Reference Guide

Ayla Embedded Agent for Android

Version: 1.0

Date Released: December 28, 2017

Document Number: AY006FAA3-1

Contact Information

Ayla Networks TECHNICAL SUPPORT and SALES

Contact Technical Support: https://support.aylanetworks.com
or via email at support@aylanetworks.com

Contact Sales: https://www.aylanetworks.com/company/contact-us

Ayla Networks REGIONAL OFFICES

GREATER CHINA
Shenzhen
Room 310-311
City University of Hong Kong
Research Institute Building
No. 8 Yuexing 1st Road
High-Tech Industrial Park
Nanshan District
Shenzhen, China
Phone: 0755-86581520

HEADQUARTERS
Silicon Valley
4250 Burton Drive, Suite 100
Santa Clara, CA 95054
United States
Phone: +1 408 830 9844
Fax: +1 408 716 2621

JAPAN
Room #701
No. 2 Ueno Building 3-7-18
Shin-Yokohama, Kohoku Ward
Yokohama City, 222-0033 Japan
Phone: 045-594-8406

EUROPE
Munich
Ludwigstr. 8
D-80539 München
Germany

TAIWAN
Taipei
5F No. 250 Sec. 1
Neihu Road, Neihu District
Taipei 11493, Taiwan

For a Complete Contact List of Our Offices in the US, China, Europe, Taiwan, and Japan:
https://www.aylanetworks.com/company/contact-us

Table of Contents

1 Introduction

The Ayla Embedded Agent for Android is a software development framework that allows a device running Android OS to access the Ayla Cloud Service using an Android application, calling Ayla's device API. The Android Agent software distribution contains the API libraries and a ready-to-run Android sample application, which can be used as a reference.

1.1 Audience

This document is intended for engineers who are familiar with Android application development.

1.2 Related Documentation

Refer to the following documents available on support.aylanetworks.com for additional information on the Ayla Developer Portal and OEM Dashboard.

- *Ayla Developer Portal User's Guide* (AY006UDP3-2)
- *Ayla OEM Dashboard User's Guide* (AY006UDB3-4)
- *Device Onboarding: Ayla Registration Methods* (AY006FOR3-1)
- *Ayla LAN Mode Mobile Application Note* (AY006ALM3)

1.3 Document Conventions

This document uses these Ayla documentation conventions:

- For APIs, functions returning 0 on success may return -1 or some other non-zero value on failure. If the type returned is enum ada_err, a successful return is zero and the error returns is an error number less than zero.
- Text that you type (i.e. commands) and the contents of downloaded files are shown as:

```
cd wmsdk_bundle-3.1.16.1
tar xfz ada-wmsdk-src-1.0.tgz
```

- Function prototypes, function names, variables, structure names and members, and other code fragments are shown in `courier new`, a fixed-width font.
- Network paths, file paths, menu paths and the like are shown in **bold** text and each point that you have to click to navigate to the next is separated by "/."
- Information on hazards that could damage a product or actions that may cause data loss or undesirable results.

CAUTION

Ensure that the appropriate data buffering is accounted for in deployed devices, where loss of data is critical to the core functionality or the services provided by the systems.

1.5 Abbreviations and Acronyms

The following acronyms are used in this document.

ADA	Ayla Device Agent. This is a legacy term for Ayla Embedded Agent. The ADA acronym is still used in CLI commands and sources, and descriptions of either of these may refer to the Ayla Embedded Agent as the “device agent.”
ADS	Ayla Device Service (the cloud service)
ADW	Ayla Device Wi-Fi (library for embedded systems)
ANS	Ayla Notification Service
ARM	Advanced RISC Machines
CGI	Common Gateway Interface
CRC	Cyclic Redundancy Check
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Server
EVB	Evaluation Board
MIPS	Million instructions per second
NDK	Native Develop Toolkit
RISC	Reduced Instruction Set Computing

2 Ayla Android Device Architecture

This section provides the software components of an Ayla Android device application. Also refer to Figure 1 at the end of the section.

Ayla Device Service (ayladeviceservice)

This is the cloud client background service, which provides the following functionality:

- Communicates using a secure connection with the Ayla Device Service (ADS)
- Connects to the Ayla Notification Service (ANS) and receives event notifications
- Handles JSON interface requests with an internal web server
- Supports LAN mode sessions for communication with Ayla mobile applications

Ayla Application Service (aylaappservice)

This is the application background service that transfers datapoints between the Android Application and the ADS. When the application is in `onStop` state, datapoints are cached. The application is updated when the application execution resumes.

Ayla Device Library (ayladevicelib)

This is the Android library for application development. The API invokes the Ayla device and application services.

Android Agent

This is an Android application that does the following:

- Starts the Ayla device and application services.
- Creates properties.
- Associates the properties with an Ayla template in the Ayla Cloud Service.
- Sends and receives datapoints from the Ayla Cloud.

The Android Agent serves as an example to demonstrate the Ayla APIs and can be used as a reference. Customers can modify or replace this agent with their own application.

acgi

This is a CGI utility executed by the device's primary web server. To handle all incoming .json requests , the web server should be configured to run acgi. acgi parses each request. Valid requests are forwarded to the Ayla Device Android Service and handled by its internal web server. This component is required to support LAN mode, Wi-Fi Setup, and Same-LAN registration. Some device configuration is needed for acgi to work. Also refer to 6 LAN Mode.

lighttpd

This is a lightweight web server installed in the Linux layer to operate LAN mode. This web server is an example only. Customer may replace it with their own web server or use the server as-is.

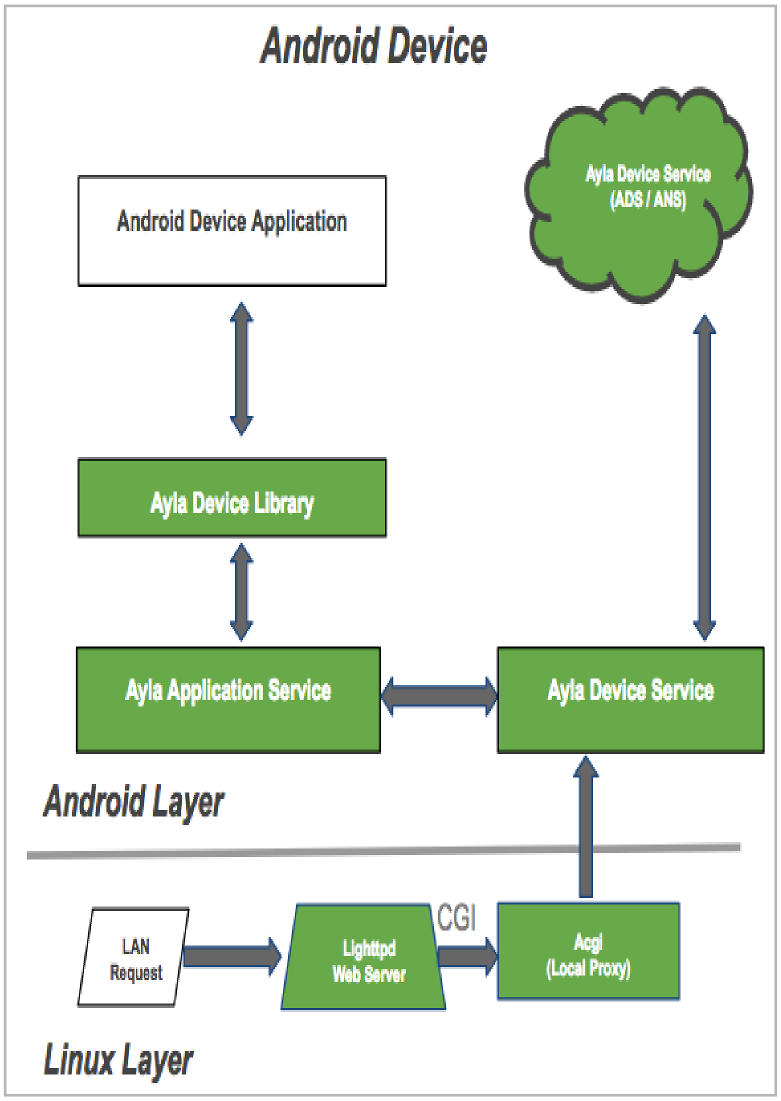


Figure 1: System Level Block Diagram

3 Android Device Configuration

Each device requires a factory configuration file to uniquely identify itself and authenticate it with the Ayla Device Service (ADS). This file contains several important fields, including the Device Serial Number (DSN), its RSA public key for authentication, and OEM-specific information and key. This section describes:

- Methods to create a factory configuration file
- Configuration file management

3.1 Create a Factory Configuration File

There are currently two methods to create a unique factory configuration file for each device:

- Factory Method, which is geared toward large volume production on a factory line.
- Development Method, which is generally used by engineers to generate small batches of configuration files to use during development.

3.1.1 Factory Method

Each device is manufactured and programmed with a firmware image containing a default (not-unique) configuration file. When the device is being provisioned, a command-line utility on the device called `ad_setup` is run using a serial console to program the factory configuration file with the device's unique identity. This is ideal for high volume production because all devices can be loaded with the same firmware image, and a quick stop along the production line can program the device using a script running in a serial console. Ayla provides a sample script called `ad_setup.py` that runs on a factory workstation to program the correct configuration.

3.1.2 Development Method

Source code is provided for a utility called `config_gen` that can be compiled and run on a Linux development machine. This utility takes several parameters, including:

- the `<DSN>.xml` file produced by the Ayla Factory Service
- an OEM info file
- the MAC address of the device to be provisioned (if a manufacturing log file is needed)

The `config_gen` utility outputs a unique device factory config and appends an entry to a factory log file that may be required by Ayla to enable the device on the Ayla Cloud Service.

To compile `config_gen`, in directory scripts, issue:

```
build.sh -f
```

This generates the Linux tool `config_gen`. A message appears on the screen to provide the file location.

3.2 Configuration File Management

Other than during factory provisioning, the factory configuration file is never written to; we therefore recommend that you set read-only file permissions for the factory configuration file or store it in a read-only file system to ensure the integrity of the file.

When a module modifies its configuration from defaults and saves the configuration changes, a new file is created called `startup config`. All future configuration changes overwrite the `startup config` file. By default, the startup configuration is created in the internal storage, which is only accessible by your application. It uses the factory file name with `.startup` appended to it.

NOTE It is important to store the `startup config` in a writeable flash file system so that `startup config` persists across reboots. Otherwise, the modules revert to factory default settings on each reboot.

When a module does a factory reset, the module deletes its `startup config` file, and loads its `factory config` file.

4 Setup Development Environment

You need a 64-bit Mac or Linux computer with following software installed for development:

- Python
- Android Studio
- Git

The three steps to build a working environment are described in this section.

4.1 Step One: Clone the Source Code

Obtain and run the git clone source code from:

https://github.com/AylaNetworks/device_android_public

When this step is completed, you should have the following in the directory called `device_android_public`:

- `README` file, which is a quick start guide to build the demo application.
- `android_agent`, which contains the Android application source code configured to be opened by Android Studio.
- `device_linux`, which are the Linux libraries developed by Ayla for the Android device.
- `mfg`, which are the manufacturing scripts for mass production.
- `scripts`, which has the scripts to build the development environment.

4.2 Step Two: Modify the build.sh Script

This entails a change to the `device_android_public/scripts` directory. In this directory, the `build.sh` script describes the entire project. Currently, the script is written for the demo application. Modify the following fields for your case:

- TARGET_CPU

  This variable describes the CPU type of your Android device. The type may be ARM, MIPS, or x86. We support both 32-bit and 64-bit CPU.

  For example, you describe ARM as:

```
export TARGET_CPU=arm
```

- PACKAGE_NAME

  This specifies the name of the Android Application package; for example, the demo app package is `com.aylanetworks.androidd`, and is assigned as follows:

```
export PACKAGE_NAME=com.aylanetworks.androidd
```

- APP_MODULE_DIR

  Scripts need the location of the application module, so they can install the compile libraries. For example, the demo project has multiple modules, and the module for application is `AppDevice`:

```
export APP_MODULE_DIR=android_agent/AppDevice
```

4.3 Step Three: Run build.sh

To run `build.sh`, change to the `device_android_public/scripts` directory and issue:

```
build.sh -a
```

The script downloads the following software packages from their official sites and compiles the software automatically.

- Android NDK r13b
- zlib v 1.2.11
- OpenSSL 1.0.2k
- curl v 7.53.0
- Jansson v 2.10
- lighted v 1.4.45

Once this completes, you are done building a working environment. The entire process may take a while. The compiled libraries are installed in the module you specified in `APP_MODULE_DIR`.

`build.sh` provides these functionalities:

-a	Builds all libraries, including third party, device Linux libraries, and tools for LAN mode
-n	Generates NDK toolchain
-s	Builds third-party support libraries
-d	Builds Ayla device Linux libraries
-l	Generates the LAN mode package
-f	Builds factory tools, the Linux `config_gen` executable

NOTE In most cases, you only need to issue `build.sh -a` once, then you can proceed to Android application development. The other options are used to rebuild portions of the libraries.

5 Design the Android Application

Developers use the Android Device application to define the following:

- Device properties
- How the device should respond to the datapoint received
- What the device should send to the cloud
- How the device deals with the various cloud events

This is achieved by calling Ayla device `lib` APIs. The distribution comes with a sample Android Java application project. Developers can use it as a starting point to speed up development. The required libraries described in Section 4 are already installed in corresponding directories.

Following are the four modules in this project.

- `AppDevice`: the Android device application module. `AndroiAgent.java` is the main application activity.
- `aylaappservice`: the Ayla application service, which runs as an Android background service.
- `ayladeviceservice`: The Ayla device service. An Android background service runs in a separate process.
- `ayladevicelib`: The Ayla device `lib` API. The application calls these APIs to start the application and device services.

These modules are in Android Studio. To locate them, click **File / Open** to open the `android_agent` directory. In most cases, you only need to modify the AppDevice module. The other tree modules function as libraries.

This section provides an overview of the Ayla device and then describes the following components of an Ayla device application:

- Devices
- Properties
- Datapoint
- Event Listener
- Ayla Device Service Callback
- Wi-Fi Connection Handling

5.1 Overview of the Ayla Device

An Ayla Device has two major fields: a list of properties and a set of device listeners.

The list of properties provides the properties defined in the Ayla Cloud and can be created by instantiating `AylaProperty` class. Each `AylaProperty` can have the Receiver method to handle datapoints received from the cloud, use sendDatapoint method to update property values in the cloud, and could have Recovery method to deal with transmission failure.

A datapoint is encapsulated by creating an AylaDatapoint. The datapoint has a property value and metadata. AylaProperty methods send and receive datapoints, and the latest value is cached in AylaProperty.

A device needs several listeners to listen to the cloud status changes (such as connectivity changes), factory reset, registration, and batch send datapoint results. These are all defined as interfaces. Users can refer to the demo code and implement their own.

5.2 Devices

A device is created in the OEM host version, and a valid device config file is provided, as shown in the following example. Refer also to Section 3 on configuring the Android device.

```
device = new AylaDevice(getApplicationContext(),
                              OEM_HOST_VERSION,
                              devdConfigFile);
```

NOTES

- The `OEM_HOST_VERSION` is the template version represented by String.
- The `devdConfigFile` is the file name with a full path of the device configuration file.

An AylaDevice must have a list of properties. Each property is added to the device by calling the `addProperty()` method, as shown in the following example:

```
AylaProperty<String> version = new AylaProperty<>("version",
                                   AylaDataType.STRING,
                                   AylaProperty.DataDirection.FROM_DEVICE);
device.addProperty(version);
```

An AylaDevice also has a set of listeners for various events. Developers can choose to implement those they need.

Finally, to start the device, call `start(Context)` to start the device, as shown in the following example:

```
device.start(AndroidAgent.this);
```

5.3 Properties

Each property has its own name, datatype, and direction. Properties may also have methods to handle the datapoints received. Pertinent details on all of this are described under the following necessary actions for properties:

- Create Property
- Receiver
- Recovery
- Ack Management

Create Property

There are five data types defined in class `AylaDataType`:

- `BOOLEAN`
- `DECIMAL`
- `INTEGER`
- `STRING`
- `FILE`

The property direction is either `TO_DEVICE` or `FROM_DEVICE`.

`AylaProperty` is a generic class, and the Java data types are represented as follows:

`AylaDataType.BOOLEAN`	Boolean
`AylaDataType.DECIMAL`	Double
`AylaDataType.INTEGER`	Integer
`AylaDataType.STRING`	String

For example, a Blue LED property is written as follows:

```
AylaProperty<Boolean>blueLED = new AylaProperty<>)
                   “blue LED”,
                   AylaDataType.BOOLEAN,
                   AylaProperty.DataDirection.TO_DEVICE);
```

The `AylaDataType.FILE` is a special design for file uploading and downloading. The value is the file name with a full path represented by the String Java data type. Following are the steps to upload a file with examples.

1. Create a FILE property, as shown:

```
AylaProperty<String>fileUp = new AylaProperty<>)
                   “fileUp”,
                   AylaDataType.FILE,
                   AylaProperty.DataDirection.FROM_DEVICE);
```

2. Create a datapoint with the file name to send, as shown:

```
AylaProperty<String>datapoint = new AylaDatapoint<>(String.class);
                          Datapoint.setValue(“file name with full path”);
                          fileUp.sendDatapoint(datapoint);
```

To download a file from the cloud, the `FILE` property needs to be a `TO_DEVICE` property. Developers need to call `setDownloadFileName(String filename)` to specify the downloaded file name. The file name should have a full path. Refer to the following example:

```
AylaProperty<String>fileDown = new AylaProperty<>)
                 “fileDown”,
                 AylaDataType.FILE,
                 AylaProperty.DataDirection.TO_DEVICE);

fileDown.setDownloadFileName(
                 “/data/data/com.aylanetworks.androidd/files/file_down”);
```

Receiver

This pertains to taking actions when receiving a datapoint from the cloud. Developers need to implement an `AylaProperty.Receiver` interface and set it to the property by the `setOnReceive()` method. The following is an example of a receiver to control the blue LED on a board.

```
BlueLED.setOnReceive(new AylaProperty.Receiver(){
      @Override
      public boolean exec(AylaProperty) {
              AylaDatapoint datapoint = property.getLatestDatapoint();
              If( (boolean)datapoint.getvalue() )
                              // Turn on blue LED
                      else
                              // Turn off blue LED
               return status;
      }
});
```

NOTES

- `AylaProperty.getLatestDatapoint()` is used to get the datapoint received from the cloud.
- `AylaDatapoint.getValue()` can retrieve the value of the datapoint. (Refer to Section 5.4 for information on datapoints).

Recovery

If the datapoint send to the cloud fails due to issues, such as a loss of connection, you can implement an `AylaProperty.Recovery` interface and assign it to a property using the `setOnRecovery()` method to deal with the issue.

For example, if you have an output property that fails when sending integer values to the cloud, you can implement the following recovery method to re-send the data whenever this issue occurs:

```
Output.setOnRecovery(new AylaProperty.Recovery(){
      @Override
      public boolean exec(AylaProperty property){
                 property.sendDatapoint(property.getLatestDatapoint());
                 return true;
      }
});
```

Or, you can request the datapoint again using `AylaProperty.request()` if the property is a `TO_DEVICE` property. For example, you have a `fileDown` property that receives files sent by the cloud. The `Recovery()` is written as:

```
fileDown.setOnRecovery(new AylaProperty.Recovery(){
      @Override
      public boolean exec(AylaProperty property){
          Log.d(TAG,"Filedownloadingfailed,request it from cloud again);
          property.request();
          return true;
      }
});
```

Acknowledgement (Ack) Management

When a property is set to `Ack Enabled` in the cloud, the Ayla device library wraps the receive datapoint into a JSON object and returns that to the cloud for verification. The whole process is handled by the library automatically. If you want to manage this flow in the application, you can set `setAppManageAck()` to `true` and return the datapoint with additional information using `AylaDatapoint.sendACK()`.

For example, if you want the `blueLED` property to send an acknowledgement with status, you write this as follows:

```
blueLED.setOnReceive(new AylaProperty.Receiver(){
      @Override
      public boolean exec(AylaProperty property){
          property.setAppManagesAck(true);
          AylaDatapoint datapoint = property.getLatestDatapoint();

        if(datapoint != null)
                datapoint.sendAck(0, 100);

        return true;

      }

});
```

NOTES

- The first parameter of `sendAck()` is an Integer status (`0` for success, and other for user defined failure cases).
- The second parameter is an Integer message defined by developers.

5.4 Datapoints

The device and the cloud exchange data through `AylaDatapoint`. The Ayla Device Library routes datapoints to the property for the particular datapoint. This section provides descriptions and examples of receiving and sending datapoints.

5.4.1 Receive Datapoints

Whenever a property receives a datapoint from the cloud, an `AylaDatapoint` is created and cached in the property. You can use `AylaProperty.getLatestDatapoint()` to get the datapoint and `AylaDatapoint.getValue()` to get the value of the datapoint. A datapoint may have metadata attached. You can therefore use `AylaDatapoint.getMetaDataListSize()` to find out the length and `getMetadataKey(index)`, `getMetadataValue()` to retrieve the content. Following is an example:

```
input.setOnReceive(new AylaProperty.Receiver(){
      @Override
      public boolean exec(AylaProperty property){
          AylaDatapoint datapoint = property.getLatestDatapoint();
                Log.d(TAG, "Property value = "+property.getValue());

                for( int I = 0; i<datapoint.getMetadataListSize(); i++){
           Log.d(TAG, "Key = " +getMetadataKey(i)+
                               "Value = "+ getMetadataValue(i));
        }
        return true;

      }
```

5.4.2 Send Datapoints

There are two ways to send datapoints: `sendDatapoint(AylaDatapoint)` to send a single datapoint or `appendDatapoint(AylaDatapoint)` to append datapoints to a batch list. If you use the latter, you then use `AylaDevice.sendBatchedDatapoint()` to send all batched datapoints at once, but first, the developer needs to create an `AylaDatapoint`.

`AylaDatapoint` is a generic type class. Its type can be Java Boolean, Integer, Double, and String. For example, if you have a property called "version", which is a String of the software version number, you can write this as follows:

```
AylaProperty<String> version = new AylaProperty<>(
                              "version",
                        AylaDataType.STRING,
                        AylaProperty.DataDirection.FROM_DEVICE);

Datapoint = new AylaDatapoint<>(String.class, "android_demo 1.1.0.beta"));
```

To send the version property immediately, you issue the following:

```
version.sendDatapoint( datapoint );
```

If you want to append several datapoints to a batch list, you issue the following:

```
version.appendDatapoint( datapoint_1 );
output.appendDatapoint( datapoint_2 );
blueButton.appendDatapoint( datapoint_3 );
          :
          :
 // These datapoint can be sent out all at once by

AylaDevice.sendBatchedDatapoints(AylaOpOptions);
```

NOTE `AylaOpOptions` is an object to configure how you want your datapoint sent. For example, `AylaOpOptions.setDests()` is to set the destination, and `AylaOpOptions.setConfirm()` is to request confirmation from the cloud. Refer to Section 8.5 for additional information on AylaOpOptions.

5.5 Event Listener

The Ayla Device Library provides several listener interfaces for device events. Developers need to implement those interfaces to get notifications. This section provides information and examples of the following listeners:

- Connectivity Listener
- Factory Reset Listener
- Registration Listener
- Exit Listener
- Batch Confirm Listener

Connectivity Listener

When the cloud connection status changes, the listener gets an update, as shown below:

```
public interface onConnectivityListener{
     boolean exec(AppConnectionType type, Boolean up);
}
```

NOTES

- The first parameter indicates the type of connection, which is either `AylaDevice.AppConnection.CLOUD_CONNECT` or `LAN_CONNECT`.
- The second parameter tells you if the connection is dropped.

You set the connection with `setOnConnectivityListener()`, for example:

```
device.setOnConnectivityListener(new AylaDevice.onConnectivityListener(){
        @Override
        public boolean exec(AylaDevice.AppConnectionType type, Boolean up) {
               //write code to handle connection status changes
               return true;
}

});
```

Factory Reset Listener

This listener gets the factory reset event. Developers can write the code to reset the device in this listener using the following interface:

```
public interface onFactoryResetListener{
     boolean exec();
}
```

The interface has no parameters. Developers need to do a factory reset sequence in method exec():

```
device.setOnFactoryResetListener(new AylaDevice.onFactoryResetListener(){
        @Override
        public boolean exec() {
               Log.d(TAG, “factoryReset()”);
               return true;
        }

});
```

Registration Listener

This listener gets the registration event using the following interface:

```
public interface onRegistrationListener{
     boolean exec(Boolean registered);
}
```

The parameter “registered” is the status reported from the cloud; for example:

```
device.setOnRegistrationListener(new AylaDevice.onFactoryResetListener(){
        @Override
        public boolean exec() {
         Log.d(TAG, “registration status “ + registered.toString());
         return true;
     }

});
```

Exit Listener

This listener gets the exit event using the following interface:

```
public interface onExitListener{
     boolean exec(int status);
}
```

The parameter registered is the Exit status reported from the cloud; for example:

```
device.setOnExitListener(new AylaDevice.onFactoryResetListener(){
        @Override
        public boolean exec(int status) {
         Log.d(TAG, "exit status " + status);
         return true;
     }

});
```

Batch Confirm Listener

You can send the datapoints of several properties as a batch to minimize the number of transactions. This listener is used to check if the batch was sent to the cloud successfully. Developer needs to implement `onBatchConfirmListener` interface and assign it by calling `setOnBatchConfirmListener`, as shown in the following example:

```
public interface onBatchListener{
     AylaConfirmInfo.Error exec(Integer batchld,
                                    AylaOpOptions opOptions,
                                    AylaConfirmInfo confirmInfo);
}
```

Following is an example of the Batch Confirm Listener:

```
device.setOnBatchConfirmListener(new AylaDevice.onBatchConfirmListener(){
    @Override
    public AylaConfirmInfo.Error exec(Integer batchId, AylaOpOptions opOptions,
                                         AylaConfirmInfo confirmInfo) {
            // Get error status. AylaConfirmInfo.Error.NONE means succeeded
            confirmInfo.getError();

            // Get delivery time
            opOptions.getDevTimeMs()

            // Get destination
            confirmInfo.getDestination()

            return error;
});
```

5.6 Ayla Device Service Callback

Devices can receive several events from Device Controls in the Ayla OEM Dashboard, which are implemented in Ayla Device Service in com.aylanetworks.ayladeviceservice, DevdThread.java

Following are functions to note:

- The `void applyPlatformSetupMode(boolean enable);` is designed to receive `"Toggle Values"'s` boolean value.
- The `void resetPlatform()` and `void factoryResetPlatform()` are called when Factory Reset is trigger set in Device Controls panel of the Ayla OEM Dashboard.
- `resetPlatform()` should do a platform-related reset, such as reboot the hardware device. `factoryResetPlatform()` should do the device software-related reset functions.

5.7 Wi-Fi Connection Handling

Whenever a Wi-Fi connection drops and resumes, the application should call `AyaDevice.bindDHCP` to re-establish the connection to the cloud. The sample application listens to Android Wi-FI interface `OFF/ON` events and demonstrates how to reconnect itself to the cloud in `createWiFiStatusListener()`. Developers may have their own Wi-Fi cases to deal with as well, such as when Wi-Fi sleeps and resumes.

6 LAN Mode

LAN Mode is an important feature in the Ayla Solution. This feature enables the device to be controlled by a mobile application directly over the local area network (LAN) when the mobile application and device are on the same network. This results in much faster performance and feedback to the end-user of the mobile application. Additionally, the device can be controlled within the network if Internet connectivity is lost.

The distribution includes a cross-complied light weight web server (lighttpd) and Ayla CGI tool (acgi). The Android application called `AndroidAgent` automatically installs these two tools.

To initiate LAN mode, you need root access rights for the Android device to execute a run script. To do this, issue the following:

```
$ adb shell
$ su
$ /data/data/com.aylanetworks.androidd/files/html/run.sh
```

For mass production, the `run.sh` should be placed in the `init.rc` file of the platform and executed while booting up.

If you want to use a different web server and connect `acgi` to this server, following are the steps:

1. Enable LAN mode on your device template in the Ayla Developer Portal.

NOTE For more information on how to do this, refer to *Ayla Developer Portal User's Guide*.

2. Install a web server on your platform (such as `lighttpd`) and configure the server to allow requests for `*.json` resources.
3. Forward any requests for `*.json` resources, such as `POST /local_reg.json)`, to the the Ayla `acgi` utility. Some web servers require you to have a file with the URI for every request you want to handle (`lighttpd` needs this in the www or www/html directory).
4. Refer to *Ayla LAN Mode Mobile Application Note* to enable LAN mode on your mobile application.

Once you have completed these steps, the mobile application should automatically switch to LAN mode when the mobile application and device are on the same network.

7 Run the Android Device

Before starting the device, you need to make sure that the device configuration file called `devd.conf` is installed in the correct directory. The sample `AndroidAgent.java` reads `devd.conf` from the `/data/local/tmp directory`.

To install the file in this directory:

1. Connect the Android device through the USB.
2. Turn on the USB developer mode of your device. On the host computer, issue:

   ```
   adb push devd.conf /data/local/tmp
   ```

   This command pushes `devd.conf` to /data/local/tmp directory.
3. After that, click the Run button in Android Studio to execute `AndroidAgent.java`.

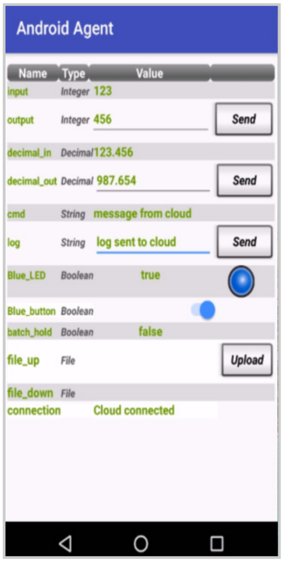
Once you start the device and LAN mode, you can use Same LAN to register the device to your account.

NOTE Refer to *Device Onboarding: Ayla Registration Methods*.

The demo application comes with a simple user interface (UI) to show property values on the device. This is particularly useful for debugging issues. All types of properties are displayed on screen. You can, for example, send the `FROM_DEVICE` property as follows:

1. Enter a value and then press Send.
2. Press Upload to send a file up to cloud.

When device receives a file from the cloud, the `file_down` property blinks on the UI to notify users. The connection status displays at the bottom of the screen, as shown in Figure 2.



Android Agent

Name	Type	Value	
input	Integer	123	
output	Integer	456	Send
decimal_in	Decimal	123.456	
decimal_out	Decimal	987.654	Send
cmd	String	message from cloud	
log	String	log sent to cloud	Send
Blue_LED	Boolean	true	●
Blue_button	Boolean		
batch_hold	Boolean	false	
file_up	File		Upload
file_down	File		
connection		Cloud connected	

Figure 2: Android User Interface

8 Ayla Device Library API

Ayla Device Library (which is `ayladevicelib)` provides APIs for Android applications to use Ayla Device and Application Services. This section includes examples and descriptions of nested classes, public methods, and public constructions for these Ayla APIs.

8.1 AylaDevice

Refer to Section 5.1 for an overview of the Ayla Device. With regard to APIs for AylaDevice:

- public class `AylaDevice`
- extends Object
- com.aylanetworks.ayladevicelib

Nested Classes	

`enum`	`AppConnectionType`
	`CLOUD_CONNECT, LAN_CONNECT`

`interface`	`onConnectivityListener {`
	`    boolean exec(AppConnectionType type, Boolean up);`
	`}`
	This method listens to the connectivity event.

`interface`	`onFactoryResetListener {`
	`boolean exec();`
	`}`
	This method listens to factory reset event.

`interface`	`onRegistrationListener {`
	`boolean exec(Boolean registered);`
	`}`
	This method listens to registration event.

`interface`	`onExitListener {`
	`boolean exec(int status);`
	`}`
	This method listen to the exit event.

`interface`	`onBatchConfirmListener {`
	`AylaConfirmInfo.Error exec(Integer batchId, AylaOpOptions opOptions,`
	`AylaConfirmInfo confirmInfo);`
	`}`

Public Constructors
`public AylaDevice(Context context, String templateVersion, String devdConfigFile)` • `Context` - The context of android application. • `templateVersion` - The version of template that is defined in the cloud. • `devdConfigFile:` - The `devd.conf` file name including the full path.

Public Methods	
`boolean`	`start(Context context)` This starts the AylaDevice.
`boolean`	`stop(Context context)` This stops the AylaDevice.
`boolean`	`addProperty(AylaProperty property)` This adds a property to the device.
`AylaProperty`	`getProperytByName(String propertyName)` This searches for the property by its name.
`boolean`	`sendFromDeviceAylaProperties()` This sends all device properties with directions `FROM_DEVICE` to the cloud. This method is used to synchronize properties in the cloud and device.
`boolean`	`requestToDeviceAylaProperties()` This requests all device properties with directions `TO_DEVICE` from the cloud. This method is used to synchronize properties in the cloud and device.
`boolean`	`requestAylaProperty(String propertyName)` This requests one property from the cloud by name.
`boolean`	`setOnConnectivityListener(onConnectivityListener listener)` This sets `onConnectivityListener` to device.
`boolean`	`setOnFactoryResetListener(onFactoryResetListener listener)` This sets `OnFactoryResetListener` to device.
`boolean`	`setOnRegistrationListener(onRegistrationListener listener)` This sets `OnRegistrationListener` to device.

Public Methods	
`boolean`	`setOnExitListener(onExitListener listener)` This sets `OnExitListener` to device.
`boolean`	`setOnBatchConfirmListener(onBatchConfirmListener listener)` This sets `OnBatchConfirmListener` to device.
`boolean`	`isFirstConnection()` This checks to make sure that the device is connected for the first time. Usually, the device needs to sync its properties with the cloud.
`boolean`	`setFirstConnection(boolean status)` This sets a `FirstConnection` flag in the device.
`int`	`bindDHCP(Context context)` This obtains the IP address from DHCP. This function is used when the Wi-Fi interface is dropped and then put back on-line. The device needs to get the IP address again and reconnect itself to the cloud.
`int`	`sendBatchedDatapoints(AylaOpOptions opOptions)` A property can use `appendDatapoint( datapoint )` to save a datapoint in a batched list. This method sends out all batched datapoint. The `OpOptions` is used to set delivery time, destination, and confirmation options. The result is informed by `onBatchConfirm` method.

8.2 AylaProperty

Refer to Section 5.3 for details on the Ayla Property. With regard to APIs for AylaProperty:

- public class `AylaProperty<T>`
- extends Object implements Serializable
- com.aylanetworks.ayladevicelib

To respond to a datapoint sent from the cloud, you need to implement a Receiver and set it using `setOnReceive(Receiver)`. After that, you can use `onReceive()` to invoke it.

There are two ways to send a datapoint:

- Using `sendDatapoint(AylaDatapoint)` to send a single datapoint.
- Using `appendDatapoint(AylaDatapoint)` to append a datapoint to a batch list, and then `AylaDevice.sendBatchedDatapoint()` to send all batched datapoints at once.

Nested Classes	
`enum`	`Destination` The values are: • `ALL` – to send the datapoint to all destinations. • `CLOUD` – to send the datapoint to the cloud. • `LAN` – to send the datapoint to the LAN.

`interface`	`Receiver {`
	`boolean exec(AylaProperty property);`
	`}`

`interface`	`Recovery {`
	`boolean exec(AylaProperty property);`
	`}`

`enum`	`DataDirection {`
	`TO_DEVICE,`
	`FROM_DEVICE`
	`}`

Public Constructors
`public AylaProperty(String name, AylaDataType type, DataDirection direction)` • `name` - The name of the property. • `type` - The data type of the property. • `direction` – This can be `TO_DEVICE` or `FROM_DEVICE`.

Public Methods	
`string`	`getName()` This returns the name of the property.
`AylaDataType`	`getType()` This gets the datatype of the property.
`DataDirection`	`getDirection()` This gets the direction of the property.
`AylaDatapoint<T>`	`getLatestDatapoint()` The latest datapoint received is cached in the property, and use this method to get it.
`boolean`	`setOnReceive(Receiver receiver)` This sets the Receiver method to property.
`boolean`	`setOnRecovery(Recovery recovery)` This sets the Recovery method to property.

Public Methods	
`boolean`	`sendDatapoint(AylaDatapoint datapoint)` This sends a single datapoint to the cloud.
`boolean`	`appendDatapoint(AylaDatapoint datapoint)` This appends the datapoint to a batch.
`boolean`	`request()` This requests the cloud property value. If the property has a Receiver, it is called.
`boolean`	`setAppMangesAck(boolean ack)` When `ack` is set to `true`, the property needs to handle `Ack` by itself. The benefit is that you can issue `AylaDatapoint.sendAck` to send extra information.
`boolean`	`setDownloadFileName(String file)` This sets the download file name with a full path.
`AylaConfirmInfo.Error`	`confirm(AylaOpOptions, AylaConfirmInfo confirmInfo)` When a datapoint sent to the cloud requires confirmation, developers can issue `AylaDatapoint.requireConfirm(true)` and this `confirm()` method receives the status from the cloud. This method only prints `.status`. You may override it for your own error handling.

8.3 AylaDatapoint

AylaDatapoint is the class to store property's value and metadata. Refer to Section 5.4 for additional details on Ayla Datapoints. With regard to APIs for AylaDatapoint:

- public class `AylaDatapoint<T>`
- extends Object
- com.aylanetworks.ayladevicelib

Public Constructors

`public AylaDatapoint(Class<T> clazz)`
`public AylaDatapoint(Class<T> clazz, T value)`
• AylaDatapoint is generic type.
• clazz : the class of type, for example Integer.class, Double.class
• value: The value of datapoint

Public Methods	
`T`	`getValue()` This gets the value of the datapoint.
`boolean`	`setValue(T value)` This sets the value of the datapoint.
`boolean`	`setDestination(Destination destination)` The destination can be ALL, CLOUD, LAN.
`Destination`	`getDestination()` This gets the destination of the datapoint.
`boolean`	`setDeliveryTimeMs(long deliverTimeMs)` This sets the delivery time. When this method is omitted, the time is set to the system time automatically.
`boolean`	`addMetadata(String key, String value)` This adds metadata to the metadata list of datapoints.
`int`	`getMetadataListSize()` This gets the number of metadata.
`String`	`getMetadataKey(int index)` This gets the Key string of metadata.
`String`	`getMetadataValue(int index)` This gets the Value string of metadata.
`long`	`getDeliverTimeMs()` This gets the deliver time in ms.
`boolean`	`sendAck(Integer status, Integer message)` When it is `AylaProperty.setAppMangesAck(true)`, the application needs to use `sendAck()` to send acknowledgement. The status is `0` for success, and other value as user-defined failure code. The message is an user-defined integer value.
`boolean`	`requireConfirm(boolean confirm)` When datapoint that is sent needs confirmation. Developers can issue `requireConfirm(true)`. The property that sends this datapoint can get confirmation through `AylaProperty.confirm()` method.

8.4 AylaDataType

AylaDataType is defined by the Ayla Cloud. Each type has its own associated Java data type as shown in this section. The AylaDataType is used to instantiate Ayla defined properties. For example, an INTEGER AylaProperty is created as follows:

```
AylaProperty<Integer> input =
      new AylaProperty<>("input",
                               AylaDataType.INTEGER,
                               AylaProperty.DataDirection.TO_DEVICE);
```

With regard to APIs for AylaDataType:

- public enum `AylaDataType`
- com.aylanetworks.ayladevicelib

AylaDataType	Represented Data Type in Java
INTEGER	Integer
BOOLEAN	Boolean
DECIMAL	Double
STRING	String
FILE	File

8.5 AylaOpOptions

AylaOpOption is a class designed to store information like delivery time, destination, and confirmation options. AylaOpOption is usually used along with datapoint; for example, as the parameter of `AylaDevice.sendBatchedDatapoints()`.

With regard to APIs for AylaOpOptions:

- public enum `AylaDataType`
- com.aylanetworks.ayladevicelib

Public Constructors
`public AylaOpOptions()` The values of AylaOpOptions are set through the methods.

Public Methods	
`void`	`setDests(AylaDatapoint.Destination dests)` This sets the destination of datapoint in AylaOpOptions object. The value can be: • AylaDatapoint.Destination.ALL AylaDatapoint.Destination.CLOUD • AylaDatapoint.Destination.LAN
`long`	`getDevTimeMs()` This gets the delivery time in ms.

`void`	`setConfirm(boolean confirm)`
	
	`setConfirm(true)` is used to ask for confirmation from the cloud. For batch sends of datapoints, the confirmation is retrieved through `AylaDevice.onBatchConfirmListener().`

8.6 AylaConfirmInfo

The AylaConfirmInfo object is created by Ayla Device library. Developers do not need to instantiate the object. They only need to read its values. With regard to APIs for AylaConfirmInfo:

- public class `AylaConfirmInfo`
- extends Object
- com.aylanetworks.ayladevicelib

Nested Classes	
`enum`	`Error` The values are: • `NONE` – no error. • `CONNECTION` – connection failure. • `APPLICATION` – application error. • `UNKNOWN` – unknown error.

Public Methods	
`AylaDatapoint.Destination`	`getDestination()` This gets information on the destination of the datapoint.
`Error`	`getError()` This method gets the error code return from the cloud.

9 Manufacturing Script

In `mfg/ad_setup`, `ad_setup.py` is a sample Python script for deploying device configuration files (`devd.conf`) to multiple Android devices. This script runs on the host machine and the device configuration file (`devd.conf`) is generated in the device through serial port at the runtime.

Since Android devices vary among vendors, you may need to modify the script for your own case. For example, some devices require root privilege to write through serial port, in which case, you need to modify the script to provide the root password.

Following are basic requirements:

- Python and pySerial must be installed on the host machine.
- There has to be serial port access on the device. (If you prefer to USB, one suggestion is to use the Google adb tools by issuing `adb push .conf file` to the /data/local/tmp directory in the device.)
- You need device config .conf files, which are generated by the Linux `config_gen` utility. You can issue `scripts/build.sh -f` to build the Linux tool. Please refer to /device_linux/rel/host_util/config_gen/README for detail usage.

To use `ad_setup.py`:

1. Put the .conf files in ./dsn directory
2. Run `ad_setup.py`. The .conf file is written to the device and the used .conf files are moved to ./used directory.
3. Run `ad_setup.py` again to write the next .conf file.

Ayla Networks

4250 Burton Drive, Santa Clara, CA 95054
Phone: +1 408 830 9844
Fax: +1 408 716 2621