Internet of Things with Python

This book lets you stay at the forefront of cutting-edge research on IoT. We'll open up the possibilities using tools that enable you to interact with the world, such as Intel Galileo Gen 2, sensors, and other hardware. You will learn how to read, write, and convert digital values to generate analog output by programming Pulse Width Modulation (PWM) in Python. You will get familiar with the complex communication system included in the board, so you can interact with any shield, actuator, or sensor.

Later on, you will not only see how to work with data received from the sensors, but also perform actions by sending them to actuators and displays. You’ll be able to connect your IoT device to the entire world with the help of Python packages and cloud-based services. With everything ready, you will see how to work in real time on your IoT device using the MQTT protocol in Python.

By the end of the book, you will be able to develop IoT prototypes with Python, libraries, and tools.

Who this book is written for

The book is ideal for Python developers who want to explore the tools in the Python ecosystem in order to build their own IoT applications and work on IoT-related projects. It is also a very useful resource for developers with experience in other programming languages that want to easily prototype IoT applications with the Intel Galileo Gen 2 board.

What you will learn from this book

- Prototype and develop IoT solutions from scratch with Python as the programming language
- Develop IoT projects with the Intel Galileo Gen 2 board along with Python
- Work with the different components included in the boards using Python and the MRAA library
- Interact with sensors, actuators, and shields
- Work with local and cloud-based storage
- Interact with any electronic device that supports the I2C bus
- Allow mobile devices to interact with the board
- Work with real-time IoT and cloud services
- Understand Big Data and IoT analytics

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'Displaying Information and Performing Actions'
- A synopsis of the book’s content
- More information on Internet of Things with Python
Gastón C. Hillar is Italian and has been working with computers since he was eight. He began programming with the legendary Texas TI-99/4A and Commodore 64 home computers in the early 80s. He has a bachelor’s degree in computer science, in which he graduated with honors, and an MBA, in which he graduated with an outstanding thesis. At present, Gastón is an independent IT consultant and freelance author who is always looking for new adventures around the world.

He has been a senior contributing editor at Dr. Dobb's and has written more than a hundred articles on software development topics. Gaston was also a Microsoft MVP in technical computing. He has received the prestigious Intel® Black Belt Software Developer award seven times.

He is a guest blogger at Intel® Software Network (http://software.intel.com). You can reach him at gastonhillar@hotmail.com and follow him on Twitter at http://twitter.com/gastonhillar.

His blog is http://csharpmulticore.blogspot.com.

He lives with his wife, Vanesa, and his two sons, Kevin and Brandon.
Preface

Internet of Things, also known as IoT, is changing the way we live and represents one of the biggest challenges in the IT industry. Developers are creating low-cost devices that collect huge amounts of data, interact with each other, and take advantage of cloud services and cloud-based storage. Makers all over the world are working on fascinating projects that transform everyday objects into smart devices with sensors and actuators.

A coffee cup is not a simple object anymore—it can send a message to your smartwatch indicating that the liquid inside has the right temperature so that you can drink it without worrying about checking whether it is too hot. In case you move the coffee cup before you receive the message, your wearable vibrates to indicate that you don't have to drink it yet.

You can check the coffee level of the coffee dispenser in your smartphone, and you won't have to worry about ordering more coffee: the coffee dispenser will automatically place an online order to request coffee when the coffee level is not enough to cover the rest of the day. You just need to approve the online order that the coffee dispenser suggests from your smartwatch. Based on certain statistical algorithms, the coffee dispenser will know the appropriate time to make the order.

What happens when more usual visitors arrive at the office? Their smartwatches or smartphones will communicate with the coffee dispensers and they will place orders in case the probable consumption of decaffeinated coffee increases too much. We have smart coffee cups, smart coffee dispensers, smartwatches, smartphones, and wearables. All of them take advantage of the cloud to create a smart ecosystem capable of providing us with all the different types of coffees we need for our day.
The Intel Galileo Gen 2 board is an extremely powerful and versatile minicomputer board for IoT projects. We can boot a Linux version and easily execute Python scripts that can interact with the different components included on the board. This book will teach you to develop IoT prototypes, from selecting the hardware to all the necessary stacks with Python 2.7.3, its libraries, and tools. In case you need a smaller board or an alternative, all the examples included in the book are compatible with Intel Edison boards, and therefore, you can switch to this board in case you need to.

Python is one of the most popular programming languages. It is open source, multiplatform, and you can use it to develop any kind of application, from websites to extremely complex scientific computing applications. There is always a Python package that makes things easier for us in order to avoid reinventing the wheel and solve problems faster. Python is an ideal choice for developing a complete IoT stack. This book covers all the things you need to know to transform everyday objects into IoT projects.

This book will allow you to prototype and develop IoT solutions from scratch with Python as the programming language. You will leverage your existing Python knowledge to capture data from the real world, interact with physical objects, develop APIs, and use different IoT protocols. You will use specific libraries to easily work with low-level hardware, sensors, actuators, buses, and displays. You will learn how to take advantage of all the Python packages with the Intel Galileo Gen 2 board. You will be ready to become a maker and to be a part of the exciting IoT world.

**What this book covers**

*Chapter 1, Understanding and Setting up the Base IoT Hardware*, start us off on our journey towards Internet of Things (IoT) with Python and the Intel Galileo Gen 2 board. We will learn the different features offered by this board and visualize its different components. We will understand the meaning of the different pins, LEDs, and connectors. We will learn to check the board’s firmware version and to update if necessary.

*Chapter 2, Working with Python on Intel Galileo Gen 2*, leads us through many procedures that make it possible to work with Python as the main programming language to create IoT projects with our Intel Galileo Gen 2 board. We will write a Linux Yocto image to a microSD card, configure the board to make it boot this image, update many libraries to use their latest versions, and launch the Python interpreter.
Chapter 3, Interacting with Digital Outputs with Python, teaches us how to work with two different libraries to control digital outputs in Python: mraa and wiring-x86. We will connect LEDs and resistors to a breadboard and write code to turn on between 0 to 9 LEDs. Then, we will improve our Python code to take advantage of Python's object-oriented features, and we will prepare the code to make it easy to build an API that will allow us to print numbers with LEDs with a REST API.

Chapter 4, Working with a RESTful API and Pulse Width Modulation, has us working with Tornado Web Server, Python, the HTTPie command-line HTTP client, and the mraa and wiring-x86 libraries. We will generate many versions of RESTful APIs that will allow us to interact with the board in computers and devices connected to the LAN. We will be able to compose and send HTTP requests that print numbers in LEDs, change the brightness levels for three LEDs, and generate millions of colors with an RGB LED.

Chapter 5, Working with Digital Inputs, Polling and Interrupts, explains the difference between reading pushbutton statuses with polling and working with interrupts and interrupt handlers. We will write code that will allow the user to perform the same actions with either pushbuttons in the breadboard or HTTP requests. We will combine code that reacts to changes in the statuses of the pushbuttons with a RESTful API built with Tornado Web Server. We will create classes to encapsulate pushbuttons and the necessary configurations with the mraa and wiring-x86 libraries.

Chapter 6, Working with Analog Inputs and Local Storage, explains how to work with analog inputs to measure voltage values. We will measure voltages with an analog pin and both the mraa and the wiring-x86 libraries. We will be able to transform a variable resistor into a voltage source and make it possible to measure the darkness level with an analog input, a photoresistor, and a voltage divider. We will fire actions when the environment light changes, and we will work with both analog inputs and outputs. We will register events by taking advantage of the logging features included in the Python standard library and the USB 2.0 connector included in the Intel Galileo Gen 2 board.

Chapter 7, Retrieving Data From the Real World with Sensors, has us working with a variety of sensors to retrieve data from the real world. We will take advantage of the modules and classes included in the upm library that will make it easy for us to start working with analog and digital sensors. We will learn the importance of considering units of measurement because sensors always provide values measured in a specific unit, which we must consider. We will measure the magnitude and direction of proper acceleration or g-force, ambient temperature, and humidity.
Chapter 8, Displaying Information and Performing Actions, teaches us about different displays the we can connect to our board through the I2C bus. We will work with an LCD display with an RGB backlight, and we will then replace it with an OLED dot matrix. We will write code that takes advantage of the modules and classes included in the upm library to work with LCD and OLED displays and show text on them. We will also write code that interacts with an analog servo. We will control the shaft to allow us to create a gauge chart to display the temperature value retrieved with a sensor. Our Python code will make things move.

Chapter 9, Working with the Cloud, teaches you how to combine many cloud-based services that will allow us to easily publish data collected from sensors and visualize it in a web-based dashboard. We will work with the MQTT protocol and its publish/subscribe model to process commands in our board and indicate when the commands are successfully processed through messages. First, we will work with the PubNub cloud that works with the MQTT protocol under the hood. Then, we will develop the same example with Mosquitto and Eclipse Paho. We will be able to write applications that can establish bidirectional communications with our IoT devices.

Chapter 10, Analyzing Huge Amounts of Data with Cloud-based IoT Analytics, explains the close relationship between IoT and Big Data. We will work with Intel IoT Analytics, a cloud-based service that allows us to organize huge amounts of data collected by multiple IoT devices and their sensors. We will use the requests package to write a few lines of Python code to interact with the Intel IoT Analytics REST API. We will learn about the different options that Intel IoT Analytics offers us to analyze huge amounts of data, and we will define rules to trigger alerts.
In this chapter, we will work with a variety of breakout boards and an actuator to display data and perform actions by writing a Python code. We shall:

- Understand LCD displays and their connection types
- Learn the most important things we must consider when choosing LCD displays
- Take advantage of the upm library with LCD displays and actuators
- Use an LCD display with an RGB backlight that works with the I²C bus
- Display and update text in a 16x2 LCD screen
- Use an OLED display that works with the I²C bus
- Display and update text on a 96-by-96 dot matrix OLED display
- Wire a standard servo motor to be controlled with PWM
- Display a value with a servo motor and a shaft

Understanding LCD displays and their connection types

Sometimes, our IoT device has to provide information to the user with any device connected to an Intel Galileo Gen 2 board. We can use different kinds of electronic components, shields, or breakout boards to achieve this goal.
For example, we can use simple LEDs to provide information that we can represent with colors. For example, a red LED that turns on can indicate that our temperature sensor connected to the board has detected that the ambient temperature is higher than 80 degrees Fahrenheit (°F) or 26.66 degrees Celsius (°C). A blue LED that turns on can indicate that our temperature sensor had detected that the ambient temperature is lower than 40 degrees Fahrenheit (°F) or 4.44 degrees Celsius (°C). A red LED turned on can indicate that the temperature is between these two values. These three LEDs allow us to provide valuable information to the user.

We can also achieve the same goal using a single RGB LED and work with pulse width modulation (PWM) to change its color based on the measured ambient temperature value, as we learned in Chapter 4, Working with a RESTful API and Pulse Width Modulation.

However, sometimes colors aren't enough to provide a detailed and accurate information to the user. For example, sometimes we want to display the humidity level with a percentage value and a few LEDs aren't enough to represent numbers from 0 to 100%. If we want to be able to display a 1% step, we would require 100 LEDs. We don't have 100 GPIO pins, and therefore, we would require a shield or breakout board with 100 LEDs and a digital interface such as an I2C bus to allow us to send commands indicating the number of LEDs that we want to be turned on.

In these cases, an LCD screen that allows us to print a specific number of characters might be an appropriate solution. For example, on an LCD screen that allows us to display 16 characters per line, with 2 lines of 16 characters, known as a 16x2 LCD module, we can display the temperature in the first line and the humidity level in the second line. The following table shows an example of each line with the text and the values considering that we have 16 columns and 2 rows for the characters.

<table>
<thead>
<tr>
<th>Temp</th>
<th>40.2°F</th>
<th>4</th>
<th>0</th>
<th>2</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humidity</td>
<td>80%</td>
<td>8</td>
<td>0</td>
<td>%</td>
<td></td>
</tr>
</tbody>
</table>

The 16x2 LCD module provides a clear description for each value, a floating point value and a unit of measure. Thus, we will use a 16x2 LCD module for our example. The following picture shows an example of the location of each character in a 16x2 LCD screen:
There are LCD modules with different features and we must consider a lot of the things we learned when we analyzed sensors in Chapter 7, *Retrieving Data from the Real World with Sensors*. The following list enumerates the most important things that we must consider when we select an LCD module and their description. As we analyzed many of these things when we learned about sensors, we won't repeat the descriptions for the common items.

- **Compatibility with Intel Galileo Gen 2 board and the voltage supply that we are using (5V or 3.3V).**
- **Power consumption.**
- **Connection type**: Some LCD displays consume too many pins, and therefore, it is very important to check all the pins that they require. The most common connection types for LCD displays are the I2C bus, the SPI bus, and the UART port. However, some LCD displays require a bus or port combined with additional GPIO pins.
- **Operating range and special environment requirements.**
- **Dimensions**: LCD displays come with different dimensions. Sometimes only specific dimensions are suitable for our project.
- **Number of columns and rows**: Based on the text we have to display, we will select the LCD display with the appropriate number of columns and rows that can display the characters.
• **Response time:** It is very important to determine how much we can wait for the LCD display to show the new content that replaces the text that is being displayed or to clear the display.

• **Protocol, support in the upm library and Python bindings.**

• **Supported character set and built-in fonts:** Some LCD displays support user-defined characters, and therefore, they allow us to configure and display custom characters. It is also important to check whether the LCD display supports characters for the languages in which we have to display the text.

• **Backlight color, text color and contrast level:** Some LCD displays allow us to change the backlight color while others have a fixed backlight color. An RGB backlight makes it possible to combine red, green, and blue components to determine the desired backlight color. In addition, it is always important to take into account whether the contrast level is appropriate for the light conditions in which you will need to display information.

• **Cost.**

### Wiring an LCD RGB backlight to the I²C bus

In our last example in *Chapter 7, Retrieving Data from the Real World with Sensors*, we worked with a multifunctional digital sensor that provided us with the temperature and relative humidity information. We worked with a breakout board that uses the FC bus to allow the Intel Galileo Gen 2 board to communicate with the sensor. Now, we will add a breakout board with a 16x2 LCD RGB backlight to allow us to display the measured temperature and humidity values with text and numbers.

The LCD RGB backlight breakout board will also be connected to the same FC bus to which the temperature and humidity digital sensor is connected. We can connect many slaves to the FC bus in the Intel Galileo Gen 2 board as long as their have different FC addresses. In fact, the LCD RGB backlight breakout board has two FC addresses: one for the LCD display and the other for the backlight.

We need the following parts to work with this example:

• A SeeedStudio Grove LCD RGB backlight breakout. The following URL provides detailed information about this breakout board: http://www.seeedstudio.com/depot/Grove-LCD-RGB-Backlight-p-1643.html.

The following diagram shows the digital temperature and humidity breakout, the LCD RGB backlight breakout, the necessary wirings, and the wirings from the Intel Galileo Gen 2 board to the breadboard. The Fritzing file for the sample is `iot_fritzing_chapter_08_01.fzz` and the following image is the breadboard view:
Displaying Information and Performing Actions

The following image shows the schematic with the electronic components represented as symbols:
As seen in the previous schematic, we have the following connections:

- The **SDA** pin is connected to both the breakout board pins labeled **SDA**. This way, we connect both the digital temperature and humidity sensor and the LCD backlight to the serial data line for the I2C bus.
- The **SCL** pin is connected to both the breakout board pins labeled **SCL**. This way, we can connect both the digital temperature and humidity sensor and the LCD backlight to the serial clock line for the I2C bus.
- The power pin labeled **3V3** is connected to the digital temperature and humidity sensor breakout board power pin labeled **VCC**.
- The power pin labeled **5V** is connected to the LCD backlight breakout board power pin labeled **VCC**.
- The ground pin labeled **GND** is connected to both the breakout board pins labeled **GND**.

Now, it is time make all the necessary wirings. Don't forget to shut down the Yocto Linux, wait for all the onboard LEDs to turn off and unplug the power supply from the Intel Galileo Gen 2 board before adding or removing any wire from the board's pins.

### Displaying text on an LCD display

The *upm* library includes support for the 16x2 LCD RGB backlight breakout board in the *pyupm_i2clcd* module. The *Jhd1313m1* class declared in this module represents a 16x2 LCD display and its RGB backlight, connected to our board. The class makes it easy to set the color components for the RGB backlight, clear the LCD display, specify the cursor location, and write text through the I2C bus. The class works with the *mraa.I2c* class under the hoods to talk with the RGB backlight and the LCD display. These two devices act as slave devices connected to the I2C bus, and therefore, each of them have a specific address in this bus.

We will take the code we wrote in the previous chapter when we read temperature and humidity values from the sensor and we will use this code as a baseline to add the new features. The code file for the sample was *iot_python_chapter_07_05.py*. 

---

[241]
We will create an `Lcd` class to represent the 16x2 LCD RGB backlight and make it easier for us to set the background color and write the text in two lines without worrying about the specific methods when working with an instance of the `Jhd1313m1` class. We will use the `Jhd1313m1` class to interact with the LCD and its RGB backlight. The following lines show the code for the new `Lcd` class that works with the `upm` library, specifically with the `pyupm_i2clcd` module. The code file for the sample is `iot_python_chapter_08_01.py`.

```python
import pyupm_th02 as upmTh02
import pyupm_i2clcd as upmLcd
import time

class Lcd:
    # The I2C address for the LCD display
    lcd_i2c_address = 0x3E
    # The I2C address for the RBG backlight
    rgb_i2c_address = 0x62

    def __init__(self, bus, red, green, blue):
        self.lcd = upmLcd.Jhd1313m1(
            bus,
            self.__class__.lcd_i2c_address,
            self.__class__.rgb_i2c_address)
        self.lcd.clear()
        self.set_background_color(red, green, blue)

    def set_background_color(self, red, green, blue):
        self.lcd.setColor(red, green, blue)

    def print_line_1(self, message):
        self.lcd.setCursor(0, 0)
        self.lcd.write(message)

    def print_line_2(self, message):
        self.lcd.setCursor(1, 0)
        self.lcd.write(message)
```

---

[242]
The Lcd class declares two class attributes: `lcd_i2c_address` and `rgb_i2c_address`. The first class attribute defines the I2C address for the LCD display, that is, the address that will process the commands that locate the cursor and write text once the cursor is located in a specific row and column. The address is \( 3E \) in hexadecimal (\( 0x3E \)). If we just see a \( 0x3E \) within the code, we don't understand that it is an I2C bus address for the LCD display. The second class attribute defines the I2C address for the RGB backlight, that is, the address that will process the commands that set the red, green, and blue components for the backlight color. The address is \( 62 \) in hexadecimal (\( 0x62 \)). If we just see a \( 0x62 \) within the code, we don't understand that it is an I2C bus address for the RGB backlight. These class attributes make it easier to read the code.

We have to specify the I2C bus number to which the both the 16x2 LCD and the RGB backlight are wired when we create an instance of the Lcd class in the `bus` required argument. In addition, it is necessary to specify the values for the red, green and blue color components to configure the background color for the RGB backlight. The constructor, that is, the `__init__` method, creates a new `upmLcd.Jhd1313m1` instance with the received `bus` argument followed by the `lcd_i2c_address` and `rgb_i2c_address` class attributes and saves the reference for the new instance in the `lcd` attribute. Then, the code calls the `clear` method for the new instance to clear the LCD screen. Finally, the code calls the `set_background_color` method with the red, green, and blue values received as arguments to configure the background color for the RGB backlight.

The class declares the `set_background_color` method that calls the `lcd.setColor` method with the red, green and blue values received as arguments. Under the hoods, the `upmLcd.Jhd1313m1` instance will write data to the slave device whose address is equal to the `rgb_i2c_address` class attribute through the I2C bus to specify the desired value for each color component. We just create a specific method to follow Python naming conventions and make our final code that uses our class easier to read.

The class defines the following two additional methods to make it easy to print text on the first and the second row of the LCD display:

- `print_line_1`
- `print_line_2`
The print_line_1 method calls the setCursor method for the upmLcd.Jhd1313m1 instance (self.lcd), with 0 as the value for both the row and the column argument, to locate the cursor in the first row and the first column. Then, a call to the write method for the the upmLcd.Jhd1313m1 instance (self.lcd) with the message reviewed as a parameter as an argument prints the received string in the LCD display. Under the hoods, the upmLcd.Jhd1313m1 instance will write the data to the slave device whose address is equal to the lcd_i2c_address class attribute through the I2C bus to specify the desired location for the cursor and then to write the specified text starting in the position in which we have located the cursor. The first row is identified with 0, but we named the method print_line_1 because it makes it easier for us to understand that we are writing a message in the first line of the LCD screen.

The print_line_2 method has the same two lines of code than the print_line_1 method with just one difference: the call to the setCursor method specifies 1 as the value for the row argument. This way, the method prints a message in the second line of the LCD screen.

Now, we will create a subclass of the previously coded Lcd class named TemperatureAndHumidityLcd. The subclass will specialize the Lcd class to allow us to easily print a temperature value expressed in degrees Fahrenheit in the first line of the LCD screen and print a humidity value expressed in percentage in the second line of the LCD screen. The following lines show the code for the new TemperatureAndHumidityLcd class. The code file for the sample is iot_python_chapter_08_01.py.

class TemperatureAndHumidityLcd(Lcd):
    def print_temperature(self, temperature_fahrenheit):
        self.print_line_1("Temp. {:5.2f}F".format(temperature_fahrenheit))

    def print_humidity(self, humidity):
        self.print_line_2("Humidity {0}%
                            .format(humidity))

The new class (TemperatureAndHumidityLcd) adds the following two methods to its superclass (Lcd):

- **print_temperature**: Calls the print_line_1 method with the formatted text that displays the temperature value expressed in degrees Fahrenheit (°F) received in the temperature_fahrenheit argument.

- **print_humidity**: Calls the print_line_2 method with the formatted text that displays the humidity level expressed in percentage received in the humidity argument.
Now, we will write a loop that will display the ambient temperature expressed in degrees Fahrenheit (ºF) and the humidity value in the LCD screen, every 10 seconds. The code file for the sample is `iot_python_chapter_08_01.py`.

```python
if __name__ == '__main__':
    temperature_and_humidity_sensor = \n        TemperatureAndHumiditySensor(0)
    lcd = TemperatureAndHumidityLcd(0, 0, 0, 128)

    while True:
        temperature_and_humidity_sensor.\n            measure_temperature_and_humidity()
        lcd.print_temperature(\n            temperature_and_humidity_sensor.temperature_fahrenheit)
        lcd.print_humidity(\n            temperature_and_humidity_sensor.humidity)
        print("Ambient temperature in degrees Celsius: {0}".\n            format(temperature_and_humidity_sensor.temperature_celsius))
        print("Ambient temperature in degrees Fahrenheit: {0}".\n            format(temperature_and_humidity_sensor.temperature_fahrenheit))
        print("Ambient humidity: {0}".\n            format(temperature_and_humidity_sensor.humidity))
        # Sleep 10 seconds (10000 milliseconds)
        time.sleep(10)
```

The highlighted lines show the changes made to the `__main__` method compared with the previous version. The first highlighted line creates an instance of the previously coded `TemperatureAndHumidityLcd` class with 0 as the value of the `bus` argument, 0 for red and green, and 128 for blue to set the background color to light blue. The code saves the reference to this instance in the `lcd` local variable. This way, the instance will establish a communication with the LCD screen and the RGB backlight through the I²C bus. The RGB backlight will display a light blue background.

Then, the code runs a loop forever and the highlighted line calls the `lcd.print_temperature` method with `temperature_and_humidity_sensor.temperature_fahrenheit`, that is, the measured temperature expressed in degrees Fahrenheit (ºF), as an argument. This way, the code displays this temperature value in the first line of the LCD screen.
The next highlighted line calls the `lcd.print_humidity` method with `temperature_and_humidity_sensor.humidity`, that is, the measured humidity expressed in percentage, as an argument. This way, the code displays this humidity value in the second line of the LCD screen.

The following line will start the example:

```
python iot_python_chapter_08_01.py
```

After you run the example, turn on an air conditioner or heating system, to generate a change in the ambient temperature and humidity. The LCD screen will display the temperature and humidity and refresh it every 10 seconds.

**Wiring an OLED dot matrix to the I²C bus**

LCD displays are not the only option when we have to display content on an external screen through the I²C or SPI buses. There are also OLED dot matrixes that allow us to control a specific number of dots. In OLED dot matrices we have control over each dot, instead of controlling each character space. Some of them are grayscale and others RGB.

The key advantage of OLED dot matrixes is that we can display any kind of graphics and not just text. In fact, we can mix any kind of graphics and images with text. The Grove OLED Display 0.96" is an example of a 16 grayscale 96-by-96 dot matrix OLED display module that works with the I²C bus. The following URL provides detailed information about this breakout board: [http://www.seeedstudio.com/depot/Grove-OLED-Display-096-p-824.html](http://www.seeedstudio.com/depot/Grove-OLED-Display-096-p-824.html). The Xadow RGB OLED 96x24 is an example of an RGB color 96-by-64 dot matrix OLED display module that works with the SPI bus. The following URL provides detailed information about this breakout board: [http://www.seeedstudio.com/depot/Xadow-RGB-OLED-96x64-p-2125.html](http://www.seeedstudio.com/depot/Xadow-RGB-OLED-96x64-p-2125.html).

Another option is to work with TFT LCD dot matrices or displays. Some of them include support for touch detection.
Now, we will replace the breakout board with a 16x2 LCD RGB backlight with a 16 grayscale 96-by-96 dot matrix OLED display module that also works with the I2C bus, and we will use this new screen to display similar values with a different configuration. The wirings are compatible with the previous breakout board.

As it happened in our previous example, the dot matrix OLED will also be connected to the same I2C bus to which the temperature and humidity digital sensor is connected. As the dot matrix OLED has an I2C address that is different than the one used by the temperature and humidity digital sensor, we don't have problems to wire the two devices to the same I2C bus.

We need the following additional part to work with this example: A SeeedStudio Grove OLED Display 0.96", 16 grayscale 96-by-96 dot matrix OLED display module. The 96-by-96 dot matrix OLED display provides us the chance to control 9,216 dots, known as pixels. However, in this case, we just want to use the OLED display to display a similar text than the one we displayed in our previous example, but with a different layout.

If we use the default 8-by-8 character box, we have 12 columns (96/8) and 12 rows (96/8) for characters. The following table shows an example of each line with the text and the values.

<table>
<thead>
<tr>
<th>Temperature</th>
<th>Fahrenheit</th>
<th>Celsius</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.2</td>
<td>4.55</td>
<td></td>
</tr>
<tr>
<td>Humidity</td>
<td>Level</td>
<td></td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table shows an example of each line with the text and the values.

<table>
<thead>
<tr>
<th>T</th>
<th>e</th>
<th>m</th>
<th>p</th>
<th>e</th>
<th>r</th>
<th>a</th>
<th>t</th>
<th>u</th>
<th>r</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>a</td>
<td>h</td>
<td>r</td>
<td>e</td>
<td>n</td>
<td>h</td>
<td>e</td>
<td>i</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>e</th>
<th>l</th>
<th>s</th>
<th>i</th>
<th>u</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>.</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
<th>u</th>
<th>m</th>
<th>i</th>
<th>d</th>
<th>i</th>
<th>t</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>e</td>
<td>v</td>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 8   | 0 | % |

---

[247]
The possibility to work with 12 columns and 12 rows of characters allows us to provide a very clear description for each value. In addition, we are able to display the temperature values expressed in both degrees Fahrenheit and degrees Celsius. The following picture shows an example of the location of each character in the 96-by-96 dot matrix OLED display module with an 8-by-8 character box.

After we replace the LCD screen breakout board with the OLED module, we will have the following connections:

- The SDA pin is connected to both breakout board pins labeled SDA. This way, we connect both the digital temperature and humidity sensor and the OLED module to the serial data line for the I2C bus.
- The SCL pin is connected to both the breakout board pins labeled SCL. This way, we connect both the digital temperature and humidity sensor and the OLED module to the serial clock line for the I2C bus.
- The power pin labeled 3V3 is connected to the digital temperature and humidity sensor breakout board power pin labeled VCC.
- The power pin labeled 5V is connected to the OLED module power pin labeled VCC.
- The ground pin labeled GND is connected to both the breakout board pins labeled GND.

Now, it is time to make all the necessary wirings. Don't forget to shutdown the Yocto Linux, wait for all the onboard LEDs to turn off, and unplug the power supply from the Intel Galileo Gen 2 board before adding or removing any wire from the board's pins.
Displaying text on an OLED display

The upm library includes support for the SeeedStudio Grove OLED display 0.96", 16 grayscale 96-by-96 dot matrix OLED display breakout board the in the pyupm_i2clcd module. As this OLED display uses SSD1327 driver integrated circuit, the SSD1327 class declared in this module represents a 96-by-96 dot matrix OLED display, connected to our board. The class makes it easy to clear the OLED screen, draw bitmap images, specify the cursor location, and write text through the I2C bus. The class works with the mraa.I2c class under the hoods to talk with the OLED display.

We will create a new Oled class that will represent the 96-by-96 dot matrix OLED and will use its default 8-by-8 character box to display text. We will use the SSD1327 class to interact with the OLED display. The following lines show the code for the new Oled class that works with the upm library, specifically with the pyupm_i2clcd module and its SSD1327 class. The code file for the sample is iot_python_chapter_08_02.py:

```python
class Oled:
    # The I2C address for the OLED display
    oled_i2c_address = 0x3C

    def __init__(self, bus, red, green, blue):
        self.oled = upmLcd.SSD1327(
            bus,
            self.__class__.oled_i2c_address)
        self.oled.clear()

    def print_line(self, row, message):
        self.oled.setCursor(row, 0)
        self.oled.setGrayLevel(12)
        self.oled.write(message)
```

The Oled class declares the oled_i2c_address class attribute that defines the I2C address for the OLED display, that is, the address that will process the commands that locate the cursor and write text once the cursor is located in a specific row and column. The address is 3C in hexadecimal (0x3C).

We have to specify the I2C bus number to which the OLED display is wired when we create an instance of the Oled class in the bus required argument. The constructor, that is, the __init__ method, creates a new upmLcd.SSD1327 instance with the received bus argument followed by the oled_i2c_address class attribute, and saves the reference for the new instance in the oled attribute. Finally, the code calls the clear method for the new instance to clear the OLED screen.
The class declared the `print_line` method to make it easy to print text on a specific row. The code calls the `setCursor` method for the `upmLcd.SSD1327` instance (self. oled), with the received row value as the value for the row argument and 0 for the column argument, to locate the cursor in the specified row and the first column. Then, a call to the `setGrayLevel` and the `write` method for the the `upmLcd.SSD1327` instance (self.oled) with the message received as a parameter as an argument prints the received string in the OLED display with the default 8-by-8 character box with the gray level set to 12. Under the hoods, the `upmLcd.SSD1327` instance will write data to the slave device whose address is equal to the `oled_i2c_address` class attribute through the I2C bus to specify the desired location for the cursor and then to write the specified text starting in the position in which we have located the cursor.

Now, we will create a subclass of the previously coded `Oled` class named `TemperatureAndHumidityOled`. The subclass will specialize the `Oled` class to allow us to easily print a temperature value expressed in degrees Fahrenheit, the temperature value expressed in degrees Celsius and a humidity value expressed in percentage. We will use the previously explained layout for the text. The following lines show the code for the new `TemperatureAndHumidityOled` class. The code file for the sample is `iot_python_chapter_08_02.py`.

```python
class TemperatureAndHumidityOled(Oled):
    def print_temperature(self, temperature_fahrenheit, temperature_celsius):
        self.oled.clear()
        self.print_line(0, "Temperature")
        self.print_line(2, "Fahrenheit")
        self.print_line(3, "{:5.2f}".format(temperature_fahrenheit))
        self.print_line(5, "Celsius")
        self.print_line(6, "{:5.2f}".format(temperature_celsius))

    def print_humidity(self, humidity):
        self.print_line(8, "Humidity")
        self.print_line(9, "Level")
        self.print_line(10, "{0}%".format(humidity))
```

The new class (`TemperatureAndHumidityOled`) adds the following two methods to its superclass (Oled):

- **print_temperature**: Calls the `print_line` method many times to display the temperature in both degrees Fahrenheit (°F) and Celsius (°C) received as arguments
- **print_humidity**: Calls the `print_line` method many times to display the humidity value received as an argument
In this case, we refresh many lines to change just a few values. As we will run a loop every 10 seconds, it won’t be a problem. However, in other cases in which we want to update values in a shorter amount of time, we can write optimized code that just clears a single line and updates the specific value in this line.

Now, we will write a loop that will display the ambient temperature expressed in Fahrenheit (°F) and Celsius (°C) and the humidity value in the OLED screen, every 10 seconds. The code file for the sample is `iot_python_chapter_08_02.py`.

```python
if __name__ == '__main__':
    temperature_and_humidity_sensor = TemperatureAndHumiditySensor(0)
    oled = TemperatureAndHumidityOled(0)

    while True:
        temperature_and_humidity_sensor.measure_temperature_and_humidity()
        oled.print_temperature(
            temperature_and_humidity_sensor.temperature_fahrenheit,
            temperature_and_humidity_sensor.temperature_celsius)
        oled.print_humidity(
            temperature_and_humidity_sensor.humidity)
        print("Ambient temperature in degrees Celsius: {0}".
            format(temperature_and_humidity_sensor.temperature_celsius))
        print("Ambient temperature in degrees Fahrenheit: {0}".
            format(temperature_and_humidity_sensor.temperature_fahrenheit))
        print("Ambient humidity: {0}".
            format(temperature_and_humidity_sensor.humidity))
        # Sleep 10 seconds (10000 milliseconds)
        time.sleep(10)
```

The highlighted lines show the changes made in the `__main__` method compared with the previous version. The first highlighted line creates an instance of the previously coded `TemperatureAndHumidityOled` class with 0 as the value of the `bus` argument. The code saves the reference to this instance in the `oled` local variable. This way, the instance will establish a communication with the OLED screen through the I2C bus.
Then, the code runs a loop forever and the highlighted line calls the `oled.print_temperature` method with `temperature_and_humidity_sensor.temperature_fahrenheit` and `temperature_and_humidity_sensor.temperature_celsius` as arguments. This way, the code displays both temperature values in the first lines of the OLED screen.

The next highlighted line calls the `oled.print_humidity` method with `temperature_and_humidity_sensor.humidity`. This way, the code uses many lines to display this humidity value at the bottom of the OLED screen.

The following line will start the example:

```
python iot_python_chapter_08_02.py
```

After you run the example, turn on an air conditioner or a heating system to generate a change in the ambient temperature and humidity. The OLED screen will display the temperature and humidity and refresh it every 10 seconds.

**Wiring a servo motor**

So far, we have been using sensors to retrieve data from the real world and we displayed information in LCD and OLED displays. However, IoT devices are not limited to sensing and displaying data, they can also move things. We can connect different components, shields, or breakout boards to our Intel Galileo Gen 2 board and write Python code to move things connected to the board.

Standard servo motors are extremely useful to precisely control a shaft and position it at various angles, usually between 0 and 180 degrees. In Chapter 4, *Working with a RESTful API and Pulse Width Modulation*, we worked with pulse width modulation, known as PWM, to control the brightness of an LED and a RGB LED. We can also use PWM to control a standard analog servo motor and position its shaft at a specific angle.

---

Standard servo motors are DC motors that includes gears and feedback control loop circuitry that provides precision positioning. They are ideal for pinion steering, robot arms and legs, among other usages that require a precise positioning. Standard servo motors don't require motor drivers.
Obviously, not all servos have the same features and we must take into account many of them when we select a specific servo motor for our project. It depends on what we need to position, the accuracy, the required torque, the optimal servo rotational velocity, among other factors. In this case, we will focus on the usage of PWM to position a standard servo motor. However, you cannot use the same servo to rotate a lighter plastic piece than the one you will need to rotate a heavy robotic arm. It is necessary to research about the appropriate servo for each task.

Now, we will wire a standard high sensitive mini servo motor to our existing project and we will rotate the shaft to display the measured temperature expressed in degrees Fahrenheit with the shaft. The shaft will allow us to display the measured temperature in a half circle protractor that measures angles in degrees and will display the number for the angle from 0 to 180 degrees. The combination of the servo with the shaft and the protractor will allow us to display the temperature with moving parts. Then, we can create our own protractor with a scale that can add colors, specific thresholds and many other visual artifacts to make temperature measurement funnier. Specifically, we can create a gauge chart, speedometer or semicircle donut, that is, a combination of a doughnut chart and a pie chart in a single chart with the different temperature values. The following picture shows an example of a half circle protractor that we can use in combination with the servo with the shaft.

![Half Circle Protractor](image)

We need the following additional part to work with this example: A SeeedStudio Grove Servo or a EMAX 9g ES08A High Sensitive Mini Servo. The following URLs provide detailed information about these servos: [http://www.seeedstudio.com/depot/Grove-Servo-p-1241.html](http://www.seeedstudio.com/depot/Grove-Servo-p-1241.html) and [http://www.seeedstudio.com/depot/EMAX-9g-ES08A-High-Sensitive-Mini-Servo-p-760.html](http://www.seeedstudio.com/depot/EMAX-9g-ES08A-High-Sensitive-Mini-Servo-p-760.html).
Displaying Information and Performing Actions

The following diagram shows the digital temperature and humidity breakout, the LCD RGB backlight breakout, the mini servo, the necessary wirings and the wirings from the Intel Galileo Gen 2 board to the breadboard. The Fritzing file for the sample is `iot_fritzing_chapter_08_03.fzz` and the following picture is the breadboard view:
The following picture shows the schematic with the electronic components represented as symbols:

As seen in the previous schematic, we added the following additional connections to our existing project:

- The power pin labeled **5V** in the board's symbol is connected to the servo's pin labeled +. Servos usually use a red wire for this connection.
- The PWM capable GPIO pin labeled **D3 PWM** in the board's symbol is connected to the servo's pin labeled **PULSE**. Servos usually use a yellow wire for this connection.
- The ground pin labeled **GND** in the board's symbol is connected to the servo's pin labeled -. Servos usually use a black wire for this connection.
Now, it is time to make all the necessary wirings. Don't forget to shut down the Yocto Linux, wait for all the onboard LEDs to turn off, and unplug the power supply from the Intel Galileo Gen 2 board before adding or removing any wire from the board's pins.

**Positioning a shaft to indicate a value with a servo motor**

We can use the `mraa.Pwm` class to control PWM on the PWM capable GPIO pin labeled ~3, as we learned in *Chapter 4, Working with a RESTful API and Pulse Width Modulation*. However, this would require us to read the detailed specs for the servo. The `upm` library includes support for both the SeeedStudio Grove Servo or the EMAX 9g ES08A High Sensitive Mini Servo in the `pyupm_servo` module. The `ES08A` class declared in this module represents any of the two mentioned servos connected to our board.

The class makes it easy to set the desired angle for the servo shaft and work with angles instead of duty cycles and other PWM details. The class works with the `mraa.Pwm` class under the hoods to configure PWM and control the duty cycle based on the desired angle for the shaft.

We will take the code we wrote in the previous example and we will use this code as a baseline to add the new features. The code file for the sample was `iot_python_chapter_08_02.py`.

We will create a `TemperatureServo` class to represent the servo and make it easier for us to position the shaft in a valid angle (from 0 to 180 degrees) based on the temperature expressed in degrees Fahrenheit. We will use the `ES08A` class to interact with the servo. The following lines show the code for the new `TemperatureServo` class that works with the `upm` library, specifically with the `pyupm_servo` module. The code file for the sample is `iot_python_chapter_08_03.py`.

```python
import pyupm_th02 as upmTh02
import pyupm_i2clcd as upmLcd
import pyupm_servo as upmServo
import time

class TemperatureServo:
    def __init__(self, pin):
        self.servo = upmServo.ES08A(pin)
        self.servo.setAngle(0)
```

[256]
def print_temperature(self, temperature_fahrenheit):
    angle = temperature_fahrenheit
    if angle < 0:
        angle = 0
    elif angle > 180:
        angle = 180
    self.servo.setAngle(angle)

We have to specify the pin number to which the servo is connected when we create an instance of the TemperatureServo class in the pin required argument. The constructor, that is, the __init__ method, creates a new upmServo.ES08A instance with the received pin as its pin argument, saves its reference in the servo attribute and calls its setAngle with 0 as the value for the angle required argument. This way, the underlying code will configure the output duty cycle for the PWM enabled GPIO pin based on the received value in the angle argument to position the shaft at the desired angle. In this case, we want the shaft to be positioned at 0 degrees.

The class defines a print_temperature method that receives a temperature value expressed in degrees Fahrenheit (°F) in the temperature_fahrenheit argument. The code defines an angle local variable that makes sure that the desired angle for the shaft is in a valid range: from 0 to 180 (inclusive). If the value received in the temperature_fahrenheit argument is lower than 0, the angle value will be 0. If the value received in the temperature_fahrenheit argument is greater than 180, the angle value will be 180. Then, the code calls the setAngle method for the upmServo.ES08A instance (self.servo) with angle as an argument. Under the hoods, the upmServo.ES08A instance will configure the output duty cycle for the PWM enabled GPIO pin based on the received value in the angle argument to position the shaft at the desired angle. This way, the shaft will position at an angle that will be the same than the received temperature in degrees Fahrenheit (°F), as long as the temperature value is between 0 and 180 degrees Fahrenheit (°F).

In case it is too cold, (less than 0 degrees Fahrenheit) the shaft will stay at a 0 degrees angle. In case the temperature is higher than 180 degrees Fahrenheit, the shaft will stay at a 180 degrees angle.

Now, we will make changes to our main loop to display the ambient temperature expressed in Fahrenheit (°F) with the shaft, every 10 seconds. The code file for the sample is iot_python_chapter_08_03.py.

if __name__ == "__main__":
    temperature_and_humidity_sensor = \n        TemperatureAndHumiditySensor(0)
    oled = TemperatureAndHumidityOled(0)
    temperature_servo = TemperatureServo(3)
    while True:
Displaying Information and Performing Actions

```python
temperature_and_humidity_sensor.m
measure_temperature_and_humidity()
oled.print_temperature(
    temperature_and_humidity_sensor.temperature_fahrenheit,
    temperature_and_humidity_sensor.temperature_celsius)
oled.print_humidity(
    temperature_and_humidity_sensor.humidity)
temperature_servo.print_temperature(
    temperature_and_humidity_sensor.temperature_fahrenheit)
print("Ambient temperature in degrees Celsius: {0}".
    format(temperature_and_humidity_sensor.temperature_celsius))
print("Ambient temperature in degrees Fahrenheit: {0}".
    format(temperature_and_humidity_sensor.temperature_fahrenheit))
print("Ambient humidity: {0}".
    format(temperature_and_humidity_sensor.humidity))
# Sleep 10 seconds (10000 milliseconds)
time.sleep(10)
```

The highlighted lines show the changes made to the main method compared with the previous version. The first highlighted line creates an instance of the previously coded TemperatureServo class with 3 as the value of the pin argument. The code saves the reference to this instance in the temperature_servo local variable. This way, the instance will configure PWM for pin number 3 and position the shaft at 0 degrees.

Then, the code runs a loop forever and the highlighted line calls the temperature_servo.print_temperature method with temperature_and_humidity_sensor.temperature_fahrenheit as an argument. This way, the code makes the shaft point to the temperature value in the protractor.

The following line will start the example.

```python
python iot_python_chapter_08_03.py
```

After you run the example, turn on an air conditioner or a heating system and generate a change in the ambient temperature. You will notice how the shaft starts moving to reflect the changes in the temperature every 10 seconds.
Test your knowledge

1. The Intel Galileo Gen 2 board works as an I²C bus master and allows us to:
   1. Connect many slaves to the I²C bus as long as their have different I²C addresses.
   2. Connect many slaves to the I²C bus as long as their have the same I²C addresses.
   3. Connect a maximum of two slaves to the I²C bus as long as their have different I²C addresses.

2. A 16x2 LCD module allows us to display:
   1. Two lines of text with 16 characters each.
   2. Sixteen lines of text with 2 characters each.
   3. Sixteen lines of text with 3 characters each.

3. A 16 grayscale 96-by-96 dot matrix OLED display module allows us to control:
   1. 96 lines of text with 96 characters each.
   2. A single line with 96 dots or 96 characters, based on how we configure the OLED display.
   3. 9,216 dots (96*96).

4. A 16 grayscale 96-by-96 dot matrix OLED display with an 8-by-8 character box allows us to display:
   1. 96 lines of text with 96 characters each: 96 columns and 96 rows.
   2. 16 lines of text with 16 characters each: 16 columns and 16 rows.
   3. 12 lines of text with 12 characters each: 12 columns and 12 rows.

5. Standard servos allow us to:
   1. Display text on an OLED display.
   2. Position the shaft at various specific angles.
   3. Move the shaft to a specific location by specifying the desired latitude and longitude.
Summary
In this chapter, we learned about different displays the we could connect to our board through the I2C bus. We worked with an LCD display, an RGB backlight, and then replaced it with an OLED dot matrix.

We wrote the code that took advantage of the modules and classes included in the upm library that made it easier for us to work with LCD and OLED display and show text on them. In addition, we wrote the code that interacted with an analog servo. Instead of writing our own code to set the output duty cycle based on the desired position for the shaft, we took advantage of a specific module and a class in the upm library. We could control the shaft to allow us to create a gauge chart to display the temperature value retrieved with a sensor. Our Python code could make things move.

Now that we are able to show data next to the board and work with servos, we will connect our IoT device to the entire world and work with cloud services, which is the topic of the next chapter.
Where to buy this book

You can buy Internet of Things with Python from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.