Programming Notes

Integrating MIT App Inventor 2 with Particle products.

Author: Bob Glicksman, date: 12/27/2017

Contents

1. Introduction.

This project is about building apps using MIT App Inventor 2 that can communicate with Particle's Photon or Electron modules. Information about Particle and their products can found at: https://www.particle.io/. Information about MIT App Inventor 2 can be found at: http://ai2.appinventor.mit.edu.

2. Overview.

Particle produces devices and modules designed for Internet-of-Things (IoT) applications. The Particle Photon is a small, inexpensive module that contains an advanced 32 bit microcontroller and WiFi communication capability. The Particle Electron is similar to the Photon except that it communicates over the Internet using 2G or 3G cellular data rather than WiFi. The user can program both the Photon and the Electron in an Arduino-like language. Particle provides both a Web IDE and a Desktop IDE and the ability to "flash" firmware onto your registered Particle devices over the Internet.

Particle makes Internet connectivity easy via Particle's "Cloud" service. The Cloud is free to use for people who own Particle devices. Particle's Cloud service allows users and developers to "flash" their firmware onto their devices over the Internet, so that firmware can be updated from anywhere that there is Internet access. Particle's Cloud service also provides a means for any Internet connected app to communicate with a user's registered Particle devices, over the Internet, even when the Particle device in question is NATed to the Internet and behind a firewall.

Particle uses an encrypted communication link for device-to-Particle Cloud communication. Particle's firmware "system" embeds Particle Cloud communication capability into every Particle device. The user can have their Particle device communicate over the Internet using Particle's firmware IDE (either the web or desktop versions) using simple high level constructs. Low level communication capabilities are also available, but these will not be covered in this project.

What this project is about is understanding what a user needs to write in their Particle device firmware in order to communicate with an external app and how to code MIT App Inventor 2 apps to integrate with Particle devices.

The important takeaway here is that your Particle device integrates with the Particle Cloud and that your app must communicate with your Particle device using Particle's REST API to the Particle Cloud. These concepts are explained below.

3. Particle Firmware Internet Communication.

Particle provides four high level Internet connectivity functions: (a) expose a variable on the device to the Cloud, (b) call a function on the device from the Particle Cloud (and optionally return the result), (c) publish and event to the Particle Cloud, and (c) subscribe to a Particle Cloud event. This project deals only with the first two capabilities, where the app is running and the user is controlling communication with the Particle device(s). Publish/subscribe capabilities are useful to communicate with a server that is always listening for published events and always sending notifications to devices that subscribe to server events, as opposed to an app that is under user control.

In order for an app to communicate with Particle devices in this manner, the user has to write some firmware on the Particle device and the user has to write some software on the app to go along with the device firmware. The firmware that you need to write integrates your device with the Particle cloud. The code that you need to include in your app needs to use Particle's REST interface to communicate with the Particle Cloud. Don't be concerned if you do not know what a REST interface is or how to code one in MIT App Inventor 2. That is what this project is all about!

4. Particle Device Firmware for Internet Communication.

4.1.Overview.

Particle's documentation about their firmware Cloud functions can be found at: https://docs.particle.io/reference/firmware/photon/

This project focuses on two of these firmware functions: Particle.variable() and Particle.function().

4.2.Particle.variable()

You use the Particle.variable() declaration in setup() to declare that a variable that you use in your firmware is to be exposed to the Particle Cloud and thereby accessible for your app to read. Per Particle's documentation, you can expose up to 20 variables in your firmware program to the Cloud and each variable can be of type int, double, or String. Only String types have been used in this project example, but all variable values ultimately come to your app as text (strings) anyway, owing to the nature of the REST interface.

Generally, variables that you expose to the Cloud should be global variables, i.e. variables that are declared outside of any function and are accessible anywhere within your firmware program. The variables that you expose to the Cloud are no different in any way than are any other global variables that you declare and use in your firmware. In order to make a variable in your firmware into a Cloud accessible variable, you simply add the statement *Particle.variable("external name", internalName)* somewhere within setup(). "*external name*" is simply a string literal that you will use in your app to designate the variable that you want to read over the Internet. *internalName* is the name of the global variable that you gave that variable when you declared it in your firmware.

The Particle firmware for this project is in the file called Test_MIT.ino within this repository. You can examine this file using any text editor. If you do so, you will see the line of code *Particle.variable("message", message)* around line 46. This line of code exposes the global variable to the Cloud that was declared in line 30 of the firmware:

String message = version + Time.timeStr() + "Z\n";

In this example code, "version" was a previously declared String constant in line 28 of the firmware and we append to it the system time (the time when setup() is run, thus the time of the last reset of the Particle device) and the constant String " *Z\n*" which denotes that the time is in Zulu (UTC) and then adds in a newline at the end of the string. This allows an app to read out both the firmware version number (e.g. to check that the firmware version is compatible with the app version) and the time when the device was last reset (e.g. the app can inform the user that the information supplied by the device is only good from this past time to the present).

4.3.Particle.function()

You use the Particle.function() declaration in setup() to declare that any function that you define anywhere in your firmware be exposed to the Particle Cloud. This means that an external app can call this function at any time and it will execute right then. The Cloud functions that you write in your firmware are exactly like any other function, with a few limitations described below. You can use Cloud functions inside your firmware just like any other function (e.g. call the function from loop()), but you don't need to use the function anywhere else in your firmware if you don't want to (i.e. if you only want to call it from the Cloud).

Functions that you expose to the Cloud must take only one argument and that argument must be of type String. This is not a huge limitation, since you can include multiple fields within your String argument using some delimiter, such as a comma, to separate the fields. An example of how this can be done is in our SIS project; see:

https://github.com/TeamPracticalProjects/SISProject/blob/master/SISSoftware/Firmware/SaratogaSIS.ino

The cloud exposed function "*registrar()*" is found around line 918 in this firmware and it is parsed into its component parts using the "*parser()*" internal function that is found around line 858.

In addition to a single string argument, a Cloud exposed function can return an INT value to the cloud.

Like Particle.variable(), you provide two arguments to *Particle.function("external name", internalName)*. "*external name*" is just a string constant that is the name that your app will use to call this function over the Internet. *internalName* is the function name that you use in your firmware and it need not be the same as the external name exposed to the Cloud.

In the Test_MIT.ino file, you will see two functions declared as Cloud functions at lines 44 and 45. At line 44, *Particle.function("on-off", ledRemote);* declares the internal function "*ledRemote()*" to be Cloud-accessible. *ledRemote()* is coded starting at line 75 in this firmware. However, the MIT app inventor 2 app will refer to it as "*on-off*". *ledRemote* takes a String argument and the firmware tests to see if this string is "on" or "off" (case insensitive). If it is "ON" or "on", the firmware function turns the Photon/Electron D5 LED on and if the argument string is "OFF" or "off" it turns this LED off. If the LED is turned on, the function returns the integer value 1; if the LED is turned off, the function returns the integer value 0, and if some unrecognizable string was sent to this function, the value -1 is returned.

At line 45, *Particle.function("move", servoRemote);* declares the internal function "*servoRemote()*" to be cloud accessible. *servoRemote()* is coded starting at line 89 in this firmware. However, the MIT app inventor 2 app will refer to it as "*move*". *servoRemote* accepts a string argument that is assumed to be an integer and is converted accordingly. It returns -1 if the string argument is not an integer. If the argument is an integer, this value is clamped between the limits of SERVO_MAX and SERVO_MIN in order to protect the servo. The resulting value is sent to the servo to move it to the commanded angle. The function *servoRemote* returns an integer value that is the actual angle that the servo was commanded to, unless the argument was a non-integer in which case the returned value is -1.

4.4. Final Notes about the Particle Firmware (Test_MIT.ino)

This firmware expects an LED (with appropriate current limiting resistor) to be connected to Photon/Electron pin D5. It expects a servo to be connected to pin A5. These pins are chosen for compatibility with the Water Leak Sensor hardware from one of our previous projects (see: https://github.com/TeamPracticalProjects/WaterLeakSensor). If you have been following Team Practical Projects and building out projects, then you already have hardware that you can use for this project. Just compile and flash Test_MIT.ino to your WLD Photon. If you do not have (or don't wish to use) a WLD, you can connect an LED/resistor to D5 and a servo to A5 of a Photon all connected together on a solderless breadboard.

Test_MIT.ino uses the on-board D7 LED on a Photon or Electron to indicate that the firmware is installed and running. It does this by flashing the D7 LED once per second. This is handy for testing. You don't want to be scratching your head about why your app isn't making things happen when the device isn't running this firmware.

After connecting together the hardware and flashing the firmware to it, the Particle Console can (and should) be used to test out that everything is functioning properly. Always do this whenever you are trying out some Cloud communication and it doesn't appear to be working.

The Particle Console (see: https://docs.particle.io/guide/tools-and-features/console/) is a wonderful tool for testing out your Cloud variables and functions to see that the device and firmware are operating exactly as you expect them to operate before you try and integrate with your own Cloud-connected app.

5. MIT App Inventor 2 Software (ParticleTest.aia).

5.1. Getting and preparing the ParticeTest.aia Source Code into MIT App Inventor 2 on your computer.

The MIT App Inventor 2 "source code" for this project is contained in the file *ParticleTest.aia*. You need the source code for two reasons:

1. To understand how to write MIT App Inventor 2 code to communicate with your device, and
2. To put your Particle "access token" and your Particle device's "device ID" into your app.

There are two global variables up at the top of the Blocks window for you to do this. Simply edit the text data in the global variables "deviceID" and "accessToken" accordingly. You can find the deviceID for your Particle devices either in the Particle Console "devices" or in the Particle Web IDE (in the IDE, select the Devices tool at the left of the IDE window and open up the device that you want the ID of). You can find your access token in the Particle Web IDE by clicking on the Settings tool at the left of the IDE window. The deviceID and accessToken are long, complex strings and it is best to copy and paste these directly into the right places in the *ParticleTest.aia* Build window of MIT App Inventor 2.

In order to do this, you first need to copy the file ParticleTest.aia to some place on your computer. Then, you need to open MIT App Inventor 2 on your computer and select Projects/Import project (.aia) from my computer. This will give you complete access to the source code.

5.2. Understanding the Particle Cloud REST Interface.

In order to obtain Cloud variables from your device and in order to call Cloud functions on your device, you need to use Particle's REST interface. REST stands for "Representational State Transfer" and you can read up on this at:
https://en.wikipedia.org/wiki/Representational_state_transfer

However, you do not need to understand REST in order to create apps that communicate with your Particle device. It is sufficient for you to understand that Particle's REST interface uses the *https:* protocol and, specifically, uses the *https:* GET command to obtain the data from a Cloud variable on your device and the *https:* POST command to call Cloud functions on your device.

You use a Web block under "Connectivity" in the MIT App Inventor 2 toolbox to perform these GET and POST operations, using appropriately coded strings. It is strongly recommended that you use different a Web tool for each Cloud variable that you want to read and for each Cloud function that you want to call. This will ensure that events resulting from your GET and POST commands are properly associated with the right command in your app code. So drag as many Web tools to your Designer screen as you need, one for each Cloud variable that you want to access and one for each Cloud function that you want to call. You can leave them as "Web 1", "Web 2" etc. or you can rename them to help you remember which is which when coding in the Blocks editor.

While it is not necessary to understand the details of *https:*, you will benefit from reading about the Particle Cloud API at: https://docs.particle.io/reference/api/

5.3. Obtaining the value of a Particle Cloud variable using GET

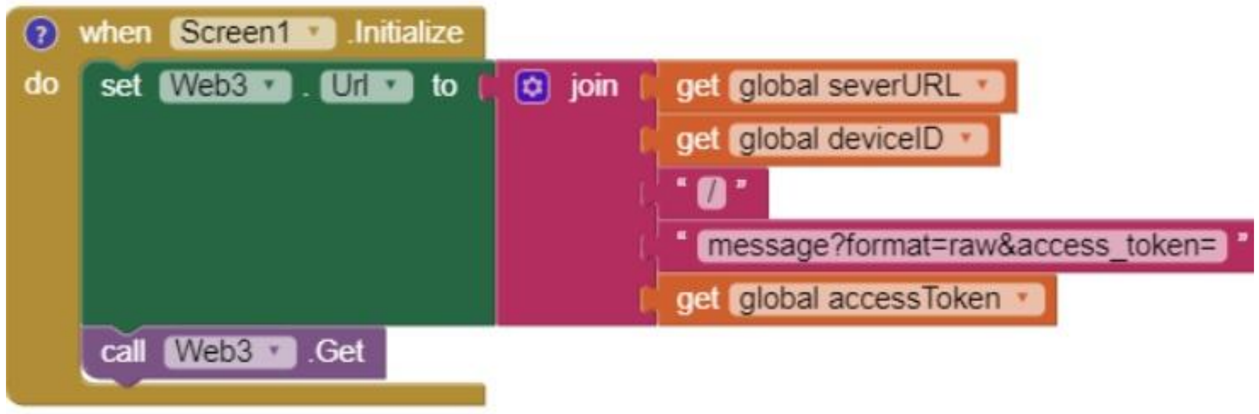
The *https:* GET command requires an appropriately structured string of text to be send over the Internet to the Particle Cloud. The GET string is structured as follows:

https://serverURL/YourDeviceID/variableName?format=raw&access_token=YourAccessToken

where:

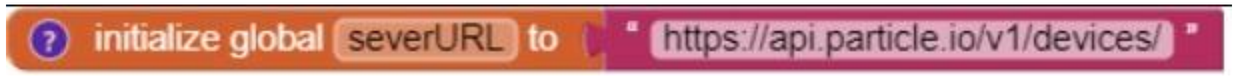
- *server URL* is always: api/particle.io/v1/devices/
- *YourDeviceID* is the Particle device ID that you get from the Particle Console or the Particle Web IDE, as explained above, followed by a "/".
- *variableName* is the Cloud name for the variable that you designed as a Cloud variable in the Particle.variable() declaration in your firmware.
- *?format=raw&* is the literal string "?format=raw&". It tells the Particle Cloud that the data that you are requesting is to be returned in "raw" format, meaning just whatever data is in your Cloud variable (note: if you leave this out, the Particle Cloud will return a JSON formatted object containing a bunch of information that you will then need to parse in your app code).
- *access_token=* is the literal string "access_token="
- *YourAccessToken* is your Particle access token that you obtain from the Particle Web IDE, as explained above.

In order to send this out over the Internet as a GET command, you join all of these pieces to make a long text string and use the appropriate Web block: *set Webx.url to {joined text string that you created above}*. Then you simply call add: *Webx.Get* to issue the GET command over the Internet with this “URL” text string as the argument. For example, if the Web block used is “Web 3”, the following MIT App Inventor 2 block will obtain the value of the Cloud variable that is called “*message*” from your device when the app is first opened:



```
when Screen1 .Initialize
do  set Web3 . Url to  join  get global severURL
                             get global deviceID
                             " / "
                             " message?format=raw&access_token= "
                             get global accessToken
    call Web3 .Get
```

In order to use this code in your app, you must put your device ID and your access token in global variables, as well as have a global variable preset to the Particle Cloud serverURL:



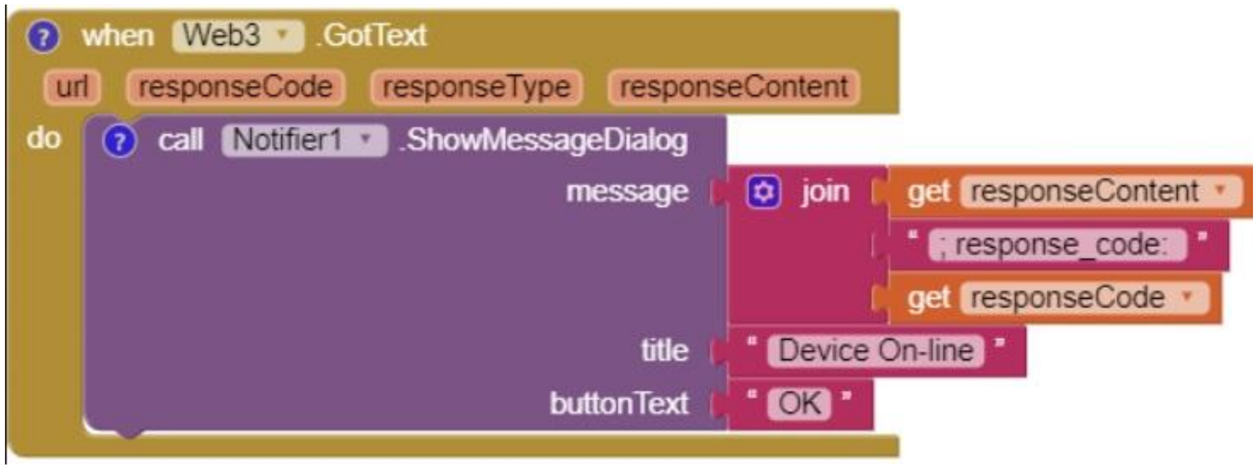
```
initialize global severURL to " https://api.particle.io/v1/devices/ "
```

Your Cloud variable must also be defined in your device firmware using:

Particle.variable(“message”, internalVariableName);

Note that MIT App Inventor 2 requires event driven programming, so you need some event to trigger off the set and call blocks. You can use the app opening event, as shown above, or any other event, such as a button press, shaking the phone or whatever you need.

The above program blocks send the GET request out to the Particle Cloud, but you need an event to tell you when the response is received. You use the “*when Webx.GotText*” as the event of receiving a text response from the Particle Cloud and you code into this block whatever you want to do with this received text. This trigger also provides you with the url that was sent in the GET command, the HTTP response code, the HTTP response type and, most importantly, the “response content” which is the value of the Cloud variable that you wanted to read. For example, the following block of code opens a Notifier when the text response is received from your device and displays the variable value and the response code that was received from the Particle Cloud:


when Web3 .GotText
url responseCode responseType responseContent
do call Notifier1 .ShowMessageDialog
message join get responseContent
"; response_code: "
get responseCode
title "Device On-line"
buttonText "OK"

Note again that if you do not include "*format=raw&*" in the url text string that you send out to the Particle Cloud, the *responseContent* that you get back will not be just the Cloud variable content but will be a JSON object with a number of parameters in it, of which the variable value will be only one.

5.4.Calling a Particle Cloud function using POST

The *https:* POST command is a little bit more complicated than the GET command. The Particle Cloud API requires a url string similar to GET, but you also have to specify your access_token and the Cloud function argument as FORM data. For example, the url part is a string looks like:

https://serverURL/YourDeviceID/functionName?format=raw

where:

- *server URL* is always: api/particle.io/v1/devices/
- *YourDeviceID* is the Particle device ID that you get from the Particle Console or the Particle Web IDE, as explained above.
- *functionName* is the Cloud name for the function that you designated as a Cloud function in the *Particle.function()* declaration in your device firmware.
- *?format=raw* tells the Particle Cloud that the data that you are requesting is to be returned in "raw" format, meaning just whatever data is in your Cloud function return value (note: if you leave this out, the Particle Cloud will return a JSON formatted object containing a bunch of information that you will then need to parse in your app code).
- *NOTE: you do not place your access_token in the url string for a POST. Rather, you put in as form data.*

You then need to construct a string of name/value pairs and send this as "form data" over the Internet to the Particle Cloud. The form data string that you need to create looks like the following:

access_token=YourAccessToken&params=FunctionArgumentString

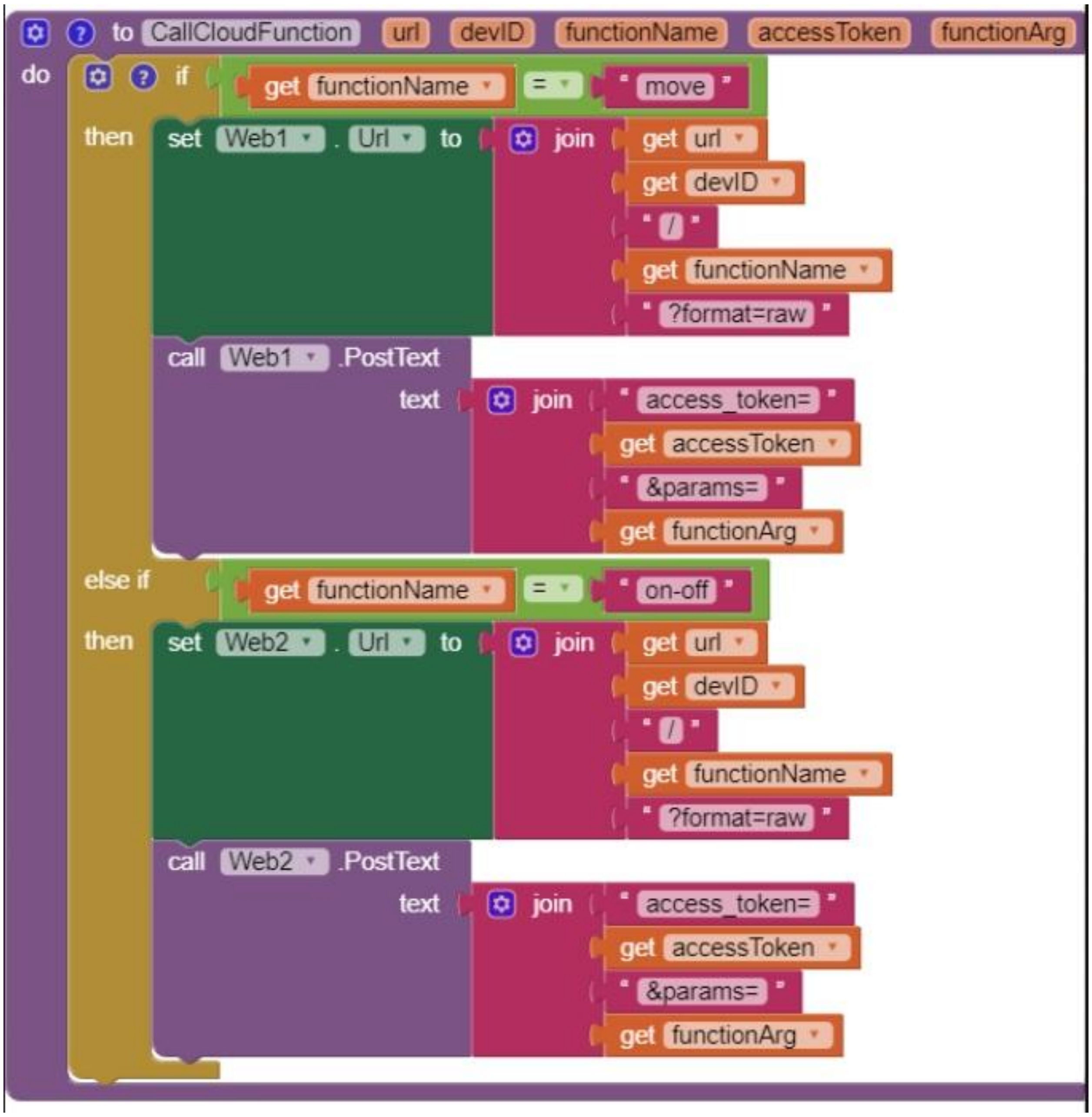
where:

- *access_token=* is the literal string "access_token="
- *YourAccessToken* is your Particle access token that you obtain from the Particle Web IDE, as explained above.
- *&params=* is the literal string "&params="
- *FunctionArgumentString* is the string variable that your Cloud function expects as its argument.

The following MIT App Inventor 2 block of code is a procedure called "CallCloudFunction" that accepts a number of arguments and calls the appropriate Particle Cloud function on your device based upon the *functionName* argument. In this example, two functions are supported inside of this procedure: one is called "*move*" and the other is called "*on-off*". You may expand and use this procedure for all of your Particle Cloud function calls, or you may wish to code up each Particle Cloud function calls as its own procedure (without the IF-ELSE control block). The choice is up to you.

Referring to the procedure below, the following arguments must be passed into the procedure:

- *url* is the literal string "api/particle.io/v1/devices/", which can be placed in a global variable if you wish.
- *deviceID* is the Particle device ID of the device that you wish to make function calls on via the Particle Cloud
- *functionName* is the Cloud name for the function that you designated as a Cloud function in the *Particle.function()* declaration in your firmware.
- *accessToken* is your Particle access token that you obtain from the Particle Web IDE, as explained above.
- *functionArg* is the string variable that your Cloud function expects as its argument.



```
to CallCloudFunction url devID functionName accessToken functionArg
do
  if      get functionName = "move"
  then    set Web1.Url to join get url
                               get devID
                               "/"
                               get functionName
                               "?format=raw"
          call Web1.PostText
                    text join "access_token="
                              get accessToken
                              "&params="
                              get functionArg
  else if get functionName = "on-off"
  then    set Web2.Url to join get url
                               get devID
                               "/"
                               get functionName
                               "?format=raw"
          call Web2.PostText
                    text join "access_token="
                              get accessToken
                              "&params="
                              get functionArg
```

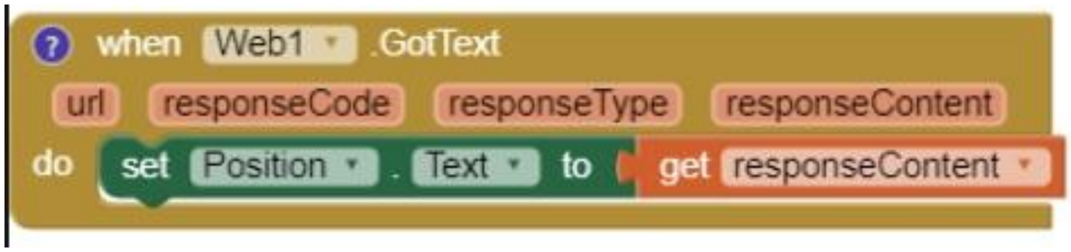
You should place a Web block into your Designer window for each Cloud function that you want to call. And, of course, you need an event to trigger off the procedure. In this case, any procedure call can be placed in any event creating block, such as a button push or shaking the phone.

Inside of your procedure, you need a "*set Webx.Url to*" block to send out the url string that you create by joining the url, deviceID, functionName, and the literal "?format=raw". As with GET, "*format=raw*" tells the Particle Cloud that the return value of your device function should be sent back "raw", that is, as just what you coded in your device firmware. Note that the

Particle.function() return value must always be an int, but what you get back is a text string that is the value of that int.

After setting the url, you then must "*call Webx.PostText*" to send out the POST name/value pairs. You create this string by joining the literal "access_token=" with *YourAccessToken* and appending the literal "&params=" followed by the string argument for your function call.

Just like GET, when you "*call Webx.PostText*" the POST command goes out over the Internet and you need to process the event which is the return value from the Particle Cloud. Like for GET, use "*when Webx.GotText*" to handle the event and place the code that does what you want inside this block; see example below:



Once again, "*responseContent*" is the return value from the Cloud function, "*responseType*" and "*responseCode*" are automatically returned status information from the *https:* protocol itself. Whether you actually want to include this event handler in your app is up to you. However, it is recommended to have your Cloud functions return something so that you can know that the Cloud function call actually got through to your device and was processed there. If you don't actually need data returned from a Cloud function call, just return the value 0 if the function executed OK and -1 if it didn't.

5.5.The rest of the App

The ParticleTest app (*ParticleTest.aia*) is used in conjunction with hardware and firmware (*Test_MIT.ino*) to demonstrate getting information from a Cloud variable and calling Cloud functions. The actual means of performing these actions has been described above. The rest of the app just provides a demonstration of these capabilities and, in particular, shows the benefits of writing your own app to control Particle devices over the Internet. When you write your own app, you not only decide when and under what circumstances to read data from your device and to call functions to execute actions on your device, but you can process responses from your device to know what actually happened.

This app uses the response data from calling the Cloud function "*on-off*" to change the color of the *LED ON* and *LED OFF* buttons to reflect the actual state of the device's D5 LED. Initially, both buttons are gray, indicating that the app does not know the status of the LED. However, clicking on either button changes the color of both buttons so that the active LED state is green and the inactive LED state is white. Since these button colors are set according to the return value from the "*on-off*" cloud function, we can be sure that the Photon/Electron executed the function and set the LED accordingly.

Likewise, this app uses the return value from the "*move*" Cloud function to display the actual angle of the servo on device pin A5 on the app screen. The device firmware protects the servo by ensuring that the command value in the move function call is between 5 and 175 degrees. This way, an inexpensive servo cannot be pinned at one end and possibly strip the internal plastic gears. By including this limit code in the device firmware itself, the device is protected from programming errors in the app. Since the app has a text box into which the user can type any number, it is beneficial to have a label that displays the actual setting of the servo angle, since this may be different from the what was commanded. Try this out: put some out-of-range value in the servo text box (say 1000) and click the SET button. You will see that the actual angle commanded of the servo is 175, which is the upper limit enforced by the firmware.

This app demonstrates getting the value of a Cloud variable by reading out some device initialization data. It is very handy for an app user to know the version of the firmware that is flashed onto their device, because as the project evolves and the firmware changes, the firmware and app code may get out of sync. It helps debugging to know what firmware version is actually flashed onto the Particle device. It also helps to know when the device was last reset; particularly if the device is depended upon to monitor something important. If the device has recently been reset, e.g. because of a power outage or WiFi interruption, some historical information may be lost from the device. If you want to know that nobody opened your front door between the hours of 9 am and 5 pm via some door open status in a Cloud variable, it is important to know that the device wasn't reset (and the Cloud variable along with it) in between these times.

6. Follow On Work.

Reading out Cloud variables and calling Cloud functions form the basis of creating an app to interact with Particle Photons and Electrons over the Internet. The app provided here does not require the user to log into their Particle account but does require the user to put their own *access_token* and *deviceID* into the MIT App Inventor 2 source code and build the app for transfer to their device. This is very inconvenient.

One approach to not having to hard code the *access_token* and *deviceID* into the app is to store this information in a file on the mobile device. This file could contain one or more comma delimited strings or JSON objects that are written to the file with a text app on the mobile device or created on a host computer and transferred to some particular place on the mobile device. A better approach, however, would be to have a second screen on every such app that contained the ability for the user to login to their Particle account and download the device information for all of their devices into a file on the mobile device. The code in the other screen could then open this file and display the user’s devices, e.g. in a pick list, allowing the user to select which device to use. Selection of the device might then trigger the app to check in with the Particle Cloud that the selected device is indeed online and if it is, then retrieve the device information such as firmware version and last reset time. All of this appears possible using additional capabilities documented in the Particle Cloud API documentation. The author intends to explore this further and to update this project with such additional features are developed.