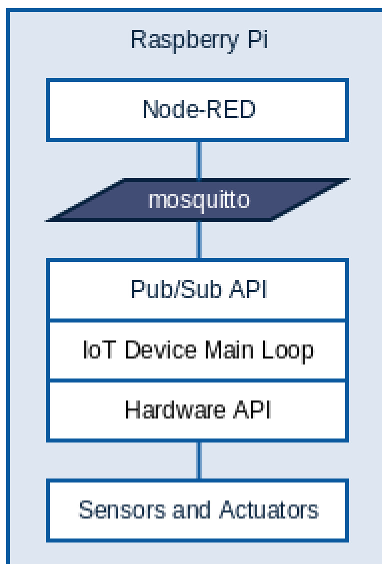
Module 8 Example – IoT Device and Architecture

NOTE: This example is not a project as in previous modules. It is being presented as an example of how a generalized IoT Device may be architected, designed and implemented. The goal is to provide you with ideas for your projects.

As you spent time on your projects this week you may have discovered, or wondered, is there a better way to write my python module? As is often the case with a 'main loop' style program, that loop can become complex rather quickly. In addition, when processing asynchronous messages from multiple sources, you may find you need to structure your code to prioritize message processing or to interface with hardware that has specific requirements. For example, the GrovePi hardware requires coordinated read and write access (e.g. in a multithreaded application only one thread should access the GrovePi at a time) and this access needs to be enforced by your code.

You may also wonder where you should place different pieces of application logic. As we have seen, you can write logic in Python, create Flows in Node-RED, and create full scale web applications using Meteor. So, what logic should go in each of these tiers? To start, a general rule of thumb is latency sensitive calculations are kept close (from a latency standpoint) to the data and systems they operate upon. For example, if real-time response are needed those calculations will generally be done on the IoT device, not in a cloud service where large and variable network latencies are present.

Once latency requirements are met, more general application requirements are considered. In this case, we would like to create a generic framework that will allow us to program, and reprogram, our IoT device at run-time and from the cloud (safely of course). While this could be accomplished by loading a new python module or modules containing your program, in this example we will leverage the flexibility and ease of use of Node-RED. In fact, in a well-designed system, Node-RED could even be used by technically savvy end-users if desired. Thus, our generalized IoT Device Architecture contains the following:



Starting at the bottom of this stack, the Sensors and Actuators are realized by the GrovePi module and any other hardware components you would like to integrate (e.g. a Raspberry Pi camera module). Next, the hardware API is realized by the GrovePi python modules (i.e. grovepi, grove_rgb_lcd, etc.) and any other python modules you integrate (e.g. the python modules used to integrate a Raspberry Pi camera).

The IoT Device Main Loop implements generic IoT device monitoring and control. It reads from, and writes to, the Hardware API to gather sensor data and to control actuators. Sensor and actuator state is published using the Pub/Sub API while commands are received via subscriptions.

Finally, mosquitto and Node-RED are used to provide access to and control of the generic IoT Device. Note that these services are running on the Raspberry Pi in this example. However, as we have seen in previous modules, they can also be run in a local network, in the cloud, or in both. When working with resource constrained embedded devices, the generic stack up to the Pub/Sub API can be embedded in the device and mosquitto an
This architecture provides a great deal of flexibility, allowing processing and logic to be moved to where it is needed given an application's requirements.

Generalized IoT Device Implementation

First, we will model the Grove Pi's ports:

ports.py

```
1 """
2 Defines constants for each Grove Pi port.
3 """
4
5 # Analog Ports
6 A0 = 0
7 A1 = 1
8 A2 = 2
9
10 # Digital Ports
11 D2 = 2
12 D3 = 3
13 D4 = 4
14 D5 = 5
```

```
15 D6 = 6
16 D7 = 7
17 D8 = 8
18
19 # I2C Ports
20 I2C_0 = 0
21 I2C_1 = 1
22 I2C_2 = 2
23
24 # Mode
25 INPUT = "INPUT"
26 OUTPUT = "OUTPUT"
```

Next, we will create a Grove Device abstraction that models each Grove Pi sensor and actuator:

GroveDevices.py

```
1 """
2 A GroveDevice abstraction is created for each Grove Pi sensor and actuator.  The
3 GroveDevice provides a uniform read and write interface to each component and
4 maintains a representation of each component's state. Additionally, each
5 GroveDevice configures the underlying hardware as necessary for proper device
6 operation.
7 """
8
9 import grovepi
10 import grove_rgb_lcd
11 import ports
12
13
14 class GroveDevice(object):
```

```
15    def __init__(self, port=None, value=0):
16        self.port = port
17        self.value = value
18
19    def read(self):
20        return self.value
21
22    def write(self, value):
23        self.value = value
24
25
26class LED(GroveDevice):
27    def __init__(self, port=ports.D5):
28        GroveDevice.__init__(self, port)
29        grovepi.pinMode(port, ports.OUTPUT)
30
31    def write(self, value):
32        self.value = int(value)
33        grovepi.analogWrite(self.port, self.value)
34
35
36class LCD(GroveDevice):
37    def write(self, value):
38        self.value = value
39        try:
40            r, g, b = value["rgb"]
41            grove_rgb_lcd.setRGB(r, g, b)
42        except KeyError:
```

```
43            pass
44        try:
45            text = value["text"]
46            grove_rgb_lcd.setText(text)
47        except KeyError:
48            pass
49
50
51class DHTSensor(GroveDevice):
52    DHT11 = 0
53    DHT22 = 1
54    DHT21 = 2
55
56    def __init__(self, port=ports.D7, dht_type=DHT11):
57        GroveDevice.__init__(self, port)
58        self.dht_type = dht_type
59
60    def read(self):
61        temperature, humidity = grovepi.dht(self.port, self.dht_type)
62        self.value = {
63            "temperature": temperature,
64            "humidity": humidity
65        }
66        return self.value
67
68
69class AnalogSensor(GroveDevice):
70    def __init__(self, port):
```

```
71         GroveDevice.__init__(self, port)
72 
73     def read(self):
74         self.value = grovepi.analogRead(self.port)
75         return self.value
76 
77 
78 class Potentiometer(AnalogSensor):
79     def __init__(self, port=ports.A2):
80         AnalogSensor.__init__(self, port)
81 
82 
83 class LightSensor(AnalogSensor):
84     def __init__(self, port=ports.A1):
85         AnalogSensor.__init__(self, port)
86 
87 
88 class SoundSensor(AnalogSensor):
89     def __init__(self, port=ports.A0):
90         AnalogSensor.__init__(self, port)
91 
92 
93 class Button(GroveDevice):
94     def __init__(self, port=ports.D3):
95         GroveDevice.__init__(self, port)
96         grovepi.pinMode(port, ports.INPUT)
97 
98     def read(self):
```

```
 99         self.value = grovepi.digitalRead(self.port)
100         return self.value
101
102
103 class Buzzer(GroveDevice):
104     def __init__(self, port=ports.D2):
105         GroveDevice.__init__(self, port)
106         grovepi.pinMode(port, ports.OUTPUT)
107
108     def write(self, value):
109         self.value = value
110         grovepi.digitalWrite(self.port, value)
111
112
113 class Relay(GroveDevice):
114     def __init__(self, port=ports.D6):
115         GroveDevice.__init__(self, port)
116         grovepi.pinMode(port, ports.OUTPUT)
117
118     def write(self, value):
119         self.value = value
120         grovepi.digitalWrite(self.port, value)
121
122
123 class UltrasonicRanger(GroveDevice):
124     def __init__(self, port=ports.D4):
125         GroveDevice.__init__(self, port)
126
```

```
127     def read(self):
128         self.value = grovepi.ultrasonicRead(self.port)
129         return self.value
```

As in previous modules, we will use a UUID to identify IoT devices:

uuidgen.py

```
1 """
2 Generates UUIDs in a consistent manner.
3 """
4 
5 import uuid
6 import netifaces
7 
8 
9 def generateUuid(namespace='', domain='snhu.edu'):
10     """ Generate a device specific Type 5 UUID within a namespace and domain.
11 
12     :param namespace: The namespace where the UUID is being generated
13     :param domain: The domain where the UUID is being generated
14     :return: Type 5 UUID
15     """
16     mac = netifaces.ifaddresses('eth0')[netifaces.AF_LINK][0]['addr']
17     return str(uuid.uuid5(uuid.NAMESPACE_DNS, mac+'.'+namespace+'.'+domain))
```

Perhaps the most interesting part of this example is the generalized IoT device code that implements a generic, reusable main loop:

IoTDevice.py

```
1 """
2 Implements a generalized IoT Device architecture and main loop. This IoT device
3 receives actuator control messages over MQTT and publishes changing sensor and
```

```
 4 actuator data over MQTT.  The default configuration utilizes all Grove Pi
 5 interfaces and demonstrates all sensor and actuator types provided in the
 6 starter kit.
 7 """
 8 
 9 import GroveDevices
10 import ports
11 import paho.mqtt.client as mqtt
12 import time
13 import json
14 import uuidgen
15 import Queue
16 
17 DEVICE_UUID = uuidgen.generateUuid()
18 SENSOR_DATA_TOPIC = "SNHU/IT697/sensor/data/"+DEVICE_UUID
19 ACTUATOR_TOPIC = "SNHU/IT697/actuator/control/"+DEVICE_UUID
20 
21 # Actuators
22 BUZZER = GroveDevices.Buzzer(ports.D2)
23 BLUE_LED = GroveDevices.LED(ports.D5)
24 RELAY = GroveDevices.Relay(ports.D6)
25 RED_LED = GroveDevices.LED(ports.D8)
26 LCD = GroveDevices.LCD()  # I2C-1 port
27 
28 ACTUATORS = {
29     "blue_led": BLUE_LED,
30     "red_led": RED_LED,
31     "lcd": LCD,
```

```
32      "buzzer": BUZZER,
33      "relay": RELAY
34  }
35
36  # Sensors
37  SOUND_SENSOR = GroveDevices.SoundSensor(ports.A0)
38  LIGHT_SENSOR = GroveDevices.LightSensor(ports.A1)
39  POTENTIOMETER = GroveDevices.Potentiometer(ports.A2)
40  BUTTON = GroveDevices.Button(ports.D3)
41  ULTRASONIC_RANGER = GroveDevices.UltrasonicRanger(ports.D4)
42  DHT_SENSOR = GroveDevices.DHTSensor(ports.D7)
43
44  SENSORS = [
45      ("potentiometer", POTENTIOMETER, 1),
46      ("light_sensor", LIGHT_SENSOR, 5),
47      ("sound_sensor", SOUND_SENSOR, 5),
48      ("button", BUTTON, 1),
49      ("ultrasonic_ranger", ULTRASONIC_RANGER, 5),
50      ("dht_sensor", DHT_SENSOR, 100)
51  ]
52
53
54  def on_connect(client, userdata, flags, rc):
55      """Called each time the client connects to the message broker
56      :param client: The client object making the connection
57      :param userdata: Arbitrary context specified by the user program
58      :param flags: Response flags sent by the message broker
59      :param rc: the connection result
```

```
60    :return: None
61    """
62    # subscribe to the ACTUATOR topic when connected
63    client.subscribe(ACTUATOR_TOPIC)
64
65# A message queue is required to coordinate reads and writes from/to
66# the GrovePi
67MSG_QUEUE = Queue.Queue()
68
69
70def on_message(client, userdata, msg):
71    """Called for each message received
72    :param client: The client object making the connection
73    :param userdata: Arbitrary context specified by the user program
74    :param msg: The message from the MQTT broker
75    :return: None
76    """
77    MSG_QUEUE.put(msg.payload)
78
79
80def calculate_delta(sensor_name, value, last_values, changed_values):
81    """Determine which values have changed from their last values
82    :param sensor_name: The sensor name the value was read from
83    :param value: The new value
84    :param last_values: The last set of checked values
85    :param changed_values: The set of values that have changed
86    """
87    try:
```

```
 88            if last_values[sensor_name] == value:
 89                return
 90        except KeyError:
 91            pass
 92        changed_values[sensor_name] = value
 93        last_values[sensor_name] = value
 94
 95
 96read_count = 0
 97last_values = {}
 98
 99
100def read_sensors_and_actuators():
101    """ Reads sensors and actuators and returns the values that have changed
102    since last read
103    :return: The changed values since last read
104    """
105    global read_count
106    changed_values = {}
107    for sensor_name, sensor, priority in SENSORS:
108        if not (read_count % priority):
109            calculate_delta(sensor_name, sensor.read(), last_values,
110                            changed_values)
111    read_count += 1
112    for actuator_name, actuator in ACTUATORS.iteritems():
113        calculate_delta(actuator_name, actuator.read(), last_values,
114                        changed_values)
115    return changed_values
```

```
116
117
118def publish_sensor_data(values):
119    """ Publishes data over MQTT to the sensor data topic
120    :param values: The sensor values to send
121    :return:  None
122    """
123    values["timestamp"] = int(time.time()*1000)
124    out_str = json.dumps(values)
125    mqtt_client.publish(SENSOR_DATA_TOPIC, out_str)
126    print("==>> " + out_str)
127
128
129def process_received_messages():
130    """ Processes all MQTT messages placed in the message queue.
131    :return: None
132    """
133    while True:
134        try:
135            payload = MSG_QUEUE.get(False)
136            print("<<== " + payload)
137            payload = json.loads(payload)
138            for actuator, msg in payload.iteritems():
139                try:
140                    ACTUATORS[actuator].write(msg)
141                except KeyError:
142                    pass
143        except Queue.Empty:
```

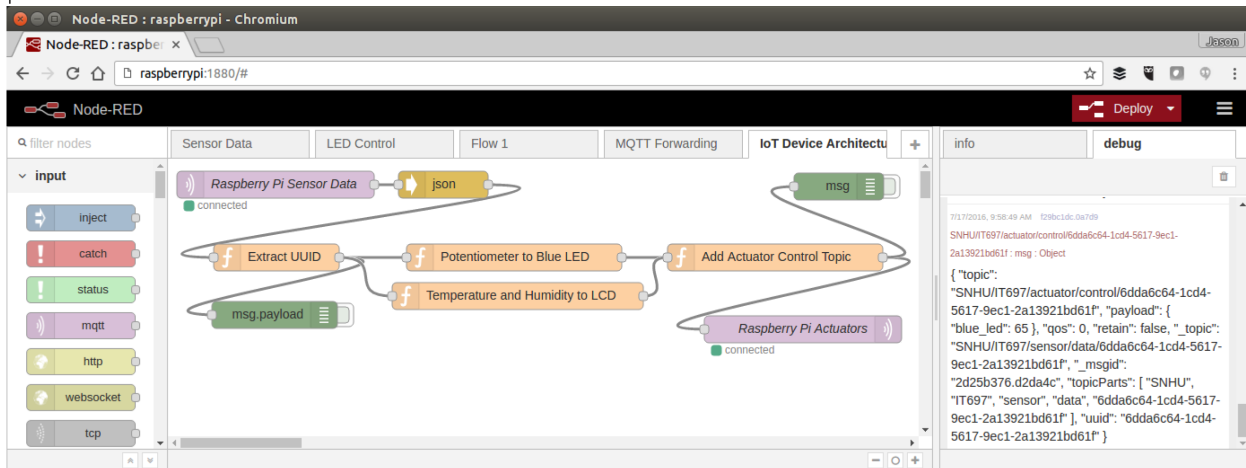
```
144            break
145
146
147MESSAGE_BROKER_URI = "localhost"
148mqtt_client = mqtt.Client()
149mqtt_client.on_connect = on_connect
150mqtt_client.on_message = on_message
151mqtt_client.connect(MESSAGE_BROKER_URI)
152mqtt_client.loop_start()
153
154time.sleep(1)  # give the hardware time to initialize
155
156while True:
157    try:
158        changedValues = read_sensors_and_actuators()
159
160        if changedValues:
161            publish_sensor_data(changedValues)
162
163        process_received_messages()
164
165    except (IOError, TypeError) as e:
166        print("Error", e)
```

Place all four of the above files in a directory of your choice (e.g. IoTDeviceArchitecture) and run IoTDevice.py:

```
python IoTDevice.py
```

Implementing an Application in Node-RED

Once you have a generic IoT device, application logic can be implemented in Node-RED. In this example we will re-implement the Temperature and Humidity Sensor of Module One and LED control via the potentiometer similar to Module Four.



Copy the following json node definitions to your clipboard and import into a new Node-RED flow (the import menu is to the right of the Deploy button):

IoT Device Architecture Nodes


```
[
   {
      "id":"fcd4d5d4.f1f3c8",
      "type":"mqtt-broker",
      "z":"1294dea6.abda71",
      "broker":"localhost",
      "port":"1883",
      "clientid":"",
      "usetls":false,
      "verifyservercert":true,
      "compatmode":true,
      "keepalive":"60",
      "cleansession":true,
      "willTopic":"",
      "willQos":"0",
      "willRetain":null,
      "willPayload":"",
      "birthTopic":"",
      "birthQos":"0",
      "birthRetain":null,
      "birthPayload":""
   },
   {
      "id":"4a38b38a.722c0c",
```

```
      "type":"mqtt out",
      "z":"1294dea6.abda71",
      "name":"Raspberry Pi Actuators",
      "topic":"",
      "qos":"0",
      "retain":"",
      "broker":"fcd4d5d4.f1f3c8",
      "x":708,
      "y":190,
      "wires":[

      ]
   },
   {
      "id":"87c27b64.367eb8",
      "type":"mqtt in",
      "z":"1294dea6.abda71",
      "name":"Raspberry Pi Sensor Data",
      "topic":"SNHU/IT697/sensor/data/#",
      "broker":"fcd4d5d4.f1f3c8",
      "x":120,
      "y":29,
      "wires":[
         [
            "8429a230.c79dd"
         ]
      ]
   },
   {
      "id":"c4ab8779.c202b8",
      "type":"function",
      "z":"1294dea6.abda71",
      "name":"Add Actuator Control Topic",
      "func":"msg.topic = \"SNHU/IT697/actuator/control/\" + msg.uuid; \nreturn msg;",
      "outputs":1,
      "noerr":0,
      "x":677,
      "y":110,
      "wires":[
         [
            "4a38b38a.722c0c",
            "f29bc1dc.0a7d9"
         ]
```

```
      ]
   },
   {
      "id":"61ec95f0.d06a9c",
      "type":"function",
      "z":"1294dea6.abda71",
      "name":"Extract UUID",
      "func":"msg.topicParts = msg.topic.split('/');\nmsg.uuid = msg.topicParts[msg.topicPart
      "outputs":1,
      "noerr":0,
      "x":121,
      "y":110,
      "wires":[
         [
            "e8ecef47.b5248",
            "7503f9dd.86b598",
            "91cf4d59.c489"
         ]
      ]
   },
   {
      "id":"e8ecef47.b5248",
      "type":"function",
      "z":"1294dea6.abda71",
      "name":"Potentiometer to Blue LED",
      "func":"if (msg.payload.potentiometer === undefined) {\n    return;\n}\n\nmsg.payload =
      "outputs":1,
      "noerr":0,
      "x":386,
      "y":110,
      "wires":[
         [
            "c4ab8779.c202b8"
         ]
      ]
   },
   {
      "id":"8429a230.c79dd",
      "type":"json",
      "z":"1294dea6.abda71",
      "name":"",
      "x":307,
      "y":29,
```

```
      "wires":[
         [
            "61ec95f0.d06a9c"
         ]
      ]
   },
   {
      "id":"f29bc1dc.0a7d9",
      "type":"debug",
      "z":"1294dea6.abda71",
      "name":"",
      "active":false,
      "console":"false",
      "complete":"true",
      "x":748,
      "y":31,
      "wires":[

      ]
   },
   {
      "id":"7503f9dd.86b598",
      "type":"debug",
      "z":"1294dea6.abda71",
      "name":"",
      "active":false,
      "console":"false",
      "complete":"false",
      "x":119,
      "y":174,
      "wires":[

      ]
   },
   {
      "id":"91cf4d59.c489",
      "type":"function",
      "z":"1294dea6.abda71",
      "name":"Temperature and Humidity to LCD",
      "func":"if (msg.payload.dht_sensor === undefined) {\n    return;\n}\n\nvar temperature
lcd = {\n    rgb: [0, 255, 0],\n    text: \"Temp: \"+temperature+\"C\\n\"+\"Humidity: \"+humi
      "outputs":1,
      "noerr":0,
```

```
      "x":390,
      "y":153,
      "wires":[
         [
            "c4ab8779.c202b8"
         ]
      ]
   }
]
```

Deploy the flow and observe the temperature and humidity sensor and adjust the potentiometer to see the LED dim and brighten.

As you can see, the sensors and actuators are now completely reprogrammable at run-time via Node-RED.

Create your own demo applications and enjoy!