...]

    @Override
 public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceStace);

        // Instantiate the Memory manager object.
        memoryManager = new MemoryManager(this);

        [...]
    }

    [...]
}
```

Obtain the memory values

You can read the total, free, cached and available memory values using the following methods:

Method	Description
`getTotalMemory()`	Returns the device's total usable memory in kB (i.e. physical RAM minus a few reserved bits and the kernel binary code)
`getFreeMemory()`	Returns the device's free memory in kB

`getCachedMemory()`	Returns the device's in-memory cache for files read from the disk in kB
`getAvailableMemory()`	Returns the device's available memory in kB. An estimate of how much memory is available for starting new applications, without swapping.

These methods may fail if there is any error while reading the memory values throwing an `IOException`.

Getting the memory values

```
import com.digi.android.system.memory.MemoryManager;

[...]

MemoryManager memoryManager = ...;

[...]

System.out.println("Total memory: " + memoryManager.getTotalMemory() +
"kB");
System.out.println("Free memory: " + memoryManager.getFreeMemory() +
"kB");
System.out.println("Cached memory: " + memoryManager.getCachedMemory() +
"kB");
System.out.println("Available memory: " +
memoryManager.getAvailableMemory() + "kB");

[...]
```

Memory Example

This API has not a separate example. Instead, it is integrated in the **CPU Management Sample Application**, which demonstrates the usage of the CPU and memory APIs.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the <TODO verify link GitHub repository>.

PWM

A **PWM** (**Pulse-Width Modulator**) is a component used for controlling power to inertial electrical devices. It generates a periodic waveform with positive width which can be controlled and thus, the waveform's average value modified.

The average value of voltage fed to the load is controlled by turning the switch between the supply and the load on and off at a fast pace. The longer the switch is on compared to the off, the higher the power supplied to the load will be.
The ConnectCore 6 has several PWM (Pulse Width Modulation) interfaces to manage devices such as servos and voltage regulators. In the ConnectCore 6 Hardware Reference Manual you can find information about the available PWM channels.

Digi adds to Android an API to manage these PWM channels. You can configure the PWM duty cycle, the frequency, and the polarity among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all PWM API methods require the **`com.digi.android.permission.PWM`** permission.

> If your application does not have the `com.digi.android.permission.PWM` permission it will not have access to any PWM service feature.

First of all, a new `PWMManager` object must be instantiated by passing the Android Application Context.

Instantiating the PWMManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.pwm.PWMManager;

public class PWMSampleActivity extends Activity {

 PWMManager pwmManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the PWM manager object.
        pwmManager= new PWMManager(this);

  [...]
 }

 [...]
}
```

Instantiate a PWM channel

The `PWMManager` allows you to create a `PWM` object representing a physical module's PWM channel. If you don't know how many channels are available in the device you can list them.

Method	Description
`createPWM(int)`	Creates and returns a PWM object with the given interface number
`listChannels()`	Lists all available PWM channels in the device

The `createPWM(int)` method may fail if the provided channel is not available throwing an `PWMException`.

Getting PWM channels

```
import com.digi.android.pwm.PWM;
import com.digi.android.pwm.PWMManager;

[...]

PWMManager pwmManager = ...;

// Create a PWM object for each available PWM channel.
int[] channels = pwmManager.listChannels();
PWM[] pwms = new PWM[channels.length];

for (int i = 0; i < channels.length; i++)
 pwms[i] = pwmManager.createPWM(channels[i]);

[...]
```

Manage PWM duty cycle

You can get and set a PWM duty cycle using the following methods:

Method	Description
`getDutyCycle()`	Returns the duty cycle percentage of the PWM signal (0 to 100%)
`setDutyCycle(double)`	Changes the duty cycle percentage of the PWM signal

These methods may fail if there is any error while reading or configuring the PWM duty cycle throwing a `PWMException`.

Getting and setting the PWM duty cycle

```
import com.digi.android.pwm.PWM;
import com.digi.android.pwm.PWMManager;

[...]

PWMManager pwmManager = ...;
PWM pwm = ...;

[...]

System.out.println("PWM channel: " + pwm.getChannelIndex());
System.out.println("Duty cycle: " + pwm.getDutyCycle());

pwm.setDutyCycle(42);
System.out.println("New duty cycle: " + pwm.getDutyCycle());

[...]
```

PWM Example

The **PWM Sample Application** demonstrates the usage of the PWM API. In this example you can list all the available PWM channels and set the duty cycle of that signal.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Serial port

A **Serial Communications Interface** or Universal Asynchronous Receiver-Transmitter (UART) is a serial communications peripheral that implements the asynchronous serial communications protocol.

The serial communications interface can be used to communicate with several devices such as displays, sensors, data acquisition systems, etc.

The ConnectCore 6 SBC provides access to three UART interfaces on UART expansion connector. This connector provides access to the following interfaces:

- UART1: 4 wire, RS232 level UART
- UART3: 4 wire, RS232 level UART
- UART5: 4 wire, TTL UART shared with XBee interface

These three UART interfaces have software flow control lines (RTS and CTS). UART1 and UART3 have RS232 levels and they are configured in DTE mode (CTS input and RTS output). The UART5 interface has TTL levels and it is configured in DCE mode (CTS output and RTS input).
In the ConnectCore 6 Hardware Reference Manual you can find information about the available Serial Ports.

Digi adds to Android an API to manage these Serial Ports interfaces. You can configure the connection, send, and receive data among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all serial API methods require the **`com.digi.android.permission.SERIAL`** permission.

> If your application does not have the `com.digi.android.permission.SERIAL` permission it will not have access to any serial port service feature.

First of all, a new `SerialPortManager` object must be instantiated by passing the Android Application Context.

Instantiating the SerialPortManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.serial.SerialPortManager;

public class SerialPortSampleActivity extends Activity {

 SerialPortManager serialPortManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the Serial Port manager object.
        serialPortManager= new SerialPortManager(this);

  [...]
 }

 [...]
}
```

The Serial Port API allows you to:

- Open/close a serial port
- Configure a serial port
- Monitor serial port events
- Communicate with serial devices
- Manage serial port lines

Serial Port Example

The **Serial Port Sample Application** demonstrates the usage of the Serial Port API. In this example you can list all the available Serial ports, configure them, send, and receive data.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Open/close a serial port

The `SerialPortManager` allows you to create a `SerialPort` object for a specific port name. The `listSerialPorts()` method lists all the available serial ports in the device.

You can also specify other serial port parameters using a `SerialPortParameters` object or directly in the `openSerialPort` method.

Method	Description
`openSerialPort(String)`	Creates and opens a `SerialPort` object for the provided port name
`openSerialPort(String, SerialPortParameters)`	Creates and opens a `SerialPort` object for the provided port name with the given configuration
`openSerialPort(String, int, int, int, int, int, int)`	Creates and opens a `SerialPort` object for the provided port name with the given port parameters

When they are not specified, the serial port parameters default values are the following:

- Baud rate: 9600
- Data bits: 8
- Stop bits: 1
- Parity: None
- Flow control: None
- Read timeout: 3 seconds

> The `SerialPort` class includes constant definitions for the possible values of data bits, stop bits, parity, and flow control. For example: `SerialPort.DATABITS_8`, `SerialPort.STOPBITS_1`, `SerialPort.PARITY_NONE`, `SerialPort.FLOWCONTROL_NONE`.

The three methods that open the serial port may fail for the following reasons:

- If the serial port name is invalid or does not exist, the `openSerialPort()` method throws a `NoSuchPortException`.
- If the serial port is already in use by other application(s), the `openSerialPort()` method throws a `PortInUseException`.

Opening serial ports

```
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortManager;
import com.digi.android.serial.SerialPortParameters;

[...]

SerialPortManager serialPortManager = ...;

// Get the list of available serial ports.
String[] portNames = serialPortManager.listSerialPorts();

// Define a serial configuration: 115200/8/N/1 and hardware flow
control.
SerialPortParameters params = new SerialPortParameters(
  9600,                  /* baudrate:     9600 */
  SerialPort.DATABITS_8,              /* data bits:    8 */
  SerialPort.STOPBITS_1,              /* stop bits:    1 */
  SerialPort.PARITY_NONE,              /* parity:       none */
  SerialPort.FLOWCONTROL_RTSCTS_IN | SerialPort.FLOWCONTROL_RTSCTS_OUT,
/* flow ctrl:    hardware */
  2000                 /* read timeout: 2s */);

// Open the available serial ports with the previous configuration.
SerialPorts[] ports = new SerialPorts[portNames.length];
for (int i = 0; i < portNames.length; i++)
 ports[i] = serialPortManager.openSerialPort(portNames[i], params);

[...]
```

Once you have finished with a serial port, you must close it. This frees the port so that other applications can use it.

To close a serial port, use the `close()` method of the `SerialPort` object.

Closing serial ports

```
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortManager;

[...]

SerialPortManager serialPortManager = ...;
SerialPort port = ...;

[...]

// Close the serial port.
port.close();

[...]
```

Configure a serial port

Before sending and receiving data, the serial port must be configured. As explained in the Open/Close a serial port topic, when opening a port you can set the configuration, but the `SerialPort` class also offers methods to get their values and to change them once the port is opened:

Method	Description
`getName()`	Retrieves the serial port name
`getBaudRate()`	Retrieves the configured baud rate
`getDataBits()`	Retrieves the configured number of data bits
`getStopBits()`	Retrieves the configured number of stop bits
`getParity()`	Retrieves the configured parity
`getFlowControl()`	Retrieves the configured flow control
`setPortParameters(int, int, int, int, int)`	Configures the serial port with the given values for baud rate, number of data bits, number of stop bits, parity, and flow control

Configuring a serial port

```
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortManager;

[...]

SerialPortManager serialPortManager = ...;
SerialPort port = ...;

[...]

// Read old configuration.
System.out.println("Port: " + port.getName() + "\n"
     + "   Baud rate: " + port.getBaudRate() + "\n"
     + "   Data bits: " + port.getDataBits() + "\n"
     + "   Stop bits: " + port.getStopBits() + "\n"
     + "   Parity: " + port.getParity() + "\n"
     + "   Flow control: " + port.getFlowControl());

// Set a new configuration: 38400/8/E/1, no flow control.
port.setPortParameters(38400, SerialPort.DATABITS_8,
SerialPort.STOPBITS_1,
          SerialPort.PARITY_EVEN, SerialPort.FLOWCONTROL_NONE);

[...]
```

Remember that when you are done with the serial port you need to close it by calling the `close()` method.

Monitor serial port events

You can monitor different serial port events, such as data available, lines state changes, or errors during the communication. To do so follow the next steps:

1. Subscribe for serial port notifications
2. Enable notifications
3. Unsubscribe for serial ports notifications

1. Subscribe for serial port notifications

Use the `registerEventListener(ISerialPortEventListener)` method of the `SerialPort` object to subscribe for serial port notifications.

Registering for serial port events notifications

```
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;

// Create the serial port listener.
MySerialPortListener mySerialPortListener = ...;

// Register the serial port listener.
port.registerEventListener(mySerialPortListener);

[...]
```

The subscribed listener class, `MySerialPortListener`, must implement the `ISerialPortEventListener` interface. This interface defines the `serialEvent(SerialPortEvent)` method that is called every time a new serial port event occurs.

The received event is represented by the `SerialPortEvent` object. To distinguish between the different notifications use the `getEventType()` method to get the `EventType` and filter.

ISerialPortEventListener implementation example, MySerialPortListener

```
import com.digi.android.serial.ISerialPortEventListener;
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortEvent;

public class MySerialPortListener implements ISerialPortEventListener {
 @Override
 public void serialEvent(SerialPortEvent event) {
  switch (event.getEventType()) {
   case BI:
    System.out.println("Break interrupt received");
    break;
   case CD:
    System.out.println("Carrier detect received");
    break;
   case CTS:
    System.out.println("CTS line activated");
    break;
   case DSR:
    System.out.println("DSR line activated");
    break;
   case RI:
    System.out.println("Ring Indicator received");
    break;
   case FE:
    System.out.println("Received framing error");
    break;
   case PE:
    System.out.println("Received parity error");
    break;
   case OE:
    System.out.println("Buffer overrun error");
    break;
   case DATA_AVAILABLE:
    System.out.println("Data to read available");
    break;
   default:
    System.out.println("Unknown event");
    break;
  }
 }
}
```

Only one listener can be subscribed per `SerialPort` object, if you try to register more a `TooManyListenerException` will be thrown.

2. Enable notifications

Once the listener is implemented, it can be used to listen to particular serial port events.

By default, none of the previous events is notified to the listener. To do so, the reception of each `SerialPort` event type needs to be requested individually. Use the following methods:

Method	Description

`notifyOnBreakInterrupt(boolean)`	Enables/disables Break interrupt notifications
`notifyOnCarrierDetect(boolean)`	Enables/disables Carrier Detect notifications
`notifyOnCTS(boolean)`	Enables/disables CTS notifications
`notifyOnDataAvailable(boolean)`	Enables/disables data to read is available
`notifyOnDSR(boolean)`	Enables/disables DSR notifications
`notifyOnFramingError(boolean)`	Enables/disables framing error notifications
`notifyOnOutputEmpty(boolean)`	Enables/disables output buffer empty notifications
`notifyOnOverrunError(boolean)`	Enables/disables overrun error notifications
`notifyOnParityError(boolean)`	Enables/disables parity error notifications
`notifyOnRingIndicator(boolean)`	Enables/disables Ring Indicator notifications

Enabling serial port notifications

```
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;
MySerialPortListener mySerialPortListener = ...;

port.registerEventListener(mySerialPortListener);

// Enable notifications.
port.notifyOnDataAvailable(true);
port.notifyOnCTS(true);
port.notifyOnDSR(true);
port.notifyOnRingIndicator(true);
port.notifyOnBreakInterrupt(true);
port.notifyOnCarrierDetect(true);
port.notifyOnFramingError(true);
port.notifyOnOverrunError(true);
port.notifyOnParityError(true);

[...]
```

3. Unsubscribe for serial ports notifications

If you no longer wish to be notified about any serial port event, use the `unregisterEventListener()` method to unsubscribe the registered listener.

Unregistering serial port events notifications

```
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;
MySerialPortListener mySerialPortListener = ...;

port.registerEventListener(mySerialPortListener);

[...]

// Remove the serial port event listener.
port.unregisterEventListener();

[...]
```

Remember that when you are done with the serial port you need to:

1. Unsubscribe your registered listener (if any) with `unregisterEventListener()` method.
2. Then close the port by calling the `close()` method.

Communicate with serial devices

When the serial port is open you can communicate with the serial device connected to it, transmitting and receiving data:

- Send data
- Receive data

Send data

You can send data through the serial port. To do so, you need to get the serial port's output stream. From the `OutputStream` object invoke one of the existing `write()` methods.

Sending serial data

```
import java.io.OutputStream;
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;

String dataToSend = "This is the data to send";

// Get the port output stream.
OutputStream outStream = port.getOutputStream();

// Send data through the serial port.
outStream.write(dataToSend.getBytes());

[...]
```

Remember that when you are done with the serial port you need to:

1. Unsubscribe your registered listener (if any) with `unregisterEventListener()` method.
2. Then close the port by calling the `close()` method.

Receive data

You can receive data from the serial port by getting its input stream. From the `InputStream` object invoke one of the existing `read()` methods.

Receiving serial data

```
import java.io.InputStream;
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;

[...]

private void readData() {
 // Get the port input stream.
 InputStream inStream = port.getInputStream();

 int availableBytes = inStream.available();
 if (availableBytes > 0) {
  byte[] readBuffer = new byte[availableBytes];

  // Read the serial port.
  int numBytes = inStream.read(readBuffer, 0, availableBytes);

  if (numBytes > 0)
   System.out.println("Read: " + new String(readBuffer, 0,
availableBytes));
 }
}
[...]
```

Use the data available serial port event (`EventType.DATA_AVAILABLE`) of your registered `ISerialPortEventListener` to know when to read data:

ISerialPortEventListener implementation example, MySerialPortListener

```
import com.digi.android.serial.ISerialPortEventListener;
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortEvent;

public class MySerialPortListener implements ISerialPortEventListener {
 @Override
 public void serialEvent(SerialPortEvent event) {
  switch (event.getEventType()) {
   case DATA_AVAILABLE:
    readData();
    break;
  }
 }
}
```

Do not forget to subscribe your listener to receive data available events and to enable this notification on your `SerialPort` object.

Registering for serial port events notifications

```
import com.digi.android.serial.SerialPort;

[...]

SerialPort port = ...;

// Create the serial port listener.
MySerialPortListener mySerialPortListener = ...;

// Register the serial port listener.
port.registerEventListener(mySerialPortListener);

// Enable data available notifications.
port.notifyOnDataAvailable(true);

[...]
```

Remember that when you are done with the serial port you need to:

1. Unsubscribe your registered listener (if any) with `unregisterEventListener()` method.
2. Then close the port by calling the `close()` method.

Manage serial port lines

The `SerialPort` class also offers different methods to get the current state of the port lines and set some of them:

Method	Description
`isCD()`	Gets the state of the CD (Carrier Detect) line
`isCTS()`	Gets the state of the CTS (Clear To Send) line
`isDSR()`	Gets the state of the DSR (Data Set Ready) line
`isDTR()`	Gets the state of the DTR (Data Terminal Ready) line
`setDTR(boolean)`	Sets or clears the DTR (Data Terminal Ready) line
`isRI()`	Gets the state of the RI (Ring Indicator) line
`isRTS()`	Gets the state of the RTS (Request To Send) line
`setRTS(boolean)`	Sets or clears the RTS (Request To Send) line
`sendBreak(int)`	Sends a break signal of the specified milliseconds duration

Managing serial port lines

```
import com.digi.android.serial.SerialPort;
import com.digi.android.serial.SerialPortManager;

[...]

SerialPortManager serialPortManager = ...;
SerialPort port = ...;

[...]

// Read lines state.
System.out.println("CD: " + port.isCD());
System.out.println("CTS: " + port.isCTS());
System.out.println("DSR: " + port.isDSR());
System.out.println("DTR: " + port.isDTR());
System.out.println("RI: " + port.isRI());
System.out.println("RTS: " + port.isRTS());

// Set lines state.
port.setDTR(true);
port.setRTS(true);

// Send a break signal of 1 second.
port.sendBreak(1000);

[...]
```

Remember that when you are done with the serial port you need to:

1. Unsubscribe your registered listener (if any) with `unregisterEventListener()` method.
2. Then close the port by calling the `close()` method.

SPI

The **Serial Peripheral Interface Bus** (**SPI**) is a synchronous serial data link standard, named by Motorola, which operates in full duplex mode. Devices communicate in master/slave mode, where the master device initiates the data frame.

The SPI bus can operate with a single master device and with one or more slave devices. The ConnectCore 6 device is always the SPI master of an SPI bus and, by default, only can have 1 SPI slave connected.
The ConnectCore 6 has several SPI interfaces to communicate with other SPI devices using this protocol. In the ConnectCore 6 Hardware Reference Manual you can find information about the available SPI interfaces. Digi adds to Android an API to manage these SPI interfaces.

Digi adds to Android an API to manage these SPI interfaces. You can configure the SS (Slave Select) behaviour, write, transfer, and read among other things. In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all SPI API methods require the **`com.digi.android.permission.SPI`** permission.

> If your application does not have the `com.digi.android.permission.SPI` permission it will not have access to any SPI service feature.

First of all, a new `SPIManager` object must be instantiated by passing the Android Application Context.

Instantiating the SPIManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.spi.SPIManager;

public class SPISampleActivity extends Activity {

 SPIManager spiManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the SPI manager object.
        spiManager= new SPIManager(this);

  [...]
 }

 [...]
}
```

Instantiate an SPI Interface

The `SPIManager` allows you to create an `SPI` interface object to communicate with a slave device

connected to a physical interface. For that purpose you need to provide the SPI interface number and SPI slave device ID number as parameters. You can use the `SPIManager` to list all the SPI interfaces of your ConnectCore 6 as well as the slave device IDs connected to each SPI interface.

Method	Description
`createSPI(int, int)`	Creates and returns an `SPI` object with the given interface and slave device
`listInterfaces()`	Lists all the available SPI interface numbers
`listSlaveDevices(int)`	Lists all the available slave devices of the given SPI interface number

The `createSPI(int, int)` method may fail if the interface number or slave device ID are 0 or lower than 0, throwing an `IllegalArgumentException`.

Getting SPI interfaces

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIManager;

import java.util.ArrayList;

[...]

SPIManager spiManager = ...;

// Create an SPI object for each available SPI interface and connected
slave.
int[] nInterfaces = spiManager.listInterfaces();
ArrayList<SPI> interfaces = new ArrayList<SPI>();

for (int i = 0; i < nInterfaces.length; i++) {
    int spiInterface = nInterfaces[i];
    int[] nSlaves = spiManager.listSlaveDevices(spiInterface);
    for (int j = 0; j < nSlaves.length; j++)
        interfaces.add(spiManager.createSPI(spiInterface, nSlaves[j]));
}

[...]
```

Manage the SPI interface

The next step is to open the SPI interface. To do so, you just need to call the `open(SPIConfig)` method. Prior to open the interface you can get its interface and slave number and check its current status (open or closed). When you are done with the interface, you must close it. All this management can be performed using the following methods:

Method	Description
`getInterface()`	Gets the SPI interface number
`getSlaveDevice()`	Gets the SPI slave device number
`isInterfaceOpen()`	Gets the status of the SPI interface

`open(SPIConfig)`	Opens the SPI interface with the given configuration. Refer to the Configure an SPI interface section for more information about the SPI configuration.
`close()`	Attempts to close the SPI interface

The `open(SPIConfig)` method may fail for the following reasons:

- There is an error opening the the SPI interface throwing an `IOException`.
- The configured SPI interface does not exist throwing a `NoSuchInterfaceException`.
- The provided SPI configuration object is `null` throwing a `NullPointerException`.

The `close()` method may fail if there is an error closing the SPI interface throwing an `IOException`.

Managing the SPI interface

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

// Print interface and slave number.
System.out.println("Created SPI interface " +
spiInterface.getInterface() + " at slave " +
spiInterface.getSlaveDevice());

[...]

// Check if the interface is open, if not, open it.
if (!spiInterface.isInterfaceOpen()) {
    SPIConfig spiConfig = ...;
    spiInterface.open(spiConfig);
}

[...]

// Close the SPI interface.
spiInterface.close();
```

Configure an SPI interface

The `open(SPIConfig)` method of an SPI interface requires an object that represents the configuration of the SPI interface as parameter. This object is an instance of the `SPIConfig` class. The `SPIConfig` object can be instantiated providing the SPI clock mode, bit order, chip select, clock frequency and word length. If the chip select and bit order parameters are not provided, the constructor will use the `SPIChipSelect.ACTIVE_LOW` and `SPIBitOrder.MSB_FIRST` default values.

Creating an SPIConfig object

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIBitOrder;
import com.digi.android.spi.SPIChipSelect;
import com.digi.android.spi.SPIClockMode;
import com.digi.android.spi.SPIConfig;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

[...]

// Create the SPIConfig object with the following configuration:
//  - Clock mode: CPOL_1_CPHA_0
//  - Chip select: ACTIVE_LOW
//  - Bit order: MSB_FIRST
//  - Clock frequency: 50 KHz
//  - Word length: 8 bits per word
SPIConfig spiConfig = new SPIConfig(SPIClockMode.CPOL_1_CPHA_0,
SPIChipSelect.ACTIVE_LOW, SPIBitOrder.MSB_FIRST, 50000, 8);

// Open the interface with the configuration specified in spiConfig.
spiInterface.open(spiConfig);
```

The values used for the clock mode, chip select and bit order are constants that can be found in the SPIClockMode, SPIBitOrder and SPIChipSelect enumerator classes.

Once the `SPIConfig` object has been declared, you can take the values of the different customization parameters with their corresponding get methods.

Method	Description
`getBitOrder()`	Gets the configured bit ordering
`getChipSelect()`	Gets the configured chip select active level
`getClockFrequency()`	Gets the clock frequency in Hz
`getClockMode()`	Gets the configured clock mode
`getWordLength()`	Gets the word length

Getting SPIConfig parameters

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIConfig;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

SPIConfig spiConfig = ...;
spiInterface.open(spiConfig);

// Print configuration values.
System.out.println("Open SPI interface with the following
configuration:");
System.out.println(" - Clock mode: " + spiConfig.getClockMode().name());
System.out.println(" - Chip select: " +
spiConfig.getChipSelect().name());
System.out.println(" - Bit order: " + spiConfig.getBitOrder().name());
System.out.println(" - Clock frequency: " +
spiConfig.getClockFrequency() + " Hz");
System.out.println(" - Word length: " + spiConfig.getWordLength() + "
bits per word");
```

SPIClockMode

This is the list of the available SPI clock modes to be used when declaring the `SPIConfig` object.

Value	Description
`CPOL_0_CPHA_0`	Data captured on the clock's rising edge and propagated on a falling edge
`CPOL_0_CPHA_1`	Data captured on the clock's falling edge and propagated on a rising edge
`CPOL_1_CPHA_0`	Data captured on clock's falling edge and propagated on a rising edge
`CPOL_1_CPHA_1`	Data captured on clock's rising edge and propagated on a falling edge

SPIBitOrder

This is the list of the available SPI bit order values to be used when declaring the `SPIConfig` object.

Value	Description
`MSB_FIRST`	The most significant bit is transferred in first place
`LSB_FIRST`	The less significant bit is transferred in first place

SPIChipSelect

This is the list of the available SPI chip select configurations to be used when declaring the `SPIConfig` obj ect.

Value	Description
`ACTIVE_LOW`	Chip select line active low
`ACTIVE_HIGH`	Chip select line active high
`NOT_CONTROLLED`	Chip select not controlled by driver

Communicate with the SPI interface

When the `SPI` object connection is open you can communicate with the slave device corresponding to that object using the following actions:

- Read data
- Write data
- Transfer data

Read data

To read data from an SPI interface you only need to use the read(int) method providing the number of bytes to read as parameter.

Method	Description
`read(int)`	Reads the specified number of bytes from the SPI slave device

The `read(int)` method may fail for the following reasons:

- The number of bytes to read is lower than 0 throwing an `IllegalArgumentException`.
- The SPI interface is closed or there is an error reading from the SPI interface throwing an `IOException`.

Reading data from the SPI interface

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

[...]

// Read 8 bytes of data.
byte[] readData = spiInterface.read(8);

[...]
```

> Remember that when you are done with the SPI interface you need to close it calling the `close()` method.

Write data

The SPI API allows you to write data to the SPI slave device associated to the SPI interface object. For that

purpose you need to call the `write(byte[])` method providing the array of bytes to write as parameter.

Method	Description
`write(byte[])`	Writes the given bytes in the SPI slave device

The `write(byte[])` method may fail for the following reasons:

- The SPI interface is closed or there is an error writing in the SPI slave device throwing an `IOException`.
- The array of bytes to write in the SPI interface is `null` throwing a `NullPointerException`.

Writing data to the SPI interface

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

[...]

// Write "Hello SPI".
spiInterface.write("Hello SPI".getBytes());

[...]
```

Remember that when you are done with the SPI interface you need to close it calling the `close()` method.

Transfer data

The third communication method provided by the SPI interface object allows you to read and write data simultaneously in one operation. There are 2 methods for such operation, both of them behave the same way, but in one method you can also specify the clock frequency and word length. If they are not specified, the operation will use the values of the `SPIConfig` object provided in the `open(SPIConfig)` method.

Method	Description
`transfer(byte[])`	Simultaneous write (of the given bytes) and read (of the same number of bytes) using the default clock frequency and word length parameters
`transfer(byte[], int, int)`	Simultaneous write (of the given bytes) and read (of the same number of bytes) using the given clock frequency and word length parameters (these parameters are only used for this transfer, but their default values remain the same)

Previous methods may fail for the following reasons:

- The SPI interface is closed or there is an error transferring data throwing an `IOException`.
- The byte array of data to transfer is `null` throwing a `NullPointerException`.

In addition, the `transfer(byte[], int, int)` method may fail if the clock frequency or word length provided are lower than 1 throwing an `IllegalArgumentException`.

Transferring data to the SPI interface

```
import com.digi.android.spi.SPI;
import com.digi.android.spi.SPIManager;

[...]

SPIManager spiManager = ...;

SPI spiInterface = ...;

[...]

// Transfer "Hello SPI" reading the same amount of bytes (9) in the same
operation.
byte[] readData = spiInterface.transfer("Hello SPI".getBytes());

// Transfer "Hello SPI" reading the same amount of bytes (9) and
configure the clock
// frequency with 500KHz and word length with 8bits per word.
byte[] readData = spiInterface.transfer("Hello SPI".getBytes(), 500000,
8);

[...]
```

Remember that when you are done with the SPI interface you need to close it calling the `close()` method.

SPI Example

The **SPI Sample Application** demonstrates the usage of the SPI API by monitoring the communication with a slave device. The application allows reading, writing and transferring data to the slave device..

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Watchdog

A **watchdog** is a timer that must be reset periodically to indicate that the system is working properly. If it is not reset due to a hardware fault or program error, it will generate a reboot signal to restart the device and bring the system to a known operation mode.

The ConnectCore 6 provides a hardware watchdog interface used to restart the module when there is a system or application malfunction.

Digi extends the Android APIs with a **System Watchdog Service** that is not available in the common Android distributions and provides an API to manage the system watchdog. In addition to this, Digi includes a new software watchdog interface called **Application Watchdog Service** that is used to restart applications when malfunction is reported instead of restarting the entire system. The system watchdog is common to all applications running in the device, as it relies in the hardware layer, while the application watchdog is specific for each application.

In the Javadoc documentation you can find a complete list of the available methods in this API.

Unless noted, all Watchdog API methods (system or application watchdogs) require the **`com.digi.android.permission.WATCHDOG`** permission.

> If your application does not have the `com.digi.android.permission.WATCHDOG` permission it will not have access to any Watchdog service feature.

The Watchdog Service includes:

- System watchdog
- Application watchdog

Watchdog Example
The **Watchdog Sample Application** demonstrates the usage of the Watchdog API. The sample allows you to interact with the watchdog service by registering the application either to the system or the application watchdog services and report application failure at any time. You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

System watchdog

The system watchdog service allows applications to initialize and refresh the system watchdog (Android's system watchdog is not initialized by default). If the system watchdog is not refreshed within the configured timeout, the system will reboot.

First of all, a new `SystemWatchdogManager` object must be instantiated by passing the Android Application Context.

Instantiating the SystemWatchdogManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.watchdog.SystemWatchdogManager;

public class SystemWatchdogSampleActivity extends Activity {

 SystemWatchdogManager systemWatchdogManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the system watchdog manager object.
        systemWatchdogManager = new SystemWatchdogManager(this);

  [...]
 }

 [...]
}
```

Initialize the system watchdog

Once the system watchdog manager is instantiated, the next step is to initialize the system watchdog. For this operation, a timeout parameter is required. This timeout establishes the maximum time that must elapse without refreshing the system watchdog before restarting the device.

Method	Description
`init(long)`	Initializes the system watchdog service with the given timeout (in milliseconds)
`isRunning()`	Returns whether the system watchdog service has been initialized or not
`getTimeout()`	Returns the configured system watchdog service timeout

The `init(long)` method may fail if the provided timeout is lower than `500 milliseconds` throwing an `IllegalArgumentException.` If the system watchdog has been already initialized before calling this method, a `UnsupportedOperationException` will be thrown.

The `getTimeout()` method may fail with a `UnsupportedOperationException` if the system watchdog service has not been initialized yet.

Initializing the system watchdog

```
import com.digi.android.watchdog.SystemWatchdogManager;

[...]

SystemWatchdogManager systemWatchdogManager = ...;

// Check if system watchdog is already running. If not, initialize it to
5 seconds.
if (!systemWatchdogManager.isRunning())
 systemWatchdogManager.init(5000);

// Print configured timeout.
System.out.println("Configured System Watchdog timeout:
" + systemWatchdogManager.getTimeout();

[...]
```

Once initialized, the system watchdog service cannot be stopped and must be refreshed periodically to avoid a system reset. The only way to stop the system watchdog service is to reboot the device.

Refresh the system watchdog

The last step after initializing the system watchdog is to periodically refresh it to avoid a system reset. The `refresh()` method is used for this purpose:

Refreshing the system watchdog

```
import com.digi.android.watchdog.SystemWatchdogManager;

[...]

SystemWatchdogManager systemWatchdogManager = ...;

systemWatchdogManager.init(5000);

Thread refreshThread = new Thread() {
 @Override
 public void run() {
  systemWatchdogManager.refresh();
  // Sleep no more time than the configured timeout.
  Thread.sleep(4000);
 }
};
refreshThread.start();

[...]
```

If the system watchdog was not initialized before calling the `refesh()` method, an `UnsupportedOperationException` will be thrown.

Ensure that your application refreshes the system watchdog periodically with a period lower than the configured timeout.

Watchdog Example

The **Watchdog Sample Application** demonstrates the usage of the Watchdog API. The sample allows you to interact with the watchdog service by registering the application either to the system or the application watchdog services and report application failure at any time.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

Application watchdog

The application watchdog service allows you to initialize the application watchdog for your application in order to shut it down in case of failure. It also allows to execute a pending intent after application is terminated due to failure.

First of all, a new `ApplicationWatchdogManager` object must be instantiated by passing the Android Application Context.

Instantiating the ApplicationWatchdogManager

```
import android.app.Activity;
import android.os.Bundle;

import com.digi.android.watchdog.ApplicationWatchdogManager;

public class ApplicationWatchdogSampleActivity extends Activity {

 ApplicationWatchdogManager applicationWatchdogManager;

 [...]

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Instantiate the application watchdog manager object.
        applicationWatchdogManager = new
ApplicationWatchdogManager(this);

  [...]
 }

 [...]
}
```

Initialize the application watchdog

Once the application watchdog manager is instantiated, the next step is to initialize application watchdog for your application. For this operation, a timeout parameter is required and optionally a `PendingIntent`. The timeout establishes the maximum time that must elapse without refreshing the application watchdog before shutting down the application. The pending intent allows you to execute an action after application is shut down due to failure, for example an application restart intent.

Method	Description
`init(long, PendingIntent)`	Initializes the application watchdog service for the calling application with the given timeout (in milliseconds) and an optional pending intent (can be `null`)
`isRunning()`	Returns whether the application watchdog service has been initialized for the calling application or not

`getTimeout()`	Returns the configured application watchdog service timeout for the calling application
`stop()`	Stops the application watchdog service for the calling application

The `init(long, PendingIntent)` method may fail if the provided timeout is invalid, throwing an `IllegalArgumentException`. If the application watchdog service has been already initialized for the calling applicationg before invoking this method, a `UnsupportedOperationException` will be thrown.

The `getTimeout()` and `stop()` methods may fail with a `UnsupportedOperationException` if the application watchdog service has not been initialized yet for the calling application.

Initializing the application watchdog

```
import com.digi.android.watchdog.ApplicationWatchdogManager;

[...]

ApplicationWatchdogManager applicationWatchdogManager = ...;

// Generate an intent to start this activity again as a new task after
failure.
Intent intent = new Intent(this, MyActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
// Build pending intent.
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent,
0);

[...]

// Check if application watchdog is already running. If not, initialize
it to 5 seconds.
if (!applicationWatchdogManager.isRunning())
 applicationWatchdogManager.init(5000, pendingIntent);

// Print configured timeout.
System.out.println("Configured Application Watchdog timeout:
" + applicationWatchdogManager.getTimeout();

[...]

// Stop the application watchdog service.
if (applicationWatchdogManager.isRunning())
 applicationWatchdogManager.stop();

[...]
```

> Unlike the system watchdog, the application watchdog can be stopped at any time by invoking the `stop()` method. The reason is that the application watchdog is software controlled while the system watchdog is controlled by hardware.

Refresh the application watchdog

The last step after initializing the application watchdog is to periodically refresh it to avoid the application to

shut down. The `refresh()` method is used for this purpose:

Refreshing the application watchdog

```
import com.digi.android.watchdog.ApplicationWatchdogManager;

[...]

ApplicationWatchdogManager applicationWatchdogManager = ...;

applicationWatchdogManager.init(5000, null);
Thread refreshThread = new Thread() {
 @Override
 public void run() {
  applicationWatchdogManager.refresh();
  // Sleep no more time than the configured timeout.
  Thread.sleep(4000);
 }
};
refreshThread.start();

[...]
```

If the application watchdog was not initialized before calling the `refesh()` method, an `UnsupportedOperationException` will be thrown.

Ensure that your application refreshes the application watchdog periodically with a period lower than the configured timeout.

Watchdog Example

The **Watchdog Sample Application** demonstrates the usage of the Watchdog API. The sample allows you to interact with the watchdog service by registering the application either to the system or the application watchdog services and report application failure at any time.

You can easily import the example using Digi's Android Studio plugin. For more information, see Import a Digi sample application. To look at the application source code, go to the GitHub repository.

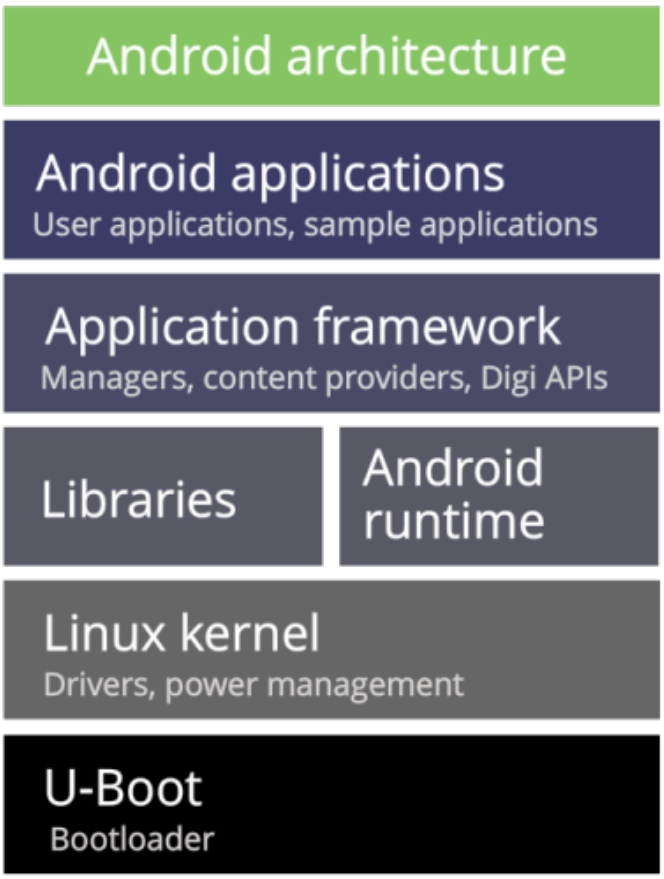
System development [Android]

With Digi Embedded for Android, you can customize and build all the firmware components needed to run Android on your embedded system, allowing you to meet the needs of your project. This section describes the Android system architecture and explains how to build and program your customized Android firmware.

- System architecture
- Build the Android firmware
- Program the Android firmware

System architecture

Android is architected in the form of a software stack comprising applications, an operating system, run-time environment, middleware, services and libraries. All these layers can be simplified in this form:



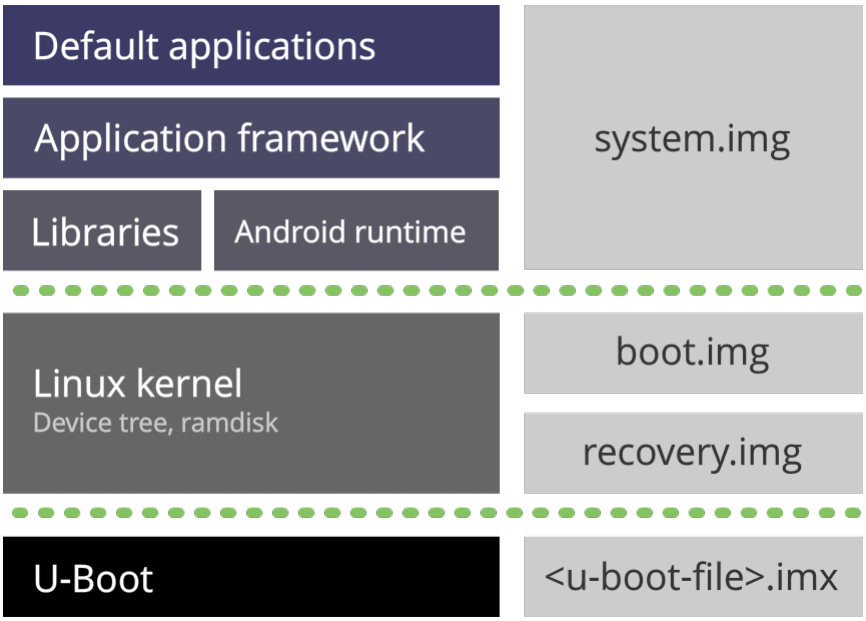
- **Android applications**: The top of the stack is where all the built-in applications, third-party applications, and your own applications live. Digi provides a set of sample applications demonstrating the new APIs.
- **Application framework**: The second layer includes the application framework, which gives you the tools to write applications such as the SDK, activity managers, location managers, notification managers, and the view system. This level includes the set of new built-in Digi API extensions to access the hardware interfaces: I2C, SPI, CAN, ADC, GPIO, CPU and GPU management, Ethernet, PWM, serial port, and watchdog.
- **Libraries** and **Android runtime**: The hardware access layer is where libraries are located for interacting with Ethernet, Wi-Fi, Bluetooth, and other hardware features. The runtime is also at this layer. Each application runs in it's own virtual machine (VM) instance, which means that applications can not directly invade each other's process space. The system provides Linux-style permission controls while maintaining an open system through Intents and other interprocess communication capabilities.
- **Linux kernel**: At it's lowest level, the Linux kernel handles drivers, power management, and other lo w level processes.
- **U-Boot**: Residing below all these layers, U-Boot is not part of Android, but it is needed to load the Android components.

Build artifacts

When the Digi Embedded for Android platform is built, it generates the following build artifacts (files):

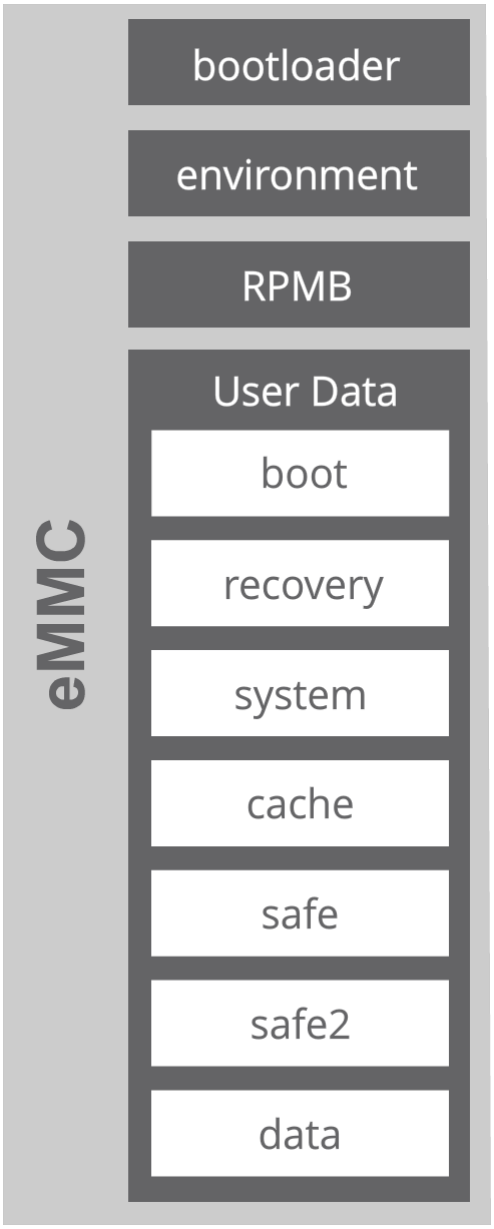
- **system.img**. This is the Android operating system image. It includes all the Android libraries, runtimes, API frameworks, and default applications that run in the device.

- **boot.img**. This image includes the kernel, the ramdisk, and the device tree configuration. U-Boot executes this image after boot, and it mounts the rest of the Android partitions.
- **recovery.img**. This image is very similar to the `boot.img`. It also includes the kernel, a ramdisk, and a device tree configuration to boot in recovery mode. U-Boot only executes this image if the recovery boot mode is requested, usually to install update packages or wipe data and cache partitions.
- **<u-boot-file>.imx**. This file contains the U-Boot bootloader. This is the first piece of code that is executed when the device boots. It initializes the device and launches the operating system.



eMMC layout

The eMMC (internal memory of the ConnectCore 6 device) is split in four physical partitions:



- **bootloader** : This partition contains the U-Boot bootloader image that is executed when the device is powered on. It starts the installed operating system and allows some device configurations.
- **environment** : This partition contains the U-Boot environment and its redundant copy.
- **RPMB**: This is the replay-protected memory-block partition, used to manage data in an authenticated and replay-protected manner. *It is not used at the moment.*
- **User Data**: User Data holds the operating system divided in logical partitions:
  - **boot**: This partition contains Android's kernel, ramdisk, and one or more device tree configurations. This is the partition where the `boot.img` file is programmed.
  - **recovery**: This partition contains an Android's kernel, ramdisk, and the device tree configurations for the recovery boot mode. This is the partition where the `recovery.img` file is programmed.
  - **system**: This partition contains the entire Android operating system, other than the kernel and the ramdisk. This includes the Android user interface as well as all the system applications that come pre-installed on the device. This partition is mounted as read-only to avoid system malfunction. The `system.img` file is programmed into this partition.
  - **cache**: This partition is where Android stores frequently accessed data and application components.
  - **safe**: Empty placeholder partition for storing encrypted user sensitive data. Refer to the TrustFence™ documentation for help on encrypted partitions.

- **safe2**: Empty placeholder partition for storing encrypted user sensitive data. Refer to the TrustFence™ documentation for help on encrypted partitions.
- **data**: Also called *userdata*, the data partition contains the user's data – this is where your settings and applications are located. This partition is mounted as read-write. The SD Card folder is virtually emulated and mounted in this partition.

Build the Android firmware

This section describes how to build Digi Embedded for Android firmware for your device. You have to build your own firmware images to customize them to meet the specific needs of your system. For instance:

- You have designed a custom carrier board for a Digi embedded module and need to customize the U-Boot bootloader or Android kernel.
- You need to customize the default Android root file system to add, remove, or modify software packages.
- You want to integrate your application into Digi Embedded for Android so that it is built and included in the root file system.

Follow these steps to create your own images:

- Set up your development computer
- Build your Android custom images
- Create an update package

Set up your development computer

To build Digi Embedded for Android you need the following:

- A Linux machine with the hardware requirements listed at http://source.android.com/source/requirements.html.

  > Ubuntu 14.04 is the recommended distribution to build Android 5.1, the version used in the ConnectCore 6. **The instructions below are for this distribution**.

- The Digi Embedded for Android source code. To get instructions on how to download the source code, fill out this form.

Once you have the Linux machine set up with the correct hardware requirements, and the Digi Embedded for Android sources, set up your development computer to build the Android firmware:

1. Install OpenJDK 7.
   a. Android version for ConnectCore 6 is 5.1 (Lollipop) and requires Java 7. Install OpenJDK 7 with the following command:

   ```
   $ sudo apt-get install openjdk-7-jdk
   ```

   b. Update the default Java version by running:

   ```
   $ sudo update-alternatives --config java
   $ sudo update-alternatives --config javac
   ```

2. Install required packages.

   Issue the following command to install the required packages in Ubuntu 14.04:

   ```
   $ sudo apt-get install git-core gnupg flex bison gperf
   build-essential zip curl zlib1g-dev gcc-multilib g++-multilib
   libc6-dev-i386 lib32ncurses5-dev x11proto-core-dev libx11-dev
   lib32z-dev ccache libgl1-mesa-dev libxml2-utils xsltproc unzip
   uuid-dev zlibc zlib1g zlib1g-dev liblzo2-dev lzop mingw32 tofrodos
   ```

3. Install the Digi Embedded for Android sources.
   a. Change the permission of the sources file and execute it to install:

   ```
   $ cd ${HOME}
   $ chmod +x dea-<version>.bin
   $ ./dea-<version>.bin
   ```

   b. Accept the license agreement and the sources will be installed inside a directory called `dea-<version>`.

> For more information on how to set up your local work environment to build the Android source files, go to http://source.android.com/source/initializing.html.

Build your Android custom images

Follow these steps to build your Android custom image based on Digi's source code:

1. Set up your environment and install the sources. If you have not already done so, see Set up your development computer.
2. Change to the directory where the source code is installed.

```
$ cd dea-<version>
```

3. Initialize the build environment:

```
$ source build/envsetup.sh
```

4. Select a ConnectCore 6 target to build:
   - `imx6_ccimx6_sbc-eng` creates development images with root access and additional debugging tools.
   - `imx6_ccimx6_sbc-user` creates images with no root access, suited for production.

   You will see information about the selected target:

```
$ lunch imx6_ccimx6_sbc-eng
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.1.1
TARGET_PRODUCT=imx6_ccimx6_sbc
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.16.0-55-generic-x86_64-with-Ubuntu-14.04-trus
ty
HOST_BUILD_TYPE=release
BUILD_ID=DAK-5.1.1-r1
OUT_DIR=out
============================================

$
```

5. Use `make` to build the Android images.
   - To build the complete Android firmware, execute:

```
$ make -j<Number_Of_Jobs>
```

> Building Android can take several hours, depending mainly on the number of CPUs of the development machine and the parallelization level used in the `make` command.

- To only build the bootloader image, execute:

```
$ make -j<Number_Of_Jobs> bootloader
```

- To only build the kernel image, execute:

```
$ make -j<Number_Of_Jobs> kernelimage
```

> The command `make` can handle parallel tasks with a `-jN` argument, and it's common to use a number of tasks `N` that's between 1 and 2 times the number of hardware threads on the computer used for the build.

6. Once the build process finishes, check that the resulting images are inside the `dea-<version>` sources directory at `out/target/product/imx6_ccimx6_sbc`. List them with the following command:

```
$ ls -lh out/target/product/imx6_ccimx6_sbc/
```

> To program these files into your ConnectCore 6, follow the steps in Program the firmware from U-Boot.

Create an update package

Android devices in the field can receive and install over-the-air (OTA) updates to the system and application software. Devices have a special recovery partition with the software needed to unpack a downloaded update package and apply it to the rest of the system.

To build an update package for your platform based on Digi's source code follow these steps:

1. Set up your environment and install the sources as explained in Set up your development computer .
2. Change to the directory where the source code is installed.

```
$ cd dea-<version>
```

3. Initialize the build environment:

```
$ source build/envsetup.sh
```

4. Select a ConnectCore 6 target to build:
   - `imx6_ccimx6_sbc-eng` creates development images with root access and additional debugging tools.
   - `imx6_ccimx6_sbc-user` creates images with no root access, suited for production.

   You will see information about the selected target:

```
$ lunch imx6_ccimx6_sbc-eng
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.1.1
TARGET_PRODUCT=imx6_ccimx6_sbc
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.16.0-55-generic-x86_64-with-Ubuntu-14.04-trus
ty
HOST_BUILD_TYPE=release
BUILD_ID=DAK-5.1.1-1.b1
OUT_DIR=out
============================================

$
```

5. Usually when you develop locally, you would use plain `make` with no particular target to compile. When you prepare for release, however, you need to do this instead:

```
$ make -j<Number_Of_Jobs> dist
```

It compiles the whole source tree, as a plain `make` does. Then it generates several zip files inside the `out/dist` folder of `dea-<version>` directory:

- `imx6_ccimx6_sbc-ota-<build_id>.zip` is the update package that can be installed through recovery. The package contains all the files needed by `system`, `boot` and `recovery` partition.
- `imx6_ccimx6_sbc-target_files-<build_id>.zip` contains all the target files (apk, binaries, libraries, etc.) that will go into the final release package.
- `imx6_ccimx6_sbc-apps-<build_id>.zip` contains all the apks.
- `imx6_ccimx6_sbc-img-<build_id>.zip` contains image files for `system`, `boot`, and `recovery`.
- `imx6_ccimx6_sbc-symbols-<build_id>.zip` contains all files in `out/target/product/imx6_ccimx6_sbc/symbols`.

The file **`imx6_ccimx6_sbc-ota-<build_id>.zip`** is the update package to be installed in your ConnectCore 6.

To install the update package `imx6_ccimx6_sbc-ota-<build_id>.zip` in your ConnectCore 6, follow the steps in Update the Android firmware.

Update package with wipe data support

The default `imx6_ccimx6_sbc-ota-<build_id>.zip` update package generated with `make dist` does not wipe the user data partition. To do so, you have to regenerate this file using the **`ota_from_target_files`** tool.

The `ota_from_target_files` tool generates an update package from the `imx6_ccimx6_sbc-target_files-<build_id>.zip` produced by the Android build system in the Create an update package topic.

Use the `--wipe_user_data` option of this tool to generate an update package with wipe data partition support:

```
$ ./build/tools/releasetools/ota_from_target_files --wipe_user_data
--no_signing --use_raw_images
out/dist/imx6_ccimx6_sbc-target_files-<build_id>.zip
out/dist/imx6_ccimx6_sbc-ota-wipe_data.zip
```

Block-based firmware updates are not supported.

Program the Android firmware

Digi Embedded for Android offers different ways to program the Android system in the ConnectCore 6 device.

If you don't have any system flashed in your device or the one you have is corrupt and does not boot, you should perform a firmware update from U-Boot. You can also update the firmware from U-Boot if you want to use the build result images obtained during the Build the Android firmware process.

If you have an Android system already running and you want to update it using an OTA update package, you can do it directly from the Android settings menu. To learn more on how to generate OTA update packages, follow the steps in the Create an update package topic,

- Program the firmware from U-Boot
- Update the Android firmware

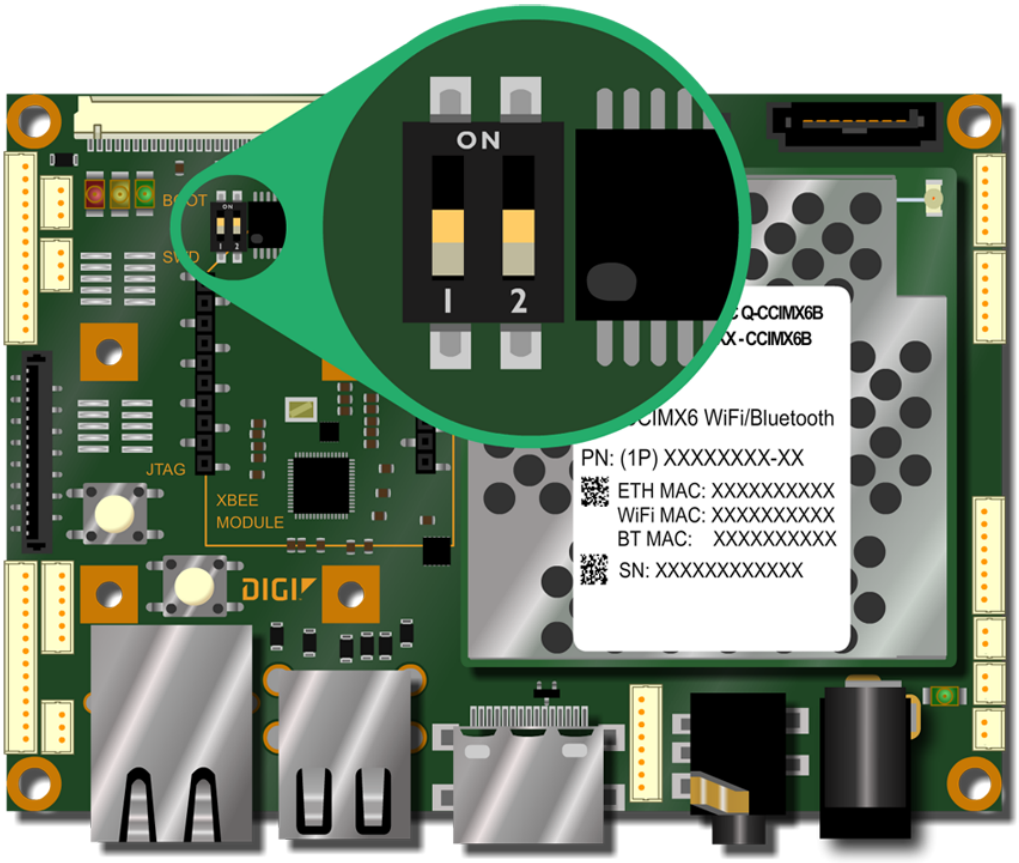
Program the firmware from U-Boot

Once you have the `boot.img`, `recovery.img` and `system.img` already generated, you can use them to update your Android system from U-Boot. Refer to the Build the Android firmware topic for more information about building the Android system.

To flash the Android firmware from U-Boot, you must program a kernel (`boot.img`), a recovery boot image (`recovery.img`), and a root file system image (`system.img`) in the device's eMMC. Follow these steps to program Android in the eMMC of your ConnectCore 6:

> See supported software to verify that your hardware is compatible with Android Lollipop.

1. Power off the device.
2. Change the **boot mode** configuration to boot from the internal eMMC. To do so, set the boot mode micro-switches as follows:
   - **SW3.1** OFF
   - **SW3.2** OFF



3. Place the `system.img`, the `boot.img`, and the `recovery.img` in the the root of a FAT-formatted micro SD card and insert it in the micro SD socket of the ConnectCore 6 SBC.

   After you build the Android firmware, these images are located inside the sources directory at `out/target/product/imx6_ccimx6_sbc` .
4. Connect the Serial adapter cable to the console port [CONS]. Connect a serial cable from the adapter to the development computer.
5. Open a serial connection to the serial port to which the ConnectCore 6 is connected. Use the following settings:
   - **Port**: Serial port to which ConnectCore 6 SBC is attached

- **Baud rate**: 115200
- **Data Bits**: 8
- **Parity**: None
- **Stop Bits**: 1
- **Flow control**: None

6. Power on the device and immediately press a key in the serial terminal to stop the auto-boot process. You will be stopped at the U-Boot bootloader prompt:

```
U-Boot dub-2015.04-r3.1 (Mar 14 2016 - 17:06:20)

CPU:   Freescale i.MX6Q rev1.5 1200 MHz (running at 792 MHz)
CPU:   Extended Commercial temperature grade (-20C to 105C) at 48C
Reset cause: POR
I2C:   ready
DRAM:  1 GiB
MMC:   FSL_SDHC: 0 (eMMC), FSL_SDHC: 1
In:    serial
Out:   serial
Err:   serial
ConnectCore 6 SOM variant 0x02: Consumer quad-core 1.2GHz, 4GB
eMMC, 1GB DDR3, -20/+70C, Wireless, Bluetooth, Kinetis
Board: ConnectCore 6 SBC, version 3, ID 129
Boot device: MMC4
PMIC:  DA9063, Device: 0x61, Variant: 0x60, Customer: 0x00, Config:
0x56
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
=>
```

7. Configure the partition table of the eMMC to hold Android images by executing these commands:

```
=> setenv mmcdev 0
=> run partition_mmc_android
```

8. Update the kernel partition issuing this command:

```
=> update boot mmc 1 fat boot.img
```

9. Wait until the process ends, then execute the following command to update the Android file system:

```
=> update system mmc 1 fat system.img
```

10. Wait until the process ends, then update the recovery partition issuing this command:

```
=> update recovery mmc 1 fat recovery.img
```

11. Wait until this process finishes to force the format of `cache` and `data` partitions:

```
=> bootargs_once="androidboot.cache=format androidboot.data=format"
```

If the `cache` and `data` partitions are already formatted or you wish to preserve their contents, you can skip this command.

12. Change the default boot command in U-Boot to boot from the eMMC issuing these commands:

```
=> setenv bootcmd dboot android mmc
=> saveenv
```

13. Boot the device executing this command:

```
=> boot
```

The first Android boot takes several minutes due to the system deployment.

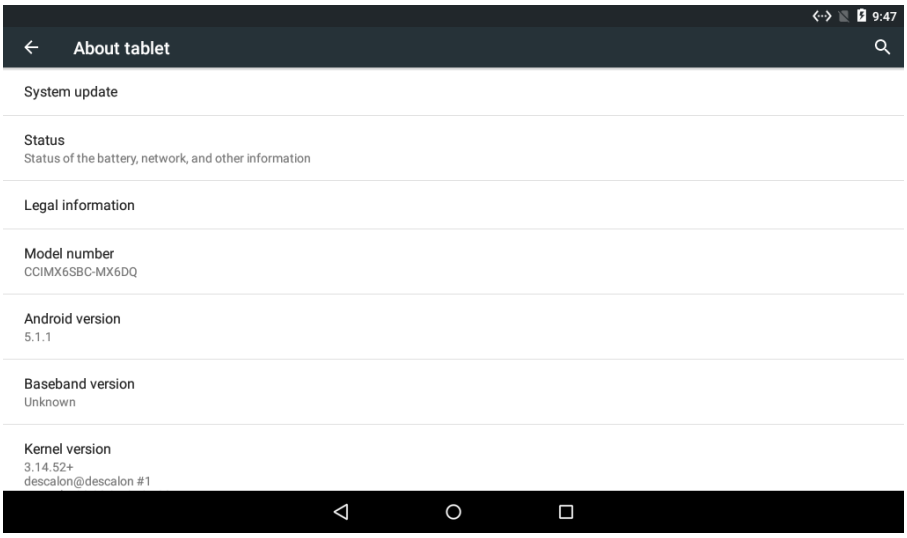
See also Update firmware from TFTP.

Update the Android firmware

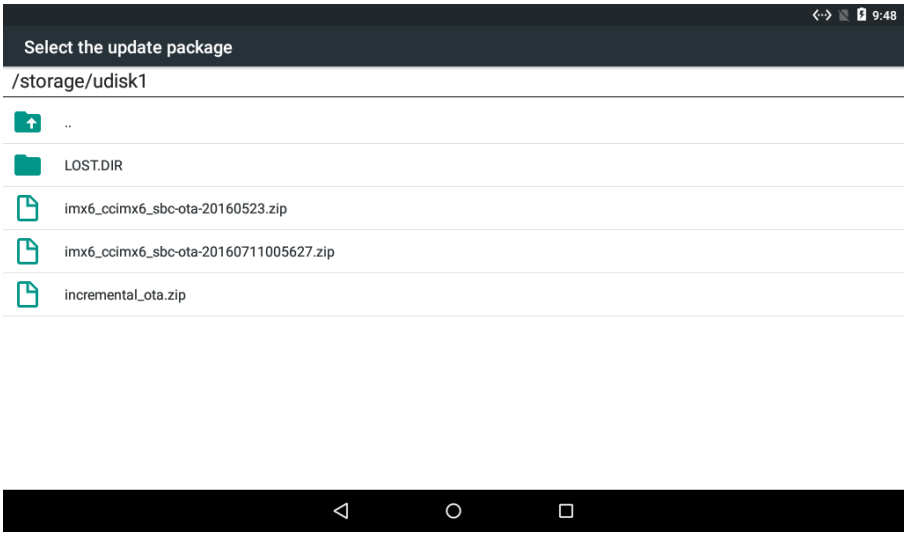
Once you have an update package already generated, you can use it to update your system from the Android settings menu. Refer to Create an update package topic for more information about compiling an Android update package.

Follow these steps to update the Android system using an update package:

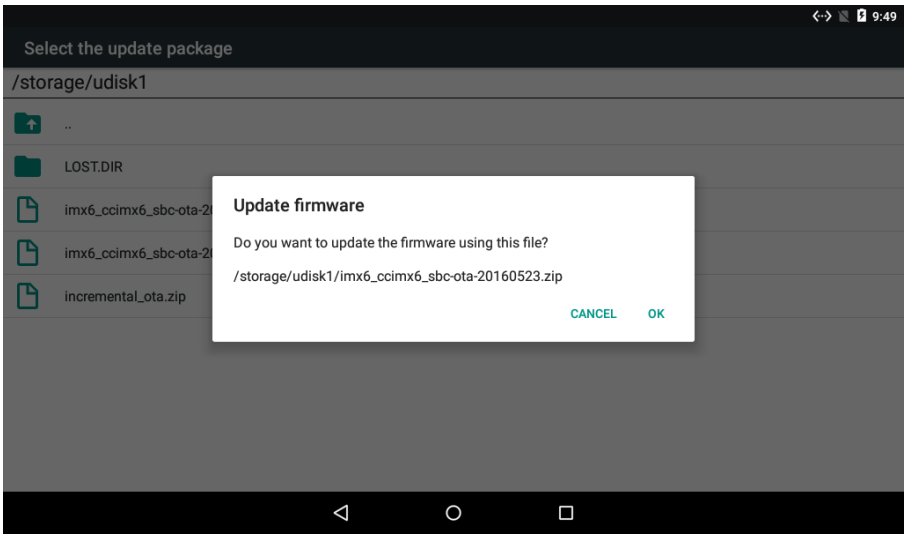
1. Place the update package you have generated anywhere inside the Android file system. It can be located either in the internal memory or in any removable device such as micro SD or USB stick.
2. Navigate to the Android settings and select the **About tablet** category.



3. Select the **System update** option. A file explorer opens.
4. Browse your update package file in the Android file system and select it.



5. Confirm that it is the update package you want to use by selecting the **OK** option in the confirmation dialog.



The firmware update process will start just after the confirmation.

> Remember that the system will be restarted during the update process.

It is also possible to install an update package from a custom Android application using the Firmware update service API provided by Digi. Refer to the Firmware update (API) topic to learn more about it.

Remote management

Digi Embedded for Android includes support to remotely manage your ConnectCore 6 devices. You can remotely monitor and analyze multiple devices, manage their configuration, or update the entire Android system, through the integrated Device Cloud support.

Thanks to this support, you can keep all your ConnectCore 6 hardware up and running by programming alarms based on device conditions or scheduling operations on multiple remote devices quickly and effectively.

Remote management features for ConnectCore 6

These are the main features that Device Cloud offers for ConnectCore 6 modules:

- **Monitor the system**. Analyze system values, such as the CPU usage & temperature or available memory, on a fixed time schedule in order to create a timeline of the device's health.
- **Access the Android file system**. List, upload, download, or remove files of your Android system from Device Cloud or a custom application using the Device Cloud web services.
- **Manage the system**. Remotely control the configuration of some interfaces such as Ethernet, Wireless or Bluetooth, or reboot the module.
- **Update an Android application or the entire system**. Upload an APK file to Device Cloud in order to install or update the application in the module, or upload an Android firmware image to update the system.
- **Schedule and automate operations on multiple remote devices with Web Services**. Write web pages or applications that send requests using the provided API over HTTP (or HTTPS).

Connect to Device Cloud

Before connecting your ConnectCore 6 to Device Cloud, you have to create a new account (if you haven't one). Digi offers a free developer edition account, with a limit of 5 devices, 30 SMS messages per month (billing period), unlimited web services, and data streams (data stored for 30 days). When you are ready to expand your device network and add your 6th device, you will be prompted to upgrade your account.

> To create a free developer edition account, go to http://myacct.digi.com/.

Add your ConnectCore 6 to Device Cloud

Before you can monitor and manage your ConnectCore 6 device, you need to add it to your Device Cloud account. To do so:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Device Management** tab and select **Devices**.
3. Click **Add Devices**. The Add Devices dialog appears.
4. Write the Ethernet MAC address of your ConnectCore 6 SBC and click **Add**. Your device appears in the list pf devices to add.

   > You can find the Ethernet MAC address in the front white label of the ConnectCore 6 SBC.

5. Click **OK** to register the device to your account and exit the Add Devices dialog. A new entry for your device appears in your device inventory.

Obtain a Vendor ID

A Vendor ID is a unique 32-bit code identifying the manufacturer of a device. You need to obtain a Vendor ID in order to connect your ConnectCore 6 device to Device Cloud.

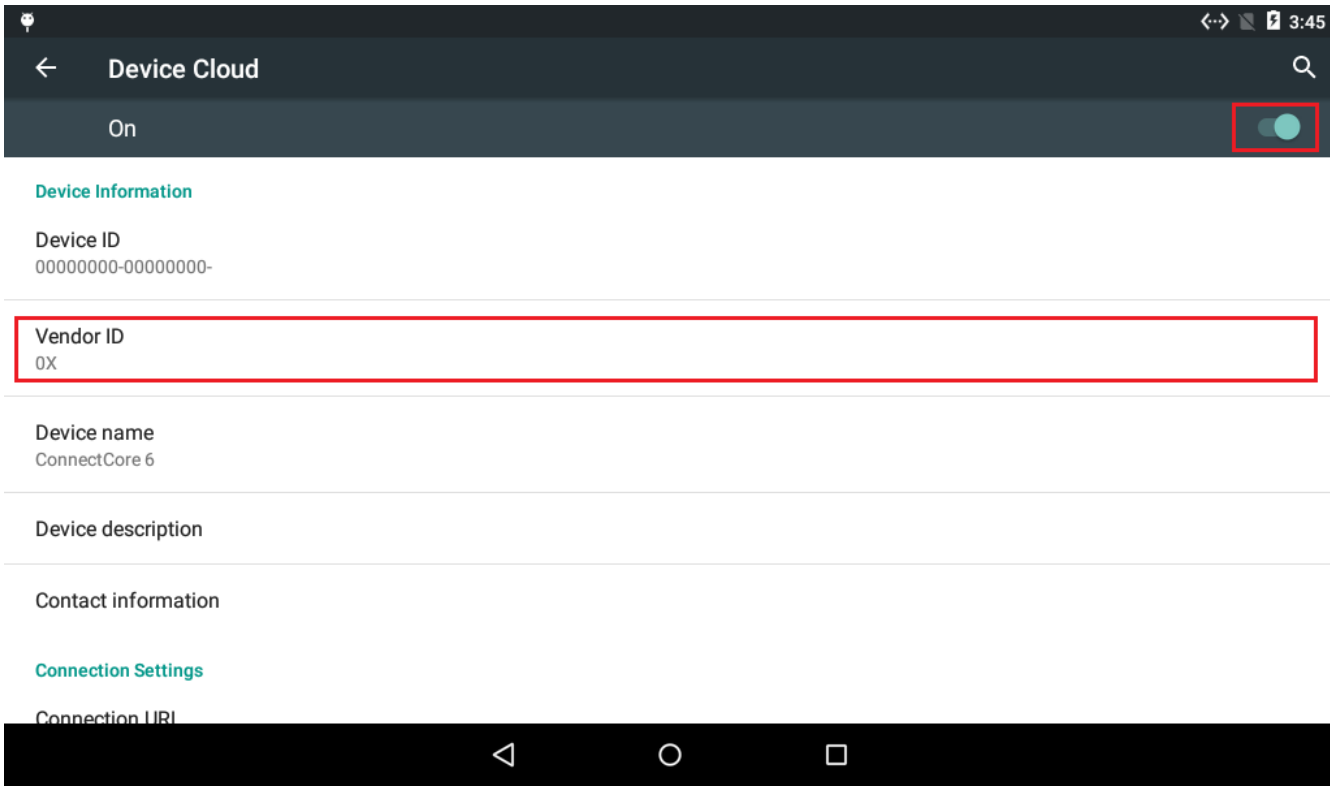
You can obtain a Vendor ID for your Device Cloud account by following these steps:

1. Log in to your Device Cloud account if you haven't already. Once logged in, select the **Admin** tab from the navigation menu.
2. Within the Vendor Information section of the page, click **Register for new vendor id**.
3. The button you just clicked will be replaced by your assigned Vendor ID number (0xXXXXXXXX).

Connect your ConnectCore 6 to Device Cloud

Once the ConnectCore 6 is added to your account and you have a Vendor ID, you have to connect the device to Device Cloud doing the following:

1. Power on the ConnectCore 6.
2. Open the Android **Settings** and go to **Device Cloud** .
3. Click **Vendor ID** and enter the value that appears on your Device Cloud account (note that you have to enter the prefix `0x` as well).
4. Turn on the switch located at the right top to connect the ConnectCore 6 to Device Cloud.



Enable the auto-connect option if you want your ConnectCore 6 automatically connects to Device Cloud when it boots next times.

Monitor the system

System monitoring is one of the key features of the Device Cloud support in ConnectCore 6 modules. It allows you to remotely control some parameters that determine the device's health, such as the CPU usage, CPU temperature, or the available memory. Device Cloud represents these parameters in dynamic graphics and you can also retrieve their values using the Device Cloud web services.

The system monitoring is disabled by default. Enable it following these steps:

1. In the ConnectCore 6 module, open the Android **Settings**.
2. Go to **Device Cloud**.
3. Check the **Enable system monitor** setting.

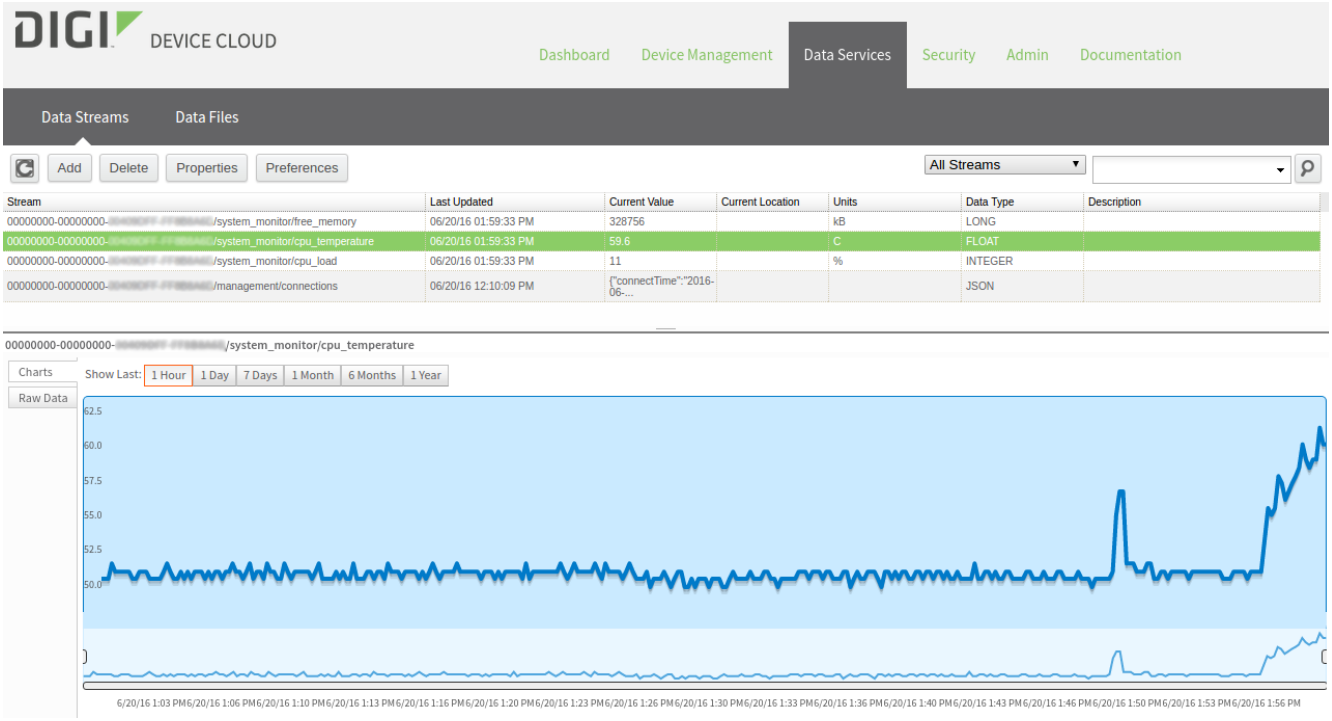
You can also configure the **Sample rate** at which the samples are captured. Another important setting is the **Number of samples to collect before uploading** (within System monitor advanced settings). Default settings are a sample rate of 10 seconds and 6 samples to collect before uploading. This means the module takes a sample for each monitored parameter every 10 seconds and stores it. When there are 6 stored samples of each monitored parameter, the module uploads all the samples. That is, the device uploads 18 samples every 60 seconds.

If the module is disconnected, it stores locally the samples and uploads them when it connects again to Device Cloud (up to a limit of 5000 samples).

Analyze the samples from Device Cloud

You can view the recording of data (in charts and in text format) from Device Cloud. To do so:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Data Services** tab and select **Data Streams**.
3. Select the parameter you want to analyze from the list of streams.
4. In the box below, select the format to represent the values (**Charts** or **Raw Data**).



Update the system

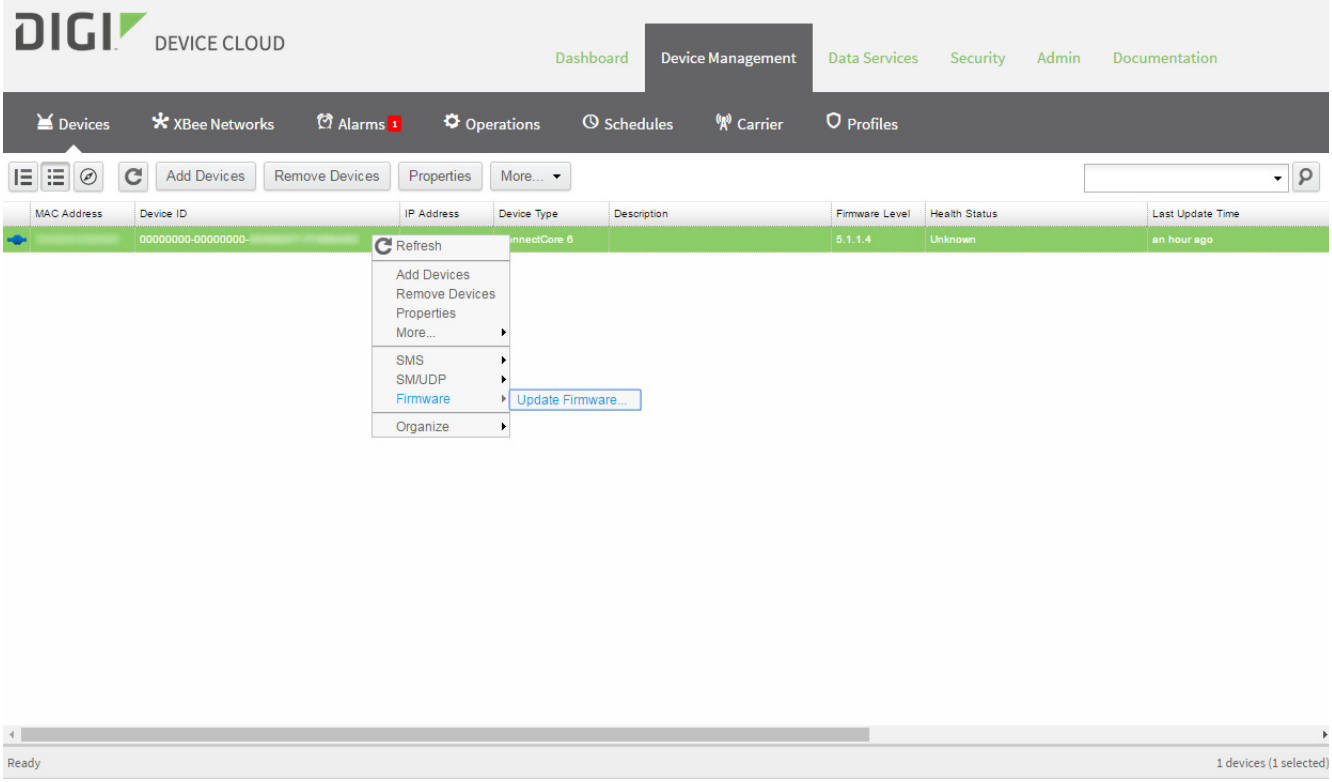
From time to time, new firmware may need to be programmed on your devices. You can remotely update your ConnectCore 6 through Device Cloud:

- Update a single application
- Update the firmware of the entire Android system

Update an application

Device Cloud allows you to install or update an Android application. It is as easy as uploading the APK file to the device using the Update firmware tool:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Device Management** tab and select **Devices**.
3. Right click on your device, select **Firmware** and then **Update Firmware...**.
4. In the Update Firmware window, click **Browse** and select the APK file you want to install or update.
5. Click **Update Firmware**. The application will be installed in the ConnectCore 6 device.



Update the firmware

In addition to be able to install an application, you can update the entire Android system. To do so, you need to create an update package and upload it to the ConnectCore 6 through Device Cloud.

Create the update package

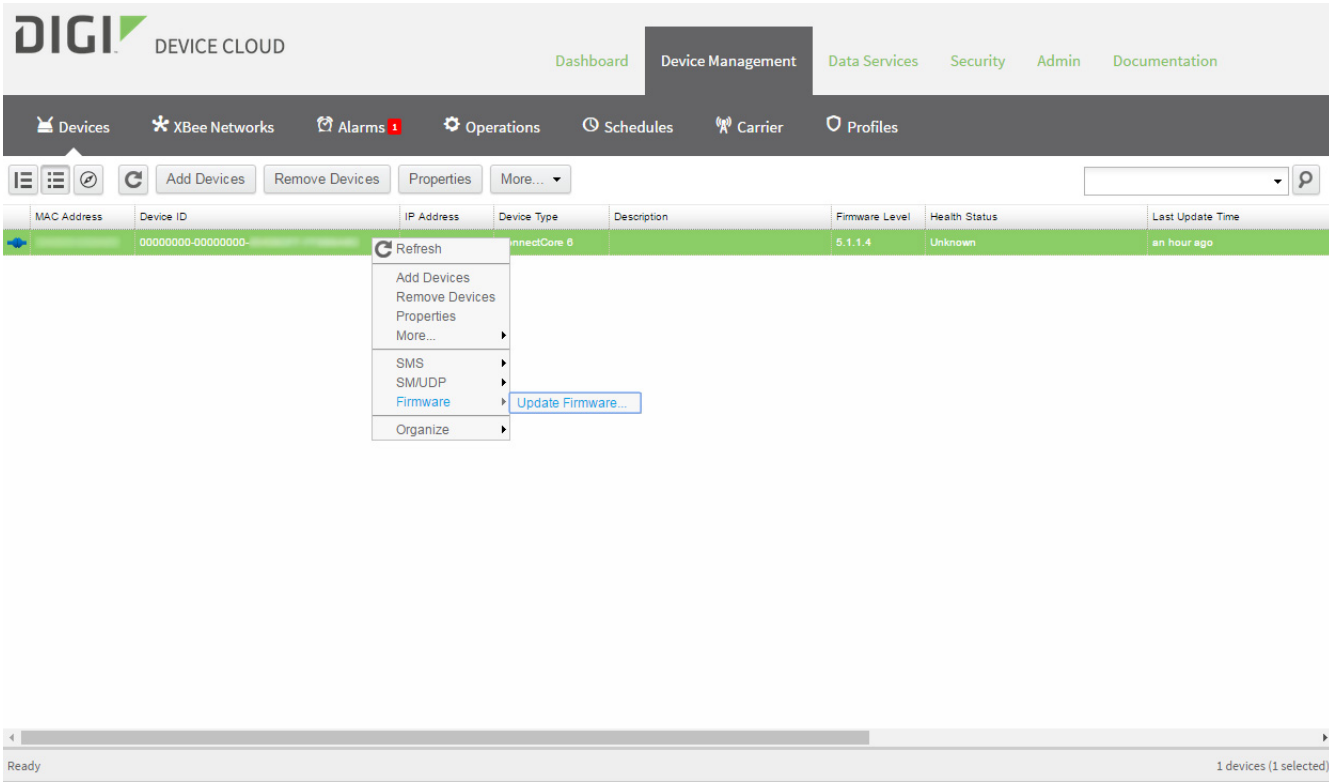
To learn how to create the update package, see the instructions in Create an update package.

Upload the package and update

Once you have created the update package, you have to upload it to Device Cloud to start the update process.

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Device Management** tab and select **Devices**.
3. Right click on your device, select **Firmware**, and then **Update Firmware**.
4. In the Update Firmware window, select the ZIP file of the update package and click **Update Firmware**. The firmware update process starts.

> If the update package is larger than 100 MB, you have to split it into several fragments as explained in the next section.



Fragment the update package and update

Device Cloud does not allow uploading files larger than 100 MB, so if your update package exceeds that limit you have to split it in several fragments. To do so, download the Firmware Fragmenter Java application and run the following command in a shell:

```
java -jar FirmwareFragmenter.jar <update_package.zip>
```

By default, the application will generate fragments of 40 MB and you will have to put them at `/storage/emulated/legacy`, that is, the device's internal storage. You can change this behavior by using the `-s <fragment_size_bytes>` and `-d <device_directory>` parameters respectively. You can find the fragments inside the `out` directory.

Once you have fragmented the update package, do the following:

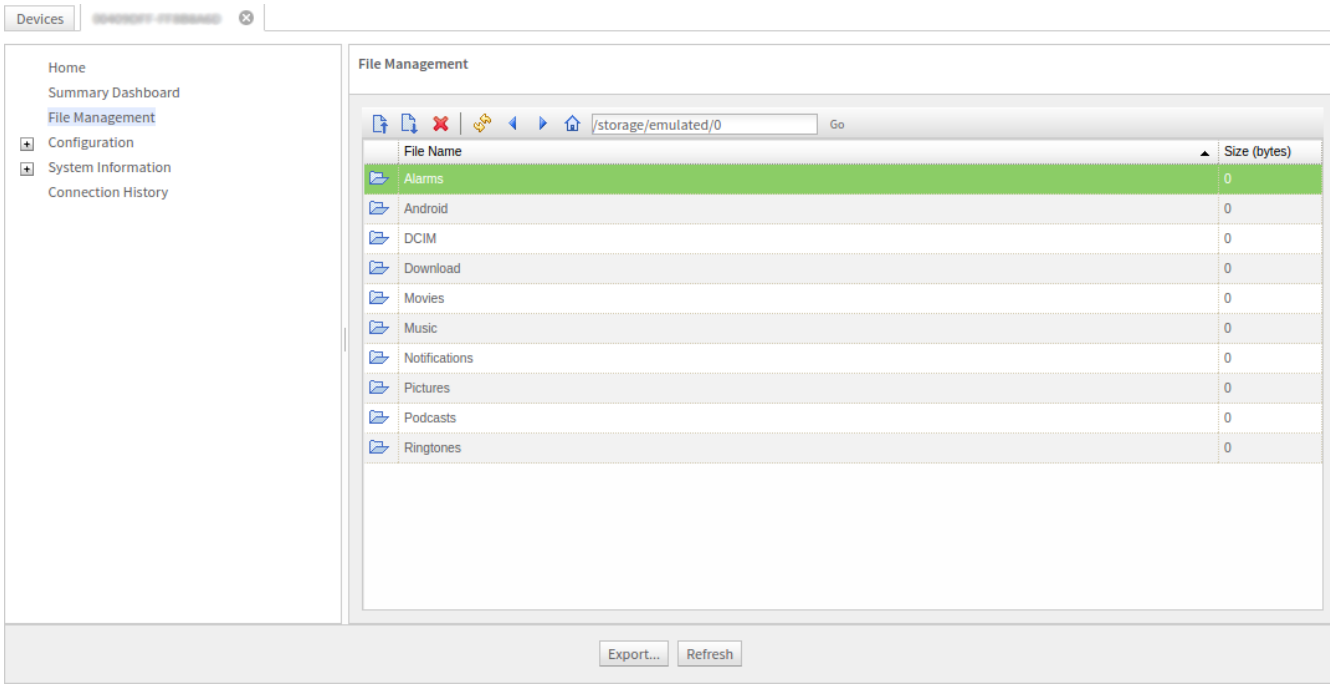
1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Through the File Management tool, upload all the fragments except the manifest file to the location you specified when fragmented the package (by default the device's internal storage).
   To learn how to use this tool, see the instructions in Access the file system.
3. Once they are uploaded to the ConnectCore 6, go to the **Device Management** tab and select **Devices**.
4. Right click on your device, select **Firmware**, and then **Update Firmware**.
5. In the Update Firmware window, select the manifest file of the update package and click **Update Firmware**. The firmware update process starts.

Access the file system

Through the Device Cloud File Management tool you can interact with the Android file system and list, upload, download, or remove files. To use this tool do the following:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Device Management** tab and select **Devices**.
3. Double click your ConnectCore 6. The device's page is loaded.
4. Click **File Management** on the left menu.

By default, the tool loads the content of `/storage`. You can navigate through the file system and use the upper button bar to upload, download or remove a file, refresh the view, go back, forward, to home, or to a specific location.



> Note that some directories are restricted by Android and cannot be accessed.

Get Android information

Device Cloud allows you to get some information related to the Android system installed in your device that could be useful for a better identification of the module. Follow these steps to find that information within the Device Cloud platform:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to the **Device Management** tab and select **Devices**.
3. Double click your ConnectCore 6. The device's page is loaded.
4. In the device's page, open the **System Information** menu and click **Android Information**.

Device Cloud will display some parameters related to the hardware of the module and the Android system installed such as Android version, Kernel version, Build ID, Hardware name and model.

Automate operations with Web Services

You can write HTTP clients that remotely manage all your ConnectCore 6 devices. Examples of such clients include web pages and programs written in a language such as Python or Java.

These clients send requests to the Device Cloud server using standard HTTP requests to retrieve data and be able to remotely control your devices, as you can do from Device Cloud web interface.

These Web Services allows you to **schedule operations to devices en masse**:

- Monitor all your ConnectCore 6 devices. Remotely retrieve and visualize data to determine your devices' health.
- Update your devices firmware. Refresh the firmware or the applications of one or more devices at the same time.
- Access the file system. Interact with files on your devices and list, upload, download, or remove files.
- Manage all your devices. Remotely configure and send specific requests to trigger processes in your devices.

A good starting point is the **API Explorer**. This tool allows you to run any Web Services request and view the response data, as well as export code as Python, Java, Ruby, Perl, or C## code.

For example, the firmware and application update can be automated using these HTTP requests. To do so, follow these steps:

1. Log in to your Device Cloud account (https://devicecloud.digi.com).
2. Go to **Documentation > API Explorer**.
3. Select **Examples > SCI > Firmware > Update firmware**.

   Device Cloud automatically creates the necessary code
4. Replace the "device id" value with the ID of your device.
5. Edit the code block with the following changes:
   a. Add the `filename` attribute to the `<update_firmware>` element. Its value should be either `manifest.txt` for firmware updates or any name ending in `.apk` for application updates.
   b. Replace `<file>~/file.bin</file>` with `<data>[Base 64 encoded data]</data>`. You have to encode the `manifest.txt` file or the Android application you want to install in Base 64 and put it within the `<data>` element.

```
<sci_request version="1.0">
  <update_firmware filename="[manifest.txt|app.apk]">
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <data>[Base 64 encoded data]</data>
  </update_firmware>
</sci_request>
```

6. Click **Export** and then select the programming language you prefer. Device Cloud will generate the necessary code to automate this process.
7. You can test the firmware or application update by clicking **Send**.

For web services usage and reference information go to Device Cloud Programmer Guide.

FAQ [Android]

Learn how to perform the following tasks and get answers to other common issues while working with your ConnectCore 6:

- Recover your device
- Perform a factory reset
- Update firmware from TFTP
- Update firmware from micro SD card
- Boot from micro SD card
- Change Android boot images
- Auto-start custom Android applications
- Override default video mode
- Optimize Android runtime
- Add and remove default Android applications and libraries
- Connect to the device with ADB using network interface

Recover your device

If the bootloader has been erased from the storage media (or written with an invalid image) and the target does not boot, you can recover the target booting from an alternative source, such as a micro SD card. To create a bootable micro SD card, follow these steps:

1. Create a bootable micro SD card from a U-Boot image
2. Boot U-Boot from the micro SD card
3. Update firmware images

1. Create a bootable micro SD card from a U-Boot image

Requirements:

- Root/Administrator permissions in your development computer
- A micro SD card with a minimum capacity of 2 GB

> The following procedure will destroy existing data in the micro SD card.

To create a bootable micro SD card from a U-Boot image:

1. Download the U-Boot bootloader binary image file: *< URL TO THE ANDROID IMAGE>*
   See hardware variants to verify which U-Boot binary you need
2. Extract the **.imx** file into a folder of your choice
3. Raw write the image on the micro SD card using one of the following methods:
   - Using a Linux host
     a. Insert the micro SD card into your computer and check the node Linux assigns to it (`/dev/[sdcard]`) using `dmesg`:

```
$ dmesg
[1413652.901270] sd 41:0:0:0: [sdc] 7744512 512-byte
logical blocks: (3.96 GB/3.69 GiB)
[1413652.903140] sd 41:0:0:0: [sdc] No Caching mode
page present
[1413652.903144] sd 41:0:0:0: [sdc] Assuming drive
cache: write through
[1413652.905638] sd 41:0:0:0: [sdc] No Caching mode
page present
[1413652.905642] sd 41:0:0:0: [sdc] Assuming drive
cache: write through
[1413652.915154] sdc: sdc1
```

> **CAUTION!** Using an incorrect device node in the next step might destroy all data on your computer hard drive.

     b. Raw write the image file to the micro SD card with this command:

```
$ sudo dd if=<path/filename.imx> of=/dev/<sdcard>
bs=512 seek=2 oflag=sync
```

     where:

- `<path/filename.imx>` must be substituted with the path and filename to the U-Boot image.
- `<sdcard>` must be substituted with the device node assigned by Linux to your micro SD card.

The micro SD card is now ready.

- Using a Windows host
  a. Download and uncompress the `dd` application for Windows from www.chrysocome.net/dd.
  b. Open a Command Prompt console in Windows.
  c. Insert the micro SD card into your computer.
  d. Change to the directory where the `dd` application was uncompressed. Then run the `dd` application to list the available drives and identify the device node assigned to your SD card. For instructions, see the `dd` documentation:

```
dd --list
```

  e. Copy the U-Boot image file to the same directory as the `dd` program.
  f. Write the U-Boot image file to the micro SD card with this command:

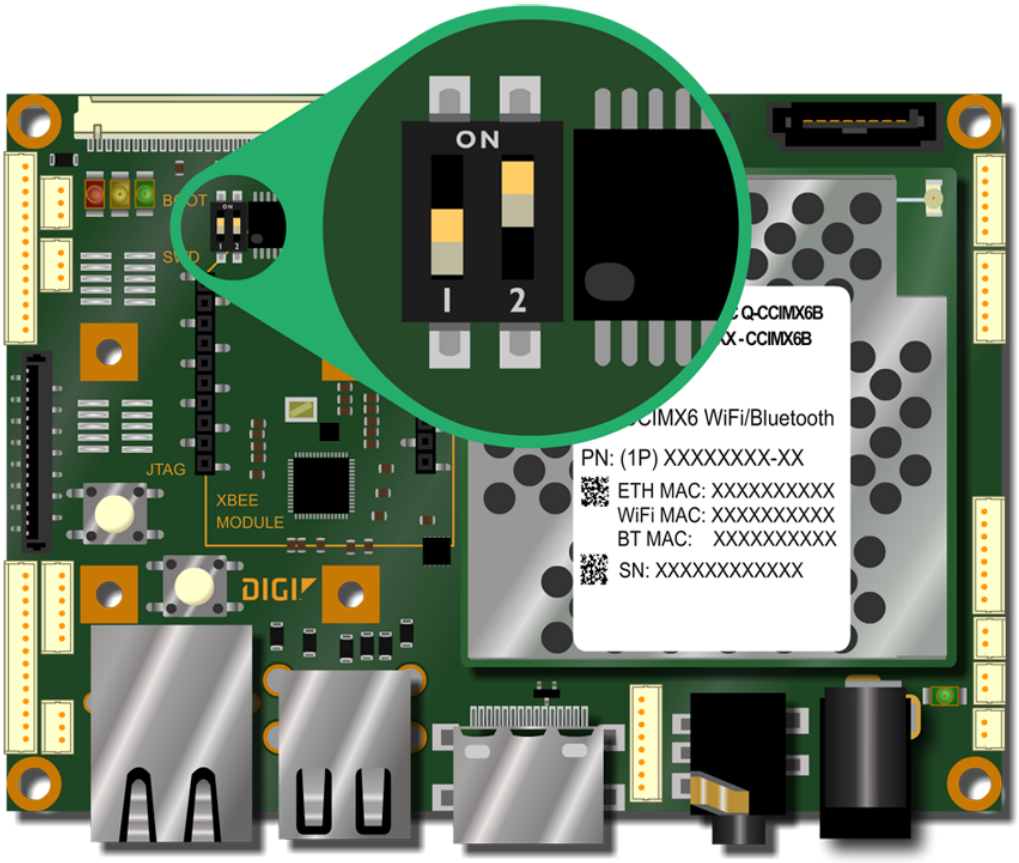
```
dd if=<filename.imx> \\.\Volume{UUID}\ bs=512 seek=2
```

  where:
  - `<filename.imx>` must be substituted with the filename of the U-Boot binary image.
  - `UUID` must be substituted by the identifier given by Windows to the micro SD card on the `--list` command.

The micro SD card is now ready.

2. Boot U-Boot from the micro SD card

1. Power off the device.
2. Insert the micro SD card into the micro SD card holder (bottom side of the board).
3. Change the **boot mode** configuration to boot from the micro SD card. To do so, set the boot mode micro-switches as follows:
   - **SW3.1** OFF
   - **SW3.2** ON



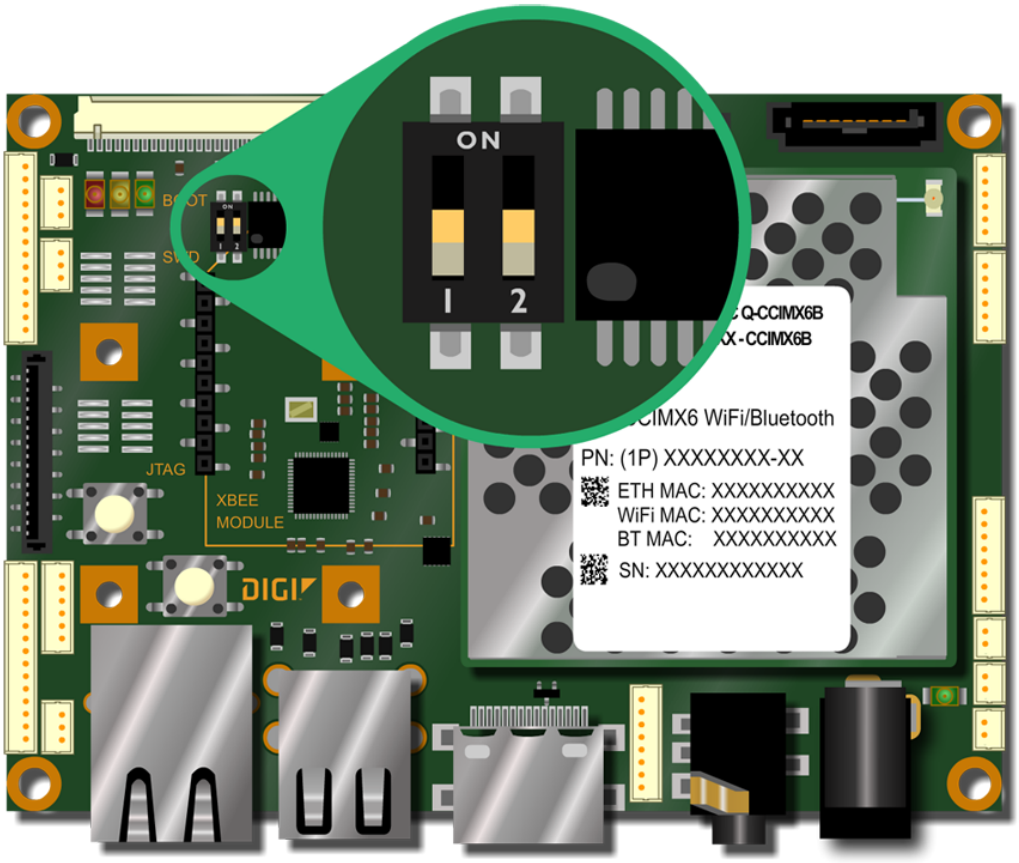
4. Power on the device.

3. Update firmware images

Once U-Boot has started from the micro SD card, you can program U-Boot and Android images in your ConnectCore 6 device. For instructions, see Update firmware from TFTP.

Once the firmware images are loaded in the device's eMMC, you can restore the boot configuration of the module:

- Change the **boot mode** configuration to boot from the internal eMMC. To do so, set the boot mode micro-switches as follows:
  - **SW3.1** OFF
  - **SW3.2** OFF



Perform a factory reset

You can remove all data from your ConnectCore 6 device by resetting it to factory settings. To do this, you can either use the Settings menu to erase all your data or clean the `data` and `cache` partitions from U-Boot.

> By performing a factory data reset, you are wiping all data (apps, documents, photos) from the device. Make sure you back up your data before performing the factory reset.

Option 1: Erase data using the Settings menu

To perform a factory data reset using the Settings menu:

1. In ConnectCore 6, open the **Settings** menu.
2. Under **Personal**, select **Backup & reset**.
3. Select **Factory data reset**.
4. Read the information on the screen and select **Reset tablet**.
5. When prompted, select **Erase everything** to erase all data from your device's internal storage. The module resets automatically.

Option 2: Clean the data and cache partitions using U-Boot

To clean the data and cache partitions, enter these commands in U-Boot:

```
=> bootargs_once="androidboot.data=format androidboot.cache=format"
=> boot
```

Update firmware from TFTP

U-Boot boot loader allows you to update the firmware of your ConnectCore 6 device over Ethernet. U-Boot uses the TFTP protocol to get the firmware images from a TFTP server running in your computer and program them into the eMMC of the device.

> See supported software to verify that your hardware is compatible with Android Lollipop.

This update process requires a TFTP server running in your computer with a configured exposed folder. The devices look for the firmware files in this folder when performing the update.

These instructions do not include setting up a TFTP server in your computer. We assume that you have alr eady installed and configured a TFTP server.

Once you have the TFTP server running in your computer, you can start the update process:

1. Copy the Android firmware files (previously downloaded or compiled) in your TFTP server's exposed folder (typically **/tftpboot**):
   - <u-boot-file>.imx
   - boot.img
   - system.img
   - recovery.img

   After building the Android firmware, you can find the image files inside the sources directory at:

   `out/target/product/imx6_ccimx6_sbc`
2. Connect the serial adapter cable to the console port [CONS] and a serial cable from the adapter to the development computer.
3. Open a serial connection to the serial port to which the ConnectCore 6 is connected. Use the following settings:
   - **Port**: Serial port to which ConnectCore 6 SBC is attached
   - **Baud rate**: 115200
   - **Data Bits**: 8
   - **Parity**: None
   - **Stop Bits**: 1
   - **Flow control**: None
4. Reset the device (press the **Reset** button on the board) and immediately press a key in the serial terminal to stop the auto-boot process. You will be stopped at the U-Boot bootloader prompt:

```
U-Boot dub-2015.04-r3.1 (Mar 14 2016 - 17:06:20)

CPU:   Freescale i.MX6Q rev1.5 1200 MHz (running at 792 MHz)
CPU:   Extended Commercial temperature grade (-20C to 105C) at 48C
Reset cause: POR
I2C:   ready
DRAM:  1 GiB
MMC:   FSL_SDHC: 0 (eMMC), FSL_SDHC: 1
In:    serial
Out:   serial
Err:   serial
ConnectCore 6 SOM variant 0x02: Consumer quad-core 1.2GHz, 4GB
eMMC, 1GB DDR3, -20/+70C, Wireless, Bluetooth, Kinetis
Board: ConnectCore 6 SBC, version 3, ID 129
Boot device: MMC4
PMIC:  DA9063, Device: 0x61, Variant: 0x60, Customer: 0x00, Config:
0x56
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
=>
```

5. Configure the network settings of the ConnectCore 6 device by doing one of the following:
   - Use a DHCP server executing the **dhcp** command:

     ```
     => dhcp
     ```

   - Configure a static IP address, where **a.b.c.d** is the IP address of your Connectcore 6 device:

     ```
     => setenv ipaddr a.b.c.d
     ```

6. Configure the IP address of the machine where the TFTP server is running and save the configuration. In this example, **w.x.y.z** is your machine IP address, where the TFTP server is running:

   ```
   => setenv serverip w.x.y.z
   => saveenv
   ```

7. Optional**.** Update the U-Boot image:

a. Execute the following command to update the U-Boot image:

```
=> update uboot tftp <u-boot-file>.imx
```

See hardware variants to verify which U-Boot binary you need.

b. Reset the board to boot into the recently updated U-Boot, and press any key to stop the auto-boot process.
c. Reset the U-Boot environment to default values (this will not reset protected variables like the MAC address). To do so, issue this command:

```
=> env default -a
```

d. Configure the network and TFTP settings on your ConnectCore 6 device again and save the configuration. Follow the instructions to configure network settings and IP address.

8. Configure the partition of the eMMC to hold Android images by running these commands:

```
=> setenv mmcdev 0
=> run partition_mmc_android
```

9. Update the kernel image by executing this command:

```
=> update boot tftp boot.img
```

10. Update the Android file system image by issuing this command:

```
=> update system tftp system.img
```

11. Update the recovery image by executing this command:

```
=> update recovery tftp recovery.img
```

12. Wait until this process finishes to force the format of `cache` and `data` partitions:

```
=> bootargs_once="androidboot.cache=format androidboot.data=format"
```

If the `cache` and `data` partitions are already formatted or you wish to preserve their contents, you can skip this command.

13. Boot the device with the firmware you have just programmed:

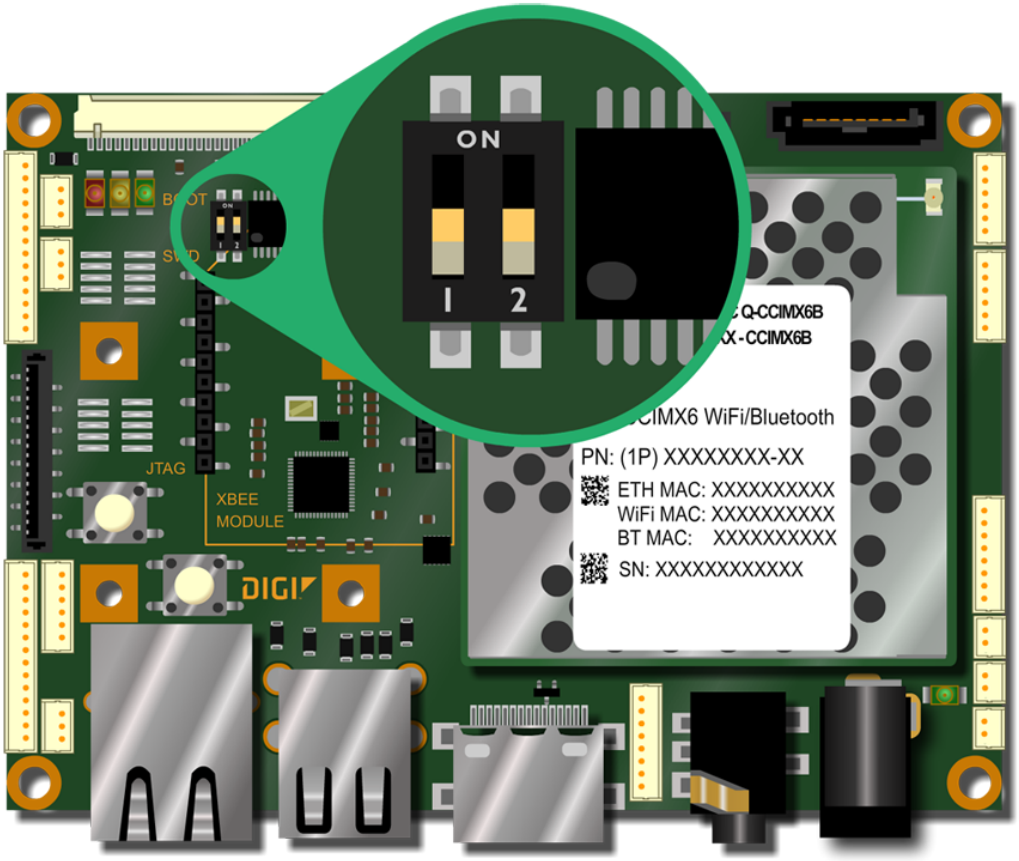
```
=> boot
```

The first Android boot takes several minutes due to the system deployment.

Update firmware from micro SD card

If you don't have a TFTP server, you can still program Android in your ConnectCore 6 device using a micro SD card. The micro SD card must be formatted and have at lease 2 GB of capacity. To program Android fr om the SD card:

1. Power off the device.
2. Change the **boot mode** configuration to boot from the internal eMMC. To do so, set the boot mode micro-switches as follows:
   - **SW3.1** OFF
   - **SW3.2** OFF



3. Place the Android firmware images in the the root of the FAT formatted micro SD card:
   - <u-boot-file>.imx
   - boot.img
   - system.img
   - recovery.img

   > See hardware variants to verify which U-Boot binary you need.
   >
   > After building the Android firmware, you can find these image files inside the sources directory at:
   >
   > `out/target/product/imx6_ccimx6_sbc`

4. Connect the Serial adapter cable to the console port [CONS] and a serial cable from the adapter to the development computer.

5. Open a serial connection to the serial port to which the ConnectCore 6 is connected. Use the following settings:
   - **Port**: Serial port to which ConnectCore 6 SBC is attached
   - **Baud rate**: 115200
   - **Data Bits**: 8
   - **Parity**: None
   - **Stop Bits**: 1
   - **Flow control**: None

6. Power on the device and immediately press a key in the serial terminal to stop the auto-boot process. You will be stopped at the U-Boot bootloader prompt:

```
U-Boot dub-2015.04-r3.1 (Mar 14 2016 - 17:06:20)

CPU:   Freescale i.MX6Q rev1.5 1200 MHz (running at 792 MHz)
CPU:   Extended Commercial temperature grade (-20C to 105C) at 48C
Reset cause: POR
I2C:   ready
DRAM:  1 GiB
MMC:   FSL_SDHC: 0 (eMMC), FSL_SDHC: 1
In:    serial
Out:   serial
Err:   serial
ConnectCore 6 SOM variant 0x02: Consumer quad-core 1.2GHz, 4GB
eMMC, 1GB DDR3, -20/+70C, Wireless, Bluetooth, Kinetis
Board: ConnectCore 6 SBC, version 3, ID 129
Boot device: MMC4
PMIC:  DA9063, Device: 0x61, Variant: 0x60, Customer: 0x00, Config:
0x56
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
=>
```

7. Optional**.** Update the U-Boot image:

   a. Execute the following command to update the U-Boot image:

   ```
   => update uboot mmc 1 fat <u-boot-file>.imx
   ```

   b. Reset the board to boot into the recently updated U-Boot, and press any key to stop the auto-boot process.
   c. Reset the U-Boot environment to default values (this will not reset protected variables like the MAC address). To do so, issue this command:

   ```
   => env default -a
   ```

   d. Configure the network and TFTP settings on your ConnectCore 6 device again and save the configuration. For instructions, see previous steps 5 and 6.

8. Configure the partition table of eMMC to hold Android images. To do so, execute these commands:

```
=> setenv mmcdev 0
=> run partition_mmc_android
```

9. Update the kernel image by executing this command:

```
=> update boot mmc 1 fat boot.img
```

10. Wait until the process ends, then update the Android file system image by issuing this command:

```
=> update system mmc 1 fat system.img
```

11. Wait until the process ends, then update the recovery executing this command:

```
=> update recovery mmc 1 fat recovery.img
```

12. Wait until this process finishes to force the format of `cache` and `data` partitions:

```
=> bootargs_once="androidboot.cache=format androidboot.data=format"
```

> If the `cache` and `data` partitions are already formatted or you wish to preserve their contents, you can skip this command.

13. Change the default boot command in U-Boot to boot from the eMMC by issuing these commands:

```
=> setenv bootcmd dboot android mmc
=> saveenv
```

14. Boot the device by executing this command:

```
=> boot
```

> The first Android boot takes several minutes due to the system deployment.

Boot from micro SD card

U-Boot is capable of starting a complete Android system from a micro SD card. To boot a system form a micro SD card follow these steps:

1. Create a bootable micro SD card from an Android image
2. Boot Android from the micro SD card

1. Create a bootable micro SD card from an Android image

Requirements:

- Root/Administrator permissions in your development computer
- A micro SD card with a minimum capacity of 2 GB

> The following procedure will destroy existing data in the micro SD card.

To create a bootable micro SD card from existing Android image:

1. Download the bootable Android image from this URL: *< URL TO THE ANDROID IMAGE>*
2. Extract the `.sdcard` file from the zip you downloaded into a folder of your choice.
3. Raw write the image on the micro SD card using one of the following methods:

- Using a Linux host
    1. Insert the micro SD card into your computer and check the node Linux assigns to it (`/dev/[sdcard]`) using `dmesg`:

```
$ dmesg
[1413652.901270] sd 41:0:0:0: [sdc] 7744512 512-byte
logical blocks: (3.96 GB/3.69 GiB)
[1413652.903140] sd 41:0:0:0: [sdc] No Caching mode
page present
[1413652.903144] sd 41:0:0:0: [sdc] Assuming drive
cache: write through
[1413652.905638] sd 41:0:0:0: [sdc] No Caching mode
page present
[1413652.905642] sd 41:0:0:0: [sdc] Assuming drive
cache: write through
[1413652.915154] sdc: sdc1
```

> **CAUTION!** Using an incorrect device node in the next step might destroy all data on your computer hard drive.

    2. Raw write the image file to the micro SD card with this command:

```
$ sudo dd if=<path/filename.sdcard> of=/dev/<sdcard>
bs=512 && sync
```

    where:
    - `<path/filename.sdcard>` must be substituted with the path and filename to the SD card image.

- `<sdcard>` must be substituted with the device node assigned by Linux to your micro SD card.

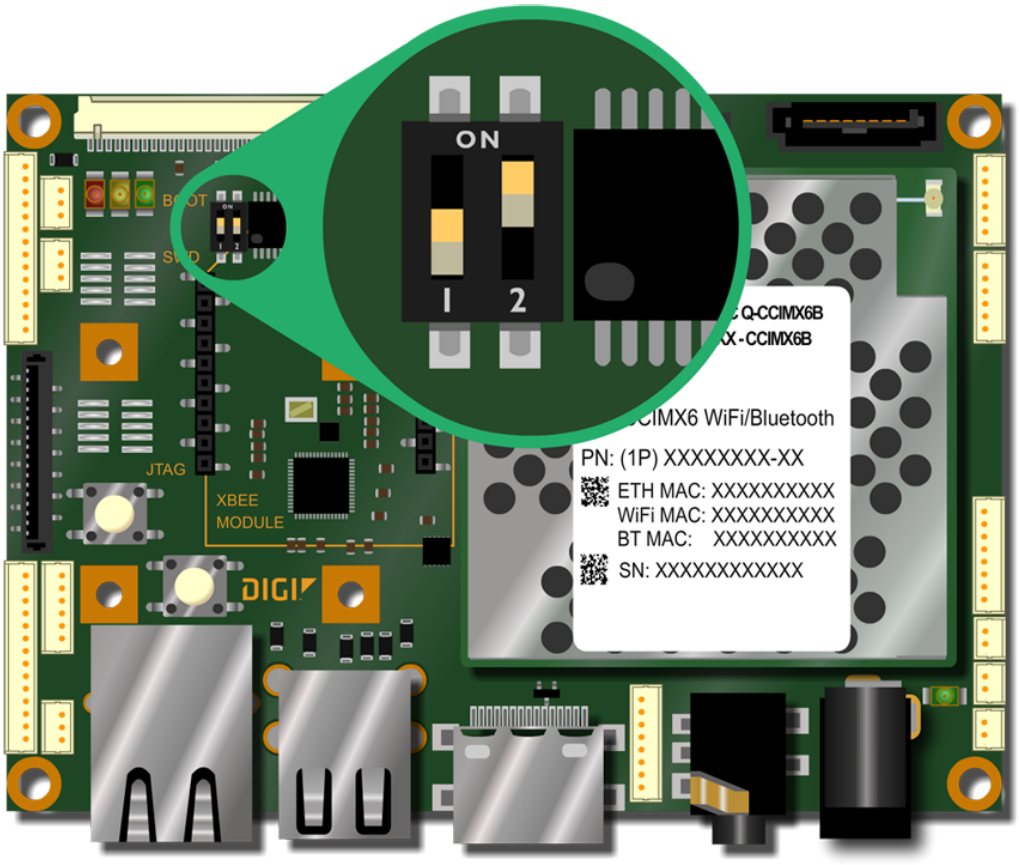
The micro SD card is now ready.

- Using a Windows host
  1. Download the `Win32DiskImager` software from http://sourceforge.net/projects/win32diskimager/files/latest/download.
  2. Uncompress the software and run it as Administrator.
  3. Insert the micro SD card into your computer.
  4. In the software, select the drive that corresponds to the micro SD card. Select the `.sdcard` image file you want to program and click the **Write** button.

  The micro SD card is now ready.

2. Boot Android from the micro SD card

1. Power off the device.
2. Insert the micro SD card into the micro SD card holder (bottom side of the board).
3. Change the **boot mode** configuration to boot from the micro SD card. To do so, set the boot mode micro-switches as follows:
   - **SW3.1** OFF
   - **SW3.2** ON



4. Power on the device.

Android now boots from the micro SD card.

Change Android boot images

When you power on your ConnectCore 6 device, it displays the manufacturing company logo (in this case Digi) for a few seconds, followed by an animation of the Digi logo. You may want to change these images for more complete product branding.

Change Linux kernel splash screen

Requirements:

- A Linux development computer
- A graphic design application
- `ppmquant` utility (part of *netpbm* package)
- `pnmnoraw` utility (part of *netpbm* package)
- The Linux kernel source code

The splash image is stored in the Linux kernel sources in PPM (Portable Pixmap Format) format. To change the splash image, follow these steps:

1. Create a splash image using a graphic design application. Save the image as **logo_custom.ppm**.

   > You can convert any existing image to *ppm* format using conversion utilities such as *mogrify.*

2. Reduce the number of colors to 224 using **ppmquant**. Execute this command:

   ```
   $ ppmquant 224 logo_custom.ppm > logo_custom_224.ppm
   ```

3. Convert the image to ASCII format using **pnmnoraw**. Issue this command:

   ```
   $ pnmnoraw logo_custom_224.ppm > logo_custom_clut224.ppm
   ```

   The final name of the image must be **logo_custom_clut224.ppm**.
4. In the Linux kernel sources, replace the current logo image with the image you have just created. For Android, it is located at `kernel_imx/drivers/video/logo/logo_custom_clut224.ppm` .
5. If you already have the kernel compiled, remove the objects to generate the images with the new image. For Android:

   ```
   $ rm -rf out/target/product/imx6_ccimx6_sbc/obj/KERNEL_OBJ
   ```

6. Build the firmware and program it in the module. The new splash is displayed when you boot your device.

Change Android boot animation

The boot animation and its configuration are contained in a ZIP file called **bootanimation.zip** that is located in the `/system/media` folder of the target root file system. This file includes the following:

- A **desc.txt** file containing the configuration of the animation.
- A **part0** sequence folder containing the PNG images that compose the animation, with filenames containing incremental numbers.
- (Optional) More `part#` folders containing other sequences of the animation.

The animation sequentially displays the images in the `part#` folders. The files located in `part0` are displayed first. Then, if there are more part folders, files within `part1` are displayed, and so on.

> The boot animation must contain at least one sequence folder (`part0`).

desc.txt file

The contents of this file define how the sequences and images are displayed during the animation. The file is configured with the following:

```
width height frame-rate
p loops pause folder-sequence-1
p loops pause folder-sequence-2
...
p loops pause folder-sequence-#
```

- **width** and **height** equal the resolution of the PNG images.
- **frame-rate** defines the number of images play per second.
- **p** indicates that they are part of the animation.
- **loops** defines the number of times that the sequence will loop. A `0` indicates that the sequence will loop infinitely. The last sequence of the animation typically has a loops value of `0`.
- **pause** defines (in number of frames) how long the scene will pause on the last frame before continuing. Divide the pause value by the `frame-rate` to convert it into time.
- **folder-sequence** is the name of the folder where the PNG images for the sequence are located.

Example:

```
420 600 30
p 1 15 part0
p 0 0 part1
```

In this example, the 420 x 600 PNG images play at 30 fps. The first scene (`part0`) plays once and pauses for 15 frames (15/30 = 0.5 seconds) before moving onto the next scene (`part1`) which loops until the boot has finished.

You can add as many sequence definition lines as necessary.

Folders

The folders contain the PNG images for each sequence of the animation. They can have a common prefix but must end with a whole number incrementing by one. For example.

```
boot_0000.png
boot_0001.png
boot_0002.png
...
boot_0085.png
```

An animation must contain at least one sequence, so there should be at least one part folder within the `bootanimation.zip` file.

Generate your own boot animation

Follow these steps to create your own boot animation:

1. Generate the sequence or sequences of PNG images with a graphic design application. Remember to name the images with an incremental whole number.
2. Divide the animation into sequences and organize the PNG files in their corresponding sequence folders. If your animation contains just one sequence, you only need one folder.
3. Create the `desc.txt` file and configure your animation settings. See desc.txt file for more information.
4. Zip your animation files (`desc.txt` and folders) in a file called **bootanimation.zip**.

   > You must use the **store** option to generate an uncompressed zip file.

5. Replace the animation of your module using one of the following methods:
   - **Replace it directly**. You can directly replace the `bootanimation.zip` of your ConnectCore 6.
     a. Open a serial connection to the serial port to which the ConnectCore 6 is connected. Use the following settings:
        - **Port**: Serial port to which ConnectCore 6 SBC is attached
        - **Baud rate**: 115200
        - **Data Bits**: 8
        - **Parity**: None
        - **Stop Bits**: 1
        - **Flow control**: None

     b. Remount the `/system` partition as read-write:

     ```
     #> mount -o rw,remount /system
     ```

     c. Copy your `bootanimation.zip` to `/system/media` .
     d. Give read permissions to the boot animation file by issuing this command:

     ```
     #> chmod 666 /system/media/bootanimation.zip
     ```

     e. Reboot the device. The new boot animation logo should be displayed.
   - **Modify the Android sources**. You can also modify the Android sources to get Android firmware with your own boot animation.
     a. In the Android sources, replace the `bootanimation.zip` file contained in `device/digi/imx_ccimx6_sbc/binaries` with your zip file.
     b. If you already have the sources compiled, remove the file `out/target/product/imx6_ccimx6_sbc/system/media/bootanimation.zip`.
     c. Build the Android firmware and program it in the module.

Auto-start custom Android applications

When creating customized Android firmware, you must typically launch a specific Android application after system boots. Android has two mechanisms for this:

1. Start an application after Android boot:
   - Valid for a standard Android system with multiple applications.
   - No need to modify and compile Android sources.
2. Replace the default Android Home application:
   - Recommended if your system consists only of this application
   - May require you to modify and compile Android sources.

Start an application after Android boot

When an Android system boots, it sends out a boot complete event. Android applications can listen and capture this event to take specific actions, such as automatically starting an activity or service.

You can use this mechanism to create an application with the required permissions to listen for the boot complete event and automatically start an activity or service every time Android starts up. To do so, follow these steps:

1. Declare the permission in the `AndroidManifest.xml`. Add the `android.permission.RECEIVE_BOOT_COMPLETED` permission to your application's manifest file just before the application declaration node:

AndroidManifest.xml

```
<uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

2. Define the Activity that will be auto-started in the `AndroidManifest.xml`. Place this declaration inside the application node:

AndroidManifest.xml

```
<activity
    android:name=".MyActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"
/>
    </intent-filter>
</activity>
```

3. Register the Receiver listening for the boot complete event in the `AndroidManifest.xml`. Place this declaration inside the application node:

AndroidManifest.xml

```
<receiver
    android:name=".StartMyActivityAtBootReceiver"
    android:label="StartMyServiceAtBootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"
/>
    </intent-filter>
</receiver>
```

4. Create the receiver class to listen for the boot complete event. This class must extend `BroadcastReceiver` abstract class. Its `onReceive()` method is called when the device boot is complete. For example, create a Java class called **StartMyActivityAtBootReceiver.java** and place it in the same package as the activity class to auto-start:

StartMyActivityAtBootReceiver

```
 public class StartMyActivityAtBootReceiver extends
BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if
(Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
   Intent activityIntent = new Intent(context, MyActivity.class);
   activityIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
   context.startActivity(activityIntent);
  }
    }
}
```

When this class receives an intent, it checks if it is the `ACTION_BOOT_COMPLETE`. If so, it creates a new activity intent and fills it with the activity class to be started. Finally, it executes the `startActivity()` method using the Android context and the activity intent.

Due to security reasons, Android does not auto-start any application until you **manually** launch it **at least once**. After that, the applications will automatically start on each Android boot.

Replace the default Android Home application

The home screen you see on your Android device after boot is a standard application that reacts to a *home* event. When Android finishes booting and is ready to start the *home* activity, the *home* event is sent and qualifying applications identify themselves as bootable candidates.

The system sends out the `android.intent.category.HOME` and `android.intent.category.DEFAULT` intents when it is done initializing.

Android looks for application manifests with these intent filters when it starts up. If there is more than one, Android lists all of them and allows you to select the one to launch.

In order to designate your application as a home application, follow these steps:

1. Add the intent filters to the `AndroidManifest.xml`. Copy these two lines into the intent filter of your application main activity:

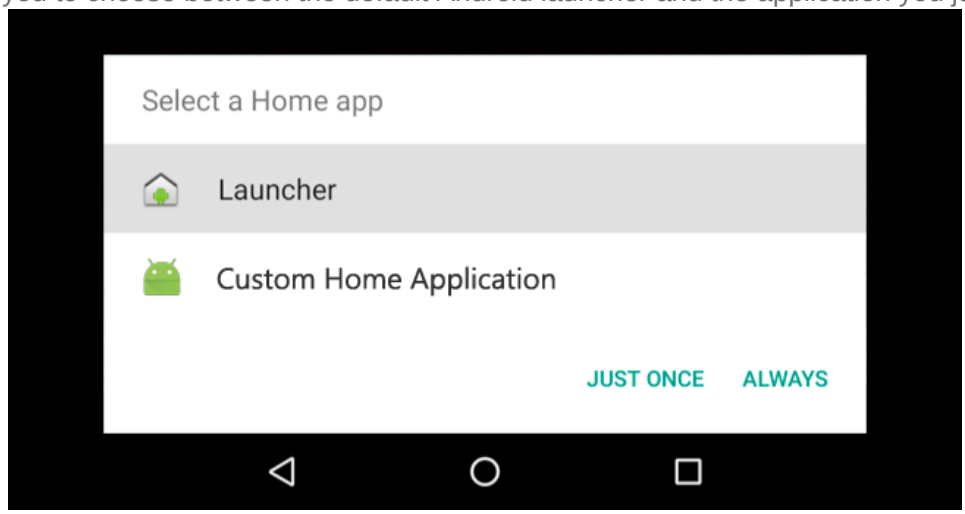
```
<category android:name="android.intent.category.HOME" />
<category android:name="android.intent.category.DEFAULT" />
```

   Your main activity definition should look similar to the following:

AndroidManifest.xml

```
<activity
    android:name=".MyActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.HOME" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.LAUNCHER"
/>
    </intent-filter>
</activity>
```

2. Install your application in the device. On the next startup, Android displays a dialog box that allows you to choose between the default Android launcher and the application you just modified:



You can set your selection as the default home application for the future.

Replace default Home application with a custom one in the sources

The Replacing the default Android Home procedure is only valid for already-deployed Android systems. If you want to deploy an Android system with a custom home application already designated, you must make additional changes to the Android BSP sources:

1. Create a custom home application and include it in the Android BSP sources. You can directly include the application source code or a pre-compiled version of it. To learn how to include custom default applications in your Android image, see <LINK_TO_ADD_CUSTOM_DEFAULT_ANDROID_APPLICATIONS_TOPIC>

Verify that your custom Android home application includes the **android.intent.category.HOME** and **android.intent.category.DEFAULT** intent filters in the application manifest file.

2. Force your application to override the default launcher applications. Add the following entry in your application's `Android.mk` file just before the `include $(BUILD_PACKAGE)` line:

```
LOCAL_OVERRIDES_PACKAGES := Home Launcher2 Launcher3
```

Your application's `Android.mk` file should look similar to the following:

Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_STATIC_JAVA_LIBRARIES := android-common android-support-v13
LOCAL_SRC_FILES := $(call all-java-files-under, src) $(call
all-renderscript-files-under, src)
LOCAL_SDK_VERSION := current
LOCAL_PACKAGE_NAME := MyApplication
LOCAL_CERTIFICATE := shared
LOCAL_PRIVILEGED_MODULE := true
LOCAL_OVERRIDES_PACKAGES := Home Launcher2 Launcher3
include $(BUILD_PACKAGE)
```

3. Include your application in the Android firmware build. Add your application's module name "MyApplication" (as defined in the `LOCAL_PACKAGE_NAME` of your Application's `Android.mk` file) to the list of packages of the firmware at **device/digi/imx6_ccimx6_sbc/imx6_ccimx6_sbc.mk**:

imx6_ccimx6_sbc.mk

```
[...]
PRODUCT_PACKAGES += MyApplication
[...]
```

4. Build the Android firmware. Issue this command sequence in the root folder of the Android sources.
   a. Clean the artifacts from the previous build:

```
$ make installclean
```

   b. Build the Android firmware:

```
$ make -j<Number_Of_Jobs>
```

   `<Number_Of_Jobs>` is the number of cores you want to use for the build process
5. The resulting firmware will boot your custom Android home application by default.

Override default video mode

Default Digi Embedded for Android settings also configure all video interfaces. You can override the configuration by using the `extra_bootargs` variable in the U-Boot environment.

> Digi recommends that you change the video mode in the kernel device tree once verification is complete.

Boot with HDMI only

```
=> setenv extra_bootargs video=mxcfb0:dev=hdmi,1920x1080M@60
video=mxcfb1:off video=mxcfb2:off video=mxcfb3:off
=> saveenv
```

Boot with LVDS only

```
=> setenv extra_bootargs video=mxcfb0:dev=ldb,bpp=16,if=RGB666
video=mxcfb1:off video=mxcfb2:off video=mxcfb3:off
=> saveenv
```

Boot with LVDS primary and HDMI secondary

```
=> setenv extra_bootargs video=mxcfb0:dev=ldb,bpp=16,if=RGB666
video=mxcfb1:dev=hdmi,1920x1080M@60 video=mxcfb2:off video=mxcfb3:off
=> saveenv
```

Boot with HDMI primary and LVDS secondary

```
=> setenv extra_bootargs video=mxcfb0:dev=hdmi,1920x1080M@60
video=mxcfb1:dev=ldb,bpp=16,if=RGB666 video=mxcfb2:off video=mxcfb3:off
=> saveenv
```

Boot with dual LVDS

```
=> setenv extra_bootargs video=mxcfb0:dev=ldb,bpp=16,if=RGB666
video=mxcfb1:dev=ldb,bpp=16,if=RGB666 video=mxcfb2:off video=mxcfb3:off
=> saveenv
```

Optimize Android runtime

Android Lollipop includes a new virtual machine called ART (Android Runtime). ART uses AOT (ahead-of-time) compilation into native code, which performs better than JIT (just-in-time) compilation into bytecode. You can configure ART to perform this optimization in different ways.

Android Lollipop includes the *dex2oat* tool for optimizing applications on deployment.

For a complete overview refer to the Android documentation.

Pre-optimize the sytem image

The Android build system includes a build parameter that allows you to pre-optimize all parts of the `system.img`. This system image optimization results in lower boot times both at deployment and normal boots.

Add the following entry to the `BoardConfig.mk` to enable this feature:

```
WITH_DEXPREOPT := true
```

The resulting image includes the pre-optimized files for each application. Pre-optimized files take more space in the system image, so this option produces a larger system image size.

Optimize system image at first boot

You can optimize the system runtime at first boot with a specific compiler filter used by the `dex2oat` tool. To do so, add the following entry to the `BoardConfig.mk`:

```
ADDITIONAL_BUILD_PROPERTIES +=
dalvik.vm.image-dex2oat-filter=<dex2oat-compiler-filter>
```

Replace the parameter value <*dex2oat-compiler-filter*> with one of the following:

Parameter value	Description
`everything`	`Compiles almost everything, excluding class initializers and some rare methods that are too large to be represented by the compiler's internal representation.`
`speed`	`Compiles most methods and maximizes runtime performance, which is the default option.`
`balanced`	`Attempts to get the best performance return on compilation investment.`
`space`	`Compiles a limited number of methods, prioritizing storage space.`
`interpret-only`	`Skips all compilation and relies on the interpreter to run code.`

verify-none	Special option that skips verification and compilation; should be used only for trusted system code.

During compilation, the build script adds this setting to the `build.prop` file in the system partition. On the first deployment/boot, the `dex2oat` optimizes the system applications with the set compiler filter. The system then starts up faster on subsequent boots due to the optimized code.

> Filter `dalvik.vm.image-dex2oat-filter` only applies if pre-optimization `WITH_DEXPREOPT` is disabled.

Optimize new applications

The ART VM also includes parameters that increase the performance of new applications installed in a running Android system. You must add the following compiler filter to the `BoardConfig.mk`.

```
ADDITIONAL_BUILD_PROPERTIES +=
dalvik.vm.dex2oat-filter=<dex2oat-compiler-filter>
```

Replace <*dex2oat-compiler-filter*> with one of the values described in the parameter value table in Optimize system image at first boot.

During compilation, the build script adds this setting to the `build.prop` file in the system partition. The `dex2oat` tool uses this filter to optimize any application during installation on the running system.

Default Android images optimization

The following table lists default configuration of the **eng** and **user** images:

Image	Pre-optimization	Optimization at first boot	Optimization new apps
eng	`WITH_DEXPREOPT := true`	`dalvik.vm.image-dex2oat-filter=verify-none`	`dalvik.vm.dex2oat-fil`
user	`WITH_DEXPREOPT := true`	`dalvik.vm.image-dex2oat-filter=speed`	`dalvik.vm.dex2oat-fil`

Add and remove default Android applications and libraries

One of the most common customization actions to perform in an Android firmware is to establish the default applications and libraries that will be compiled and included in the final Android image. These items are defined in the PRODUCT_PACKAGES variable that is declared and extended in several Android makefiles along the sources. This is a brief list of makefiles containing the most important applications and libraries that will be included in the final Android firmware:

- device/digi/imx6_ccimx6_sbc/imx6_ccimx6_sbc.mk
- device/digi/imx6_ccimx6_sbc/imx6.mk
- build/target/product/generic.mk
- build/target/product/generic_no_telephony.mk
- build/target/product/telephony.mk
- build/target/product/core.mk
- build/target/product/core_base.mk
- build/target/product/core_minimal.mk
- build/target/product/base.mk
- build/target/product/embedded.mk

> The makefiles of the list are sorted by calling order, from product specific to Android core.

Each application or library is identified by its LOCAL_PACKAGE_NAME, which defined in the component makefile. This is an example of the *Calculator* application makefile:

Calculator application makefile

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_STATIC_JAVA_LIBRARIES := libarity android-support-v4 guava
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_SDK_VERSION := current
LOCAL_PACKAGE_NAME := Calculator
include $(BUILD_PACKAGE)
##################################################
include $(CLEAR_VARS)
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := libarity:arity-2.1.2.jar
include $(BUILD_MULTI_PREBUILT)
# Use the following include to make our test apk.
include $(call all-makefiles-under,$(LOCAL_PATH))
```

Note that the LOCAL_PACKAGE_NAME of the Calculator application is **Calculator**, this will identify the component along all the Android sources.

Remove a default application or library

In order to remove a default application or library from your custom Android firmware, you must delete all the entries of that component from the PRODUCT_PACKAGES variable. To do so, look for the component identifier in the list of makefiles presented before and remove all the entries you find.

> The same application or library could be declared in more than one makefile. It is recomended to

check the list of makefiles listed before to ensure that the component is fully removed.

For example, to remove the Calculator application you need to remove the `Calculator \` entry from the `build/target/product/core.mk` file (it is not defined anywhere else)

build/target/product/core.mk

```
[...]
PRODUCT_PACKAGES += \
    BasicDreams \
    Browser \
    Calculator \
    Calendar \
    CalendarProvider \
    CaptivePortalLogin \
[...]
```

Removing applications or libraries may cause other Android components to stop working properly, do it carefully.

It is highly recommended to clean the exported components before attempting a new product build in order to apply the changes. To do so issue this command in the root of your Android sources:

```
$  make installclean
```

Add a default application or library

To add a default Android application or library to your custom Android firmware, you need to add the component identifier to the PRODUCT_PACKAGES variable. You can do that in any of the makefiles listed before, but it is highly recommended to add it to the product specific makefile to track all the new components there.

For example, if you have an Android application whose LOCAL_PACKAGE_NAME defined in its makefile is **MyApplication**, you only need to add that entry to the PRODUCT_PACKAGES variable in the `device/digi/imx6_ccimx6_sbc/imx6_ccimx6_sbc.mk` file:

device/digi/imx6_ccimx6_sbc/imx6_ccimx6_sbc.mk

```
[...]
# Your custom applications
PRODUCT_PACKAGES += \
    MyApplication
[...]
```

It is highly recommended to clean the exported components before attempting a new product build in order to apply the changes. To do so issue this command in the root of your Android sources:

```
$  make installclean
```

Connect to the device with ADB using network interface

By default, the ConnectCore 6 device is configured to connect to the ADB (Android Debug Bridge) daemon using the USB interface. However, it can be changed in order to connect to ADB using a network interface (Wi-Fi or Ethernet). Follow these steps to change the interface used to communicate with the device's ADB daemon to network:

1. Open a serial terminal with the device using this configuration `115200/8/N/1/N`
2. Execute the following commands:

   ```
   setprop service.adb.tcp.port 5555
   stop adbd
   start adbd
   ```

3. Open a system command shell and try to connect to the device over Ethernet executing this command:

   ```
   adb connect <ipAddress>:5555
   ```

   Where `<ipAddress>` is the IP of your ConnectCore 6 device.

Connect to the device with ADB over USB

If you have already performed a network connection and you want to connect back to ADB using the USB interface, follow these steps:

1. Open a serial terminal with the device using this configuration `115200/8/N/1/N`
2. Execute the following commands:

   ```
   setprop service.adb.tcp.port -1
   stop adbd
   start adbd
   ```

3. Connect the USB device cable to the computer.

> Rebooting the device will also configure ADB to use the USB interface by default.

Known issues and limitations [Android]

Software limitations

- Android Lollipop requires at least 1GB of RAM to run properly. Variants with 512MB of RAM have resource allocation issues while running this version of Android.
- Telit, Huawei, and Sierra Wireless RILs that were provided in binary format in Android KitKat are not yet available in Android Lollipop, so the cellular enhancement feature is not available.
- Block-based firmware updates are not supported.

Hardware limitations

- The Ethernet PHY used in the ConnectCore 6 SBC has a known delay of up to five seconds when autonegotiating at Gigabit speed.
- The maximum number of clients able to use the Wi-Fi hotspot mode in the AR6233 wireless chipset firmware is limited to five.
- The maximum number of devices able to connect to the Bluetooth Low Energy interface on the the AR6233 is limited on the chipset firmware to 10.

Known Android Studio issues

- Android Studio does not automatically update examples to the latest version.
- Android Studio has an issue with compilation when you build your first application. You can find the documented workaround in 4.2. Create the Android application.