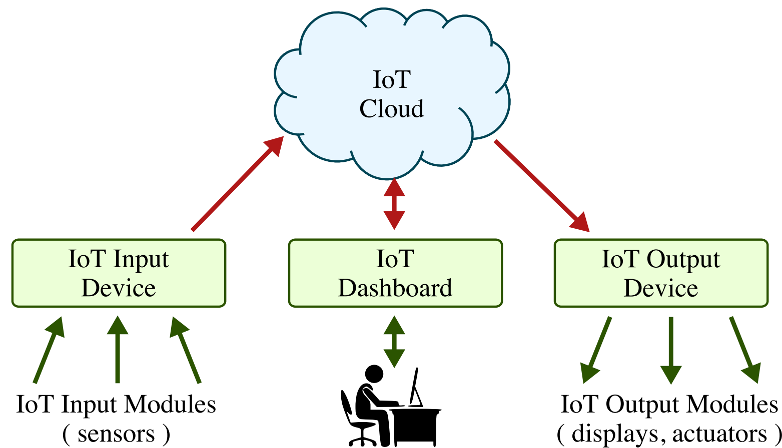
CURIE Academy, Summer 2021
Design Project Input/Output Module Tutorial

Nick Cebry and Christopher Batten
School of Electrical and Computer Engineering
Cornell University

Each IoT design project will involve building an IoT system comprised of an IoT input device, IoT cloud, IoT output device, and IoT dashboard. The IoT input devices will have various input modules attached that can sense what is going on in the environment (e.g., light, temperature, humidity, button press, rotation, water, acceleration, distance, soil moisture) and be able to upload data into the cloud. Each IoT output device will be able to download data from the could and will have various output modules attached to display data (e.g., LEDs, piezo buzzer, 4-digit display). The IoT dashboard can be used to monitor or even configure the IoT system. The following diagram illustrates the overall approach we will be using in our IoT systems.



This document describes each input/output module and provides some example code to get you started. Revisit the Lab 2 notes and handout to learn about the Particle Argon ports and how to use the Particle development environment (including how to use Particle variables and the Particle console). The various input and output modules are listed below.

1 Light Input Module 2
2 Temperature and Humidity Input Module 3
3 Button Input Module 5
4 Rotation Input Module 6
5 Water Input Module 7
6 Accelerometer Input Module 8
7 Ultrasonic Range Input Module 11
8 Soil Moisture Input Module 12
9 LED Output Module 13
10 RGB LED Output Module 14
11 Piezo Buzzer Output Module 15
12 4-Digit Display Output Module 17

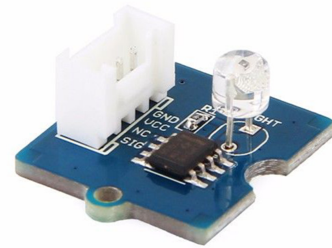
1. Light Input Module

This input module enables your IoT device to sense the amount of light in the environment. The module uses a small photo-resistor where the resistance will vary based on how much light reaches the device. Since the Particle Argon cannot directly sense resistance, the input module includes an integrated circuit that will convert the sensed resistance into a voltage which can then be read by the Particle Argon.

The example code below illustrates how to read the current light value using the Particle Argon. You must connect the light input module to analog port A0, A2, or A4. This code assumes the light input module is connected to analog port A0. In the `setup` function, we first configure analog port A0 to be an input port, and then we create a Particle variable which enables us to monitor this variable from the Particle console. In the `loop` function, we read the temperature once per second.

Use this example code to program your Particle Argon, then use the Particle console to monitor the light value. Note how the sensor reading varies when the device is covered with your hand or when the device is placed under a bright light. The reading from the sensor should get smaller when the device is in a dark environment, and should get larger when the device is in a bright environment. You will need to experiment on your own to determine what values to expect in your specific environment.

```
int light_pin   = A0;
int light_value = -1;

void setup()
{
  pinMode( light_pin, INPUT );
  Particle.variable( "light_value", light_value );
}

void loop()
{
  light_value = analogRead( light_pin );
  delay( 1000 );
}
```



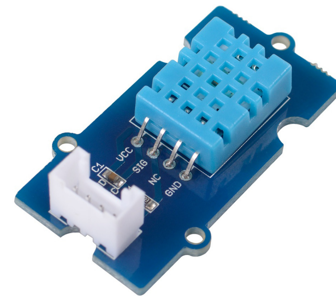
2. Temperature and Humidity Input Module

This input module enables your IoT device to sense the temperature and humidity of the environment it is placed in. The module uses a capacitor to measure the relative humidity and a thermistor to measure the temperature. The capacitor consists of two small metal plates, separated by a porous material. The capacitance value changes based on the amount of water in the air between the plates. The thermistor is a device whose resistance changes based on temperature. The Particle Argon cannot directly measure either capacitance or resistance, so the module includes integrated circuits on board to measure the these values and then send them to the Particle Argon as a series of pulses (1s and 0s) that the Particle Argon can decode into a number.

The example code below illustrates how to read the current temperature and humidity values using the Particle Argon. You will need to include the *Grove_Temperature_And_Humidity_Sensor library*, which will handle the conversion from the stream of pulses into a number for you. You can add the library to your project by clicking the library button in the IDE, searching for "Grove_Temperature", and clicking on the appropriate library. The sample code below assumes the module is connected to digital port D2. The module can also work when connected to digital port D4 or analog ports A0, A2, or A4 if necessary. Before the setup function, we create an object for controlling the interface with the sensor. Then, in the setup function we set set the mode of our pin to an input, and declare two Particle variables, one for the temperature and one for the humidity. In the loop function, we update the temperature and humidity variables by reading values from the sensor every two seconds.

You can program your Particle Argon with this example code, and then use the Particle Console to monitor the temperature and humidity values. You can try putting the sensor in your fridge or freezer to reduce the temperature, or directly under a light bulb to increase the temperature. If you exhale on the sensor, you will likely see the humidity increase relative to the air in the room.

```
#include <Grove_Temperature_And_Humidity_Sensor.h>

int dht_pin = A0;

// Creates a C++ object to manage sensor
DHT dht_sensor( dht_pin );

double temperature = 0.0;
double humidity    = 0.0;

void setup()
{
  pinMode( dht_pin, INPUT );
  Particle.variable( "temperature", temperature );
  Particle.variable( "humidity",    humidity    );
}

void loop()
{
  temperature = dht_sensor.getTempFarenheit();
  humidity    = dht_sensor.getHumidity();
  delay( 2000 );
}
```



NOTE: If the sensor has trouble reading the temperature and/or humidity, it will return either a special NaN value or zero. You may need to explicitly check to see if the data returned by the sensor is valid. Here is an example which makes sure the returned temperature is greater than 0 °F and less than 110 °F

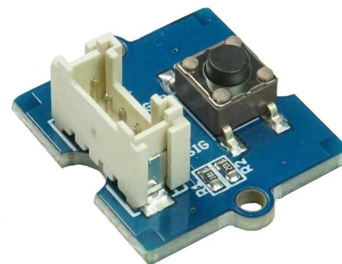
```
#include <Grove_Temperature_And_Humidity_Sensor.h>

int dht_pin = A0;

// Creates a C++ object to manage sensor
DHT dht_sensor( dht_pin );

double temperature = 0.0;

void setup()
{
  pinMode( dht_pin, INPUT );
}

void loop()
{
  temperature = dht_sensor.getTempFarenheit();
  if ( (temperature > 0) && (temperature < 110) ) {
    ... do something with the temperature ...
  }
  delay(2000);
}
```

3. Button Input Module

This input module enables your IoT device to sense a button press. It can potentially be used to configure your IoT device (e.g., pressing and holdling the button might put your IoT device into a different mode) or with some creativity it could be used to sense the presence or absence of pressure.

The example code below illustrates how to read the button state using the Particle Argon. You should usually connect the button input module to digital ports D2 or D4, but you can also connect the button input module to analog ports A0, A2, or A4 if necessary. This code assumes the button input module is connected to digital port D2. In the `setup` function, we first configure digital port D2 to be an input port, and then we create a Particle variable which enables us to monitor this variable from the Particle console. In the `loop` function, we read the button state once per second.

```
int button_pin   = D2;
int button_state = -1;

void setup()
{
  pinMode( button_pin, INPUT );
  Particle.variable( "button_state", button_state );
}

void loop()
{
  button_state = digitalRead( button_pin );
  delay( 1000 );
}
```

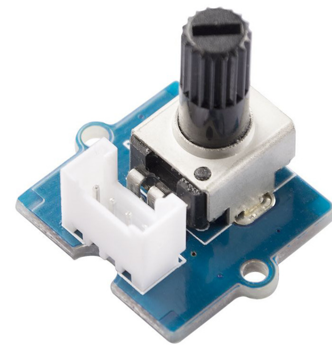


4. Rotation Input Module

This input module enables your IoT device to sense the rotation of the knob. It works by moving a conductor along a resistive material, causing the resistance value to change as the knob rotates. It can be used like the volume knob on a stereo to set the intensity of something (for instance perhaps the brightness of the RGB LED, or the desired temperature on a thermostat), or with some creative mechanical design could be attached to a door to sense how far open it is.

The example code below shows how to read the state of the rotation sensor using the Particle Argon. The rotation sensor must be connected to one of the analog ports, that is port A0, A2, or A4. This example code assumes the sensor is connected to port A0. In the setup portion of the code, we configure port A0 to be an input port, then create a Particle variable that allows us to monitor the value from the Particle console. In the loop portion of the code, we read the state of the rotation sensor once per second.

```
1  int rotation_pin   = A0;
2  int rotation_state = -1;
3
4  void setup()
5  {
6    pinMode( rotation_pin, INPUT );
7    Particle.variable( "rotation_state", rotation_state );
8  }
9
10 void loop()
11 {
12   rotation_state = analogRead( rotation_pin );
13   delay( 1000 );
14 }
```



5. Water Input Module

This input module enables your IoT device to sense the presence of water. It works by detecting when water causes the silver lines on the module to be electrically connected to one another. It could be used to check for flooding (either in a basement or on the bank or a river/lake), or to detect if a sink or water tank is full/overflowing.

The example code below shows how to read the state of the water sensor using the Particle Argon. You will usually want to connect the sensor to one of the digital ports, D2 or D4, but it can also be connected to one of the analog ports, A0, A2, or A4, if necessary. This code assumes the water sensor is connected to digital port D2. In the setup portion of the code, we configure port A0 to be an input port, then create a Particle variable that allows us to monitor the value from the Particle console. In the loop portion of the code, we read the state of the water sensor once per second.

```
int water_pin   = D2;
int water_state = -1;

void setup()
{
  pinMode( water_pin, INPUT );
  Particle.variable( "water_state", water_state );
}

void loop()
{
  water_state = digitalRead( water_pin );
  delay( 1000 );
}
```

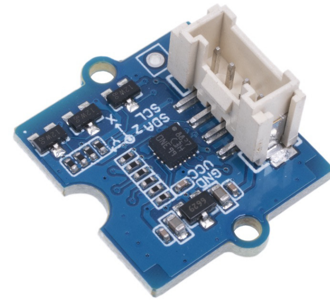
NOTE: It is also possible to use the water sensor to determine the amount of water present if the water is only causing some of the silver lines on the module to be electrically shorted. This requires connecting the module to an analog port and using the `analogRead` function instead of the `digitalRead` function.

6. Accelerometer Input Module

This input module enables your IoT device to sense acceleration, or changes in speed. Inside the sensor integrated circuit are some tiny capacitors, with one side of the capacitor attached to a moving piece and the other anchored in place. When the integrated circuit experiences acceleration, the moving piece gets pushed in one direction, causing the sides of the capacitors to get either closer together or farther apart. This causes the capacitance to change. The Particle Argon communicates with the integrated circuit using a special pattern of pulses called the inter integrated circuit communication protocol, or I2C protocol for short. The Particle Argon sends a message to the accelerometer asking for its current state, and the accelerometer measures the capacitance of its capacitors and returns the data the Particle Argon asked for.

The example code below shows how to read the current acceleration using the Particle Argon. It requires a special library that is not available through the Particle IDE, so you may want to ask a teaching assistant or the instructor to help you set up the library. The instructions for installing the library appear below the code. Before the setup section of the code, we create a C++ object for managing the communication with the sensor module. The accelerometer must be connected to one of the I2C ports, either I2C_1 or I2C_2. The example code assumes the accelerometer is connected to I2C_1. In the setup section, we configure the accelerometer, then create 3 Particle variables for recording the three different directions the accelerometer measures acceleration in. In the loop section, we read each of the three acceleration values from the accelerometer once every second.

You can program your Particle Argon with this example code and then use the Particle Console to monitor the acceleration values. Gravity causes a constant acceleration towards the center of the Earth, so you can try setting the accelerometer on a table in a couple different orientations and see how the acceleration values change. You can also try reading the values while you are shaking the sensor in your hand.



```
#include "LIS3DHTR.h"

double x_acceleration;
double y_acceleration;
double z_acceleration;

// Creates a C++ object to manage sensor
LIS3DHTR<TwoWire> accel;

void setup()
{
  // The next line of code indicates which port the module
  // is connected to. Use the value Wire for the I2C_1 port
  // and the value Wire1 for the I2C_2 port
  accel.begin( Wire, LIS3DHTR_ADDRESS_UPDATED );
  delay( 100 );

  // This sets the maximum amount of acceleration the module
  // will report, where 1G is the amount of acceleration
  // due to Earth's gravity. Available options are:
  // LIS3DHTR_RANGE_2G
  // LIS3DHTR_RANGE_4G
  // LIS3DHTR_RANGE_8G
  // LIS3DHTR_RANGE_16G
  // Using higher values will let you measure bigger numbers,
  // but lower the precision of your measurement.
  accel.setFullScaleRange( LIS3DHTR_RANGE_2G );

  accel.setOutputDataRate( LIS3DHTR_DATARATE_50HZ );
  Particle.variable( "x_acceleration", x_acceleration );
  Particle.variable( "y_acceleration", y_acceleration );
  Particle.variable( "z_acceleration", z_acceleration );
}

void loop()
{
  x_acceleration = accel.getAccelerationX();
  y_acceleration = accel.getAccelerationY();
  z_acceleration = accel.getAccelerationZ();
  delay(1000);
}
```

Library Setup Instructions: The library we need to use for the accelerometer is not included in the Particle IDE by default. The library can be found at this link: `https://github.com/Seeed-Studio/Seeed_Arduino_LIS3DHTR/tree/master/src` To add the library to your project, click the (+) icon in the upper right hand corner of the IDE while your project code is open. Two new tabs will appear at the top of your screen for a .h file and a .cpp file. Rename the files to LIS3DHTR.h and LIS3DHTR.cpp, then copy the code from the github URL into these two files.

NOTE: For the accelerometer, you might want to be able to read the values more often than is possible with the code using the Particle Console above. There is a tool called the Particle command line interface, AKA the Particle CLI. This program will let you send messages from your Particle Argon straight to your computer over the USB cable without using the cloud. Talk to an instructor or TA if you would like to set this up. Below is some sample code that will periodically send the acceleration values to your computer over the USB cable. You can use the command `% particle serial monitor` in the Particle CLI to see the values.

```
#include "LIS3DHTR.h"

// Creates a C++ object to manage sensor
LIS3DHTR<TwoWire> accel;

void setup()
{
  Serial.begin( 9600 );
  delay( 5000 );

  // The next line of code indicates which port the module
  // is connected to. Use the value Wire for the I2C_1 port
  // and the value Wire1 for the I2C_2 port
  accel.begin( Wire, LIS3DHTR_ADDRESS_UPDATED );
  delay( 100 );

  // This sets the maximum amount of acceleration the module
  // will report, where 1G is the amount of acceleration
  // due to Earth's gravity. Available options are:
  // LIS3DHTR_RANGE_2G
  // LIS3DHTR_RANGE_4G
  // LIS3DHTR_RANGE_8G
  // LIS3DHTR_RANGE_16G
  // Using higher values will let you measure bigger numbers,
  // but lower the precision of your measurement.
  accel.setFullScaleRange( LIS3DHTR_RANGE_2G );

  accel.setOutputDataRate( LIS3DHTR_DATARATE_50HZ );

  if ( !accel.isConnection() )
    Serial.println( "LIS3DHTR didn't connect." );
}

void loop()
{
  float x_acceleration = accel.getAccelerationX();
  float y_acceleration = accel.getAccelerationY();
  float z_acceleration = accel.getAccelerationZ();

  Serial.printlnf( "x:%f,y:%f,z:%f", x_acceleration, y_acceleration, z_acceleration );

  delay( 1000 );
}
```

7. Ultrasonic Range Input Module

This module enables your IoT device to sense the distance to a solid object it is pointed at. The module uses ultrasonic sound waves to determine the distance to an object. Ultrasonic waves are sound waves with a pitch so high that the human ear cannot detect them. The sensor sends out an ultrasonic wave, and then waits for the wave to bounce back or echo. By measuring the time it takes for the wave to bounce back, and using the speed of sound in air, the module can determine how far away the object that the wave bounced off of is. This is just like how bats and dolphins use echolocation.

The example code below shows how to get the distance to an object using the Particle Argon. The ultrasonic range module must be connected to either an analog or a digital port on the Particle Argon. This example assumes it is connected to analog port A4. This module has an integrated circuit on it to handle the task of measuring the time between when the wave is sent out and when the echo comes back. In order to communicate with the integrated circuit on the module, we will use the Grove-Ultrasonic-Ranger library. Add this library to your project by clicking the library button in the IDE, searching for ranger, and then clicking on it to add it to your project. Before the setup portion of the code, we create a C++ object that we will use to talk to the integrated circuit on the module. In the setup portion, we declare a Particle variable which will hold the distance returned by the sensor. In the loop function, we use the library to get the distance value from the sensor every two seconds.

Program your Particle Argon with the example code below, and then use the Particle Console to monitor the distance value. You can place your hand in front of the sensor and move it back and forth to see how the value changes. You can also try pointing the sensor at a door or window and seeing if you can tell based on the value returned whether it is open or closed. The value from the sensor should be larger when objects are farther away, and smaller when they are closer.

```
1  #include <Grove-Ultrasonic-Ranger.h>
2
3  int ranger_pin = A4;
4  int ranger_cm  = -1;
5
6  // Creates a C++ object to manage sensor
7  Ultrasonic range_sensor( ranger_pin );
8
9  void setup()
10 {
11   Particle.variable( "ranger_cm", ranger_cm );
12 }
13
14 void loop()
15 {
16   ranger_cm = range_sensor.MeasureInCentimeters();
17   delay( 2000 );
18 }
```

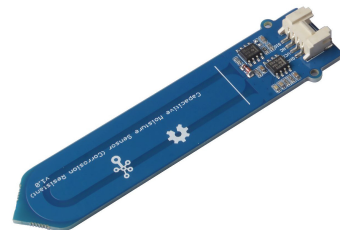
8. Soil Moisture Input Module

This module enables your IoT device to sense the moisture content of soil. The module uses a capacitor to measure moisture content. The water and soil being measured form the dielectric, or material between the plates of the capacitor. As the amount of water in the soil changes, the capacitance of the capacitor changes. An integrated circuit on the module changes this capacitance value to an analog voltage that we can read with the Particle Argon.

The example code below shows how to get readings from the capacitive moisture sensor using the Particle Argon. The moisture sensor must be connected to an analog port on the Particle Argon, one of: A0, A2, or A4. The example code assumes the sensor is connected to analog port A0. In the setup section of the code, we configure the A0 port as an input and declare a Particle variable. In the loop portion of the code, we update the variable once every second with the value from the moisture sensor.

Program your Particle Argon with the example code below, and then use the Particle console to monitor the moisture value. You can put the sensor in a potted plant or your yard outside and see what moisture value you get. Pour some water on the soil (but not the sensor!) and see how the value changes. You should see the value go down as the moisture increases.

```
1  int moisture_pin   = A0;
2  int moisture_value = -1;
3
4  void setup()
5  {
6    pinMode( moisture_pin, INPUT );
7    Particle.variable( "moisture_value", moisture_value );
8  }
9
10 void loop()
11 {
12   moisture_value = analogRead( moisture_pin );
13   delay( 1000 );
14 }
```

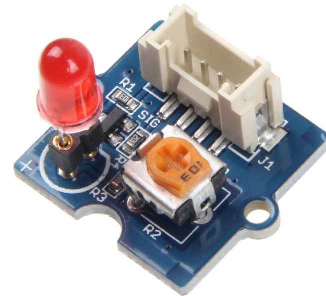


9. LED Output Module

This output module enables your IoT device to turn on/off a red LED. It can be used to indicate the status of your device (potentially with different blink rates as well) or as an alarm.

The example code below illustrates how to blink the LED output module using the Particle Argon. You should usually connect the LED output module to digital ports D2 or D4, but you can also connect the LED output module to analog ports A0, A2, or A4 if necessary. This code assumes the LED output module is connected to digital port D2. In the `setup` function, we first configure digital port D2 to be an output port. In the `loop` function, we continuously turn on the LED, wait one second, turn off the LED, and wait one second.

```
int led_pin = D2;

void setup()
{
  pinMode( led_pin, OUTPUT );
}

void loop()
{
  digitalWrite( led_pin, HIGH );  // Turn on the LED
  delay( 1000 );                  // Wait 1 second
  digitalWrite( led_pin, LOW );   // Turn off the LED
  delay( 1000 );                  // Wait 1 second
}
```



10. RGB LED Output Module

This output module enables your IoT device to turn an RGB LED on/off and change its color. It can be used to indicate the status of your device, or to indicate a variety of different alarms based on color. You could also use it to represent a color changing smart lightbulb.

The example code below illustrates how to blink the RGB LED several different colors using the Particle Argon. You will usually want to connect the RGB LED to one of the digital ports, D2 or D4, but you can also connect it to the analog ports, A0, A2, or A4. This code assumes the RGB LED is connected to digital port D2. You will want to connect the wire to the "in" side of the RGB LED module. The Particle Argon communicates with an integrated circuit on the module through a special series of pulses (a pattern of 1s and 0s) to tell the integrated circuit what to do with the RGB LED. We will use a library to handle the details of this communication. To include the library in your project, click the library button in the Particle IDE, search for "chainable" and then click on the Grove_ChainableLED library to add it to your project. Before the setup region, we create a C++ object that will handle communication with our RGBN LED module. In the setup region, we initialize the C++ object. Then in the loop region we cycle the RGB LED between red, green, and blue, displaying each color for one second.

Program your Particle Argon with the example code below, and see how the LED changes to display different colors. Then, try programming in some different colors for your LED to display. You can use either the RGB or the HSB functions to define your colors. In either function, the first value should always be 0 because you have only a single RGB LED. In the RGB function, the next three values represent the amount of red, green, and blue present in the color respectively. In HSB, the three values correspond to hue, saturation, and brightness respectively. You can use this color picker: `https://colorpicker.me/#e31a1a` to find values for the colors you want. The red, green, and blue values can be copied directly from the color picker into the RGB function. The color picker will produce values between 0 and 360 for hue, representing degrees in a circle, so divide that number by 360 to get the hue value for the HSB function. This color picker shows numbers between 0 and 100 for "saturation" and "value", so you should divide them by 100 before putting them in the saturation and brightness fields of the HSB function.

```
1  #include <Grove_ChainableLED.h>
2
3  // Creates a C++ object to manage RGB LED
4  ChainableLED rgb_led( D4, D5, 1 );
5
6  void setup()
7  {
8    rgb_led.init();
9  }
10
11 void loop()
12 {
13   rgb_led.setColorRGB( 0, 255, 0, 0 ); // red
14   delay( 1000 );
15   rgb_led.setColorRGB( 0, 0, 255, 0 ); // green
16   delay( 1000 );
17   rgb_led.setColorHSB( 0, 0.65, 1.0, 0.5 ); // blue
18   delay( 1000 );
19 }
```

11. Piezo Buzzer Output Module

This output module enables your IoT device to make sound. It can be used for status indication through particular patterns or as an alarm. With a little extra work, you can also get it to play some simple music.

The example code below illustrates how to get the buzzer to make some simple buzzing noises that could be useful for status notifications or an alarm. You will usually want to connect the buzzer to a digital pin, D2 or D4, but you can also connect it to an analog pin, A0, A2, or A4. The example code assumes that the buzzer is connected to port D2. The setup segment of the code configures the pin the buzzer is connected to as an output. The loop portion turns the buzzer on for half a second, then turns it off for a full second, and then repeats.

Program your Particle Argon with the example code below and see how the buzzer reacts. You can change the pattern of the buzzing by changing the lengths of the delays and optionally adding more calls to `digitalWrite`.

```
1  int buzzer_pin = D2;
2
3  void setup()
4  {
5    pinMode( buzzer_pin, OUTPUT );
6  }
7
8  void loop()
9  {
10   digitalWrite( buzzer_pin, HIGH );
11   delay( 500 );
12   digitalWrite( buzzer_pin, LOW );
13   delay( 1000 );
14 }
```

Note: if you want to get your buzzer to play a tune, the code below shows one example of how to do so. You can change the song by changing the note values in the `melody` array. You will also need to update the length of the notes in the `noteDurations` array. If you change the number of notes, change the value in the `for` statement from 12 to whatever number of notes you use. If you would like help coding your own song, ask a teaching assistant or the instructor for assistance.

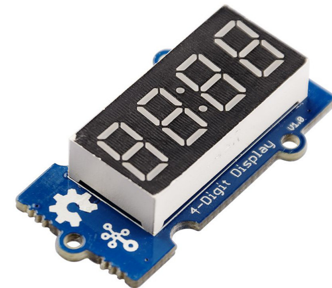
```
int buzzer_pin = D2;

int notes[] =
{ 0,
// C,  C#,   D,  D#,   E,   F,  F#,   G,  G#,   A,  A#,   B
3817,3597,3401,3205,3030,2857,2703,2551,2404,2273,2146,2024,   // 3 (1-12)
1908,1805,1701,1608,1515,1433,1351,1276,1205,1136,1073,1012,   // 4 (13-24)
 956, 903, 852, 804, 759, 716, 676, 638, 602, 568, 536, 506,   // 5 (25-37)
 478, 451, 426, 402, 379, 358, 338, 319, 301, 284, 268, 253,   // 6 (38-50)
 239, 226, 213, 201, 190, 179, 169, 159, 151, 142, 134, 127 }; // 7 (51-62)

#define NOTE_G3  2551
#define NOTE_G4  1276
#define NOTE_C5  956
#define NOTE_E5  759
#define NOTE_G5  638
#define RELEASE  20
#define BPM      100

// notes in the melody:
int melody[] = { NOTE_E5, NOTE_E5, 0, NOTE_E5, 0, NOTE_C5, NOTE_E5,
                 0, NOTE_G5, 0, 0, NOTE_G4 };

// note durations: 4 = quarter note, 2 = half note, etc.:
int noteDurations[] = { 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 4, 4 };

void setup() {
  // iterate over the notes of the melody:
  for ( int thisNote = 0; thisNote < 12; thisNote++ ) {

    // to calculate the note duration, take one second
    // divided by the note type.
    // e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
    int noteDuration = 60 * 1000 / BPM / noteDurations[thisNote];
    int note_value;
    if ( melody[thisNote] == 0 )
        note_value = 0;
    else
        note_value = 500000 / melody[thisNote];
    tone( buzzer_pin, note_value, noteDuration - RELEASE );

    // blocking delay needed because tone() does not block
    delay( noteDuration );
  }
}
```

12. 4-Digit Display Output Module

This output module enables your IoT device to display a sequence of 4 digits. This type of display is called a seven-segment display because each digit has 7 different segments that can light up. You can use the display for showing 4-digit sensor readings, error codes, or warnings. By default, it can display the numbers 0-9 and the letters A-F, which lets you represent all the values in hexadecimal, or base 16 numbers. With some extra work, you can get it to scroll through longer messages and display more characters.

The example code below illustrates how to make the display show the value 5432. You will usually want to connect the display to a digital port, D2 or D4, but you can also connect it to an analog port, A0, A2, or A4. The example code below assumes the display is connected to digital port D2. The display module has an integrated circuit on it which controls the display, and the Particle Argon communicates with this integrated circuit using a series of pulses (a special pattern of 1s and 0s). We will use a library to take care of this communication for us. To add the library to your project, click the library button in the Particle IDE, type "display" into the search box, and click on Grove_4Digit_Display to add it to your project. Before the start of the setup segment, we create a C++ object to handle the communication with the integrated circuit on the display module. In the setup segment, we initialize the display, set it's brightness to typical, and turn the "point" in the middle of the display off. In the loop segment, we assign the values 5, 4, 3, and 2 to the 4 segments of our display.

```
#include <Grove_4Digit_Display.h>

// Creates a C++ object to manage the display
TM1637 display( D2, D3 );

void setup()
{
  display.init();
  display.set( BRIGHT_TYPICAL );
  display.point( POINT_OFF );
}

void loop()
{
  display.display( 0, 5 );
  display.display( 1, 4 );
  display.display( 2, 3 );
  display.display( 3, 2 );
}
```



NOTE: if you want improved functionality from the display, you will need to install a library that is not included in the Paritcle IDE by default. We recommend you ask a TA or the instructor to help you with this, but the steps are as follows. The library can be found at this link: `https://github.com/Seeed-Studio/Grove_4Digital_Display`. To add the library to your project, click the (+) icon in the upper right hand corner of the IDE while your project code is open. Two new tabs will appear at the top of your screen for a .h file and a .cpp file. Rename the files to TM1637.h and TM1637.cpp,then copy the code from the github URL into these two files. Some example code that uses this advanced library is below.

```
1   #include "TM1637.h"
2
3   TM1637 display( D2, D3 );
4
5   void setup()
6   {
7     display.init();
8     display.set( BRIGHT_TYPICAL );
9     display.point( POINT_OFF );
10  }
11
12  void loop()
13  {
14    display.displayStr( "CURIE2021" );
15    delay( 1500 );
16    display.clearDisplay();
17    delay( 500 );
18  }
```