Lesson 3: Afero + Arduino

In Lesson 2 we looked at a Device Profile that used only GPIO attributes, and so ran without an external MCU. In this lesson we'll add an Arduino as an example of an MCU that communicates with Afero Secure Radio. This will demonstrate how you can incorporate ASR into a more complex product.

This project will provide the mobile user an on/off control for LED blinking. When the user taps ON, an MCU attribute is set, which tells the Arduino to start a loop that blinks the LED on and off. This blinking will continue until the app user taps the OFF control to halt it.

We'll run through the example, then take a closer look at how it all works.

If you need code examples that use the deprecated C++ afLib, download the archive file (.zip).

Before You Begin

Be sure you've done the following before starting the steps below:

- You've downloaded, installed, and signed in to the Afero mobile app and the Afero Profile Editor.
- You have an Afero Modulo board.
- You've got an Arduino Uno plus an Afero Plinto shield OR an Arduino Teensy, and you've connected the Afero Modulo to your Arduino. Refer to the Data Sheet appropriate for your Modulo if needed.
- You've got the Arduino IDE (1.8 or later) up and running.

The Steps

1 Download and install afLib2 for Arduino:

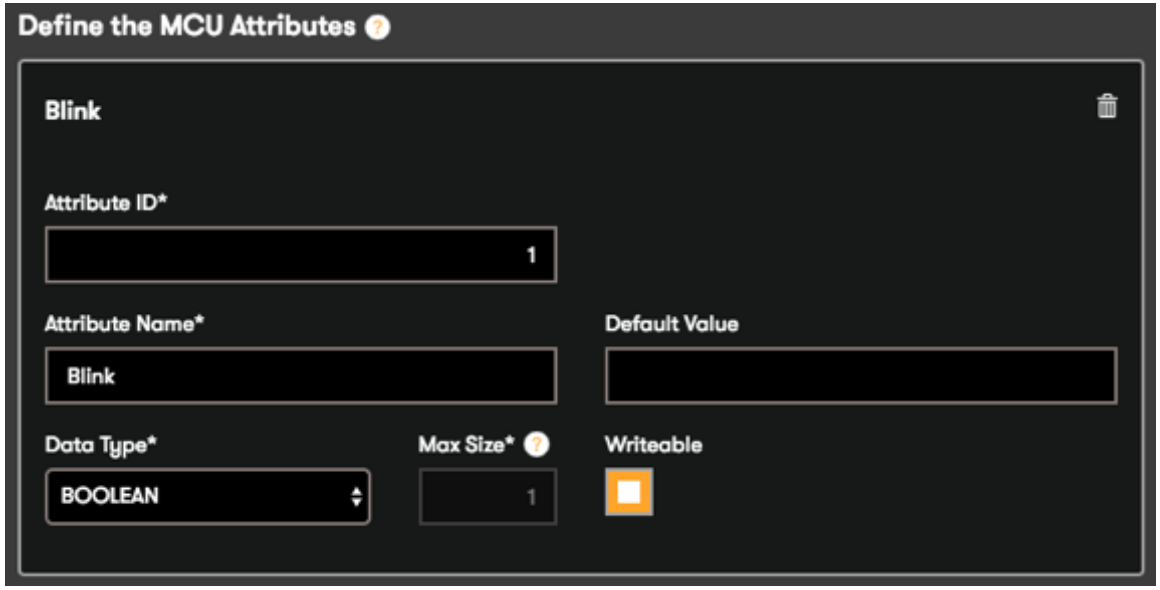
   a. You can obtain afLib2 by going to http://github.com/aferodeveloper/afLib2.
   b. Follow your IDE instructions on how to install the library.
   c. afLib2 contains an Examples directory. In this directory, you'll find an Arduino sketch as well as a directory containing an Afero Device Profile that can be published to your Modulo.
   d. The examples live in your Documents directory under Arduino/libraries/afLib2/examples/.
2. If you haven't already done so, register your Modulo to your account by scanning the QR code on your Modulo using the Afero mobile app.
3. Load the afBlink profile in Afero Profile Editor:
   a. From the Profile Editor start page, select the OPEN button.
   b. In the Open dialog, navigate to Arduino/libraries/afLib2/examples/afBlink/profile/afBlink/profile.
   c. Open the afLib2 device profile that is appropriate for your Modulo: The directory named "afBlink" contains a profile for the Modulo-1; the directory named "afBlink" is for the Modulo-2.
4. From the Afero mobile app, make sure your Modulo is connected.
5. Go to the PUBLISH tab in the Afero Profile Editor and check that your device is online and selected.
6. Click PUBLISH. The profile will be uploaded over the air and in about a minute you should see the UI on your smartphone update to the new profile UI.
7. Now that the Modulo is all set, let's update the Arduino:
   a. Open the Arduino IDE and from the File menu, select EXAMPLES > AFLIB > AFBLINK.
   b. Make sure the BOARD and PORT are set correctly in the Tools menu.
   c. With the afBlink sketch open, select UPLOAD from the Sketch menu.
   d. Once the sketch has uploaded, open the Serial Monitor to see output from the example.
8. Open the Afero mobile app and have some fun controlling your LED!

How It Works

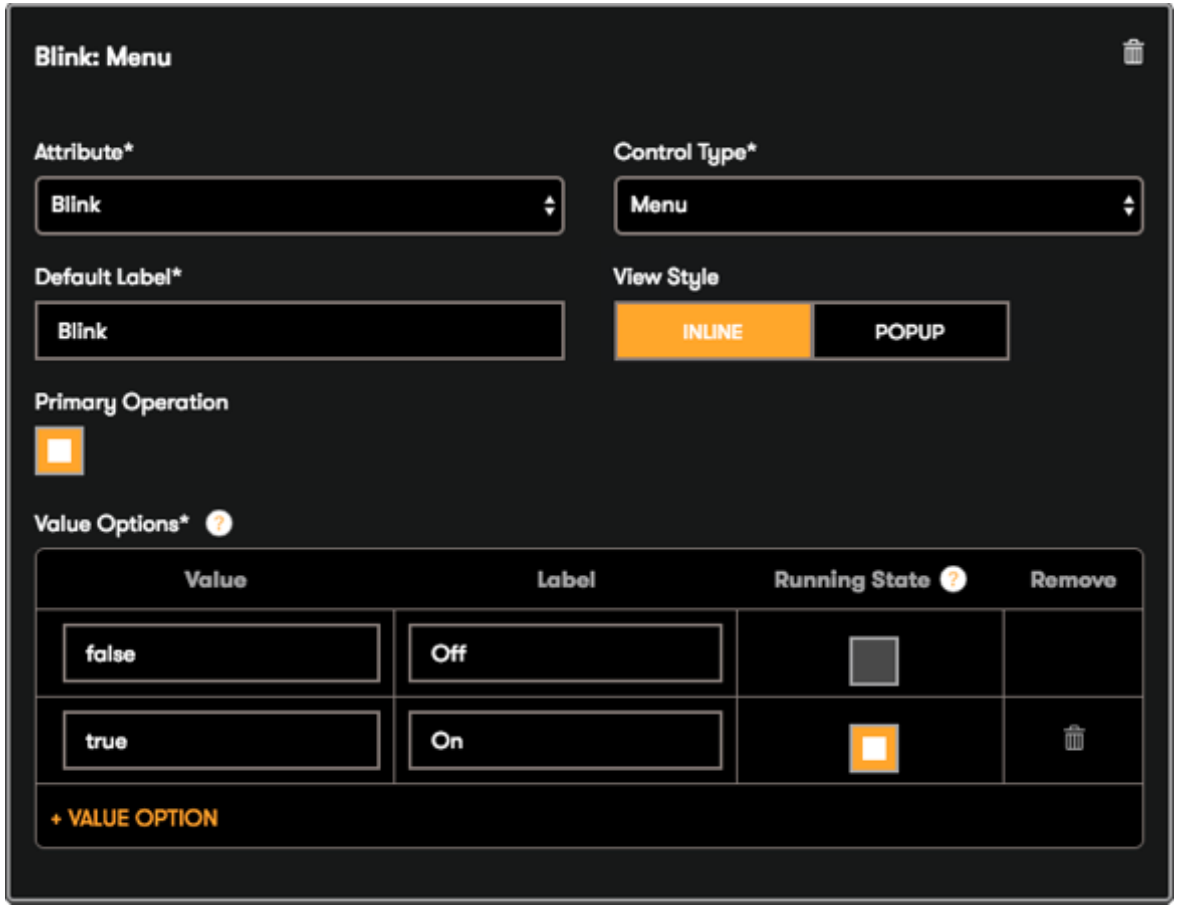
This section gives you a little more insight into what's happening behind the scenes.

The Device Profile and the App UI Work Together

The Device Profile in this lesson has the same GPIO attribute definitions as in Lesson 2, but has an additional MCU attribute. This MCU attribute is a Boolean, made WRITEABLE so that clicks in the mobile app UI can set the attribute value. We've named that attribute "Blink" because it will turn on/off the blinking of the LED.



Note that although this Device Profile has three attributes, we define only one UI control: a menu control linked to the Blink attribute. In other words, only the Blink attribute will be exposed to the end-user through the UI.



The UI Control definition should be familiar if you worked through Lesson 2: Creating a Device Profile. It's a Menu control, meaning it has a few discrete states selectable through the UI. It has two VALUE OPTIONS: ON and OFF. And since IS RUNNING is set to the ON state, the device icon will highlight when blinking is on.

The Menu control is the sole member of a UI control group; remember that every UI control must be a member of a UI control group even if it's the only member, as in this case.

As you saw when you ran through the lesson, this Profile results in an app UI consisting of buttons that control the Modulo LED: tap ON and the LED starts blinking; tap OFF and the blinking stops.

What's Happening on the MCU

Recall that when the UI gets a tap on the ON control, a message to the Afero Cloud tells the MCU program to start blinking the LED on the Modulo. Here's the Arduino console output when we tap the ON button in the app UI to start the blinking, and after letting the LED blink a few times, tap OFF to halt:

```
attrSetHandler id: 1 value: 1
attrNotifyHandler id: 1024 value: 0
attrNotifyHandler id: 1024 value: 1
attrNotifyHandler id: 1024 value: 0
attrNotifyHandler id: 1024 value: 1
attrNotifyHandler id: 1024 value: 0
attrSetHandler id: 1 value: 0
```

In the first line of output we see the MCU program logging execution of `attrSetHandler()`. We know from the afLib2 for Arduino API that the MCU runs `attrSetHandler()` when ASR has executed `setAttribute()`. So we know that ASR must have called `setAttribute()` to tell MCU to set attribute 1 to value 1, and MCU is handling that.

Now take a look at the MCU's `attrSetHandler()` definition (as pseudocode):

```
attrSetHandler(attributeId, value) {
   console_print("attrSetHandler id: ", attributeId, " value: ", value);
   // AF_BLINK is defined in device-description.h, which was created when you published your
profile.
   if (attributeId == AF_BLINK) {
      blinking = (value == 1)
   }
}
```

In response to the message from ASR, MCU sets a local variable, `blinking`, to the value of the AF_BLINK attribute.

Back to the console output, where we see several lines containing `attrNotifyHandler`. It's a fair guess that the alternating 0 and 1 reflect that the LED is being blinked.

Again, based on the afLib2 for Arduino API, we know that `attrNotifyHandler()` is executed by the MCU whenever ASR sends an update message about an attribute change. So we deduce that ASR is sending

updates every time it changes the value of GPIO 0 (the LED). The one piece of the puzzle we haven't seen is what's making ASR change that value. One more look at the MCU pseudocode:

```
void loop() {
   pause_seconds(0.5)
   // AF_MODULO_LED is defined in device-description.h, created when you published your
profile.
   if (blinking) {
      af_lib_set_attribute_16(af_lib, AF_MODULO_LED, !last_value)
   } else {
      af_lib_set_attribute_16(af_lib, AF_MODULO_LED, False)
}
```

And there it is: while variable "blinking" is true, the MCU calls `setAttribute()` every 0.5 seconds to set the GPIO attribute to the opposite state. In response to that `setAttribute()` call, ASR updates the attribute, and then sends an update message, which causes the MCU to execute `attrNotifyHandler()`.

In a typical product containing an MCU, any LED indicator in the device would be connected directly to the MCU, whereas in this example we have used the LED on the Modulo. The difference is that in this lesson, the MCU changes the LED state by making a `af_lib_set_attribute()` call, which causes ASR to make the change and send an update; whereas in a product, the MCU would probably set the LED directly. We used this design not only for setup simplicity, but also to emphasize the way attributes are affected by making `af_lib_set_attribute()` calls.

The flow above illustrates the basic messaging pattern:

1. A user action on the mobile app UI becomes a message to set the value of an attribute on a specific device.
2. The app sends the "set attribute value" message to the Afero Cloud, which broadcasts the message.
3. The ASR for the targeted device receives the message that the attribute value should be set.
4. ASR does a couple of things:
    a. Stores the attribute's current value and the new desired value.

    b. Tells the MCU that the attribute value should be set to the desired value.

5. When the MCU gets the message, `attrSetHandler()` executes. In that call, you must write code to enable the MCU to make a state change that will corresponds to the desired attribute value. This will typically involve some device action (e.g., starting LED blinking).
6. After `attrSetHandler()` runs, afLib2 informs ASR, which then:
    a. Stores the new current value of the attribute, which should equal the desired value.
    b. Sends the attribute value back to the Afero Cloud.
7. The Afero Cloud broadcasts the new attribute value.
8. The mobile app receives the broadcast and updates the UI, so the end-user knows the request has been filled.

System Attributes

Up to this point, we've confined ourselves to discussing attributes that you, the developer, define using the Afero Profile Editor. It turns out that every Device Profile you define also includes several other attributes defined automatically by the system. These are called **system attributes**.

Attribute Types and ID Ranges

As you know, when you author an MCU sketch for Afero Secure Radio, you must include the `device-description.h` file generated by the Afero Profile Editor. The `device-description.h` file consists of #defines for all attributes, both user-defined and system-defined. Different types of attributes are organized into ranges based on ID. Let's take a look in that file:

```
#define ATTRIBUTE_TYPE_SINT8                                    2
#define ATTRIBUTE_TYPE_SINT16                                   3
#define ATTRIBUTE_TYPE_SINT32                                   4
#define ATTRIBUTE_TYPE_SINT64                                   5
#define ATTRIBUTE_TYPE_BOOLEAN                                  1
#define ATTRIBUTE_TYPE_UTF8S                                   20
#define ATTRIBUTE_TYPE_BYTES                                   21
#define ATTRIBUTE_TYPE_FIXED_16_16                              6

// Attribute Blink
```

```
#define AF_BLINK                                                    1
#define AF_BLINK_SZ                                                 1
#define AF_BLINK_TYPE                              ATTRIBUTE_TYPE_BOOLEAN

// Attribute Modulo LED
#define AF_MODULO_LED                                            1024
#define AF_MODULO_LED_SZ                                            2
#define AF_MODULO_LED_TYPE                          ATTRIBUTE_TYPE_SINT16

//...snip...//

// Attribute Command
#define AF_SYSTEM_COMMAND                                       65012
#define AF_SYSTEM_COMMAND_SZ                                        4
#define AF_SYSTEM_COMMAND_TYPE                      ATTRIBUTE_TYPE_SINT32

// Attribute ASR State
#define AF_SYSTEM_ASR_STATE                                     65013
#define AF_SYSTEM_ASR_STATE_SZ                                      1
#define AF_SYSTEM_ASR_STATE_TYPE                     ATTRIBUTE_TYPE_SINT8

//...snip...//
```

In the sample above, you can see that the file begins with a set of defines that simply provide names for the data types that will be described in the remainder of the file.

Following that, you should see something that looks familiar: the define for the AF_BLINK attribute. We used the name AF_BLINK to refer to attribute #1 in the sketch we developed earlier in this exercise. At this point we'll note two features of this attribute:

- The Blink attribute you defined using the Profile Editor is an MCU attribute, and
- The attribute ID is 1.

It turns out that any MCU attributes you define will have ID numbers from 1 to 1023. Of course, you should use the #define names for the attributes and not their ID numbers, but we raise this point here because it defines the number of MCU attributes you can create, and to introduce the fact that different types of attributes have different ID ranges.

After the definition of AF_BLINK, you see another attribute from the profile you created: AF_MODULO_LED. This is one of the GPIO attributes you defined in your profile. GPIO attributes start at ID 1024, and each

GPIO has a pair of attributes: one for the base definition, and one for additional attribute definition data. Thus, GPIO 0 owns IDs 1024 and 1025, GPIO 1 owns 1026 and 1027, and so on.

Starting with ID 2001 and above, you'll see attribute definitions that you did not create when you defined your device profile. Above 65000, the attributes have names that start with "AF_SYSTEM_". These are the system attributes. We won't describe all of the system attributes here, though most have names that explain their purpose clearly enough. In general, you can ignore these attributes, but because they are defined in the `device-description.h`, you can access them in your sketch. In fact, one of these attributes is critically important for you as the author of MCU code: the AF_SYSTEM_ASR_STATE attribute.

The AF_SYSTEM_ASR_STATE Attribute

In most cases, you can ignore the system attributes, but when your project includes an MCU, you'll need to pay attention to the AF_SYSTEM_ASR_STATE (a.k.a. ASR_STATE) attribute. That's because this attribute is used to provide your MCU code important status information about your ASR in real time.

The ASR_STATE attribute can have one of a small range of values:

- 0 = Rebooted
- 1 = Linked
- 2 = Updating
- 3 = Update Ready to Apply (Reboot Requested)

And that last value is our primary interest here: the status "Reboot Requested" means that ASR has received an over-the-air (OTA) software update, and requires rebooting for that update to be installed. If your project does not include an MCU, then the reboot will execute automatically, as soon as possible after the update has been received. However, if your project includes an MCU, then responsibility for triggering the reboot falls on the MCU code. This allows your MCU to restrict the ASR reboot to times that are safe for your application.

Responding to a Reboot Request

So, you need to watch for reboot requests, and you need to respond by signaling ASR to reboot – how exactly do you go about this? Well, of course, this is all about communication via attribute values!

1. When ASR receives an OTA, it signals receipt by changing the value of the ASR_STATE attribute.
2. As we saw earlier in the example, whenever ASR sends an update message about an attribute, the MCU executes the `attrNotifyHandler()` callback.
3. So first thing, we'll need to use our `attrNotifyHandler()` code to watch for a change to AF_SYSTEM_ASR_STATE, and specifically, for that attribute to change to value 3.
4. Once we see that condition, we'll want to write something into our `attrNotifyHandler()` to trigger an ASR reboot. How can we do that? Set an attribute! It turns out that there's another system attribute, AF_SYSTEM_COMMAND, that provides a way to trigger that reboot.

Here's a detailed example:

```
#define AF_MODULE_STATE_REBOOTED       0
#define AF_MODULE_STATE_LINKED         1
#define AF_MODULE_STATE_UPDATING       2
#define AF_MODULE_STATE_UPDATE_READY   3

#define AF_MODULE_COMMAND_REBOOT       1

void attrNotifyHandler(const uint8_t requestId, const uint16_t attributeId, const uint16_t
valueLen, const uint8_t *value) {

        switch (attributeId) {
                // snip //

                case AF_SYSTEM_ASR_STATE:
                        Serial.print("ASR state: ");
                        switch (value[0]) {
                                case AF_MODULE_STATE_REBOOTED:
                                        Serial.println("Rebooted");
                                        break;

                                case AF_MODULE_STATE_LINKED:
                                        Serial.println("Linked");
                                        break;

                                case AF_MODULE_STATE_UPDATING:
                                        Serial.println("Updating");
                                        break;

                                case AF_MODULE_STATE_UPDATE_READY:
```

```
                                        Serial.println("Update ready - rebooting");
                                        af_lib_set_attribute_32(af_lib, AF_SYSTEM_COMMAND,
AF_MODULE_COMMAND_REBOOT);
                }
                                        break;

                                default:
                                        break;
                        }
                        break;

                default:
                        break;
        }
}
```

In the above definition of `attrNotifyHandler()`, we check the supplied attribute ID. If that ID is AF_SYSTEM_ASR_STATE, we check the attribute value. For values 0-2, we simply print helpful information, but if the value is 3 (AF_MODULE_STATE_UPDATE_READY), then we know we've been asked to reboot ASR. We trigger that update by calling `af_lib_set_attribute_32()` for the AF_SYSTEM_COMMAND attribute, with value 1 (which is the value that signals a reboot). Note that in our simple script, we have triggered the reboot as soon as requested, but in a more complicated project we might wait until we've completed an ongoing operation, or are in some idle state or similar.

Updated June 6, 2018