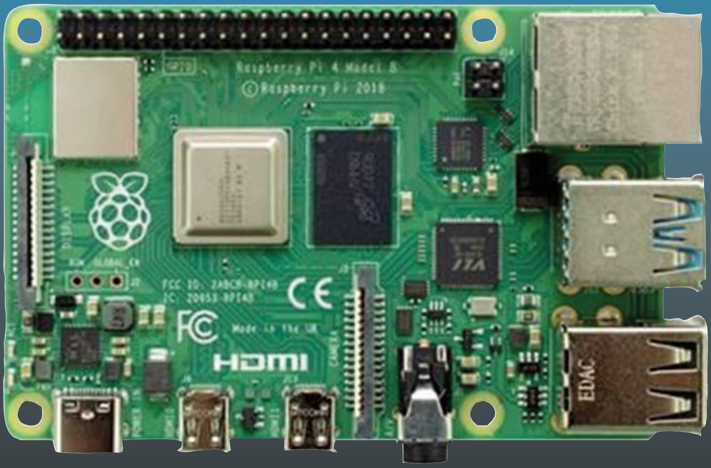
TECHNOLOGY IN ACTION™



Raspberry Pi IoT Projects

Prototyping Experiments for Makers

Second Edition

John C. Shovic

Apress®

Raspberry Pi IoT Projects

Prototyping Experiments for Makers

Second Edition

John C. Shovic

Apress®

Raspberry Pi IoT Projects: Prototyping Experiments for Makers

John C. Shovic
Spokane Valley, WA, USA

ISBN-13 (pbk): 978-1-4842-6910-7 ISBN-13 (electronic): 978-1-4842-6911-4
https://doi.org/10.1007/978-1-4842-6911-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 NY Plaza, New York, NY 10014. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6910-7. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my best friend Laurie and to my students that inspire me every day.

Table of Contents

About the Author

Dr. John C. Shovic is currently a Professor of Computer Science at the University of Idaho specializing in AI and robotics. He is also Chief Technical Officer of SwitchDoc Labs, a company specializing in technical products for the Maker Movement and the IoT. He was also Chief Technology Strategist at Stratus Global Partners with a focus on supplying expertise in computer security regulatory and technical areas to health-care providers. He has worked in the industry for over 30 years and has founded seven companies: Advanced Hardware Architectures, TriGeo Network Security, Blue Water Technologies, MiloCreek, InstiComm, SwitchDoc Labs, and bankCDA. As a founding member of the bankCDA board of directors, he currently serves as the Chairman of the Technology Committee. He has also served as a Professor of Computer Science at Eastern Washington University and Washington State University. Dr. Shovic has given over 80 invited talks and has published over 70 papers on a variety of topics on Arduinos/Raspberry Pis, HIPAA, GLB, computer security, computer forensics, robotics, AI, and embedded systems.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/ SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Acknowledgments

I would like to acknowledge the hard work of the Apress editorial team in putting this book together. I would also like to acknowledge the hard work of the Raspberry Pi Foundation and the Arduino group for putting together products and communities that help to make the Internet of Things more accessible to the general public. Hurray for the democratization of technology! Of course, I have to mention my grandchildren (Lincoln, Hazel, Makenna and Madelyn) that I am constantly building projects to entertain and education them.

Introduction

The Internet of Things (IoT) is a complex concept made up of many computers and many communication paths. Some IoT devices are connected to the Internet and some are not. Some IoT devices form swarms that communicate among themselves. Some are designed for a single purpose, while some are more general-purpose computers. This book is designed to show you the IoT from the inside out. By building IoT devices, the reader will understand the basic concepts and will be able to innovate using the basics to create their own IoT applications.

These included projects will show the reader how to build their own IoT projects and to expand upon the examples shown. The importance of computer security in IoT devices is also discussed as well as various techniques for keeping the IoT safe from unauthorized users or hackers. The most important takeaway from this book is in building the projects yourself.

Chapters at a Glance

In this book, we build examples of all the major parts of simple and complex IoT devices.

In Chapter 1, the basic concepts of IoT are explained in basic terms, and you will learn what parts and tools are needed to start prototyping your own IoT devices.

In Chapter 2, you'll learn how to sense the environment with electronics and that even the behavior of simple LightSwarm type of devices can be very unpredictable.

Chapter 3 introduces important concepts about how to build real systems that can respond to power issues and programming errors by the use of good system design and watchdogs.

Chapter 4 turns a Raspberry Pi into a battery-powered device that senses iBeacons and controls the lighting in a house while reporting your location to a server.

In Chapter 5, you'll do IoT the way the big boys do by connecting to the IBM Bluemix IoT server and sending your biometric pulse rates for storage and display.

In Chapter 6, we'll build a small RFID inventory system and use standard protocols like MQTT to send information to a Raspberry Pi, a complete IoT product.

Chapter 7 shows the dark side of the IoT, computer security. The way you protect your IoT device from hackers and network problems is the most difficult part of IoT device and system design.

Are you totally secure? You will never know. Plan for it.

The reference appendix provides resources for further study and suggestions for other projects.

CHAPTER 1

Introduction to IoT

Chapter Goal: Understand What the IoT Is and How to Prototype IoT Devices

Topics Covered in This Chapter:

- What is IoT
- Choosing a Raspberry Pi Model
- Choosing your IoT device
- Characterization of IoT devices
- Buying the right tools to deal with hardware
- Writing code in Python and in the Arduino IDE

The IoT is a name for the vast collection of "things" that are being networked together in the home and workplace (up to 20 billion by 2022 according to Gardner, a technology consulting firm). That is a very vast collection. And they may be underestimating it.

We all have large numbers of computers in a modern house. I just did a walk-through of my house, ignoring my office (which is filled with about another 100 computers). I found 65 different devices having embedded computers. I'm sure I missed some of them. Now of those computer-based devices, I counted 20 of them that have IP addresses, although I know that I am missing a few (such as the thermostat). So in a real sense, this house has 20 IoT devices. And it is only 2020 as of the writing of this book. With over 100 million households in the United States alone, 20 billion IoT devices somehow don't seem so many.

J. C. Shovic, *Raspberry Pi IoT Projects*, https://doi.org/10.1007/978-1-4842-6911-4_1

So what are the three defining characteristics of the IoT?

- Networking – These IoT devices talk to one another (M2M communication) or to servers located in the local network or on the Internet. Being on the network allows the device the common ability to consume and produce data.
- Sensing – IoT devices sense something about their environment.
- Actuators – IoT devices that do something, for example, lock doors, beep, turn lights on, or turn the TV on.

Of course, not every IoT device will have all three, but these are the characteristics of what we will find out there.

Is the IoT valuable? Will it make a difference? Nobody is sure what the killer application will be, but people are betting huge sums of money that there will be a killer application. Reading this book and doing the projects will teach you a lot about the technology and enable you to build your own IoT applications.

Choosing a Raspberry Pi Model

The Raspberry Pi family of single-board computers (see Figure 1-1) is a product of the Raspberry Pi Foundation (RaspberryPi.org). They have sold over 9 million of these small, inexpensive computers. The Raspberry Pi runs a number of different operating systems, the most common of which is the Raspbian release of Ubuntu Linux.

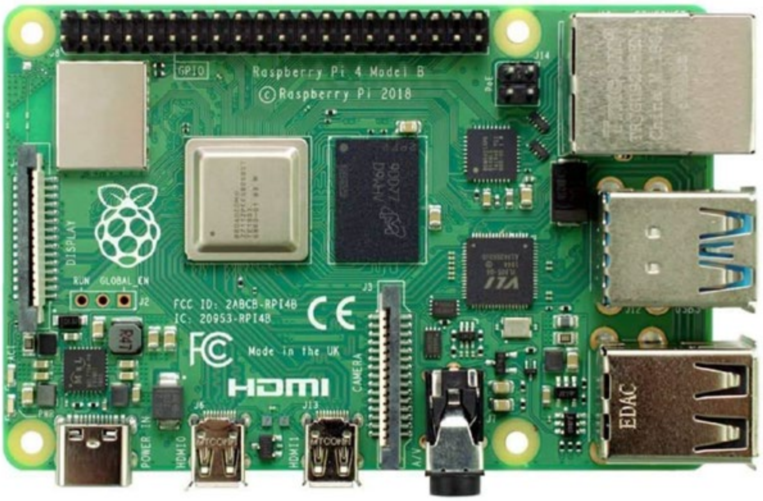


Figure 1-1. *Raspberry Pi 4*

Like Windows, Linux is a multitasking operating system, but unlike Windows, it is an open source system. You can get all the source code and compile it if you wish, but I would not recommend that to a beginner.

One of the best parts of the Raspberry Pi is that there are a huge number of device and sensor drivers available, which makes it a good choice for building IoT projects, especially using it as a server for your IoT project. The Raspberry Pi is not a low-power device, which limits its usage as an IoT device. However, it is still a great prototyping device and a great server.

There is a rather bewildering variety of Raspberry Pi boards available. I suggest for this book that you get a Raspberry PI 3B+ or Raspberry Pi 4. While the $10 Raspberry Pi Zero W is tempting, it takes quite a bit of other hardware to get it to the point where it is usable. While the Raspberry Pi 4 is more expensive ($35), it comes with a WiFi interface built in and extra USB ports.

Note that we are using the Raspberry Pi 4 for building the IoT SkyWeather Weather Station later in this book. The reason for that is computer power! We are doing a lot with the CPU in that project, using cameras and decoding 433MHz signals!

There are many great tutorials on the Web for setting up your Raspberry Pi and getting the operating system software running.

Choosing an IoT Device

If you think the list of Raspberry Pi boards available is bewildering, then wait until you look at the number of IoT devices that are available. While each offering is interesting and has unique features, I am suggesting the following devices for your first projects in the IoT. Note that I selected these based upon the ability to customize the software and to add your own devices without hiding *all* the complexity, hence reducing the learning involved. That is why I am not using Lego-type devices in this book.

We will be using the following:

- ESP8266-based boards (specifically the Adafruit Huzzah ESP8266)
- Arduino Uno and Arduino Mega2560 boards

Characterizing an IoT Project

When looking at a new project, the first thing to do to understand an IoT project is to look at the six different aspects for characterizing an IoT project:

- Communications
- Processor power
- Local storage
- Power consumption
- Functionality
- Cost

When I think about these characteristics, I like to rate each one on a scale from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications. Scoring each one forces me to think carefully about how a given project falls on the spectrum of suitability.

Communications

Communications are important to IoT projects. In fact, communications are core to the whole genre. There is a trade-off for IoT devices. The more complex the protocols and higher the data rates, the more powerful processor you need and the more electrical power the IoT device will consume.

TCP/IP based communications (think web servers; HTTP-based communication (like REST servers); streams of data; UDP – see Chapter 2) provide the most flexibility and functionality at a cost of processor and electrical power.

Low-power Bluetooth and Zigbee types of connections allow much lower power for connections with the corresponding decrease in bandwidth and functionality.

IoT projects can be all over the map with requirements for communication flexibility and data bandwidth requirements.

IoT devices having full TCP/IP support are rated the highest in this category, but will probably be marked down in other categories (such as power consumption).

Processor Power

There are a number of different ways of gauging processor power. Processor speed, processor instruction size, and operating system all play in this calculation. For most IoT sensor and device applications, you will not be limited by processor speed as they are all pretty fast. However, there is one exception to this. If you are using encryption and decryption techniques (see Chapter 7), then those operations are computationally expensive and require more processor power to run. The trade-off can be that you have to transmit or receive data much more slowly because of the computational requirements of encrypting/decrypting the data. However, for many IoT projects, this is just fine.

Higher processor power gives you the highest ratings in this category.

Local Storage

Local storage refers to all three of the main types of storage: RAM, EEPROM, and Flash Memory.

RAM (Random Access Memory) is high-data rate, read/writable memory, generally used for data and stack storage during execution of the IoT program. EEPROM (Electrically Erasable Programmable Read-Only Memory) is used for writing small amounts of configuration information for the IoT device to be read on power up. Flash Memory is generally used for the program code itself. Flash is randomly readable (e.g., as the code executes), but can only be written in large blocks and very slowly. Flash is what you are putting your code into with the Arduino IDE (see Chapter 2).

The amount of local storage (especially RAM) will add cost to your IoT device. For prototyping, the more, the merrier. For deployment, less is better as it will reduce your cost.

Power Consumption

Power consumption is the bane of all IoT devices. If you are not plugging your IoT device in the wall, then you are running off of batteries or solar cells and every single milliwatt counts in your design. Reducing power consumption is a complex topic that is well beyond the introductory projects in this book. However, the concepts are well understood by the following:

- Put your processor in sleep mode as much as possible.
- Minimize communication outside of your device.
- Try to be interrupt driven and not polling driven.
- Scour your design looking for every unnecessary amount of current.

The higher the number in this category, the less power the IoT unit uses.

Functionality

This is kind of a catch-all category that is quite subjective. For example, having additional GPIOs (General-Purpose Input/Outputs) available is great for flexibility. I am continuously running into GPIO limitations with the Adafruit Huzzah ESP8266 as there are so few pins available. Having additional serial interfaces is very useful for debugging. Special hardware support for encryption and decryption can make device computer security much simpler. One of the things that I miss in most IoT prototyping system is software debugging support hardware.

I also include the availability of software libraries for a platform into this category. A ten means very high functionality; low numbers mean limited functionality.

Cost

What is an acceptable cost for your IoT device? That depends on the value of the device and the market for your device. A $2.50 price can be great for prototypes, but will be the death of the product in production. You need to size the price to the product and the market. High numbers are low-cost units, and low numbers are higher-cost devices.

The Right Tools to Deal with Hardware

Anything is more difficult without the right tools. When you make the jump from just doing software to doing a software/hardware mix, here is a list of tools you should have:

- 30W adjustable temperature soldering iron – Heating and connecting wires
- Soldering stand – To hold the hot soldering iron
- Solder, rosin-core, 0.031" diameter, 1/4 lb (100g) spool – To solder with
- Solder sucker – Useful in cleaning up mistakes
- Solder wick/braid 5 ft spool – Used along with the solder sucker to clean up soldering messes
- Panavise Jr. – General-purpose 360-degree mini-vise
- Digital multimeter – A good-all-around basic multimeter
- Diagonal cutters – Trimming of wires and leads
- Wire strippers – Tool for taking insulation off wires
- Micro needle-nose pliers – For bending and forming components
- Solid-core wire, 22AWG, 25 ft spools – Black, red, and yellow for bread-boarding and wiring

Adafruit has an excellent beginner's kit for $100 [www.adafruit.com/products/136]. Figure 1-2 shows the tools that are in it.

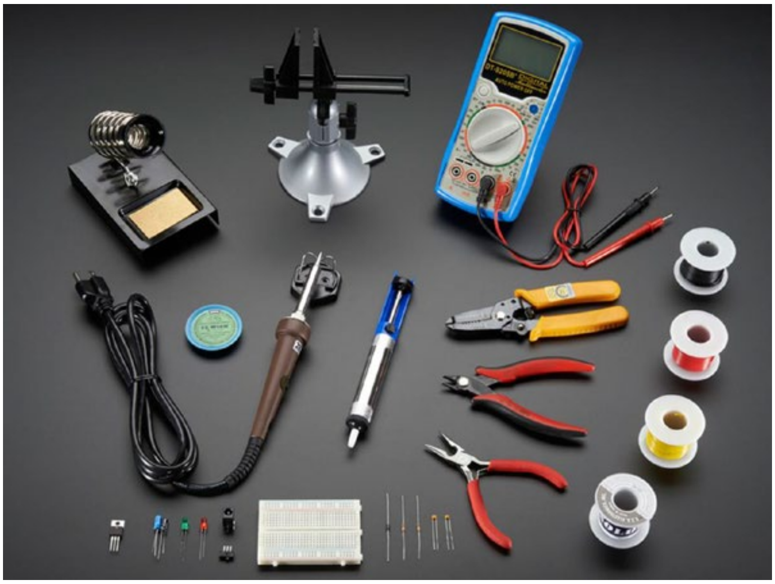


Figure 1-2. *Adafruit Electronics Toolkit*

Writing Code in Python and the Arduino IDE

All the code in this book is in two languages. Specifically, Python is used for the Raspberry Pi and C/C++ (don't be scared, there are many examples and resources) for the Arduino IDE.

Python is a high-level, general-purpose programming language. It is designed to emphasize code readability, and it especially keeps you out of having loose pointers (a curse of all C/C++ programmers) and does the memory management for you. This is the programming language of choice for the Raspberry Pi. Python has the largest set of libraries for IoT and embedded system devices of any language for the Raspberry Pi. All of the examples in this book that use the Raspberry Pi use Python. I am using Python 2.7 in this book, but it is relatively easy to convert to Python 3.5. However, it is not so trivial to find all the libraries for Python 3.5, so I suggest staying with Python 2.7.

Why are we using C++ for the majority of the IoT devices? There are four reasons for this:

- C programs are compiled into native code for these small devices, giving you much better control over size and timing. Python requires an interpreter, which is a large amount of code that would not fit on small IoT devices, such as the Arduino. On a Raspberry Pi, you may have a gigabyte (GB) of RAM and 8GB of SD card storage. On an IoT device, you might only have 2000 bytes (2K) and 32KB of code storage. That is a ratio of 500,000 to 1. That is why you need efficient code on IoT devices. Yes, there is MicroPython, but it is very limited and still uses more memory than most Arduino boards.
- When you program in C/C++, you are closer to the hardware and have better control of the timing of operations. This can be very important in some situations. One of the issues of Python is that of the memory garbage collector. Sometimes, your program will run out of memory and Python will invoke the garbage collector to clean up memory and set it up for reuse. This can cause your program to not execute in the time you expected. An interesting note is that the ESP8266 used in several chapters of this book also has a memory garbage collector, which can cause some issues in critical timing sequences. None of these problems are known to exist in the code used in this book. Keeping fingers crossed, however.
- Libraries, libraries, libraries. You can find Arduino C/C++ libraries for almost any device and application you can imagine for IoT applications. The Arduino

library itself is filled with large amounts of functionality, making it much easier to get your IoT application up and running.

- Finally, the Arduino IDE (Integrated Development Environment) is a good (but not great) environment for writing code for small devices. It has its quirks and some disadvantages. The number one disadvantage of the Arduino IDE is that it does not have a built-in debugger (although the good folks at Arduino have one in beta testing). Even with this significant disadvantage, it runs on Linux, Windows, and Mac, and we will use it in this book. The Arduino IDE is widely available, and there are many resources for learning and libraries designed for this. Other alternatives include Visual Micro (runs on Windows, built on Microsoft Visual Studio) and Eclipse (runs on Linux, Windows, and Mac). Eclipse can be a nightmare to set up and update, but they have made improvements in the past few years.

In This Book

What will we be doing in future chapters? We will be building real IoT projects that actually have a lot of functionality. Yes, it is fun to blink an LED, but it is only the first step to really doing interesting and useful things with all this new technology. Build an IoT Weather Station. Build an IoT LightSwarm. Build your own IoT device with your own sensors. It is all accessible and inexpensive and within your ability whether you are an engineer or not.

CHAPTER 2

Sensing Your IoT Environment

Chapter Goal: Build Your First IoT Device

Topics Covered in This Chapter:

- Building an inexpensive IoT sensor device based on the ESP8266 and Arduino IDE
- Setting up a simple self-organizing IoT sensor net
- Reading I2C sensor (light and color) on the Arduino devices
- Reading data from remote IoT sensors on the Raspberry Pi
- Using the Raspberry Pi for monitoring and debugging
- Displaying your data on the screen and on an iPad

In this chapter, we build our first IoT device. This simple design, a light-sensor swarm, is easy to build and illustrates a number of IoT principles including the following:

- Distributed control
- Self-organization
- Passing information to the Internet
- Swarm behavior

The LightSwarm architecture is a simple and flexible scheme for understanding the idea of an IoT project using many simple small computers and sensors with shared responsibility for control and reporting. Note that, in this swarm, communication with the Internet is handled by a Raspberry Pi. The swarm devices talk to each other, but not with the Internet. The Raspberry Pi is located on the same WiFi access point as the swarm, but could be located far away with some clever forwarding of packets through your WiFi router. In this case, since we have no computer security at all in this design (see Chapter 7 for information on making your IoT swarm and device more secure), we are sticking with the local network.

IoT Sensor Nets

One of the major uses of the IoT will be building nets and groups of sensors. Inexpensive sensing is just as big of a driver for the IoT as the development of inexpensive computers. The ability for a computer to sense its environment is the key to gathering information for analysis, action, or transmittal to the Internet. Sensor nets have been around in academia for many years, but now the dropping prices and availability of development tools are quickly moving sensor nets out to the mainstream. Whole industrial and academic conferences are now held on sensor nets [`www.sensornets.org`]. The market is exploding for these devices, and opportunities are huge for the creative person or group that can figure out how to make the sensor net that will truly drive consumer sales.

IoT Characterization of This Project

As I discussed in Chapter 1, the first thing to do to understand an IoT project is to look at our six different aspects of IoT. LightSwarm is a pretty simple project and can be broken down into the six components listed in Table 2-1.

Table 2-1. *LightSwarm Characterization (CPLPFC)*

Aspect	Rating	Comments
Communications	9	WiFi connection to the Internet – can do ad hoc mesh-type communication
Processor power	7	80MHz XTensa Harvard Architecture CPU, ~80KB data RAM/~35KB of instruction RAM/200K ROM
Local storage	6	4MB Flash (or 3MB file system!)
Power consumption	7	~200mA transmitting, ~60mA receiving, no WiFi ~15mA, standby ~1mA
Functionality	7	Partial Arduino support (limited GPIO/analog inputs)
Cost	9	< $10 and getting cheaper

Ratings are from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications.

This gives us a CPLPFC rating of 7.2. This is calculated by averaging all six values together with equal weighting. This way is great for learning and experimenting and could be deployed for some applications.

The ESP8266 is an impressive device having a built-in WiFi connection, a powerful CPU, and quite a bit of RAM and access to the Arduino libraries. It is inexpensive and will get cheaper as time goes on. It is a powerful device for prototyping IoT applications requiring medium levels of functionality.

How Does This Device Hook Up to the IoT?

The ESP8266 provides a WiFi transmitter/receiver, a TCP/IP stack, and firmware to support direction connections to a local WiFi access point that then can connect to the Internet. In this project, the ESP8266 will only be talking to devices on the local wireless network. This is an amazing amount of functionality for less than $10 retail. These chips can be found for as little as $1 on the open market, if you want to roll your own printed circuit board.

What Is an ESP8266?

The ESP8266 is made by a company in China called Espressif [espressif.com]. They are a fabless semiconductor company that just came out of nowhere with the first generation of this chip and shook up the whole industry. Now all the major players are working on inexpensive versions of an IoT chip with WiFi connectivity. Why did we not use the more powerful ESP32? Two reasons, the ESP32 has way more CPU power than we need, and it is twice as expensive.

The ESP8266 chip was originally designed for connected light bulbs but soon got used in a variety of applications, and ESP8266 modules are currently now the most popular solutions to add WiFi to IoT projects. While the ESP8266 has huge functionality and a good price, the amount of current consumed by the chip makes battery-powered solutions problematic.

The ESP8266 is enabling a whole new set of applications and communities with their innovative and inexpensive design (Figure 2-1).

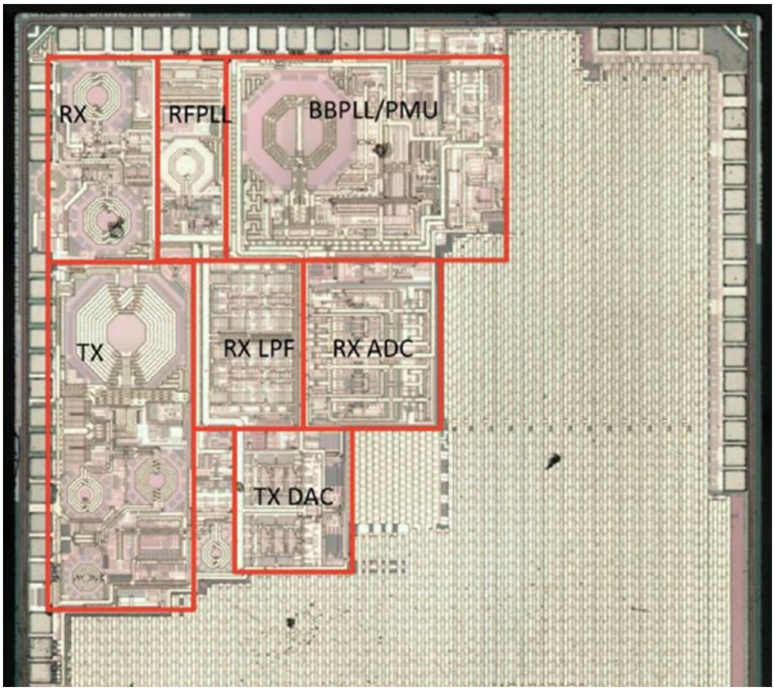


Figure 2-1. *The ESP8266 Die (Courtesy of Hackaday.io)*

We will be using a version of the ESP8266 on a breakout board designed by Adafruit (Figure 2-2). The board provides some connections, a built-in antenna, and some power regulation, all for less than $10.

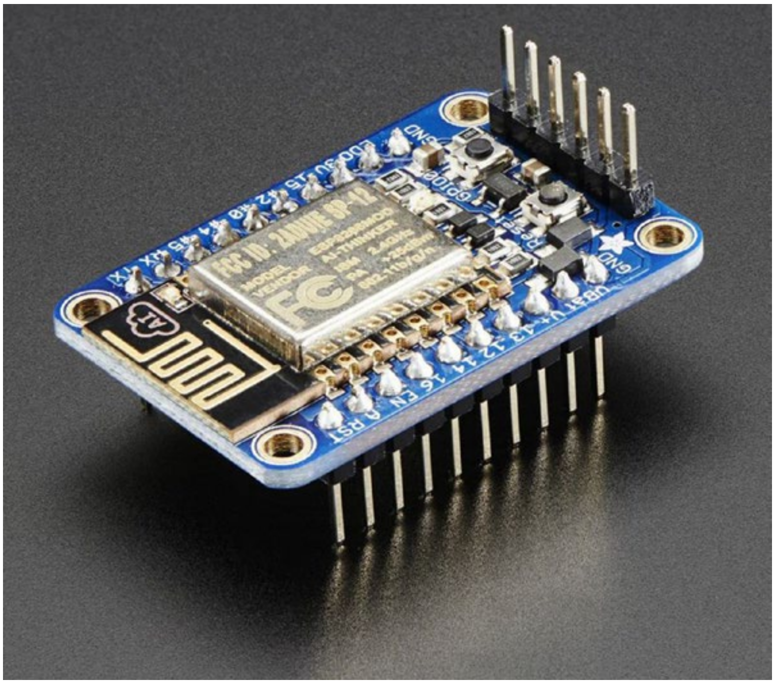


Figure 2-2. *The Adafruit Huzzah ESP8266 (Courtesy of adafruit.com)*

The LightSwarm Design

There are two major design aspects of the LightSwarm project. First of all, each element of the swarm is based on an ESP8266 with a light sensor attached. The software is what makes this small IoT device into a swarm. In the following block diagram, you can see the major two devices. I am using the Adafruit Huzzah breakout board for the ESP8266 [`www.adafruit.com/product/2471`]. Why use a breakout board? With a breakout board, you can quickly prototype devices and then move to a less expensive design in the future. The other electronic component (Figure 2-3) is a TCS34725 RGB light-sensor breakout board, also from Adafruit [`www.adafruit.com/products/1334`].

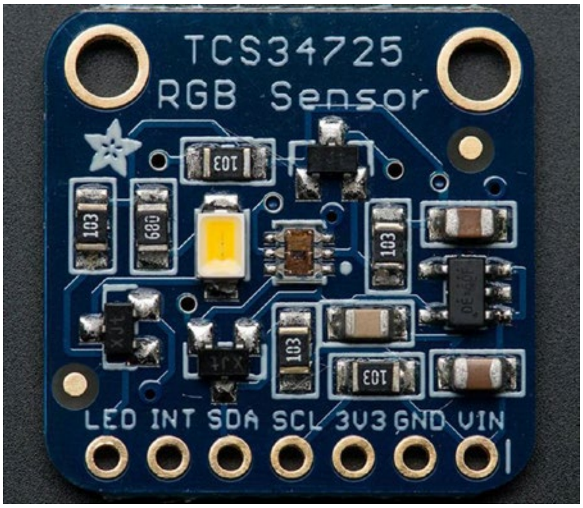


Figure 2-3. *TCS34725 Breakout Board (Courtesy of adafruit.com)*

The addition of a sensor to a computer is what makes this project an IoT device. I am sensing the environment and then doing something with the information (determining which of the Swarm has the brightest light). Figure 2-4 shows the block diagram of a single Swarm element.

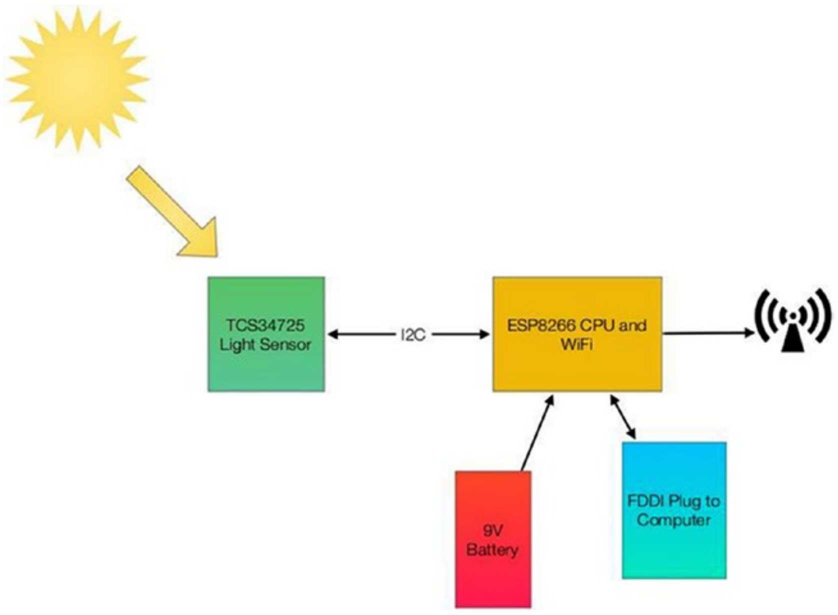


Figure 2-4. *LightSwarm Element Block Diagram*

Each of the LightSwarm devices in the swarm is identical. There are no software differences and no hardware differences. As you will see when we discuss the software, they vote with each other and compare notes and then elect the device that has the brightest light as the "Master," and then the "Master" turns the red LED on the device to show us who has been elected "Master." The swarm is designed so devices can drop out of the swarm, be added to the swarm dynamically, and the swarm adjusts to the new configuration. The swarm behavior (who is the master, how long it takes information about changing lights to propagate through the swarm, etc.) is rather complex. More complex than expected from the simple swarm device code. There is a lesson here: simple machines in groups can lead to large complex systems with higher-level behaviors based on the simple machines and the way they interact.

The behavior of the LightSwarm surprised me a number of times, and it was sometimes very difficult to figure out what was happening, even with the Raspberry Pi logger. Figure 2-5 shows the complete LightSwarm including the Raspberry Pi.

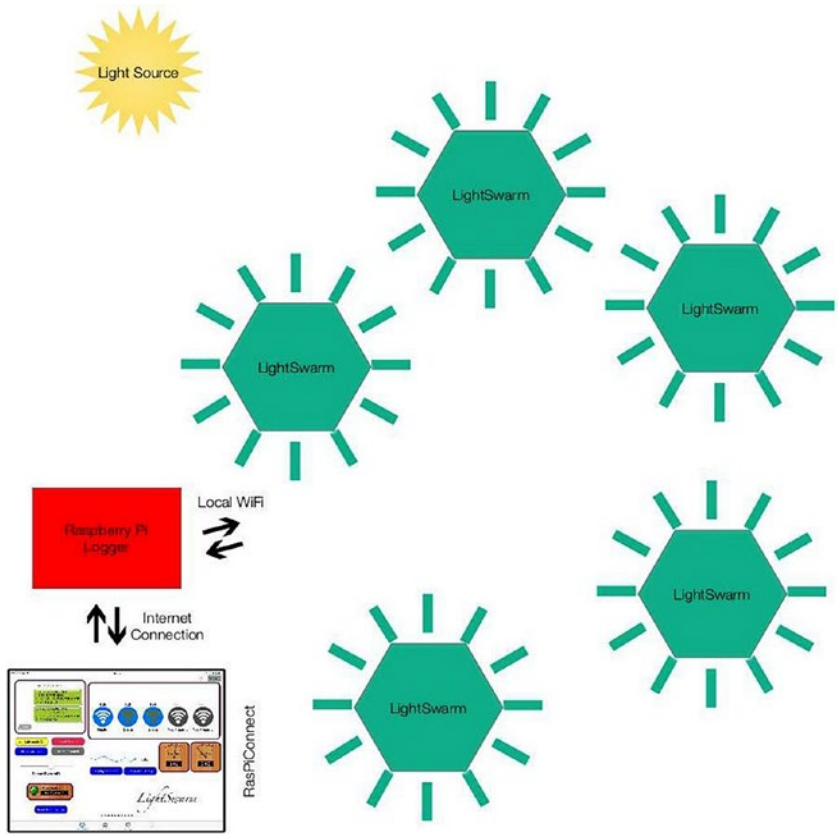


Figure 2-5. *The LightSwarm*

The Raspberry Pi in this diagram is not controlling the swarm. The Raspberry Pi gathers data from the swarm (the current "Master" device sends information to the Raspberry Pi), and then the Raspberry Pi can store the data and communicate it through the Internet, in this case to a local web page.

The LightSwarm project has an amazing amount of functionality and quirky self-organizing behavior for such a simple design.

Building Your First IoT Swarm

Table 2-2 lists the parts needed to build an IoT Swarm.

Table 2-2. *Swarm Parts List*

Part Number	Count	Description	Approximate Cost per Board	Source
ESP8266 Huzzah board	5	CPU/WiFi board	$18	`https://amzn.to/34VEgxJ`
TCS34725 breakout board	5	I2C light sensor	$10	`https://amzn.to/3n3fAt4`
FTDI cable	1	Cable for programming the ESP8266 from PC/Mac	$20	`https://amzn.to/34VEgxJ`

Installing Arduino Support on the PC or Mac

The key to making this project work is the software. While there are many ways of programming the ESP8266 (MicroPython [`micropython.org`], NodeMCU Lua interpreter [`nodemcu.com/index_en.html`], and the Arduino IDE (Integrated Development Environment) [`www.arduino.cc/en/Main/Software`]), I chose the Arduino IDE for its flexibility and the large number of sensor and device libraries available.

To install the Arduino IDE, you need to do the following:

a. Download the Arduino IDE package for your computer and install the software [www.arduino.cc/en/Guide/HomePage].

b. Download the ESP libraries so you can use the Arduino IDE with the ESP breakout board. Adafruit has an excellent tutorial for installing the ESP8266 support for the Arduino IDE [learn.adafruit.com/adafruit-huzzah-esp8266-breakout/using-arduino-ide].

Your First Sketch for the ESP8266

A great way of testing your setup is to run a small sketch that will blink the red LED on the ESP8266 breakout board. The red LED is hooked up to GPIO 0 (General-Purpose Input/Output pin 0) on the Adafruit board.

Open a new sketch using the Arduino IDE and place the following code into the code window, replacing the stubs provided when opening a new sketch. The code given here will make the red LED blink:

```
void setup() {
  pinMode(0, OUTPUT);
}

void loop() {
  digitalWrite(0, HIGH);
  delay(500);
  digitalWrite(0, LOW);
  delay(500);
}
```

If your LED is happily blinking away now, you have correctly followed all the tutorials and have the ESP8266 and the Arduino IDE running on your computer.

Next, I will describe the major hardware systems and then dive into the software.

The Hardware

The main pieces of hardware in the swarm device are the following:

- ESP8266 – CPU/WiFi interface Huzzah
- TCS34725 – Light sensor
- 9V battery power
- FTDI cable – Programming and power

The ESP8266 communicates with other swarm devices by using the WiFi interface. The ESP8266 uses the I2C interface to communicate with the light sensor. The WiFi is a standard that is very common (although we use protocols to communicate that are *not* common). See the description of UDP (User Datagram Protocol) later in this chapter. The I2C bus interface is much less familiar and needs some explanation.

Reviewing the I2C Bus

An I2C bus is often used to communicate with chips or sensors that are on the same board or located physically close to the CPU. It stands for standard Inter-IC device bus. The I2C was first developed by Philips (now NXP Semiconductors). To get around licensing issues, often the bus will be called TWI (Two-Wire Interface). SMBus, developed by Intel, is a subset of I2C that defines the protocols more strictly. Modern I2C systems take policies and rules from SMBus sometimes supporting both with minimal reconfiguration needed. The Raspberry Pi and the Arduino are both these kinds of devices. Even the ESP8266 used in this project can support both.

An I2C provides good support for slow, close peripheral devices that only need be addressed occasionally. For example, a temperature measuring device will generally only change very slowly and so is a good candidate for the use of I2C, where a camera will generate lots of data quickly and potentially changes often.

An I2C bus uses only two bidirectional open-drain lines (open-drain means the device can pull a level down to ground, but cannot pull the line up to Vdd, hence the name open-drain). Thus, a requirement of an I2C bus is that both lines are pulled up to Vdd. This is an important area, and not properly pulling up the lines is the first and most common mistake you make when you first use an I2C bus. More on pullup resistors later in the next section. The two lines are SDA (Serial Data Line) and the SCL (Serial Clock Line). There are two types of devices you can connect to an I2C bus. They are Master devices and Slave devices. Typically, you have one Master device (the Raspberry Pi in our case) and multiple Slave devices, each with their individual 7-bit address (like 0x68 in the case of the DS1307). There are ways to have 10-bit addresses and multiple Master devices, but that is beyond the scope of this book. Figure 2-6 shows an I2C bus with devices and the master connected.

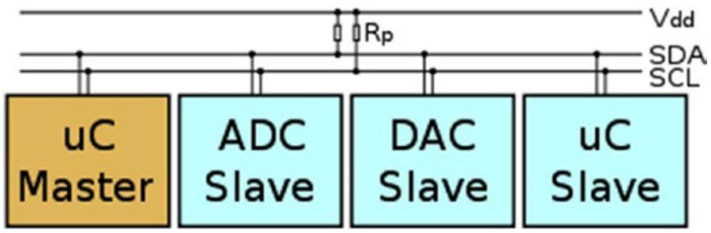


Figure 2-6. *An I2C Bus with One Master (the ESP8266 in This Case) and Three Slave Devices. Rps Are the Pullup Resistors*

SwitchDoc Note Vcc and Vdd mean the same. Gnd and Vss generally also both mean ground. There are historical differences, but today Vcc usually is one power supply, and if there is a second, they will call it Vdd.

When used on the Raspberry Pi, the Raspberry Pi acts as the Master and all other devices are connected as Slaves.

SwitchDoc Note If you connect an Arduino to a Raspberry Pi, you need to be careful about voltage levels because the Raspberry Pi is a 3.3V device and the Arduino is a 5.0V device. The ESP8266 is a 3.3V device so you also need to be careful connecting an Arduino to an ESP8266. Before you do this, read this excellent tutorial [`blog.retep.org/2014/02/15/connecting-an-arduino-to-a-raspberry-pi-using-i2c/`].

The I2C protocol uses three types of messages:

- *Single message* where a master writes data to a slave
- *Single message* where a master reads data from a slave
- *Combined messages*, where a master issues at least two reads and/or writes to one or more slaves

Lucky for us, most of the complexity of dealing with the I2C bus is hidden by drivers and libraries.

Pullups on the I2C Bus

One important thing to consider on your I2C bus is a pullup resistor. The Raspberry Pi has 1.8K (1k8) ohm resistors already attached to the SDA and SCL lines, so you really shouldn't need any additional pullup resistors. However, you do need to look at your I2C boards to find out if they have pullup resistors. If you have too many devices on the I2C bus with their own pullups, your bus will stop working. The rule of thumb from Philips is not to let the total pullup resistors in parallel be less than 1K (1k0) ohms. You can get a pretty good idea of what the total pullup resistance is by turning the power off on all devices and using an ohm meter to measure the resistance on the SCL line from the SCL line to Vdd.

Sensor Being Used

We are using the TCS34725, which has RGB and clear light-sensing elements. Figure 2-7 shows the TCS34725 die with the optical sensor showing in the center of the figure. An IR blocking filter, integrated on-chip and localized to the color-sensing photodiodes, minimizes the IR spectral component of the incoming light and allows color measurements to be made accurately. The IR filter means you'll get much truer color than most sensors, since humans don't see IR. The sensor does see IR and thus the IR filter is provided. The sensor also has a 3,800,000:1 dynamic range with adjustable integration time and gain so it is suited for use behind darkened glass or directly in the light.

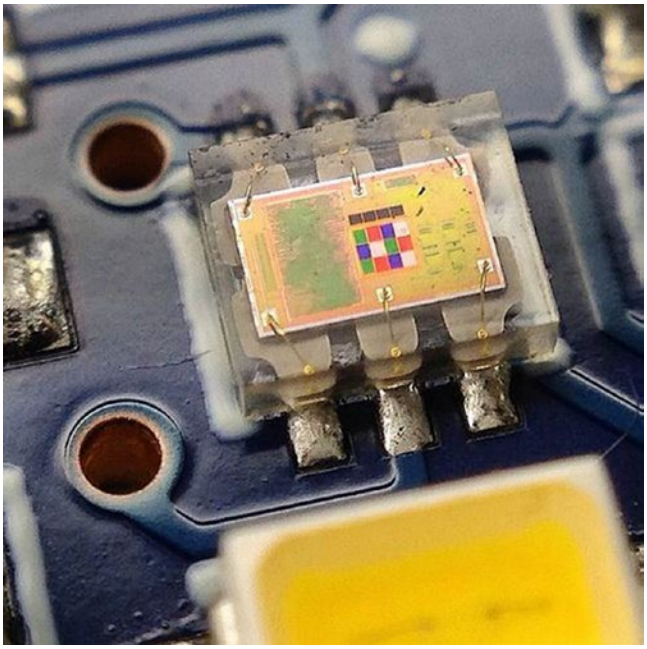
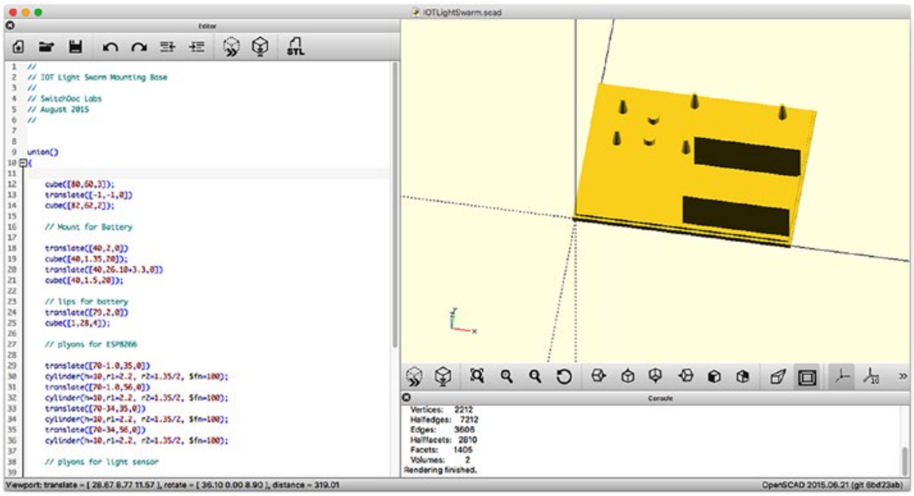


Figure 2-7. *The TCS34725 Chip Die*

This is an excellent inexpensive sensor ($8 retail from Adafruit on a breakout board) and forms the basis of our IoT sensor array. Of course, you could add many more sensors, but having one sensor that is easy to manipulate is perfect for our first IoT project. In Chapter 3, we add many more sensors to the Raspberry Pi computer for a complete IoT Weather Station design.

3D Printed Case

One of the big changes in the way people build prototypes is the availability of inexpensive 3D printers. It used to be difficult to build prototype cases and stands for various electronic projects. Now it is easy to design a case in one of many types of 3D software and then print it out using your 3D printer. For the swarm, I wanted a partial case to hold the 9V battery, the ESP8266, and the light sensor. I used OpenSCAD [www.openscad.org] to do the design. OpenSCAD is a free 3D CAD system that appeals to programmers. Rather than doing the entire design in a graphical environment, you write code (consisting of various objects, joined together or subtracted from each other) that you then compile in the environment to form a design in 3D space. OpenSCAD comes with an IDE (Integrated Development Environment), and you place the code showing in Listing 2-1 in the editor, compile the code, and then see the results in the attached IDE as shown in Figure 2-8.



```
//
// IOT Light Swarm Mounting Base
//
// SwitchDoc Labs
// August 2015
//


union()
{

    cube([80,60,3]);
    translate([-1,-1,0])
    cube([82,62,2]);

    // Mount for Battery

    translate([40,2,0])
    cube([40,1.35,20]);
    translate([40,26.10+3.3,0])
    cube([40,1.5,20]);

    // lips for battery
    translate([79,2,0])
    cube([1,28,4]);

    // plyons for ESP8266

    translate([70-1.0,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-1.0,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);

    // plyons for light sensor
```

```
Vertices:    2212
Halfedges:   7212
Edges:       3606
Halffacets:  2810
Facets:      1406
Volumes:        2
Rendering finished.
```

```
Viewport: translate = [ 28.67 8.77 11.57 ], rotate = [ 36.10 0.00 8.90 ], distance = 319.01
OpenSCAD 2015.06.21 (git 6bd23ab)
```

Figure 2-8. *OpenSCAD Display*

As shown in Listing 2-1, the OpenSCAD programming code to build this stand is quite simple. It consists of cubes and cylinders of various sizes and types.

Listing 2-1. Mounting Base for the IoT LightSwarm

```
//
// IOT Light Swarm Mounting Base
//
// SwitchDoc Labs
// December 2020
//

union()
{

    cube([80,60,3]);
    translate([-1,-1,0])
    cube([82,62,2]);

    // Mount for Battery

    translate([40,2,0])
    cube([40,1.35,20]);
    translate([40,26.10+3.3,0])
    cube([40,1.5,20]);

    // lips for battery
    translate([79,2,0])
    cube([1,28,4]);

    // pylons for ESP8266

    translate([70-1.0,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-1.0,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);

    // pylons for light sensor

    translate([10,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([10,49.5,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);

   translate([22,37,0])
    cylinder(h=6,r1=2.2, r2=2.2, $fn=100);
    translate([22,47,0])
    cylinder(h=6,r1=2.2, r2=2.2, $fn=100);
}
```

You can see the completed stand and the FTDI cable in Figure 2-9. Once designed, I quickly built five of them for the LightSwarm. Figure 2-10 shows a completed Swarm element. You can print your own from the STL file.

You can download the STL file for the LightSwarm base from github/switchdoclabs.

Use this command:

```
git clone https://github.com/switchdoclabs/SDL_STL_LightSwarm.
git
```

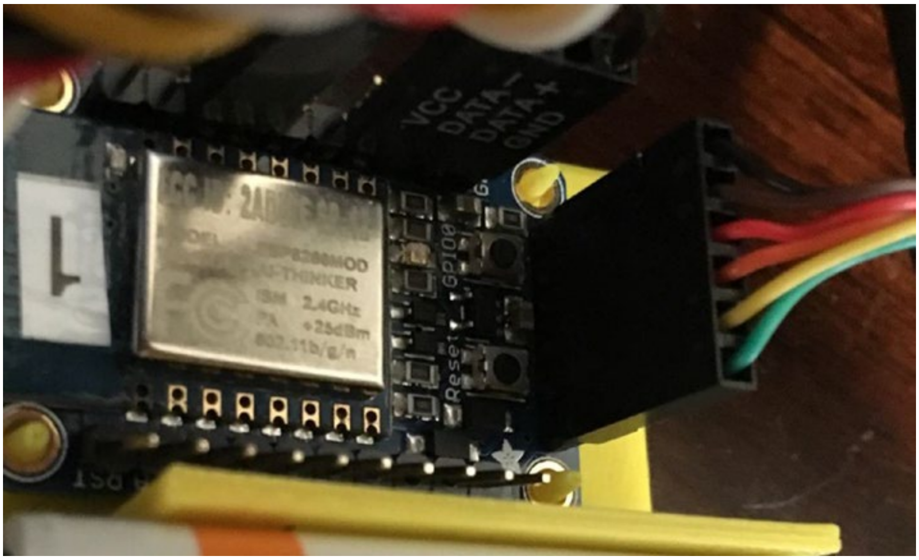


Figure 2-9. *FTDI Cable Plugged into ESP8266*

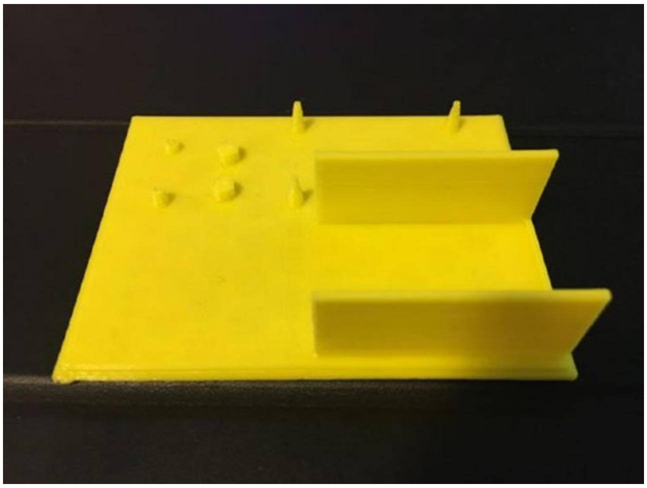


Figure 2-10. *Completed LightSwarm Stand*

The Full Wiring List

Table 2-3 provides the complete wiring list for a LightSwarm device. As you wire it, check off each wire for accuracy.

Table 2-3. *LightSwarm Wiring List*

From	To	Description
ESP8266/GND	TCS34725/GND	Ground for I2C light sensor
ESP8266/3V	TCS34725/3V3	3.3V power for I2C light sensor
ESP8266/#4	TCS34725/SDA	SDA for I2C light sensor
ESP8266/#5	TCS34725/SCL	SCL for I2C light sensor
ESP8266/GND	9VBat/"-" terminal (minus terminal)	Ground for battery
ESP8266/VBat	9VBat/"+" terminal (plus 9V)	9V from battery

The FTDI cable is plugged into the end of the Adafruit Huzzah ESP8266. Make sure you align the black wire with the GND pin on the ESP8266 breakout board as in Figure 2-9. Figure 2-11 shows the fully complete LightSwarm device.

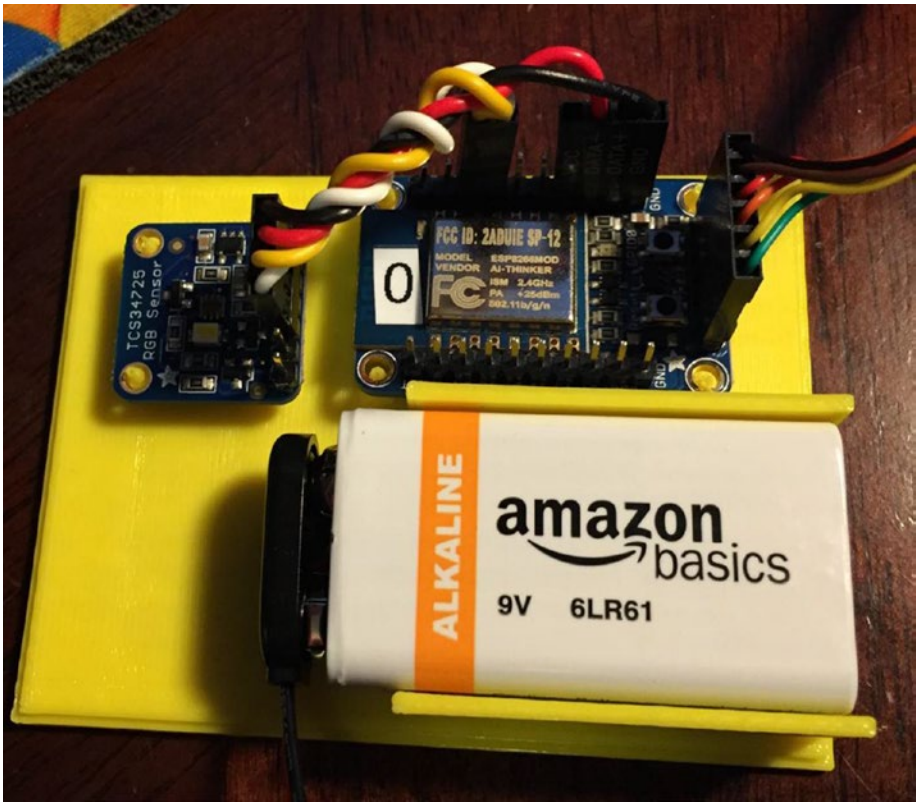


Figure 2-11. *A Complete LightSwarm Device*

The Software

There are two major modules written for the LightSwarm. The first is ESP8266 code for the LightSwarm device itself (written in the Arduino IDE – in simplified C and C++ language), and the second is the Raspberry Pi data-gathering software (written in Python3 on the Raspberry Pi). There is an excellent tutorial on how to set up the Arduino IDE and the ESP8266 libraries at `https://learn.adafruit.com/adafruit-huzzah-esp8266-breakout`.

The major design specifications for the LightSwarm device software are the following:

- Device self-discovery.
- Device becomes master when it has the brightest light; all others become slaves.
- Distributed voting method for determining master status.
- Self-organizing swarm. No server.
- Swarm must survive and recover from devices coming in and out of the network.
- Master device sends data to Raspberry Pi for analysis and distribution to the Internet.

The entire code for the LightSwarm devices is provided in Listings 2-2 through 2-11 (with the exception of the TCS74725 light-sensor driver, available here: github.com/adafruit/Adafruit_TCS34725). The latest code is also available on the Apress website [`www.apress.com`] and the SwitchDoc Labs github site [`github.com/switchdoclabs/SDL_ESP8266_LightSwarm`]. Listing 2-2 shows the beginning includes and the inital defines for parameters.

Listing 2-2. LightSwarm Code

```
/*
Cooperative IOT Self Organizing Example
SwitchDoc Labs, December 2020

 */

#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
#include <Wire.h>
#include "Adafruit_TCS34725.h"
```

```
#undef DEBUG

char ssid[] = "yyyyy";          // your wireless network SSID
                                   (name)
char pass[] = "xxxxxxx";        // your wireless network
                                   password

#define VERSIONNUMBER 28

#define SWARMSIZE 5
// 30 seconds is too old - it must be dead
#define SWARMTOOOLD 30000

int mySwarmID = 0;
```

Next in Listing 2-3, we define the necessary constants. Here are the definitions of all the Swarm commands available:

- LIGHT_UPDATE_PACKET – Packet containing current light from a LightSwarm device. Used to determine who is master and who is slave.
- RESET_SWARM_PACKET – All LightSwarm devices are told to reset their software.
- CHANGE_TEST_PACKET – Designed to change the master/slave criteria (not implemented).
- RESET_ME_PACKET – Just reset a particular LightSwarm device ID.
- DEFINE_SERVER_LOGGER_PACKET – This is the new IP address of the Raspberry Pi so the LightSwarm device can send data packets.
- LOG_TO_SERVER_PACKET – Packets sent from LightSwarm devices to Raspberry Pi.

- MASTER_CHANGE_PACKET – Packet sent from LightSwarm device when it becomes a master (not implemented).
- BLINK_BRIGHT_LED – Command to a LightSwarm device to blink the bright LED on the TCS34725.

After the constants in Listing 2-3, I set up the system variables for the devices. I am using UDP across the WiFi interface. What is UDP? UDP stands for User Datagram Protocol. UDP uses a simple connectionless model. Connectionless means that there is no handshake between the transmitting device and the receiving device to let the transmitter know that the receiver is actually there. Unlike TCP (Transmission Control Protocol), you have no idea or guarantee of data packets being delivered to any particular device. You can think of it as kind of a TV broadcast to your local network. Everyone gets it, but they don’t have to read the packets. There are also subtle other effects – such as you don’t have any guarantee that packets will arrive in the order they are sent. So why are we using UDP instead of TCP? I am using the broadcast mode of UDP so when a LightSwarm device sends out a message to the WiFi subnet, everybody gets it, and if they are listening on the port 2910 (set previously), then they can react to the message. This is how LightSwarm devices get discovered. Everybody starts sending packages (with random delays introduced), and all of the LightSwarm devices figure out who is present and who has the brightest light. Nothing in the LightSwarm system assigns IP numbers or names. They all figure it out themselves.

Listing 2-3. LightSwarm Constants

```
// Packet Types

#define LIGHT_UPDATE_PACKET 0
#define RESET_SWARM_PACKET 1
#define CHANGE_TEST_PACKET 2
```

```
#define RESET_ME_PACKET 3
#define DEFINE_SERVER_LOGGER_PACKET 4
#define LOG_TO_SERVER_PACKET 5
#define MASTER_CHANGE_PACKET 6
#define BLINK_BRIGHT_LED 7

unsigned int localPort = 2910;      // local port to listen for
                                       UDP packets

// master variables
boolean masterState = true;   // True if master, False if not
int swarmClear[SWARMSIZE];
int swarmVersion[SWARMSIZE];
int swarmState[SWARMSIZE];
long swarmTimeStamp[SWARMSIZE];   // for aging

IPAddress serverAddress = IPAddress(0, 0, 0, 0); // default no
                                                    IP Address

int swarmAddresses[SWARMSIZE];  // Swarm addresses

// variables for light sensor

int clearColor;
int redColor;
int blueColor;
int greenColor;

const int PACKET_SIZE = 14; // Light Update Packet
const int BUFFERSIZE = 1024;

byte packetBuffer[BUFFERSIZE]; //buffer to hold incoming and
                                 outgoing packets


// A UDP instance to let us send and receive packets over UDP
WiFiUDP udp;
```

```
/* Initialize with specific int time and gain values */
Adafruit_TCS34725 tcs = Adafruit_TCS34725(TCS34725_
INTEGRATIONTIME_700MS, TCS34725_GAIN_1X);

IPAddress localIP;
```

The setup() routine shown in Listing 2-4 is only run once after reset of the ESP8266 and is used to set up all the devices and print logging information to the serial port on the ESP8266, where, if you have the FTDI cable connected, you can see the logging information and debugging information on your PC or Mac.

Listing 2-4. The setup() Function for LightSwarm

```
void setup()
{

  Serial.begin(115200);
  Serial.println();
  Serial.println();

  Serial.println("");
  Serial.println("--------------------------");
  Serial.println("LightSwarm");
  Serial.print("Version ");
  Serial.println(VERSIONNUMBER);
  Serial.println("--------------------------");

  Serial.println(F(" 09/03/2015"));
  Serial.print(F("Compiled at:"));
  Serial.print (F(__TIME__));
  Serial.print(F(" "));
  Serial.println(F(__DATE__));
  Serial.println();
  pinMode(0, OUTPUT);
  digitalWrite(0, LOW);
  delay(500);
  digitalWrite(0, HIGH);
```

Here, we use the floating value of the analog input on the ESP8266 to set the pseudo-random number generation seed. This will vary a bit from device to device, and so it's not a bad way of initializing the pseudo-random number generator. If you put a fixed number as the argument, it will always generate the same set of pseudo-random numbers. This can be very useful in testing. Listing 2-5 shows the setup of the random seed and the detection of the TCS34725.

What is a pseudo-random number generator? It is an algorithm for generating a sequence of numbers whose properties approximate a truly random number sequence. It is not a truly random sequence of numbers, but it is close. Good enough for our usage.

Listing 2-5. Remainder of the setup() Function for LightSwarm

```
randomSeed(analogRead(A0));
  Serial.print("analogRead(A0)=");
  Serial.println(analogRead(A0));

  if (tcs.begin()) {
    Serial.println("Found sensor");
  } else {
    Serial.println("No TCS34725 found ... check your
    connections");

  }

  // turn off the light
  tcs.setInterrupt(true);  // true means off, false means on

  // everybody starts at 0 and changes from there
  mySwarmID = 0;
```

```
// We start by connecting to a WiFi network
Serial.print("LightSwarm Instance: ");
Serial.println(mySwarmID);

Serial.print("Connecting to ");
Serial.println(ssid);
WiFi.begin(ssid, pass);

// initialize Swarm Address - we start out as swarmID of 0

while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
Serial.println("");

Serial.println("WiFi connected");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());

Serial.println("Starting UDP");

udp.begin(localPort);
Serial.print("Local port: ");
Serial.println(udp.localPort());

// initialize light sensor and arrays
int i;
for (i = 0; i < SWARMSIZE; i++)
{

  swarmAddresses[i] = 0;
  swarmClear[i] = 0;
  swarmTimeStamp[i] = -1;
}
```

```
swarmClear[mySwarmID] = 0;
swarmTimeStamp[mySwarmID] = 1;   // I am always in time to
                                    myself
clearColor = swarmClear[mySwarmID];
swarmVersion[mySwarmID] = VERSIONNUMBER;
swarmState[mySwarmID] = masterState;
Serial.print("clearColor =");
Serial.println(clearColor);
```

Now we have initialized all the data structures for describing our LightSwarm device and the states of the light sensor. Listing 2-6 sets the SwarmID based on the current device IP address. When you turn on a LightSwarm device and it connects to a WiFi access point, the access point assigns a unique IP address to the LightSwarm device. This is done through a process called DHCP (Dynamic Host Configuration Protocol) [`en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol`]. While the number will be different for each LightSwarm device, it is not random. Typically, if you power a specific device down and power it up again, the access point will assign the same IP address. However, you can't count on this. The access point knows your specific device because each and every WiFi interface has a specific and unique MAC (Media Access Control) number, which is usually never changed.

SwitchDoc Note Faking MAC addresses allows you to impersonate other devices with your device in some cases, and you can use MAC addresses to track specific machines by looking at the network. This is why Apple Inc. has started using random MAC addresses in their devices while scanning for networks. If random MAC addresses aren't used, then researchers have confirmed that it is possible to link a specific real identity to a particular wireless MAC address [Cunche, Mathieu. "I know your MAC Address: Targeted tracking of individual using Wi-Fi". 2013].

Listing 2-6. Setting the SwarmID from IP Address

```
  // set SwarmID based on IP address

  localIP = WiFi.localIP();

  swarmAddresses[0] =  localIP[3];

  mySwarmID = 0;

  Serial.print("MySwarmID=");
  Serial.println(mySwarmID);

}
```

The second main section of the LightSwarm device code is the loop(). The loop() function does precisely what its name suggests and loops infinitely, allowing your program to change and respond. This is the section of the code that performs the main work of the LightSwarm code.

```
void loop()
{
  int secondsCount;
  int lastSecondsCount;

  lastSecondsCount = 0;
 #define LOGHOWOFTEN

  secondsCount = millis() / 100;
```

In Listing 2-7, we read all the data from the TCS34725 sensor to find out how bright the ambient light currently is. This forms the substance of the data to determine who the master in the swarm is.

After the delay (300) line in Listing 2-7, I check for UDP packets being broadcast to port 2910. Remember the way the swarm is using UDP is in broadcast mode and all packets are being received by everybody all the time. Note, this sets a limit of how many swarm devices you can have (limited to the subnet size) and also by the congestion of having too many messages go through at the same time. This was pretty easy to simulate by increasing the rate that packets are sent. The swarm still functions, but the behavior becomes more erratic and with sometimes large delays.

Listing 2-7. Reading the Light Color

```
uint16_t r, g, b, c, colorTemp, lux;

tcs.getRawData(&r, &g, &b, &c);
colorTemp = tcs.calculateColorTemperature(r, g, b);
lux = tcs.calculateLux(r, g, b);

Serial.print("Color Temp: "); Serial.print(colorTemp, DEC);
Serial.print(" K - ");
Serial.print("Lux: "); Serial.print(lux, DEC);
Serial.print(" - ");
Serial.print("R: "); Serial.print(r, DEC); Serial.print(" ");
Serial.print("G: "); Serial.print(g, DEC); Serial.print(" ");
Serial.print("B: "); Serial.print(b, DEC); Serial.print(" ");
Serial.print("C: "); Serial.print(c, DEC); Serial.print(" ");
Serial.println(" ");

clearColor = c;
redColor = r;
blueColor = b;
greenColor = g;

swarmClear[mySwarmID] = clearColor;

// wait to see if a reply is available
delay(300);
```

```
int cb = udp.parsePacket();

if (!cb) {
  //  Serial.println("no packet yet");
  Serial.print(".");
}
else {
```

In Listing 2-8, we interpret all the packets depending on the packet type.

Listing 2-8. Interpreting the Packet Type

```
// We've received a packet, read the data from it

   udp.read(packetBuffer, PACKET_SIZE); // read the packet into
   the buffer
   Serial.print("packetbuffer[1] =");
   Serial.println(packetBuffer[1]);
   if (packetBuffer[1] == LIGHT_UPDATE_PACKET)
   {
     Serial.print("LIGHT_UPDATE_PACKET received from
     LightSwarm #");
     Serial.println(packetBuffer[2]);
     setAndReturnMySwarmIndex(packetBuffer[2]);

     Serial.print("LS Packet Recieved from #");
     Serial.print(packetBuffer[2]);
     Serial.print(" SwarmState:");
     if (packetBuffer[3] == 0)
       Serial.print("SLAVE");
     else
       Serial.print("MASTER");
     Serial.print(" CC:");
     Serial.print(packetBuffer[5] * 256 + packetBuffer[6]);
```

```
      Serial.print(" RC:");
      Serial.print(packetBuffer[7] * 256 + packetBuffer[8]);
      Serial.print(" GC:");
      Serial.print(packetBuffer[9] * 256 + packetBuffer[10]);
      Serial.print(" BC:");
      Serial.print(packetBuffer[11] * 256 + packetBuffer[12]);
      Serial.print(" Version=");
      Serial.println(packetBuffer[4]);

      // record the incoming clear color

      swarmClear[setAndReturnMySwarmIndex(packetBuffer[2])] =
      packetBuffer[5] * 256 + packetBuffer[6];
      swarmVersion[setAndReturnMySwarmIndex(packetBuffer[2])] =
      packetBuffer[4];
      swarmState[setAndReturnMySwarmIndex(packetBuffer[2])] =
      packetBuffer[3];
      swarmTimeStamp[setAndReturnMySwarmIndex(packetBuffer[2])]
      = millis();

      // Check to see if I am master!
      checkAndSetIfMaster();

    }
```

The RESET_SWARM_PACKET command sets all of the LightSwarm devices to master (turning on the red LED on each) and then lets the LightSwarm software vote and determine who has the brightest light. As each device receives a LIGHT_UPDATE_PACKET, it compares the light from that swarm device to its own sensor and becomes a slave if their light is brighter. Eventually, the swarm figures out who has the brightest light. I have been sending this periodically from the Raspberry Pi and watching the devices work it out. It makes an interesting video. Listing 2-9 shows how LightSwarm interprets incoming packets. The very last section of

Listing 2-9 shows how the Swarm element updates everybody else in the Swarm what is going on with this device, and then we send a data packet to the Raspberry Pi if we are the swarm master.

Listing 2-9. LightSwarm Packet Interpretation Code

```
if (packetBuffer[1] == RESET_SWARM_PACKET)
{
  Serial.println(">>>>>>>>>RESET_SWARM_PACKETPacket Received");
  masterState = true;
  Serial.println("Reset Swarm:  I just BECAME Master (and
  everybody else!)");
  digitalWrite(0, LOW);

}

if (packetBuffer[1] == CHANGE_TEST_PACKET)
{
  Serial.println(">>>>>>>>>CHANGE_TEST_PACKET Packet
  Received");
  Serial.println("not implemented");
  int i;
  for (i = 0; i < PACKET_SIZE; i++)
  {
    if (i == 2)
    {
      Serial.print("LPS[");
      Serial.print(i);
      Serial.print("] = ");
      Serial.println(packetBuffer[i]);

    }
    else
    {
      Serial.print("LPS[");
      Serial.print(i);
      Serial.print("] = 0x");
      Serial.println(packetBuffer[i], HEX);
    }

  }


}

if (packetBuffer[1] == RESET_ME_PACKET)
{
  Serial.println(">>>>>>>>>RESET_ME_PACKET Packet Received");

  if (packetBuffer[2] == swarmAddresses[mySwarmID])
  {
    masterState = true;
    Serial.println("Reset Me:  I just BECAME Master");
    digitalWrite(0, LOW);

  }
  else
  {
    Serial.print("Wanted #");
    Serial.print(packetBuffer[2]);
    Serial.println(" Not me - reset ignored");
  }

}

}
```

```
if (packetBuffer[1] ==  DEFINE_SERVER_LOGGER_PACKET)
{
  Serial.println(">>>>>>>>>DEFINE_SERVER_LOGGER_PACKET Packet
  Received");
  serverAddress = IPAddress(packetBuffer[4], packetBuffer[5],
  packetBuffer[6], packetBuffer[7]);
  Serial.print("Server address received: ");
  Serial.println(serverAddress);

}
if (packetBuffer[1] ==  BLINK_BRIGHT_LED)
{
  Serial.println(">>>>>>>>>BLINK_BRIGHT_LED Packet Received");
  if (packetBuffer[2] == swarmAddresses[mySwarmID])
  {

    tcs.setInterrupt(false);  // true means off, false means on
    delay(packetBuffer[4] * 100);
    tcs.setInterrupt(true);  // true means off, false means on
  }
  else
  {
    Serial.print("Wanted #");
    Serial.print(packetBuffer[2]);
    Serial.println(" Not me - reset ignored");
  }

}

Serial.print("MasterStatus:");
if (masterState == true)
{
  digitalWrite(0, LOW);
  Serial.print("MASTER");
}
```

```
else
{
  digitalWrite(0, HIGH);
  Serial.print("SLAVE");
}
  Serial.print("/cc=");
  Serial.print(clearColor);
  Serial.print("/KS:");
  Serial.println(serverAddress);

  Serial.println("--------");

  int i;
  for (i = 0; i < SWARMSIZE; i++)
{
  Serial.print("swarmAddress[");
  Serial.print(i);
  Serial.print("] = ");
  Serial.println(swarmAddresses[i]);
}
  Serial.println("--------");

  broadcastARandomUpdatePacket();
  sendLogToServer();

} // end of loop()
```

Listing 2-10 is used to send out light packets to a swarm address. Although a specific address is allowed by this function, we set the last octet of the IP address (201 in the IP address 192.168.1.201) in the calling function to 255, which is the UDP broadcast address.

Listing 2-10. Broadcasting to the Swarm

```
// send an LIGHT Packet request to the swarms at the given
   address
unsigned long sendLightUpdatePacket(IPAddress & address)
{

  // set all bytes in the buffer to 0
  memset(packetBuffer, 0, PACKET_SIZE);
  // Initialize values needed to form Light Packet
  // (see URL above for details on the packets)
  packetBuffer[0] = 0xF0;   // StartByte
  packetBuffer[1] = LIGHT_UPDATE_PACKET;     // Packet Type
  packetBuffer[2] = localIP[3];     // Sending Swarm Number
  packetBuffer[3] = masterState;  // 0 = slave, 1 = master
  packetBuffer[4] = VERSIONNUMBER;  // Software Version
  packetBuffer[5] = (clearColor & 0xFF00) >> 8; // Clear High
                                                   Byte
  packetBuffer[6] = (clearColor & 0x00FF); // Clear Low Byte
  packetBuffer[7] = (redColor & 0xFF00) >> 8; // Red High Byte
  packetBuffer[8] = (redColor & 0x00FF); // Red Low Byte
  packetBuffer[9] = (greenColor & 0xFF00) >> 8; // green High
                                                   Byte
  packetBuffer[10] = (greenColor & 0x00FF); // green Low Byte
  packetBuffer[11] = (blueColor & 0xFF00) >> 8; // blue High
                                                   Byte
  packetBuffer[12] = (blueColor & 0x00FF); // blue Low Byte
  packetBuffer[13] = 0x0F;  //End Byte

  // all Light Packet fields have been given values, now
  // you can send a packet requesting coordination
  udp.beginPacketMulticast(address,  localPort, WiFi.
  localIP()); //
  udp.write(packetBuffer, PACKET_SIZE);
  udp.endPacket();
}

// delay 0-MAXDELAY seconds
#define MAXDELAY 500

void broadcastARandomUpdatePacket()
{

  int sendToLightSwarm = 255;
  Serial.print("Broadcast ToSwarm = ");
  Serial.print(sendToLightSwarm);
  Serial.print(" ");

  // delay 0-MAXDELAY seconds
  int randomDelay;
  randomDelay = random(0, MAXDELAY);
  Serial.print("Delay = ");
  Serial.print(randomDelay);
  Serial.print("ms : ");

  delay(randomDelay);

  IPAddress sendSwarmAddress(192, 168, 1, sendToLightSwarm);
  // my Swarm Address
  sendLightUpdatePacket(sendSwarmAddress);

}
```

In the function in Listing 2-11, I check if we just became master and also update the status of all the LightSwarm devices. This is where the timeout function is implemented that will remove stale or dead devices from the swarm.

Listing 2-11. Master Check and Update

```
void checkAndSetIfMaster()
{

  int i;
  for (i = 0; i < SWARMSIZE; i++)
  {

#ifdef DEBUG

    Serial.print("swarmClear[");
    Serial.print(i);
    Serial.print("] = ");
    Serial.print(swarmClear[i]);
    Serial.print("  swarmTimeStamp[");
    Serial.print(i);
    Serial.print("] = ");
    Serial.println(swarmTimeStamp[i]);
#endif

    Serial.print("#");
    Serial.print(i);
    Serial.print("/");
    Serial.print(swarmState[i]);
    Serial.print("/");
    Serial.print(swarmVersion[i]);
    Serial.print(":");
    // age data
    int howLongAgo = millis() - swarmTimeStamp[i] ;

    if (swarmTimeStamp[i] == 0)
    {
      Serial.print("TO ");
    }
    else if (swarmTimeStamp[i] == -1)
    {
      Serial.print("NP ");
    }
    else if (swarmTimeStamp[i] == 1)
    {
      Serial.print("ME ");
    }
    else if (howLongAgo > SWARMTOOOLD)
    {
      Serial.print("TO ");
      swarmTimeStamp[i] = 0;
      swarmClear[i] = 0;

    }
    else
    {
      Serial.print("PR ");

    }
  }

  Serial.println();
  boolean setMaster = true;

  for (i = 0; i < SWARMSIZE; i++)
  {

    if (swarmClear[mySwarmID] >= swarmClear[i])
    {
      // I might be master!

    }
    else
    {
      // nope, not master
      setMaster = false;
      break;
    }

  }
  if (setMaster == true)
  {
    if (masterState == false)
    {
      Serial.println("I just BECAME Master");
      digitalWrite(0, LOW);
    }

    masterState = true;
  }
  else
  {
    if (masterState == true)
    {
      Serial.println("I just LOST Master");
      digitalWrite(0, HIGH);
    }

    masterState = false;
  }

  swarmState[mySwarmID] = masterState;

}
```

```
int setAndReturnMySwarmIndex(int incomingID)
{

  int i;
  for (i = 0; i< SWARMSIZE; i++)
  {
    if (swarmAddresses[i] == incomingID)
    {
       return i;
    }
    else
    if (swarmAddresses[i] == 0)  // not in the system, so put it in
    {

      swarmAddresses[i] = incomingID;
      Serial.print("incomingID ");
      Serial.print(incomingID);
      Serial.print("  assigned #");
      Serial.println(i);
      return i;
    }

  }

  // if we get here, then we have a new swarm member.
  // Delete the oldest swarm member and add the new one in
  // (this will probably be the one that dropped out)

  int oldSwarmID;
  long oldTime;
  oldTime = millis();
  for (i = 0;  i < SWARMSIZE; i++)
 {
  if (oldTime > swarmTimeStamp[i])
  {
    oldTime = swarmTimeStamp[i];
    oldSwarmID = i;
  }

 }

 // remove the old one and put this one in....
 swarmAddresses[oldSwarmID] = incomingID;
 // the rest will be filled in by Light Packet Receive

}

// send log packet to Server if master and server address
   defined

void sendLogToServer()
{

  // build the string

  char myBuildString[1000];
  myBuildString[0] = '\0';

  if (masterState == true)
  {
    // now check for server address defined
    if ((serverAddress[0] == 0) && (serverAddress[1] == 0))
    {
      return;  // we are done.  not defined
    }
    else
    {
```

```
// now send the packet as a string with the following
   format:
// swarmID, MasterSlave, SoftwareVersion, clearColor,
   Status | ....next Swarm ID
// 0,1,15,3883, PR | 1,0,14,399, PR | ....

int i;
char swarmString[20];
swarmString[0] = '\0';

for (i = 0; i < SWARMSIZE; i++)
{

  char stateString[5];
  stateString[0] = '\0';
  if (swarmTimeStamp[i] == 0)
  {
    strcat(stateString, "TO");
  }
  else if (swarmTimeStamp[i] == -1)
  {
    strcat(stateString, "NP");
  }
  else if (swarmTimeStamp[i] == 1)
  {
    strcat(stateString, "PR");
  }
  else
  {
    strcat(stateString, "PR");
  }
```

```
    sprintf(swarmString, " %i,%i,%i,%i,%s,%i ", i,
    swarmState[i], swarmVersion[i], swarmClear[i],
    stateString, swarmAddresses[i]);

    strcat(myBuildString, swarmString);
    if (i < SWARMSIZE - 1)
    {

      strcat(myBuildString, "|");

    }
  }

}

// set all bytes in the buffer to 0
memset(packetBuffer, 0, BUFFERSIZE);
// Initialize values needed to form Light Packet
// (see URL above for details on the packets)
packetBuffer[0] = 0xF0;   // StartByte
packetBuffer[1] = LOG_TO_SERVER_PACKET;     // Packet Type
packetBuffer[2] = localIP[3];     // Sending Swarm Number
packetBuffer[3] = strlen(myBuildString); // length of
string in bytes
packetBuffer[4] = VERSIONNUMBER;  // Software Version
int i;
for (i = 0; i < strlen(myBuildString); i++)
{
  packetBuffer[i + 5] = myBuildString[i];// first string
  byte
}

packetBuffer[i + 5] = 0x0F; //End Byte
Serial.print("Sending Log to Sever:");
```

```
    Serial.println(myBuildString);
    int packetLength;
    packetLength = i + 5 + 1;

    udp.beginPacket(serverAddress,  localPort); //

    udp.write(packetBuffer, packetLength);
    udp.endPacket();

  }

}
```

That is the entire LightSwarm device code. When compiling this code on the Arduino IDE targeting the Adafruit ESP8266, we get the following:

Sketch uses 308,736 bytes (29%) of program storage space. Maximum is 1,044,464 bytes.

Global variables use 50,572 bytes (61%) of dynamic memory, leaving 31,348 bytes for local variables. Maximum is 81,920 bytes.

Still a lot of space left for more code. Most of the compiled code space earlier is used by the system libraries for WiFi and running the ESP8266.

Self-Organizing Behavior

Why do we say that the LightSwarm code is self-organizing? It is because there is no central control of who is the master and who is the slave. This makes the system more reliable and able to function even in a bad environment. Self-organization is defined as a process where some sort of order arises out of the local interactions between smaller items in an initially disordered system.

Typically, these kinds of systems are robust and able to survive in a chaotic environment. Self-organizing systems occur in a variety of physical, biological, and social systems.

One reason to build these kinds of systems is that the individual devices can be small and not very smart, and yet the overall task or picture of the data being collected and processed can be amazingly interesting and informative.

Monitoring and Debugging the System with the Raspberry Pi (the Smart Guy on the Block)

The Raspberry Pi is used in LightSwarm primarily as a data storage device for examining the LightSwarm data and telling what is going on in the swarm. You can send a few commands to reset the swarm, turn lights on, and so on, but the swarm runs itself with or without the Raspberry Pi running. However, debugging self-organizing systems like this is difficult without some way of watching what is going on with the swarm, preferably from another computer. And that is what we have done with the LightSwarm logging software on the Raspberry Pi. The primary design criteria for this software are as follows:

- Read and log information on the swarm behavior.
- Reproduce archival swarm behavior.
- Provide methods for testing swarm behavior (such as resetting the swarm).
- Provide real-time information to the Internet on swarm behavior and status.

Remember that the Raspberry Pi is a full, complex, and powerful computer system that goes way beyond what you can do with an ESP8266. First, we will look at the LightSwarm logging software and then the software that supports the display web page. Note that we are not storing the information coming from the swarm devices in this software, but we could easily add logging software that would populate a MySQL database that would allow us to store and analyze the information coming in from the swarm.

LightSwarm Logging Software Written in Python

The entire code base of the LightSwarm logging software is available off the Apress site [Apress code site] and on the SwitchDoc Labs github site [`github.com/switchdoclabs/SDL_Pi_LightSwarm`]. I am picking out the most interesting code in the logging software to comment on and explain.

First of all, this program is written in Python3. Python3 is a widely used programming language, especially with Raspberry Pi coders. There are a number of device libraries available for building your own IoT devices, and there is even a small version that runs on the ESP8266. Python's design philosophy emphasizes code readability. Indenting is important in Python, so keep that in mind as you look at the following code.

SwitchDoc Note Python is "weakly typed," meaning you define a variable and the type by the first time you use it. Some programmers like this, but I don't. Misspelling a variable name makes a whole new variable and can cause great confusion. My prejudice is toward "strongly typed" languages as it tends to reduce the number of coding errors, at the cost of having to think about and declare variables explicitly.

The first section of this program defines all the needed libraries (import statements) and defines necessary "constants." Python does not have a way to define constants, so you declare variables for constant values, which by my convention are all in uppercase. There are other ways of defining constants by using classes and functions, but they are more complex than just defining another variable. Listing 2-12 shows how the variables and constants are initialized.

Listing 2-12. Import and Constant Value Declaration

```
'''
    LightSwarm Raspberry Pi Logger
    SwitchDoc Labs
    December 2020
'''
from __future__ import print_function

from builtins import chr
from builtins import str
from builtins import range
import sys
import time
import random

from netifaces import interfaces, ifaddresses, AF_INET

from socket import *

VERSIONNUMBER = 6
# packet type definitions
LIGHT_UPDATE_PACKET = 0
RESET_SWARM_PACKET = 1
CHANGE_TEST_PACKET = 2   # Not Implemented
RESET_ME_PACKET = 3
DEFINE_SERVER_LOGGER_PACKET = 4
LOG_TO_SERVER_PACKET = 5
```

```
MASTER_CHANGE_PACKET = 6
BLINK_BRIGHT_LED = 7

MYPORT = 2910

SWARMSIZE = 5
```

Listing 2-13 defines the interface between the LightSwarm logging software and receiving commands through command files from a number of sources. These are the three important functions:

- processCommand(s) – When a command is received from the software running on the same computer (writing to this file), this function defines all the actions to be completed when a specific command is received.
- completeCommandWithValue(value) – Call function and return a value to a file when you have completed a command.
- completeCommand()– Call function when you have completed a command to tell the external program you are done with the command.

Listing 2-13. Command Interface

```
logString = ""
# command from command Code

def completeCommand():

        f = open("/home/pi/SDL_Pi_LightSwarm/state/LSCommand.
        txt", "w")
        f.write("DONE")
        f.close()
```

```
def completeCommandWithValue(value):

        f = open("/home/pi/SDL_Pi_LightSwarm/state/LSResponse.
        txt", "w")
        f.write(value)
        print("in completeCommandWithValue=", value)
        f.close()

        completeCommand()

def processCommand(s):
        f = open("//home/pi/SDL_Pi_LightSwarm/state/LSCommand.
        txt", "r")
        command = f.read()
        f.close()

        command = command.rstrip()
        if (command == "") or (command == "DONE"):
            # Nothing to do
            return False

        # Check for our commands
        #pclogging.log(pclogging.INFO, __name__, "Command %s
        Recieved" % command)

        print("Processing Command: ", command)
        if (command == "STATUS"):

            completeCommandWithValue(logString)

            return True

        if (command == "RESETSWARM"):

            SendRESET_SWARM_PACKET(s)
            completeCommand()

            return True

        # check for , commands

        print("command=%s" % command)
        myCommandList = command.split(',')
        print("myCommandList=", myCommandList)

        if (len(myCommandList) > 1):
            # we have a list command

            if (myCommandList[0]== "BLINKLIGHT"):
                SendBLINK_BRIGHT_LED(s, int(myCommandList[1]), 1)

            if (myCommandList[0]== "RESETSELECTED"):
                SendRESET_ME_PACKET(s, int(myCommandList[1]))

            if (myCommandList[0]== "SENDSERVER"):
                SendDEFINE_SERVER_LOGGER_PACKET(s)

            completeCommand()

            return True

        completeCommand()

        return False
```

Basically, the idea is that external software sends a command to the LightSwarm logging software that is running on a different thread in the same system. Remember that the Raspberry Pi Linux-based system is multitasking and you can run many different programs at once.

In Table 2-4, we show the various text commands that can be placed in the text file LSCommand.txt that will be picked up by LightSwarm.py and executed.

Table 2-4. LightSwarm Command List

Command	Syntax	Description
Reset Swarm	RESETSWARM	Resets all connected Swarm units to initial state. They then start negotiating for Master Status Again
Reset One	RESETSELECTED, <swarm id>	Resets a particular Swarm element. Uses Swarm ID 0-4, for example, RESETSELECTED, 1
Blink Bright LED	BLINKLIGHT, <swarm id>	Blinks the Bright LED on the selected Swarm unit. Uses Swarm ID 0-4, for example, BLINKLIGHT, 1
Get Status	STATUS	Returns the status of all the Swarm IDs in the file LSResponse.txt

To use these commands is simple. Under the SDL_Pi_LightSwarm/state subdirectory, edit the LSCommand.txt and enter one of the preceding commands. Shortly the main loop in LightSwarm.py will pick up the command and execute the command.

In Listing 2-14, I have one of the actual LightSwarm command implementations for sending packets. Listing 2-14 just shows the first packet type to illustrate the concepts.

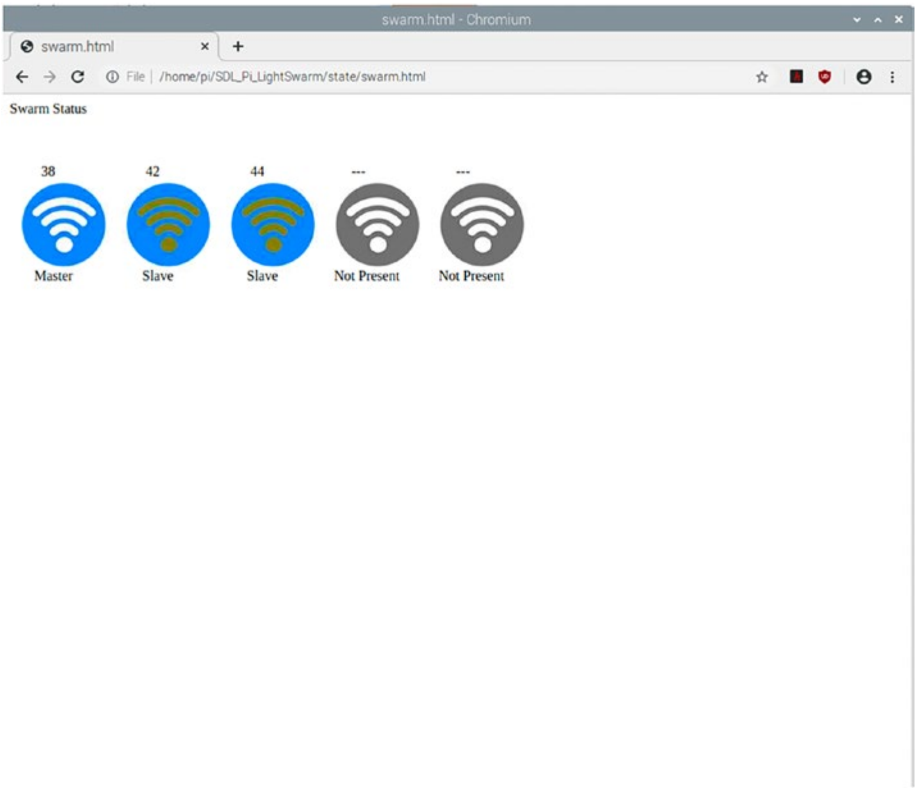
Listing 2-14. LightSwarm Command Packet Definitions

```
# UDP Commands and packets

def SendDEFINE_SERVER_LOGGER_PACKET(s):
    print("DEFINE_SERVER_LOGGER_PACKET Sent")
    s.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)

    # get IP address
    for ifaceName in interfaces():
            addresses = [i['addr'] for i in
            ifaddresses(ifaceName).setdefault(AF_INET,
            [{'addr':'No IP addr'}] )]
            print('%s: %s' % (ifaceName, ', '.join(addresses)))

    # last interface (wlan0) grabbed
    print(addresses)
    myIP = addresses[0].split('.')
    print(myIP)
    data= ["" for i in range(14)]

    data[0] = int("F0", 16).to_bytes(1,'little')
    data[1] = int(DEFINE_SERVER_LOGGER_PACKET).to_
    bytes(1,'little')
    data[2] = int("FF", 16).to_bytes(1,'little') # swarm id (FF
    means not part of swarm)
    data[3] = int(VERSIONNUMBER).to_bytes(1,'little')
    data[4] = int(myIP[0]).to_bytes(1,'little') # 1 octet of ip
    data[5] = int(myIP[1]).to_bytes(1,'little') # 2 octet of ip
    data[6] = int(myIP[2]).to_bytes(1,'little') # 3 octet of ip
    data[7] = int(myIP[3]).to_bytes(1,'little') # 4 octet of ip
    data[8] = int(0x00).to_bytes(1,'little')
    data[9] = int(0x00).to_bytes(1,'little')
    data[10] = int(0x00).to_bytes(1,'little')
    data[11] = int(0x00).to_bytes(1,'little')
    data[12] = int(0x00).to_bytes(1,'little')
    data[13] = int(0x0F).to_bytes(1,'little')
    print("data=", data)
    print("len(data)=", len(data))
    mymessage = ''.encode()
    theMessage = (mymessage.join(data), ('<broadcast>'.
    encode(), MYPORT))
    print("theMessage=", theMessage)
    print("len(theMessage)=", len(theMessage))
    mymessage = ''.encode()
    s.sendto(mymessage.join(data), ('<broadcast>'.encode(),
    MYPORT))
```

The next section of the code to be discussed is the web map that is used to display the status of the LightSwarm code. We use html to generate a display of the current Swarm status. The code in Listing 2-15 produces Figure 2-12. You can access this web page by typing "file:///home/pi/SDL_Pi_LightSwarm/state/swarm.html" into a browser on your Raspberry Pi (I used the Chromium browser).

Listing 2-15. Web Page Building Code

```
# build Webmap

def buildWebMapToFile(logString, swarmSize ):

    webresponse = ""

    swarmList = logString.split("|")
    for i in range(0,swarmSize):
        swarmElement = swarmList[i].split(",")
        print("swarmElement=", swarmElement)
        webresponse += "<figure>"
        webresponse += "<figcaption"
        webresponse += " style='position: absolute; top: "
        webresponse +=  str(100-20)
        webresponse +=  "px; left: "
        +str(20+120*i)+  "px;'/>\n"
        if (int(swarmElement[5]) == 0):
            webresponse += "&nbsp;&nbsp;&nbsp&nbsp;&nbsp;---"
        else:
            webresponse += "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp
            ;%s" % swarmElement[5]

        webresponse += "</figcaption>"
        #webresponse += "<img src='" + "http://192.168.1.40:9750"
        webresponse += "<img src='"

        if (swarmElement[4] == "PR"):
            if (swarmElement[1] == "1"):
                webresponse += "On-Master.png' style='position:
                absolute; top: "
            else:
                webresponse += "On-Slave.png' style='position:
                absolute; top: "
        else:
            if (swarmElement[4] == "TO"):
                webresponse += "Off-TimeOut.png'
                style='position: absolute; top: "
            else:
                webresponse += "Off-NotPresent.png'
                style='position: absolute; top: "

        webresponse +=  str(100)
        webresponse +=  "px; left: " +str(20+120*i)+  "px;'/>\n"

        webresponse += "<figcaption"
        webresponse += " style='position: absolute; top: "
        webresponse +=  str(100+100)
        webresponse +=  "px; left: " +str(20+120*i)+  "px;'/>\n"
        if (swarmElement[4] == "PR"):
            if (swarmElement[1] == "1"):
                webresponse += "&nbsp;&nbsp;&nbsp;&nbsp;Master"
            else:
                webresponse += "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
                Slave"
        else:
            if (swarmElement[4] == "TO"):
                webresponse += "TimeOut"
            else:
                webresponse += "Not Present"

        webresponse += "</figcaption>"

        webresponse += "</figure>"

    f = open("/home/pi/SDL_Pi_LightSwarm/state/figure.html",
    "w")

    f.write(webresponse)

    f.close()

    f = open("/home/pi/SDL_Pi_LightSwarm/state/swarm.html",
    "w")
    fh = open("/home/pi/SDL_Pi_LightSwarm/state/swarmheader.
    txt", "r")
    ff = open("/home/pi/SDL_Pi_LightSwarm/state/swarmfooter.
    txt", "r")

    webheader = fh.read()
    webfooter = ff.read()

    f.write(webheader)
    f.write(webresponse)
    f.write(webfooter)
    f.close
    fh.close
    ff.close
```



Swarm Status

***Figure 2-12.** HTML Web Page Produced by Web Map Code in LightSwarm Logging Software*

Listing 2-16 looks at incoming swarm IDs and builds a current table of matching IDs, removing old ones when adding new ones. The maximum number of swarm devices you can have is five, but can be easily increased.

Listing 2-16. Incoming Swarm Analysis Code

```
def setAndReturnSwarmID(incomingID):

    for i in range(0,SWARMSIZE):
        if (swarmStatus[i][5] == incomingID):
            return i
        else:
            if (swarmStatus[i][5] == 0):
            # not in the system, so put it in

                swarmStatus[i][5] = incomingID;
                print("incomingID %d " % incomingID)
                print("assigned #%d" % i)
                return i

    # if we get here, then we have a new swarm member.
    # Delete the oldest swarm member and add the new one in
    # (this will probably be the one that dropped out)

    oldTime = time.time();
    oldSwarmID = 0
    for i in range(0,SWARMSIZE):
        if (oldTime > swarmStatus[i][1]):
            ldTime = swarmStatus[i][1]
            oldSwarmID = i

    # remove the old one and put this one in....
    swarmStatus[oldSwarmID][5] = incomingID;
    # the rest will be filled in by Light Packet Receive
    print("oldSwarmID %i" % oldSwarmID)

    return oldSwarmID
```

Finally, Listing 2-17 is the main code for the Python program. It is very similar in function to the setup() code for the ESP8266 in the Arduino IDE. We use this to define variables, send out one-time commands, and set up the UDP interface.

Listing 2-17. LightSwarm Logger Startup Code

```
# set up sockets for UDP

s=socket(AF_INET, SOCK_DGRAM)
host = 'localhost';
s.bind(('',MYPORT))

print("--------------")
print("LightSwarm Logger")
print("Version ", VERSIONNUMBER)
print("--------------")

# first send out DEFINE_SERVER_LOGGER_PACKET to tell swarm
where to send logging information

SendDEFINE_SERVER_LOGGER_PACKET(s)
time.sleep(3)
SendDEFINE_SERVER_LOGGER_PACKET(s)

# swarmStatus
swarmStatus = [[0 for x  in range(6)] for x in
range(SWARMSIZE)]

# 6 items per swarm item

# 0 - NP  Not present, P = present, TO = time out
# 1 - timestamp of last LIGHT_UPDATE_PACKET received
# 2 - Master or slave status   M S
# 3 - Current Test Item - 0 - CC 1 - Lux 2 - Red 3 - Green
4 - Blue
```

```
# 4 - Current Test Direction  0 >=   1 <=
# 5 - IP Address of Swarm

for i in range(0,SWARMSIZE):
    swarmStatus[i][0] = "NP"
    swarmStatus[i][5] = 0

#300 seconds round
seconds_300_round = time.time() + 300.0

#120 seconds round
seconds_120_round = time.time() + 120.0

completeCommand()
```

Listing 2-18 provides the main program loop. Note this is very similar to the loop() function in the ESP8266 Swarm code. In this code, we check for incoming UDP packets, process commands from LSCommand, update status information, and perform periodic commands. It is in this loop that we would be storing the Swarm status, packets, and information if we wanted to be able to analyze the swarm behavior from the archival data.

Listing 2-18. Raspberry Pi Logger Main Loop

```
while(1) :

    # receive datclient (data, addr)
    d = s.recvfrom(1024)

    message = d[0]
    addr = d[1]

    if (len(message) == 14):

        if (message[1] == LIGHT_UPDATE_PACKET):
            incomingSwarmID = setAndReturnSwarmID((message[2]))
            swarmStatus[incomingSwarmID][0] = "P"
```

```
                swarmStatus[incomingSwarmID][1] = time.time()
                #print("in LIGHT_UPDATE_PACKET")

            if ((message[1]) == RESET_SWARM_PACKET):
                print("Swarm RESET_SWARM_PACKET Received")
                print("received from addr:",addr)

            if ((message[1]) == CHANGE_TEST_PACKET):
                print("Swarm CHANGE_TEST_PACKET Received")
                print("received from addr:",addr)

            if ((message[1]) == RESET_ME_PACKET):
                print("Swarm RESET_ME_PACKET Received")
                print("received from addr:",addr)

            if ((message[1]) == DEFINE_SERVER_LOGGER_PACKET):
                print("Swarm DEFINE_SERVER_LOGGER_PACKET Received")
                print("received from addr:",addr)

            if ((message[1]) == MASTER_CHANGE_PACKET):
                print("Swarm MASTER_CHANGE_PACKET Received")
                print("received from addr:",addr)

            #for i in range(0,14):
            #    print("ls["+str(i)+"]="+format((message[i]),
                 "#04x"))

        else:
            if ((message[1]) == LOG_TO_SERVER_PACKET):
                print("Swarm LOG_TO_SERVER_PACKET Received")

                # process the Log Packet
                logString = parseLogPacket(message)
                buildWebMapToFile(logString, SWARMSIZE )
            else:
                print("error message length = ",len(message))
        if (time.time() >  seconds_120_round):
            # do our 2 minute round
            print(">>>>doing 120 second task")
            sendTo = random.randint(0,SWARMSIZE-1)
            SendBLINK_BRIGHT_LED(s, sendTo, 1)
            seconds_120_round = time.time() + 120.0

        if (time.time() >  seconds_300_round):
            # do our 2 minute round
            print(">>>>doing 300 second task")
            SendDEFINE_SERVER_LOGGER_PACKET(s)
            seconds_300_round = time.time() + 300.0

        processCommand(s)
```

Note the last line of code, processCommands. This function is where the LSCommand.txt file is read and the commands inside the file are executed.

Results

I constructed a LightSwarm consisting of five individual swarm devices. The swarm can be seen in Figure 2-13.

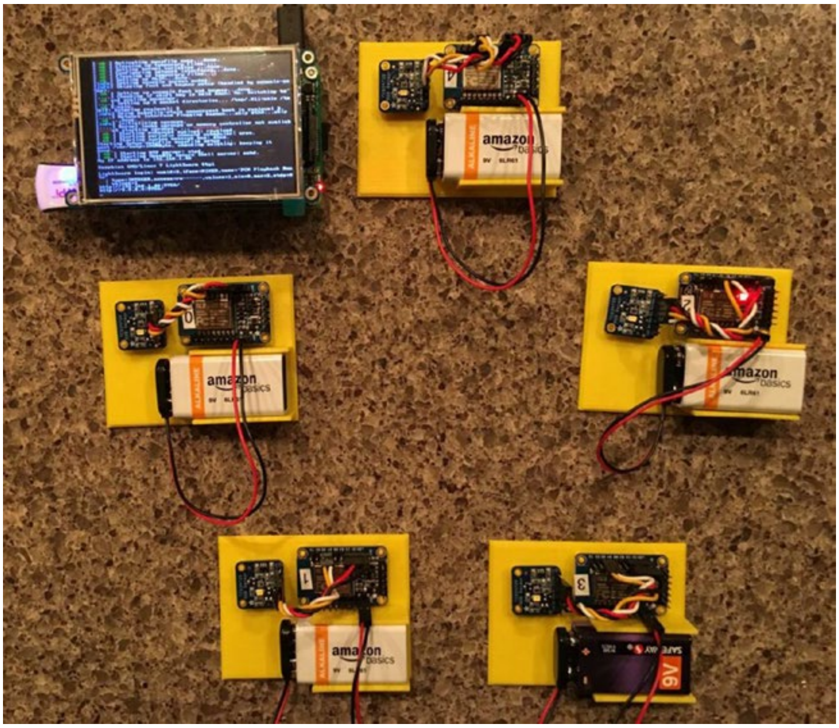


Figure 2-13. *The LightSwarm*

Finally, some results from the devices are shown in Listing 2-19. First of all is the serial debugging output from a LightSwarm device. First the device is initialized, receives an IP address from the WiFi access point (named gracie in this case), and then starts listening and sending packets. The device is swarm device #38 and receives packets from #42 and #44 but remained the master as the CC (clear color) of #28 is 9484, while the incoming packets from #42 and #44 had CC values of 253 and 626, respectively, and were both slaves.

Listing 2-19. Results from LightSwarm IoT Device Run on ESP8266

```
--------------------------
LightSwarm
Version 28
--------------------------
 12/29/2020
Compiled at:12:37:09 Dec 30 2020

analogRead(A0)=133
44
Found sensor
LightSwarm Instance: 0
Connecting to gracie
...
WiFi connected
IP address:
192.168.1.38
Starting UDP
Local port: 2910
clearColor =0
MySwarmID=0
Color Temp: 3816 K - Lux: 2321 - R: 2065 G: 2514 B: 1334 C:
6445
packetbuffer[1] =0
LIGHT_UPDATE_PACKET received from LightSwarm #44
incomingID 44  assigned #1
LS Packet Recieved from #44 SwarmState:SLAVE CC:680 RC:306
GC:176 BC:209 Version=28
swarmClear[0] = 6445  swarmTimeStamp[0] = 1
#0/1/28:ME swarmClear[1] = 680  swarmTimeStamp[1] = 3074
#1/0/28:PR swarmClear[2] = 0  swarmTimeStamp[2] = -1
```

```
#2/0/0:NP swarmClear[3] = 0  swarmTimeStamp[3] = -1
#3/0/0:NP swarmClear[4] = 0  swarmTimeStamp[4] = -1
#4/0/0:NP
MasterStatus:MASTER/cc=6445/KS:(IP unset)
--------
swarmAddress[0] = 38
swarmAddress[1] = 44
swarmAddress[2] = 0
swarmAddress[3] = 0
swarmAddress[4] = 0
--------
Broadcast ToSwarm = 255 Delay = 160ms : Color Temp: 3811 K -
Lux: 2322 - R: 2068 G: 2515 B: 1333 C: 6447
packetbuffer[1] =0
LIGHT_UPDATE_PACKET received from LightSwarm #42
incomingID 42  assigned #2
LS Packet Recieved from #42 SwarmState:SLAVE CC:593 RC:230
GC:180 BC:193 Version=28
swarmClear[0] = 6447  swarmTimeStamp[0] = 1
#0/1/28:ME swarmClear[1] = 680  swarmTimeStamp[1] = 3074
#1/0/28:PR swarmClear[2] = 593  swarmTimeStamp[2] = 4284
#2/0/28:PR swarmClear[3] = 0  swarmTimeStamp[3] = -1
#3/0/0:NP swarmClear[4] = 0  swarmTimeStamp[4] = -1
#4/0/0:NP
MasterStatus:MASTER/cc=6447/KS:(IP unset)
--------
swarmAddress[0] = 38
swarmAddress[1] = 44
swarmAddress[2] = 42
swarmAddress[3] = 0
swarmAddress[4] = 0
--------
```

```
Broadcast ToSwarm = 255 Delay = 216ms : Color Temp: 3812 K -
Lux: 2323 - R: 2070 G: 2517 B: 1335 C: 6455
packetbuffer[1] =0
LIGHT_UPDATE_PACKET received from LightSwarm #42
LS Packet Recieved from #42 SwarmState:SLAVE CC:592 RC:229
GC:179 BC:193 Version=28
swarmClear[0] = 6455  swarmTimeStamp[0] = 1
```

Listing 2-20 is the output from the Raspberry Pi LightSwarm logging software. The first thing the logging software does is send out a "DEFINE_SERVER_LOGGING_PACKET" to tell the swarm devices the IP address (192.168.1.40) of the server so the swarm master can send logging packets directly to the Raspberry Pi, rather than use the already crowded UDP broadcast ports. Finally, we see a packet coming in from SwarmID #42. Number 42 picked up the server address, and since it was the master of the swarm, it started sending in log packets to the Raspberry Pi. Note from the log results, it looks like #42 has friends nearby (#44 and #38) and is part of a swarm of at least three units.

Listing 2-20. Output from Raspberry Pi LightSwarm Logger

```
--------------
LightSwarm Logger
Version  7
--------------
DEFINE_SERVER_LOGGER_PACKET Sent
lo: 127.0.0.1
eth0: No IP addr
wlan0: 192.168.1.40
['192.168.1.40']
['192', '168', '1', '40']
DEFINE_SERVER_LOGGER_PACKET Sent
lo: 127.0.0.1
```

```
eth0: No IP addr
wlan0: 192.168.1.40
['192.168.1.40']
['192', '168', '1', '40']
Swarm DEFINE_SERVER_LOGGER_PACKET Received
received from addr: ('192.168.1.40', 2910)
incomingID 42
assigned #0
incomingID 44
assigned #1
Swarm LOG_TO_SERVER_PACKET Received
incomingID 38
assigned #2
Log From SwarmID: 38
Swarm Software Version: 28
StringLength: 87
swarmElement= [' 0', '1', '28', '9493', 'PR', '38 ']
swarmElement= [' 1', '0', '28', '591', 'PR', '42 ']
swarmElement= [' 2', '0', '28', '679', 'PR', '44 ']
swarmElement= [' 3', '0', '0', '0', 'NP', '0 ']
swarmElement= [' 4', '0', '0', '0', 'NP', '0 ']
Swarm LOG_TO_SERVER_PACKET Received
Log From SwarmID: 38
Swarm Software Version: 28
StringLength: 87
swarmElement= [' 0', '1', '28', '9490', 'PR', '38 ']
swarmElement= [' 1', '0', '28', '591', 'PR', '42 ']
swarmElement= [' 2', '0', '28', '678', 'PR', '44 ']
swarmElement= [' 3', '0', '0', '0', 'NP', '0 ']
swarmElement= [' 4', '0', '0', '0', 'NP', '0 ']
Swarm DEFINE_SERVER_LOGGER_PACKET Received
```

```
received from addr: ('192.168.1.40', 2910)
Swarm LOG_TO_SERVER_PACKET Received
Log From SwarmID: 38
Swarm Software Version: 28
StringLength: 87
swarmElement= [' 0', '1', '28', '9490', 'PR', '38 ']
swarmElement= [' 1', '0', '28', '592', 'PR', '42 ']
swarmElement= [' 2', '0', '28', '678', 'PR', '44 ']
swarmElement= [' 3', '0', '0', '0', 'NP', '0 ']
swarmElement= [' 4', '0', '0', '0', 'NP', '0 ']
Swarm LOG_TO_SERVER_PACKET Received
Log From SwarmID: 38
Swarm Software Version: 28
StringLength: 87
swarmElement= [' 0', '1', '28', '9483', 'PR', '38 ']
swarmElement= [' 1', '0', '28', '592', 'PR', '42 ']
swarmElement= [' 2', '0', '28', '678', 'PR', '44 ']
swarmElement= [' 3', '0', '0', '0', 'NP', '0 ']
swarmElement= [' 4', '0', '0', '0', 'NP', '0 ']
```

What Else Can You Do with This Architecture?

The LightSwarm architecture is flexible. You can change the sensor, add more sensors, and put in more sophisticated algorithms for swarm behavior. In Chapter 5, we extend this architecture to more complex swarm behavior, actually changing some of the physical environment of the swarm devices.

Conclusion

A good part of the IoT will be the gathering of simple, small amounts of data, some analysis on the data, and the communication of that data to servers for action and further analysis on the Internet. The projects in Chapters 3 and 4 are of more complex IoT devices gathering lots of data, processing it, acting on the data, and communicating summaries to the Internet. The LightSwarm is different in that each Swarm element is simple and cooperates without a central controller to determine which element has the brightest light and then will act on that information (turning the red LED on).

Swarms of IoT devices can be made inexpensively, can exhibit unexpected complex behavior, and can be devilishly difficult to debug. If you are trying to debug such a system, log everything during development!!!

CHAPTER 3

Building an IoT Weather Station

Chapter Goal: Gathering Data and Transmission of Data Across the Internet Topics Covered in This Chapter:

- How to build an IoT Weather Station
- What is a software-defined radio
- How to connect to 433MHz wireless sensors

Figure 3-1. *Picture of SkyWeather2 Station Deployed Outside*

- How to gather data to analyze your system performance
- How to hook up the SkyWeather2 Weather Station
- How to install and configure the SkyWeather2 Python3 software
- How to build the 3D printed parts for SkyWeather2
- How to connect your weather station to the IoT (WeatherSTEM)

Everybody talks about the weather. In this chapter, we are going to talk about the weather in much more detail than just the temperature.

In the previous chapter, we looked at building simple IoT devices that would measure temperature and share that information with a server and other IoT devices. It was a simple application, but still illustrated a number of important concepts. In this chapter, we are building a much more complex and flexible project based on using the Raspberry Pi as part of the IoT device.

The SkyWeather2 not only gathers 13 different types of weather data; it also monitors and reports its own state, status, and health.

The best part? SkyWeather2 (Figure 3-1) requires *no soldering* to assemble, test, or build because of the use of Grove connectors. More on Grove connectors later in this chapter.

IoT Characterization of This Project

As I discussed in Chapter 1, the first thing to do to understand an IoT project is to look at our six different aspects of IoT. SkyWeather2 is a more complex project, but Table 3-1 breaks it down into our six components.

Table 3-1. *SkyWeather2 Characterization (CPLPFC)*

Aspect	Rating	Comments
Communications	9	WiFi connection to the Internet – can do ad hoc mesh-type communication and Bluetooth
Processor power	5	Raspberry Pi 4B w/4GB
Local storage	8	16GB of SD card
Power consumption	2	~800mA consumption – not reasonable for small batteries or small solar system
Functionality	8	Full Linux-based system. MySQL, etc.
Cost	2	Expensive for many applications

Ratings are from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications. This gives us a CPLPFC rating of 5.7. Great for learning, not so good for deployment for most applications.

No doubt about it, the Raspberry Pi is a very flexible and powerful IoT platform. However, the power consumption, cost, and physical size of the device make it more suitable for prototyping or for stand-alone, highly functional IoT units.

How Does This Device Hook Up to the IoT?

With SkyWeather2, hook up to the Internet using the WiFi connector. We can use the SDR (software-defined radio) to hook up to 422MHz wireless local IoT weather sensors, and we can also use the Pi SkyWeather2 Hat to connect to other wired devices. In this chapter, we will be using the WiFi interface to talk to the outside world.

Data Gathering

The SkyWeather2 uses 13 different sensors to detect weather connections. Because I am using a Raspberry Pi and have good storage mechanisms and disk space, I use a MySQL database to store all the weather data for future analysis and download. We also use the dash_app to build graphs locally and use an iOS app called Blynk to display the information across the Internet. WeatherSTEM is used to show our local weather on the cloud.

What Are Grove Connectors

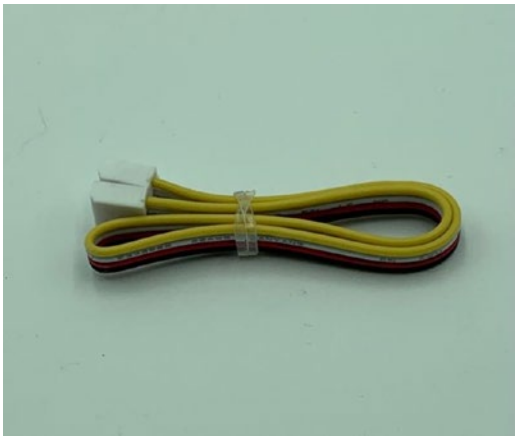


Figure 3-2. *A Grove Cable*

Grove is a modular, standardized connector prototyping system. Grove takes a building block approach to assembling electronics. Compared to the jumper or solder-based system, it is easier to connect, experiment, and build and simplifies the learning system, but not to the point where it becomes dumbed down. Some of the other prototype systems out there take the level down to building blocks. Good stuff to be learned that way, but the Grove system allows you to build real systems. It just makes it easier to hook things up.

The Grove system consists of a base unit and various modules with standardized Grove connectors.

We use Grove connectors and cable in building SkyWeather2. This means no soldering is needed to build this project.

See a full tutorial on Grove cables and connectors on SwitchDoc Labs:

www.switchdoc.com/2021/01/tutorial-intro-to-grove-connectors-for-arduinoraspberry-pi-projects/

And a great example of projects using Grove connectors:

www.switchdoc.com/2018/12/tutorial-using-an-analog-to-digital-converter-with-your-raspberry-pi-2/

The Project – SkyWeather2

SkyWeather2 is a Raspberry Pi WiFi connected weather station designed for use in the IoT by SwitchDoc Labs. This is a great system to build and tinker with. All of it is modifiable and all is open source. The following are the most important functions:

- Barometric pressure
- Outside temperature
- Outside humidity
- Altitude
- Inside temperature (at up to eight locations!)
- Inside humidity (at up to eight locations)
- Sunlight
- UV index
- Wind speed
- Wind gusts

- Wind direction
- Rain
- All your weather information on the cloud
- SkyCamera

And optionally:

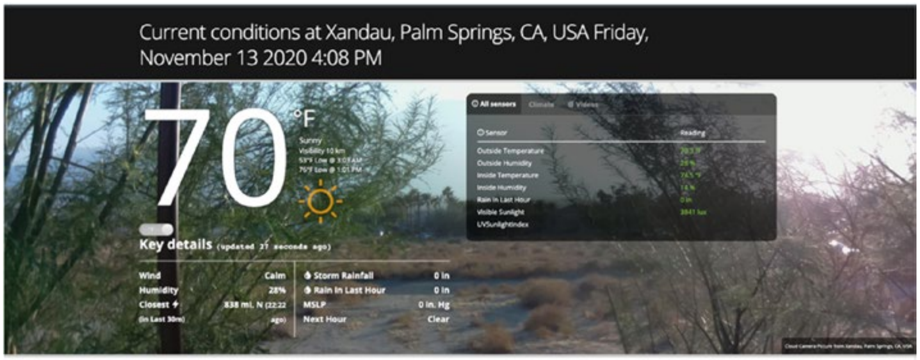
- Air quality – AQI (have your own local air quality sensor)
- ThunderBoard Lightning Detection
- Amazon Alexa integration
- WeatherUnderGround installation

This chapter will show you how to build a SkyWeather2 Raspberry Pi Weather Station. This project grew out of a number of other projects, including the massive Project Curacao [`www.switchdoc.com/project-curacao-introduction-part-1/`], a solar-powered environmental monitoring system deployed on the Caribbean tropical island of Curacao. Project Curacao was written up in an extensive set of articles in *The MagPi* magazine (starting in Issue 18 and continuing through Issue 22).

The SkyWeather2 Weather Station is an excellent education project. There are many aspects of this project that can be looked at and analyzed for educational purposes:

- Temperature, wind, and humidity data analysis.
- Add your own sensors for UV, dust and pollen count, and light color.

The included SkyCamera Pi Camera allows you to take pictures of your local conditions and upload them to WeatherSTEM. WeatherSTEM builds time-lapse movies every night and uses your latest picture as backdrop to your weather station data as shown in Figure 3-3. Figure 3-4 shows you the overall architecture of the SkyWeather2 system.


Current conditions at Xandau, Palm Springs, CA, USA Friday, November 13 2020 4:08 PM

70 °F
Sunny
Visibility 10 km
53°F Low @ 3:03 AM
75°F Low @ 1:01 PM

Key details (updated 27 seconds ago)

Wind	Calm	Storm Rainfall	0 in
Humidity	28%	Rain In Last Hour	0 in
Closest (in Last 30m)	838 mi. N (22:22 ago)	MSLP	0 in. Hg
		Next Hour	Clear

All sensors Climate Videos

Sensor	Reading
Outside Temperature	70.3 °F
Outside Humidity	28 %
Inside Temperature	74.5 °F
Inside Humidity	14 %
Rain In Last Hour	0 in
Visible Sunlight	3841 lux
UVSunlightIndex	

Cloud Camera Picture from Xandau, Palm Springs, CA, USA

Figure 3-3. *SkyWeather2 in Palm Springs with SkyCamera Picture in Background*

The Architecture of SkyWeather2

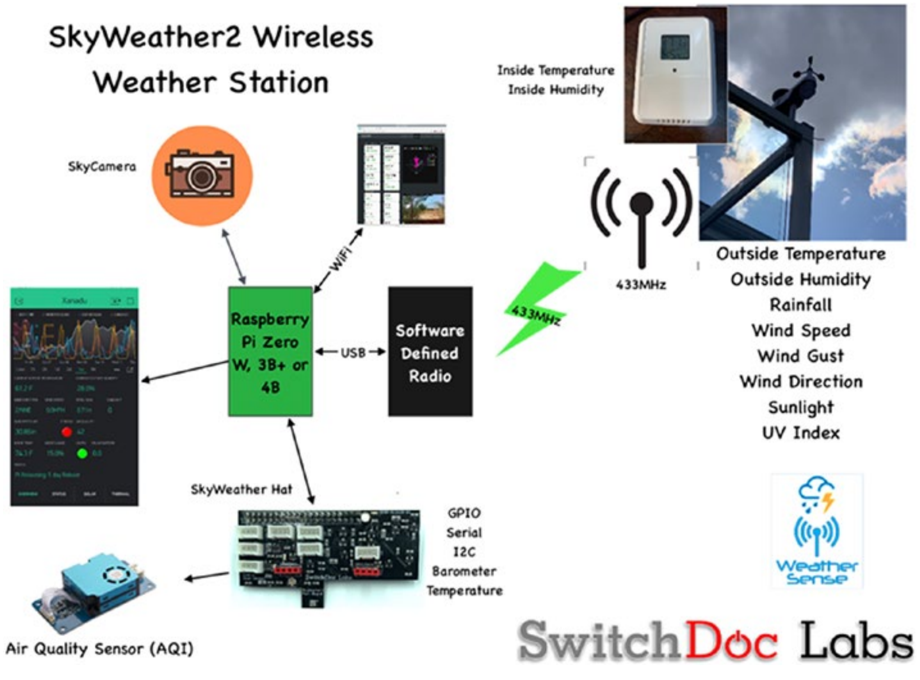


Figure 3-4. *The SkyWeather2 Architecture*

This looks really complicated! It is not. There is a step-by-step assembly and testing manual that makes it a "snap" to put together and test.

The major parts of the kit are as follows:

- WeatherRack2 433MHz wireless sensors (see Figure 3-5)
- Software-defined radio
- SkyWeather2 Hat (see Figure 3-6)
- SkyCamera (see Figure 3-7)
- Raspberry Pi (see Figure 3-7)

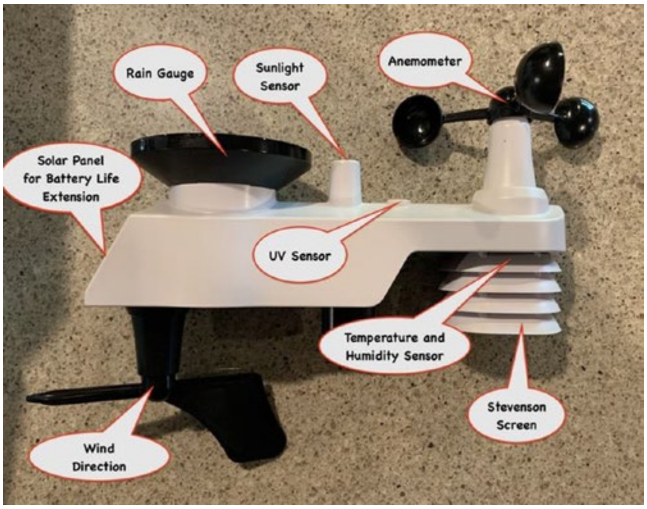


Figure 3-5. *WeatherRack2 433MHz Wireless Sensors*

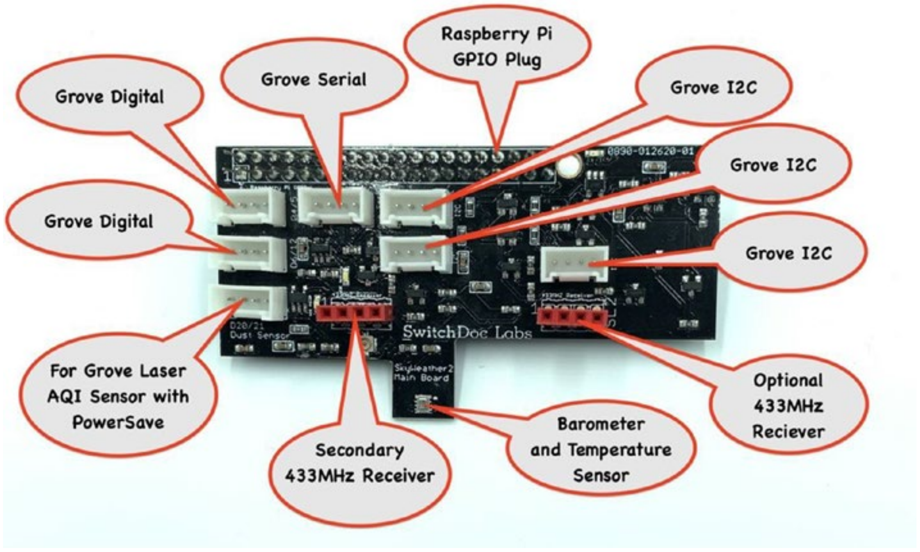


Figure 3-6. *SkyWeather2 Hat for the Raspberry Pi*

What Do You Need to Build This Project?

SwitchDoc Labs has organized all the parts for building SkyWeather into a kit and few optional items. Later on in this chapter, I will talk about the 3D printed case for SkyWeather (Figures 3-9 and 3-10) and the weatherization parts needed.

Parts List:

- SkyWeather2 kit – shop.switchdoc.com – `https://shop.switchdoc.com/products/skyweather2-raspberry-pi-based-weather-station-kit-for-the-cloud`
- Pre-loaded SD card (not required, but strongly recommended) – shop.switchdoc.com – `https://shop.switchdoc.com/products/16gb-sd-card-with-stretch-smart-garden-system-groveweatherpi`

- Raspberry Pi (you can use many different ones, but we recommend the Raspberry Pi 4B kit to begin) – amazon.com – https://amzn.to/38C2JZO
- 3 AA batteries
- 2 AAA batteries

Connecting and Testing the Hardware

SwitchDoc Labs has published excellent step-by-step assembly and testing manuals for SkyWeather 2. The latest versions of these are all available on the SkyWeather2 Product page: https://shop.switchdoc.com/products/skyweather2-raspberry-pi-based-weather-station-kit-for-the-cloud

Here are the most important manuals:

- Step-by-Step Assembly and Testing SkyWeather2 Manual: www.switchdoc.com/wp-content/uploads/2020/12/SkyWeather2AssemblyAndTestManual1.2.pdf
- Step-by-Step WeatherRack2 Assembly and Installation Manual: www.switchdoc.com/wp-content/uploads/2020/10/WeatherRack2Installation1.2.pdf

If you will be putting your SkyWeather2 station outside (highly recommended to get a good SkyCamera view), then look at the 3D printed cases and the Weatherization Manual later in this chapter. The assembly process is very straightforward (see Figures 3-6 and 3-7).

Figure 3-7. *Assembling the SkyWeather2 WeatherRack2*

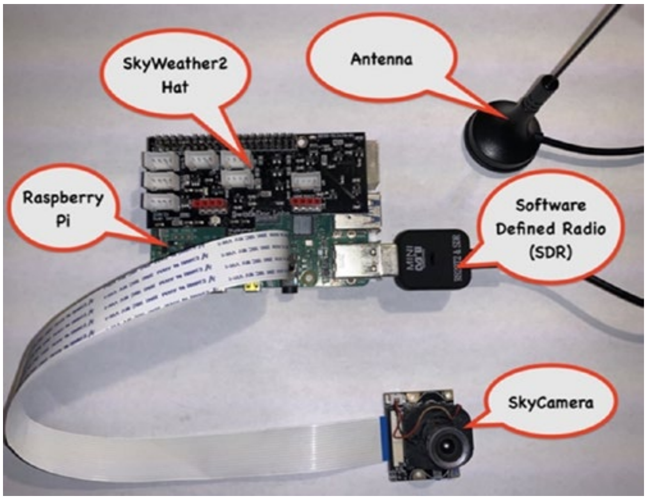


Figure 3-8. *SkyWeather2 Fully Assembled*

Weatherization and the 3D Printed Box for SkyWeather2

Because of the unique architecture of SkyWeather2, the main Raspberry Pi system does not require the entire system to be outside. However, many people prefer to mount the unit outside to capture a better sky picture, and if they are using the optional laser AQI (air quality sensor) to capture the outside air quality in their neighborhood.

Many people now have access to 3D printers so it makes sense to build custom 3D printed boxes for projects like SkyWeather2 as shown in Figure 3-10. If you don't have access to one, you can find the complete 3D printed case on shop.switchdoc.com – search for SkyWeather2 3D Prints.

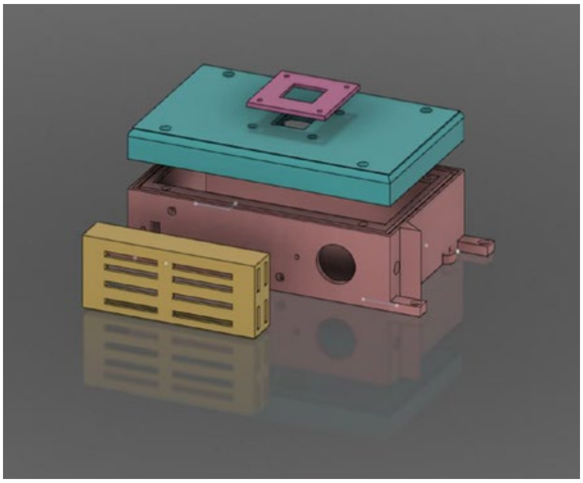


Figure 3-9. *3D Printed Box Showing Assembly*

Figure 3-9 shows all of the 3D printed parts for the SkyWeather2 kit. It takes about 25 hours to print out one full part set on our QIDI X-Max printers in ABS plastic.

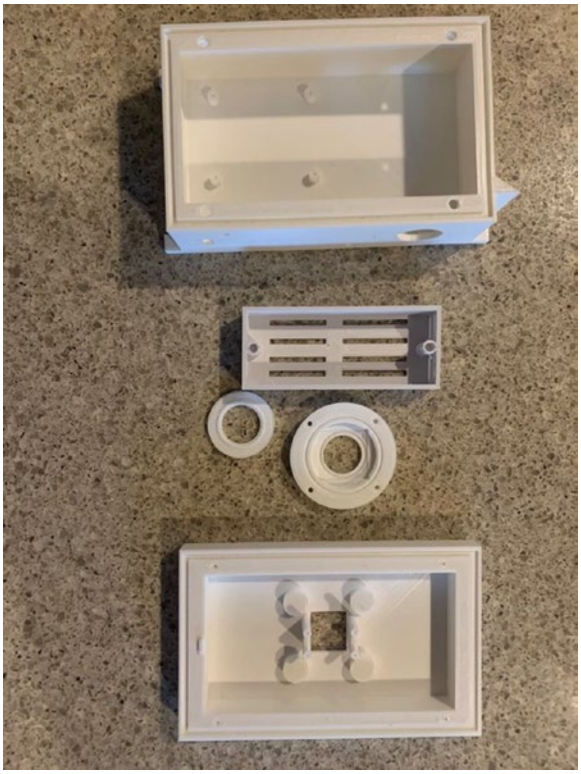


Figure 3-10. *SkyWeather2 ABS 3D Printed Parts*

All the 3D Print STL files and the Fusion360 Design files are on `https://github.com/switchdoclabs/SDL_STL_SkyWeather2` for those who want to print or modify the design.

The required assembly parts list and a step-by-step assembly and weatherization manual are located on the SkyWeather2 Product page on shop.switchdoc.com.

- SkyWeather2 WeatherProofing and Test Manual – `www.switchdoc.com/wp-content/uploads/2020/12/SkyWeather2WeatherProofingAndTestManual1.1.pdf`

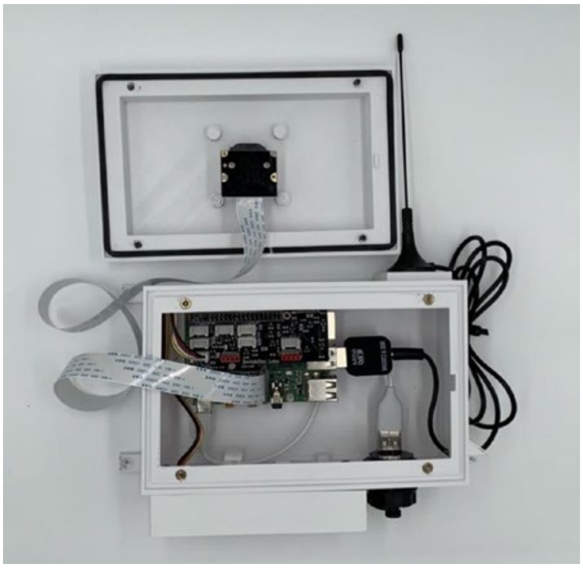


Figure 3-11. *Assembled and Weather Proofed SkyWeather2*

The Software

A big part of the SkyWeather2 project is the software. All of the Python software for this project is up on github at the switchdoclabs section [`https://github.com/switchdoclabs/WeatherPi`]. I also included all of the various libraries for the I2C devices we are using.

Non-normal Requirements for Your Pi

You will need to add the following software and libraries to your Raspberry Pi:

- MySQL – There are lots of tutorials on the Internet for installing MySQL. Here is the one we used [raspberrywebserver.com/sql-databases/using-mysql-on-a-raspberry-pi.html]. The structure of the WeatherPi MySQL database in mysqldump format is located on github [https://github.com/switchdoclabs/WeatherPiSQL]. You can use this file to build the MySQL database for the SkyWeather2 project.

The SkyWeather2 Python3 Software

The SkyWeather software is pretty simple. The application was much less complex than the Project Curacao software [www.switchdoc.com/project-curacao-software-system-part-6/]. Basically, the SkyWeather program is a set of tasks scheduled by a Python library called apscheduler.

You can download all the SkyWeather2 Python 3 software on github.com/switchdoclabs/SDL_Pi_SkyWeather2:

https://github.com/switchdoclabs/SDL_Pi_SkyWeather2

Using a terminal window, type in these two commands:

```
cd
git clone https://github.com/switchdoclabs/SDL_Pi_SkyWeather2
```

As you can see in the following, the main loop for this program is trivial because all the brains are in the apscheduler tasks:

```
# Main Loop
while True:
    time.sleep(1.0)
```

The Scheduler Tasks

In Table 3-2, all of the apscheduler tasks are defined and described. Feel free to look at the code for any of these tasks in which you might be interested. The Python3 code for adding a simple task is shown here:

```
# prints out the date and time to console
scheduler.add_job(tasks.tick, 'interval', seconds=60)
```

```
The above code adds a task (the Python functions "tasks.
tick()") to be fired every 60 seconds.   This complete tick()
function is shown below:

def tick():
    print('Tick! The time is: %s' % datetime.now())
```

The tick() function just prints out the time to the terminal window every 60 seconds. Now that is a simple task!

Table 3-2. *SkyWeather2 Tasks*

Task	Frequency	Description
tick()	60 seconds	Prints out the current time to the terminal window
wirelessSensors.readSensors	Starts once and keeps running thread	Reads all the 433MHz wireless sensors from the software-defined radio (SDR)
wiredSensors.readWiredSensors	30 seconds	Reads the wired sensors (barometric pressure, etc.)
state.printState	60 seconds	If SWDEBUG is true, prints out all the SkyWeather2 state variables
updateBlynk.blynkStateUpdate	30 seconds	Updates the Blynk app
watchdog.patTheDog	10 seconds	Tells the optional watchdog board that SkyWeather2 is still running
publishMQTT.publish	Depends on MQTT configuration	Publishes current weather data to the optional MQTT data channel
rebootPi	Every 5 days	Reboots the Raspberry Pi for reliability

(*continued*)

Table 3-2. (*continued*)

Task	Frequency	Description
util.barometricTrend	15 minutes	Calculates the current barometric pressure trend
DustSensor.read_AQI	12 minutes	Reads the option laser dust sensor to calculate local Air Quality Index (AQI)
pclogging.writeWeatherRecord	15 minutes	Writes out the current weather data to the MySQL database
pclogging.writeITWeatherRecord	15 minutes	Writes out current indoor temperature/humidity sensors to MySQL database. You can have up to eight of this T/H sensors

The Software-Defined Radio

A key part of the SkyWeather2 project is the software-defined radio. A software-defined radio (SDR) is a radio communication system where components that have been traditionally implemented in hardware (e.g., mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented by means of software on a personal computer or embedded system (still with significant hardware support).

The SDR scans, reads, demodulates, and decodes the incoming 433MHz signals from the WeatherRack2 and other optional sensors. Figure 3-12 shows the block diagram of an SDR system.

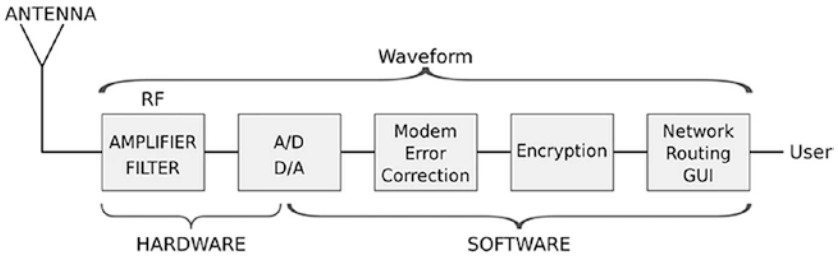
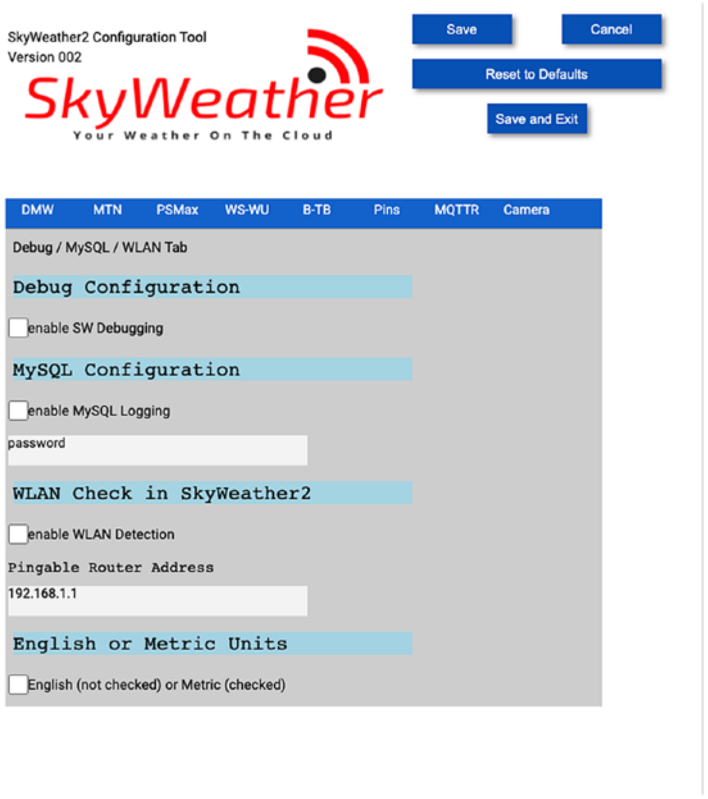


Figure 3-12. *Software-Defined Radio Architecture*

The SDR we are using with SkyWeather2 is based on the powerful RTL2832U and R820T tuner; it can tune into signals from 24MHz to 1850MHz. SwitchDoc Labs has written drivers for the WeatherSense weather sensors and supplies the drivers to the community open source.

The SkyWeather2 Configuration Software

Before you run SkyWeather2, you will use a Python configuration program showing a GUI (graphical user interface) for setting up your unit just the way you want. Figure 3-13 shows a page from the SkyWeatherConfiguration.py software.


SkyWeather2 Configuration Tool
Version 002

Debug / MySQL / WLAN Tab

Debug Configuration

☐ enable SW Debugging

MySQL Configuration

☐ enable MySQL Logging

password

WLAN Check in SkyWeather2

☐ enable WLAN Detection

Pingable Router Address

192.168.1.1

English or Metric Units

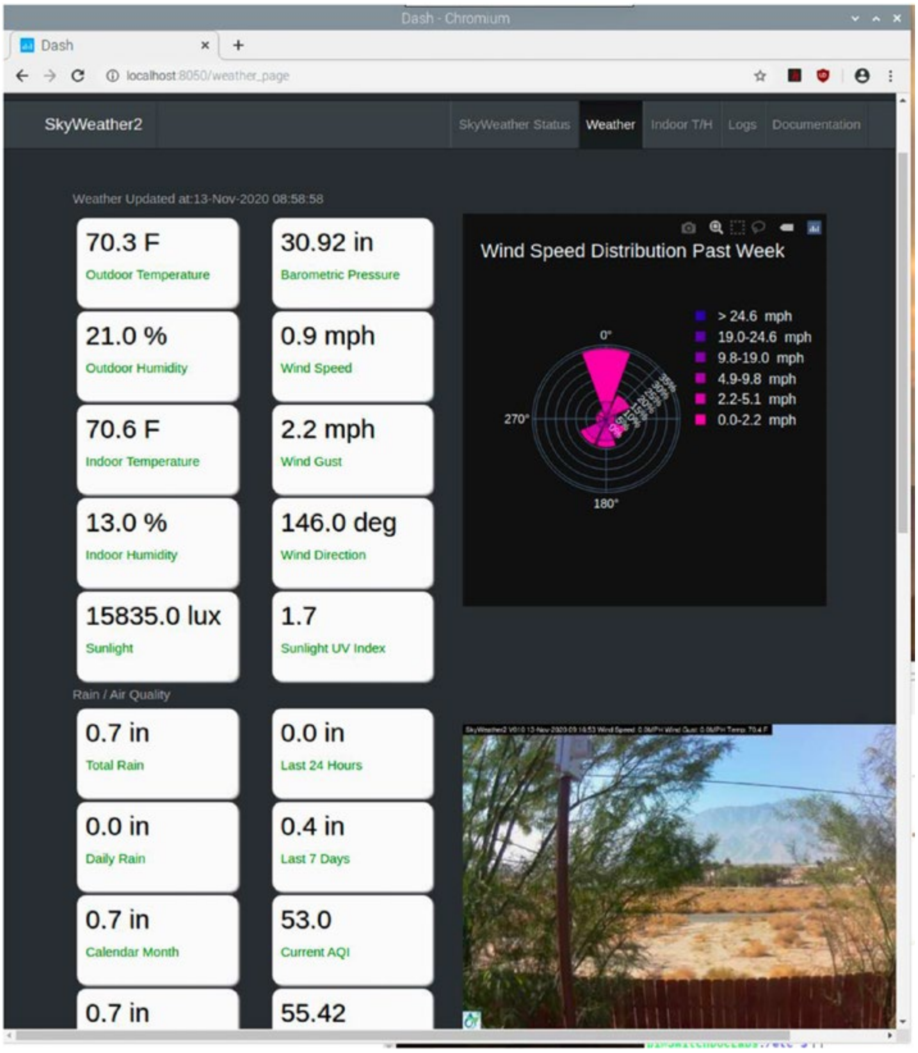
☐ English (not checked) or Metric (checked)

Figure 3-13. *SkyWeather2 Configuration Software*

The SkyWeather2 Configuration Manual contains detailed information about the SkyWeather2 Configuration software [`www.switchdoc.com/wp-content/uploads/2020/12/SkyWeather2ConfigurationAndOperationsManual1.1.pdf`].

The Dash App Local Display

The SkyWeather2 software also includes a dash_app that runs a browser-based set of displays and graphs that can be accessed on your home network. What is a dash_app? Dash is a powerful framework for building complex interactive data visualizations using pure Python. As with all of our SkyWeather2 software, the Dash App source code is supplied in github/switchdoclabs/SDL_Pi_SkyWeather2. Figure 3-14 shows one of the Dash App screens connected to SkyWeather2. Note the very, very cool wind rose in the upper right.



Weather Updated at:13-Nov-2020 08:58:58

70.3 F
Outdoor Temperature

30.92 in
Barometric Pressure

21.0 %
Outdoor Humidity

0.9 mph
Wind Speed

70.6 F
Indoor Temperature

2.2 mph
Wind Gust

13.0 %
Indoor Humidity

146.0 deg
Wind Direction

15835.0 lux
Sunlight

1.7
Sunlight UV Index

Rain / Air Quality

0.7 in
Total Rain

0.0 in
Last 24 Hours

0.0 in
Daily Rain

0.4 in
Last 7 Days

0.7 in
Calendar Month

53.0
Current AQI

0.7 in

55.42

Figure 3-14. *SkyWeather2 Dash App Main Page*

There are other Dash App pages showing a variety of historical weather information as shown in Figure 3-15. Note especially the optional local Air Quality Index graph.

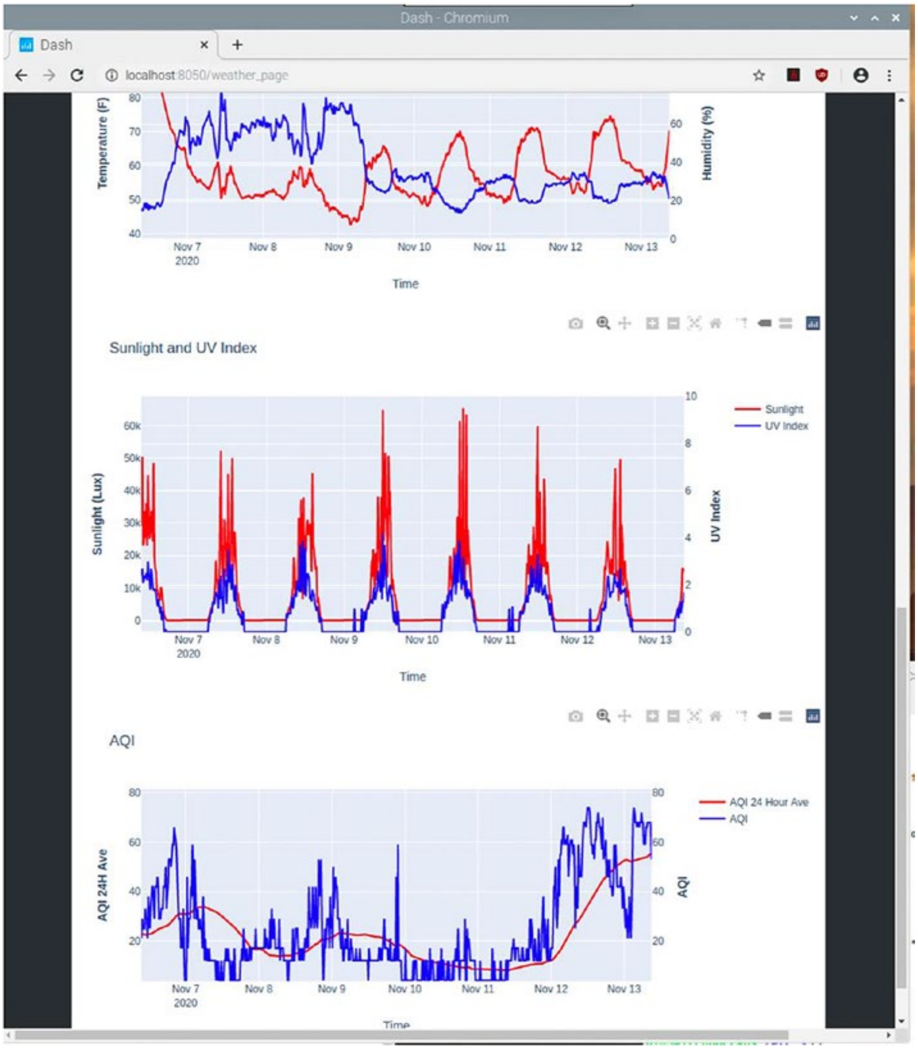


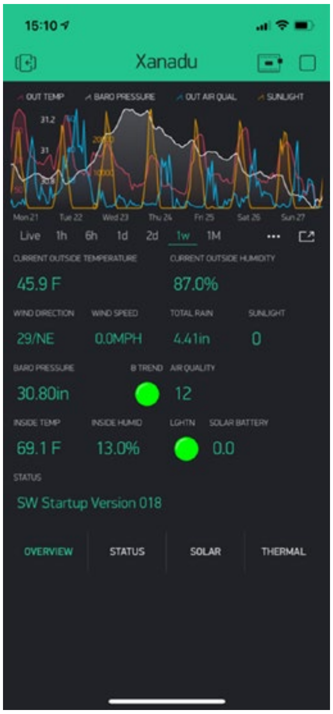
***Figure 3-15.** Dash App Historical Weather Graphs*

SkyWeather2 and Blynk

Blynk is a digital dashboard for your iOS or Android device that allows you to easily build graphical interfaces by dragging and dropping widgets. Blynk runs on iOS and Android apps to control Arduino, Raspberry Pi, and the like over the Internet.

It's a digital dashboard where you can build a graphical interface for your project by simply dragging and dropping widgets.

Blynk gives you current status of your SkyWeather station including solar power status (coming in 2021), thermal status, and current conditions. Updated to your phone every 30 seconds (see the updateBlynk status task in the scheduler task descriptions in Table 3-2). Figure 3-16 shows the front screen of the SkyWeather2 Blynk application on an iPhone.



CURRENT OUTSIDE TEMPERATURE 45.9 F
CURRENT OUTSIDE HUMIDITY 87.0%
WIND DIRECTION 29/NE
WIND SPEED 0.0MPH
TOTAL RAIN 4.41in
SUNLIGHT 0
BARO PRESSURE 30.80in
B TREND ●
AIR QUALITY 12
INSIDE TEMP 69.1 F
INSIDE HUMID 13.0%
LGHTN ●
SOLAR BATTERY 0.0
STATUS SW Startup Version 018

Figure 3-16. *SkyWeather2 Blynk App*

You can install the Blynk app on your iPhone or Android phone by following the Step-by-Step Blynk Configuration Manual located on the SkyWeather2 Product page on shop.switchdoc.com.

- SkyWeather2 Blynk Configuration Manual – `www.switchdoc.com/wp-content/uploads/2020/12/SkyWeather2WeatherBlynkConfiguration1.2.pdf`

Supplying Your Data to the World – WeatherSTEM

There are a number of websites and companies that would love you to send them your weather data. Two of these are WeatherSTEM and WeatherUnderGround. For the purposes of this project, we will send the data to WeatherSTEM.

That's right. Your SkyWeather2 station and sky pictures will be visible to everyone on the Internet, and you will be sharing your local weather with locations all over the globe!

WeatherSTEM is an innovative cloud-based weather education platform for people of all ages and professions. Our innovative curriculum is designed to help you and your students understand the myriad ways weather impacts just about everything in our lives. SkyWeather2 is a great way to become part of the WeatherSTEM community.

One of the things WeatherSTEM produces is a daily time-lapse video of your SkyCamera. Take a look at this example on YouTube:

`https://youtu.be/3Ogux1sY5Vo`

Figure 3-17 shows a still cut from a time-lapse video from WeatherSTEM.

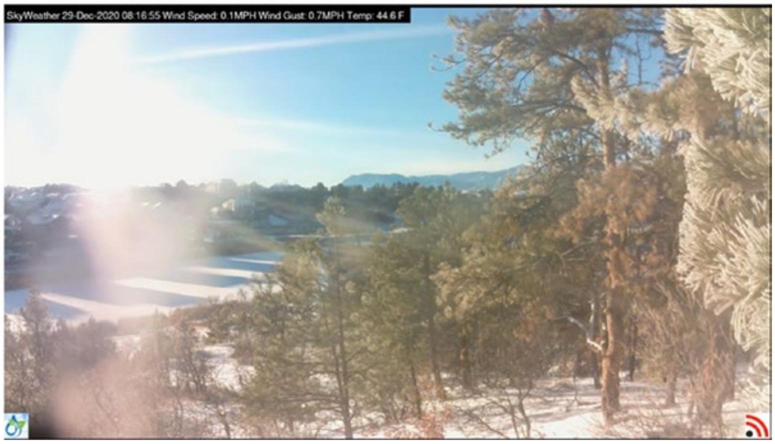
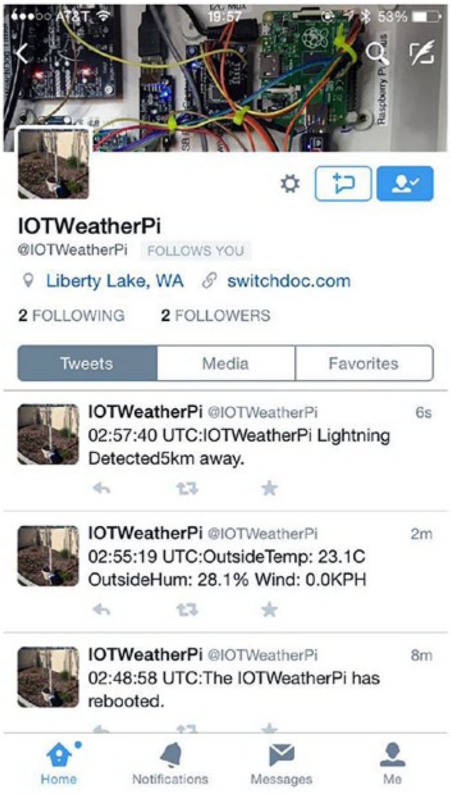

SkyWeather 29-Dec-2020 08:16:55 Wind Speed: 0.1MPH Wind Gust: 0.7MPH Temp: 44.6 F

Figure 3-17. *Still from WeatherSTEM Time-Lapse Video*

Tweeting Your Weather Data

Tweeting is a great way of reading short messages from people with little to say. With the SkyWeather2, you can join this epic project to be in on the fun. Of course, some people do have interesting things to say, and now you can start tweeting weather information. You may not have a lot of followers, but you can follow your IoT SkyWeather2 from anywhere!

Figure 3-18 shows an iPhone screenshot of the resulting Tweets from our test software. SwitchDoc Labs will be adding the Weather Tweeting capability in an upcoming SkyWeather2 release.



IOTWeatherPi
@IOTWeatherPi FOLLOWS YOU
Liberty Lake, WA switchdoc.com
2 FOLLOWING 2 FOLLOWERS

IOTWeatherPi @IOTWeatherPi 6s
02:57:40 UTC:IOTWeatherPi Lightning Detected5km away.

IOTWeatherPi @IOTWeatherPi 2m
02:55:19 UTC:OutsideTemp: 23.1C OutsideHum: 28.1% Wind: 0.0KPH

IOTWeatherPi @IOTWeatherPi 8m
02:48:58 UTC:The IOTWeatherPi has rebooted.

Figure 3-18. *SkyWeather2 Tweeting*

A Little History and Science

SkyWeather2, being open source, is very expandable – there are hundreds of Grove modules available!

SwitchDoc Labs has provided extra Grove connectors on the SkyWeather2 Hat for your own projects and sensors.

You can modify the software to do things we haven't even thought of yet!

SkyWeather2 is the last in a long series of weather projects from SwitchDoc Labs. The first weather station was Project Curacao 1, a full solar-powered station for a top ham radio station (PJ2T) down on the island nation of Curacao. This morphed into IOTWeatherPi (featured in the first edition of this book), then GroveWeatherPi, SkyWeather, and now the first wireless sensor-based station, SkyWeather2.

SwitchDoc Labs released the original SkyWeather kit in early 2019. It has been very successful, and there are installations all over the world. SkyWeather2 is a redesigned system using Python3 and the new SwitchDoc Labs Wireless WeatherRack2. It is much simpler to build and configure than the original.

The engineering on this project was great fun for the whole engineering team. Others handled the sensors and our manufacturers, but I was tasked with figuring out how to receive (demodulate) all the incoming 433MHz signals from the WeatherSense sensors. It was a heck of a challenge!

The first thing we did was to learn how to use a software-defined radio (SDR) on the Raspberry Pi. We started analyzing the signals coming into the antenna and figuring out what was going on. We had the digital data formats from our manufacturing partners, but what it looks like coming in over radio waves was quite another challenge. We first decoded the simpler indoor temperature/humidity sensor (making sure we used the proper CRC checksum to guarantee correct reception – 433MHz is noisy and there are lot of other things on the frequency band such as your car key, garage door openers, etc.). We used as a starting step a variety of tools on our Raspberry Pi (RF_Hacker and rtl_433 for two) and started to match the signals coming from the SDR to the digital bits. The coding uses something called Manchester encoding which makes it difficult to read the signal data with just the eye. We wrote software to decode it, and after a lot of effort, we were reading the indoor sensor and the WeatherRack2 correctly from the Raspberry Pi using our software-defined radio.

We had to look at the timing down on the order of about 200 usec. That is 200 millionths of a second! Using a Saleae Logic Analyzer, we figured out what parts were what and then wrote and modified software to get the data out of our sensors using an Arduino (which is a tiny computer compared to the Raspberry Pi!). Whew! One very odd thing we found out this way was that each indoor temperature/humidity message sent each message three times, with no gaps (see picture) and our WeatherRack2 sent each message twice with a gap. Explained why we would pick up multiple transmissions with the Raspberry Pi SDR!

Conclusion

In Figure 3-19, you can see the complete SkyWeather2 station out in the wild and you can see the WeatherRack2 433MHz sensors mounted outside.

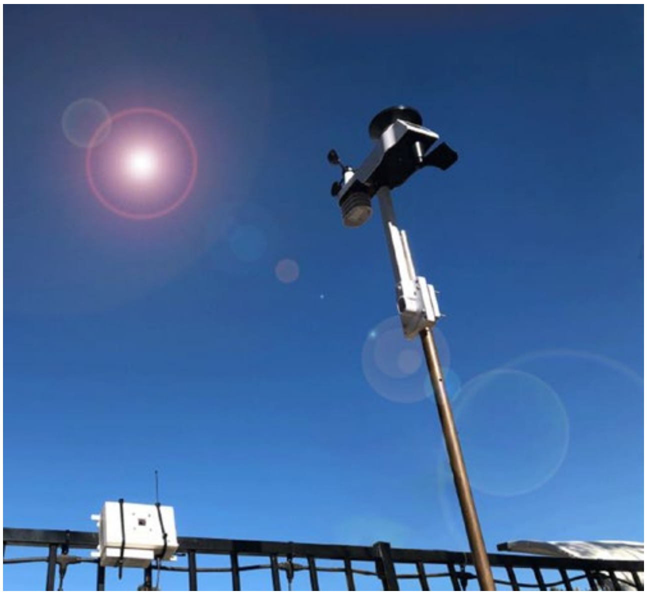


Figure 3-19. *SkyWeather2 Deployed*

It is amazing what you can do with hardware such as the Raspberry Pi and off-the-shelf sensors to sense the environment. With a CPLPFC rating of 5.7, SkyWeather2 is probably not a commercially deployable system in the IoT, but it is a great base unit to experiment with and get familiar with hardware and software for the IoT.

Here are some additional ideas for projects based on SkyWeather2:

- Replacing the WiFi with a GSM data connection – This would use the Cellular Data Network for communications.
- Making a custom Facebook posting with your weather – This would use the Facebook API to make automated postings, much like Twitter.
- Adding a GPS receiver and storing that data – You now have a mobile weather station! When it gets back to WiFi, all the stored data will be available. You could tweet your location and the local conditions on the fly with a GSM connection.
- Adding additional air quality sensors, UV sensors, and dust sensors – You have a lot of input/output pins and I2C addressing space that you can fill with more interesting sensors.

The main power expense in SkyWeather2 is the Raspberry Pi. By replacing the Raspberry Pi with a small Arduino (and dramatically reducing the functionality – Arduinos aren't anywhere near as powerful as the combination of the Raspberry Pi and Linux), you could improve your CPLPFC rating substantially. Want to see how to use an Arduino with the SkyWeather2 WeatherRack2? Check out this article on `www.switchdoc.com`:

`www.switchdoc.com/2020/11/tutorial-arduino-433mhz-weatherrack2/`

Next? We move on to using the Raspberry Pi to detect iBeacons and figure out where you are when you carry the RaspberryPi around with you. With BeaconAir, we will modify the lights using Philips Hue lights to light your way.

CHAPTER 4

Changing Your Environment with IoT and iBeacons

Chapter Goal: Gather Location Data on Your IoT Device: Locate Where You Are and Deliver Location-Dependent Data and Conduct Actions on Where You Are

Topics Covered in This Chapter:

- iBeacons and how can you use them
- Detecting and reading iBeacons from the Raspberry Pi
- Finding your location – Trilateralization in a fuzzy world
- Displaying your location on a control panel
- Turning lights off and on by your location

In Chapter 3, you saw a flexible, Raspberry Pi–based Weather Station to deliver data to the IoT (in this case, to WeatherSTEM.com and WeatherUnderGround).

In this chapter, we take our new IoT device, BeaconAir, to another level. Now we are collecting less information (and even sending less to the IoT), but we are taking data on the environment (where you are) and using it to change the environment (the amount of lighting where you are). This takes us to quite another level of interaction with the IoT.

The BeaconAir Project

BeaconAir is a portable Raspberry Pi–based project that reads the "advertising" packets emitted by iBeacons, roughly calculates your position, and then turns on lights that are close to you. The Pi then calculates the brightness based on just how close you are. The idea is that you can walk around your house with your Pi and the lights will follow you.

In other words, I am using iBeacons to figure out where my Portable Pi is physically located (in a pouch on my hip as I walk around the house), and then I control various devices with the Pi.

The unique aspect of BeaconAir vs. the many other extant Pi-based iBeacon projects is that I am not programming the Raspberry Pi to be an iBeacon; I am doing the opposite. I am using the Pi to read other iBeacons. I am using specialized iBeacons in this project, but you could also build your own iBeacons out of Raspberry Pis and then read them via Bluetooth with this project.

This project is based around a portable Raspberry Pi Model 4B which includes both an on-board Bluetooth and WiFi interface. The completed BeaconAir Portable Pi project is shown in Figure 4-1.

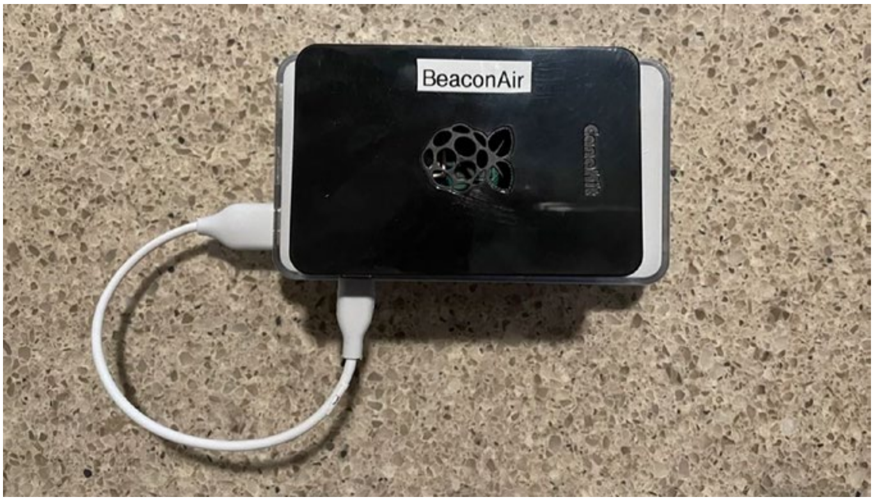


Figure 4-1. *BeaconAir Portable Pi*

IoT Characterization of This Project

As I discussed in Chapter 1, the first thing to do to understand an IoT project is to look at our six different aspects of IoT. BeaconAir is a much simpler project than IOTWeatherPi. Table 4-1 shows our six components.

Table 4-1. *Components of the BeaconAir Project*

BeaconAir Characterization (CPLPFC)		
Aspect	**Rating**	**Comments**
Communications	9	WiFi connection to the Internet – can do ad hoc mesh-type communication and Bluetooth
Processor power	9	Raspberry Pi 4B/4GB RAM
Local storage	8	16GB of SD card
Power consumption	1	~800mA consumption – not reasonable for small batteries
Functionality	8	Full Linux-based system. MySQL, etc.
Cost	1	Expensive for many applications. Board is ~$35+

Ratings are from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications.

This gives us a CPLPFC rating of 5.8, a little better than SkyWeather2 (5.7). Great for learning, not so good for deployment for most applications.

No doubt about it, the Raspberry Pi is a very flexible and powerful IoT platform. However, the power consumption, cost, and physical size of the device make it more suitable for prototyping or for stand-alone highly functional IoT units.

How Does This Device Hook Up to the IoT?

With BeaconAir, like the previous chapter, we have a lot of options. We can hook up to the Internet using the WiFi connector. We can use the Bluetooth to hook up to local devices, and we can also use the WiFi in ad hoc mode to make local connections. In this chapter, we will be using the WiFi interface to talk to the wider world of IoT devices.

Hardware List

Here is a list of the hardware you'll need in order to build this project:

- Raspberry Pi Model 4B Kit – amazon.com – `https://amzn.to/38C2JZO`
- USB Type A to USB C Cable – amazon.com – `https://amzn.to/3ptxdUM`
- Adafruit USB Battery Pack for Raspberry Pi (10000mAh) – 2 x 5V @ 2A – adafruit.com
- BlueCharm Beacons – amazon.com – `https://amzn.to/3o5NMWB`

iBeacons

iBeacon is the Apple trademark for a low-powered Bluetooth device. An iBeacon is a low-powered, low-cost transmitter that can notify nearby devices of their presence and a rough approximation of range. There are a number of manufacturers that are producing these devices, and most smartphones (and Raspberry Pis!) can be made to act as an iBeacon. iBeacons use Bluetooth Low Energy (BLE), also known as Bluetooth Smart. iBeacons can also be received on Bluetooth 4.0 devices that support dual mode (such as the IOGear dongle specified earlier.

Applications of iBeacons include location-aware advertising, social media check-ins, or notifications sent to your smartphone or Pi. An iBeacon transmits an advertising packet containing a UDID (Unique Device Identifier) that identifies the manufacturer and then a major and minor number that can be used to identify the specific device. It also sends out an RSSI (Relative Signal Strength Indicator) that can be used to approximate the distance to the iBeacon device.

It is important to note that almost all the logic behind an iBeacon deployment is through the supporting application on the device (a Raspberry Pi in our case). The only role of the iBeacon is to advertise to the device of its own existence at a physical location. In some cases, you can connect to an individual device through the iBeacon's GATT (General ATTribute profile) although some iBeacons have a proprietary interface that prohibits this.

This requirement of having the application device (like a smartphone or Raspberry Pi) read and take actions on the position of the iBeacons remains a roadblock to widespread adoption of iBeacons in the marketplace. Doing it any other way (say, for the iBeacons to detect your phone) is a big privacy concern, and so things are likely to stay this way for the foreseeable future. See Chapter 7 for a number of reasons that this is a good thing.

The BlueCharm iBeacons we used are shown in Figure 4-2.

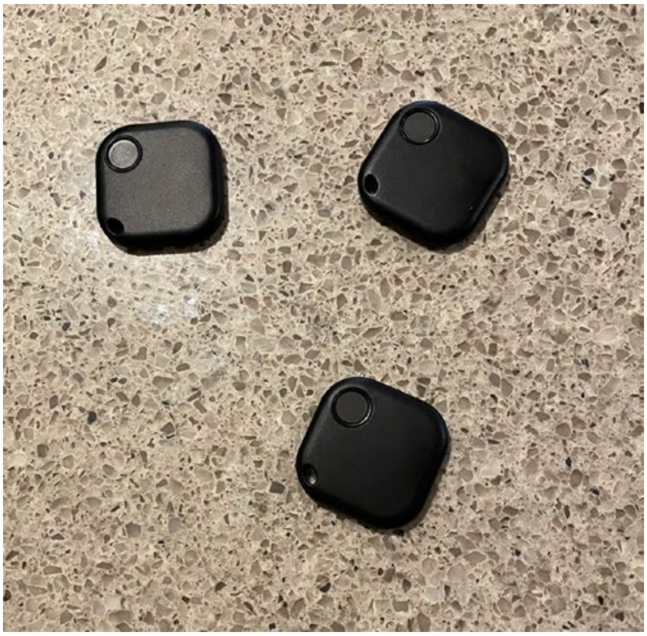


Figure 4-2. *BlueCharm iBeacons*

I used one type of iBeacons: BlueCharm. I also tested FeasyCom and an inexpensive generic iBeacon from eBay. BlueCharm worked perfectly out of the box, and their configuration app worked well. FeasyCom worked reasonably, but at least the one I ordered was dead and the others all had identical UDIDs which would be a hassle for the reader. Of course, you can always roll your own iBeacon using a Raspberry Pi [`www.wadewegner.com/2014/05/create-an-ibeacon-transmitter-with-the-raspberry-pi/`] or use your phone to emulate an iBeacon with one of the many apps in the app stores. The inside of the FeasyCom iBeacon is shown in Figure 4-3.

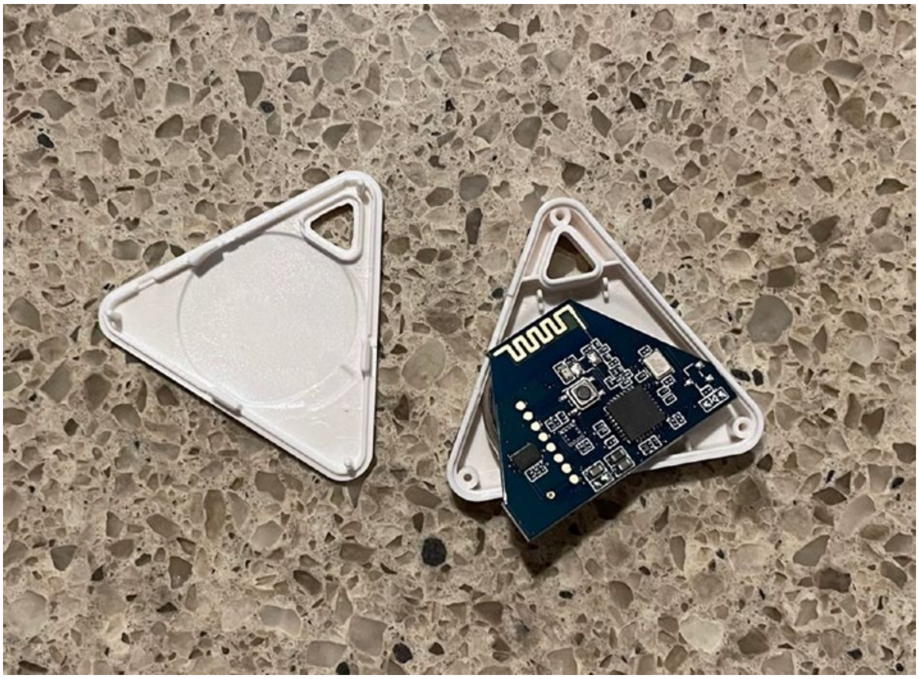
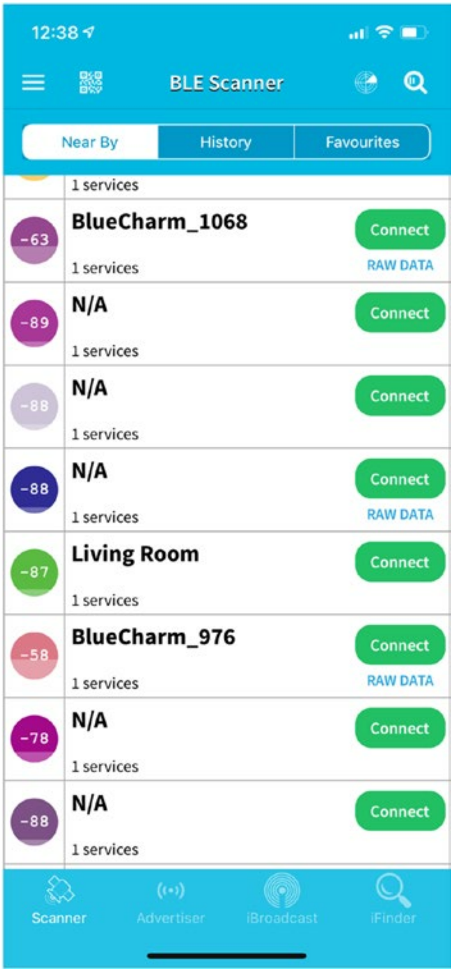


Figure 4-3. *Inside of a FeasyCom iBeacon*

There are four major pieces of software in BeaconAir: the Python Bluetooth iBeacon Scanner, the Philips Hue interface, the main BeaconAir software, and the web page server software.

Python3 Bluetooth iBeacon Scanner

Technically, this was the most difficult part of the BeaconAir system. The software available to do this was not very reliable and did not produce the kind of information I was interested in. Figure 4-4 shows the iBeacons near to my lab bench using the BLE Scanner 4.0 App on my iPhone from Bluepixel Technologies LLP.



1 services

-63	BlueCharm_1068	1 services
-89	N/A	1 services
-88	N/A	1 services
-88	N/A	1 services
-87	Living Room	1 services
-58	BlueCharm_976	1 services
-78	N/A	1 services
-88	N/A	1 services

Figure 4-4. *BLE Scanner App Showing iBeacons*

Note that we are picking up on two BlueCharm iBeacons (we identified these earlier). Interestingly enough, we are also picking up an Apple TV located about 40 feet away. I was not aware that the Apple TV was broadcasting an iBeacon packet, but on checking it is used for an undocumented way of setting up the Apple TV from your iPhone. The numbers don't make a lot of sense in the iBeacon advertising packet, but that is a problem for another day.

The biggest issue with this project was to be able to reliably read iBeacon data from the internal Bluetooth interface on the Raspberry Pi 4B. A number of the methods out there on the Web were less than satisfactory (doing hcidump scans) and often ended up hanging the Bluetooth on the Pi, requiring a reboot. Once I went to using my software library, I have zero hang-ups and the software runs for days.

iBeacons use Bluetooth Low Energy (BLE) protocols to communicate, which is a relatively new type of Bluetooth and has spotty support. Finally I stumbled upon a program using blueZ (Linux Bluetooth library) native calls, which with a lot of modifications, bug fixes, and cutting of code I didn't need, I had a iBeacon Scanner that worked every time. I have posted my working version on the SwitchDoc Labs github (`github.com/switchdoclabs/iBeacon-Scanner-`) so you can download it and test your setup.

The blescan.py program is easy to test and use but requires some setup on the Raspberry Pi. See the following section on installing all the required software on the Raspberry Pi.

Here is the output from the Bluetooth scanner program running at SwitchDoc Labs. We have a lot of iBeacons sitting around.

```
pi@SwitchDocLabs:~/SDL_Pi_iBeaconScanner $ sudo python3
testblescan.py
ble thread started
----------
43:bc:c4:82:59:1d:01:00,1d5982c4bc430e0201060aff
4c001005,17692,42096,-26,-54
43:bc:c4:82:59:1d:01:04,0104011d59,33476,48195,0,-54
dc:56:e7:49:39:f3:00:00,e756dc1102011a020a0c0aff
4c001005,2324,22049,24,-87
dc:56:e7:49:39:f3:00:04,010400f339,18919,22236,0,-88
7b:03:5f:01:4b:0c:01:00,037b1202011a020a0c0bff4c001006
0d,7585,3364,24,-26
```

```
7b:03:5f:01:4b:0c:01:04,0104010c4b,351,891,0,-26
73:df:a1:ac:0e:34:01:00,0613ff4c000c0e00e070ca209cd1f40e,62240,
22431,40,-49
73:df:a1:ac:0e:34:01:04,010401340e,44193,57203,0,-51
c8:69:cd:63:df:0f:00:00,020a0c11ff4c000f08c00a99b7fe0044,2832,
513,4,-86
----------
f8:04:2e:bf:d2:05:00:03,80a0f8042ebfd205fa042ebfd2040100,0,0,
0,-61
0a:62:b6:b2:a1:e9:01:03,fd0925f23cb3236420d2fe9f7dd48cad,38805,
14355,-81,-50
4e:a5:00:3e:a2:64:01:00,64a23e00a54e0e0201060aff4c001005,17692,
42096,-26,-55
4e:a5:00:3e:a2:64:01:04,01040164a2,15872,42318,0,-56
43:bc:c4:82:59:1d:01:00,1d5982c4bc430e0201060aff4c001005,17692,
42096,-26,-51
43:bc:c4:82:59:1d:01:04,0104011d59,33476,48195,0,-51
dc:56:e7:49:39:f3:00:00,e756dc1102011a020a0c0aff4c001005,2324,
22049,24,-89
dc:56:e7:49:39:f3:00:04,010400f339,18919,22236,0,-88
0b:a7:c3:bb:5c:3a:01:03,5cbbc3a70b0f02011a0bff4c00090603,6592,
43009,9,-75
        c8:69:cd:63:df:0f:00:00,020a0c11ff4c000f08c00a99b7
fe0044,2832,513,4,-84
```

We are finding a large number of iBeacons in the lab area. Before you can do this, you need to verify you have the latest version of blueZ, the Bluetooth stack for the Raspberry Pi (instructions in the following text).

Philips Hue Lighting System

The Philips Hue lighting system is a Zigbee-based wireless way of controlling intensity, color combinations, and on/off from a Philips Hub based on your local network. The standard apps for Android and iOS are very powerful, but for us Raspberry Pi people, the best part is that Philips has released the API for the Hub for the DIY crowd. It's somewhat expensive ($60/bulb) but robust and very easy to use and to hack. All commands are sent via wireless or Ethernet to the Philips Hue Hub, and the Hub communicates to the individual devices.

What Is Zigbee?

Zigbee is a joint specification for a suite of high-level communication protocols used to create personal area networks built from small, low-power digital radios. In this regard, it is very similar to low-power Bluetooth. The transmission distance is limited to about 10–100 meters, depending on the power output and the transmission environment. The Philips Hue Zigbee devices work well within a house, but sometimes you will see lights jump on and off the network in what seems to be humidity-related events. Given the frequencies and low-power characteristics of Zigbee, this could certainly be the case.

The Zigbee technology is meant to be simpler and less expensive than Bluetooth and WiFi, not only in dollar cost but also processor overhead to deal with the communications channel. Just imagine what the processor has to do to interpret an incoming TCP/IP packet and get to the user data. Zigbee can be used without the really “heavy” protocol stack to communicate with local devices through a mesh network to reach more distant Zigbees (see Chapter 6) and then pass to a “beefier” processor, such as a Raspberry Pi, to send information into the IoT on the Internet.

Zigbee is typically used in low-data rate IoT applications that require long battery life (a very important feature!) and secure networking. Zigbee networks are secured by 128-bit symmetric encryption keys. See Chapter 7 for a discussion of encryption keys. Zigbee has a defined rate of 250 kbit/s, which is not very fast for web access, but you can send a lot of data with that speed.

The Zigbee name refers to the waggle dance of honey bees after their return to the beehive.

Philips Hue Hub

The Philips Hue Hub communicates via authenticated JSON packets. There are a number of Python packages designed for communication with the Philips Hue Hub. We chose to use one written by Studio Imaginaire (`studioimaginaire.com/en`) called phue. It is a group of smart French people that did a great job producing the phue library. Considering the BeaconAir logo was designed in France, it seemed appropriate to use this library. You can download it at `github.com/studioimaginaire/phue`. See installation instructions in the following text.

Our test rooms for BeaconAir has ten Philips Hue A19 standard bulbs, three Philips Hue BR30 down wash lights, and two Philips Friends of Hue Bloom lights. It was expensive but worth it (the A19 bulbs are $60 apiece, BR30 bulbs $60 apiece, and the Blooms are $80 apiece). These prices should decrease in the future.

BeaconAir Hardware, Software, and Configuration

To work with the BeaconAir, you need to know about the hardware and software. You also need to know about how the software is configured. The following subsections cover each of these topics.

BeaconAir Hardware Description

The BeaconAir hardware is pretty straightforward. We use a stock Raspberry Model B+ with a Wi-Pi WiFi USB dongle and an IOGear Bluetooth 4.0 USB dongle. Everything else is done in software. Figure 4-5 shows the system, as well as how iBeacons allow us to find the approximate physical position of our BeaconAir Portable Pi.

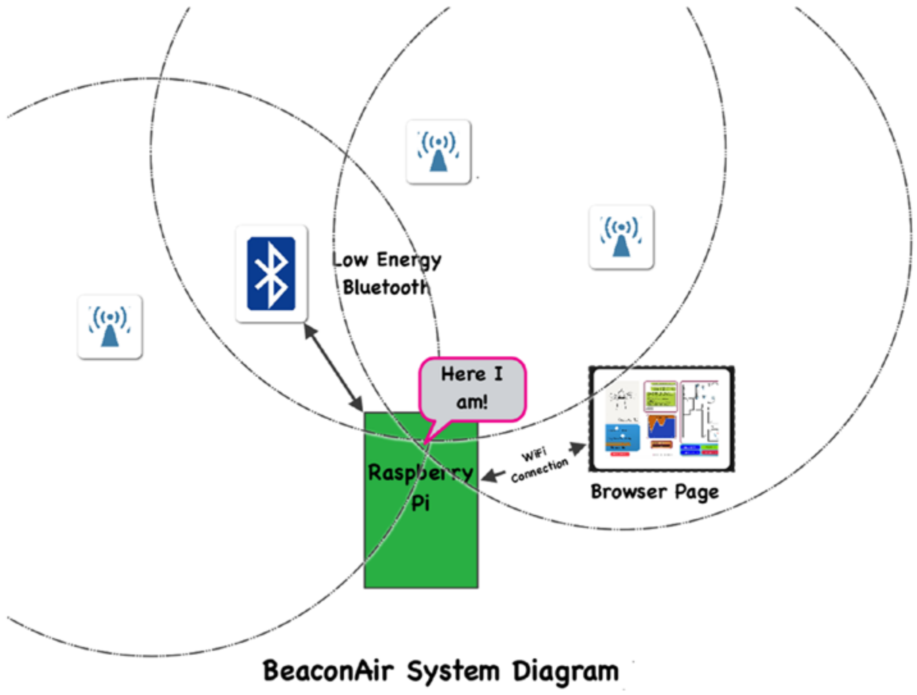


Figure 4-5. *BeaconAir System Diagram*

BeaconAir Software Description

The BeaconAir software consists of four major pieces. I have described the iBeacon Scanner and the Philips Hue Python library phue earlier. The two major pieces remaining are the main program loop and the web page.

The BeaconAir software block diagram is shown in Figure 4-6.

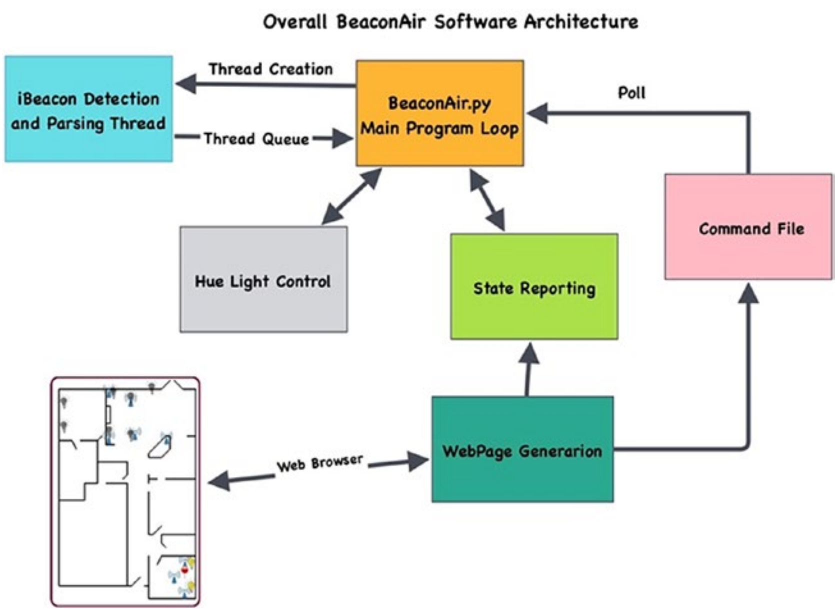

Overall BeaconAir Software Architecture
iBeacon Detection and Parsing Thread
Thread Creation
Thread Queue
BeaconAir.py Main Program Loop
Poll
Command File
Hue Light Control
State Reporting
Web Browser
WebPage Generarion

Figure 4-6. *BeaconAir Software Block Diagram*

The main software runs in a loop, with a sleep of 0.25 seconds at the end. It checks for two sources of work. First, it checks a queue that is connected to the iBeacon Scanning software running in a background thread. If the queue is empty, we have no new iBeacon reports so we go down and check to see if there are commands waiting from the RasPiConnect control panel.

```
if (queueBLE.empty() == False):
        result = queueBLE.get(False)

   # process commands from RasPiConnect
        processCommand()
```

If the queue has iBeacon results to deliver, we then go through the main loop and process the iBeacon information, set various informational parameters, build the new web page to deliver to RasPiConnect, and control the lights.

I have removed the debugging information to make things clearer. All calculations are done in meters and converted to pixels for display.

The first thing we do is process the incoming iBeacon list to fill our beacon arrays. We then clear out old values.

```
result = queueBLE.get(False)
utils.processiBeaconList(result,currentiBeaconRSSI, currentiBea
conTimeStamp,rollingiBeaconRSSI)
utils.clearOldValues(10,currentiBeaconRSSI, currentiBeaconTimeS
tamp,rollingiBeaconRSSI)
```

Next we calculate the current BeaconAir physical position but only if we have greater than three beacons.

```
# update position
if (utils.haveThreeGoodBeacons(rollingiBeaconRSSI) >= 3):
        oldbeacons = beacons
        beacons = utils.get3ClosestBeacons(rollingiBeaconRSSI)
        if (cmp(oldbeacons, beacons) != 0):
bubblelog.writeToBubbleLog("closebeacons:%i,%i,%i" %
(beacons[0], beacons[1], beacons[2]))
        myPosition = utils.XgetXYFrom3Beacons(beacons[0],beacon
        s[1],beacons[2], rollingiBeaconRSSI)
```

I now have the latest calculated position. Next I calculate the jitter in the position. A big value of jitter says either you are moving or there are significant amounts of noise in the iBeacon reports or both.

```
# calculate jitter in position
jitter = (((lastPosition[0] - myPosition[0])/lastPosition[0]) +
((lastPosition[1] - myPosition[1])/lastPosition[1]))/2.0
jitter = jitter * 100.0   # to get to percent
lastPosition = myPosition
```

Now I write out the jitter for RasPiConnect to read and send to the jitter graph on the control panel.

```
f = open("/home/pi/BeaconAir/state/distancejitter.txt", "w")

f.write(str(jitter))
f.close()
```

Next I calculate the distance from my position to all the lights and then turn the light on, change brightness, or turn it off depending on the distance.

```
lights.checkForLightTrigger(myPosition, LIGHT_DISTANCE_
SENSITIVITY, LIGHT_BRIGHTNESS_SENSITIVITY, currentLightState)
```

Next the web page is built for display on RasPiConnect.

```
# build webpage
webmap.buildWebMapToFile(myPosition, rollingiBeaconRSSI,
currentLightState, DISPLAY_BEACON_ON, DISPLAY_LIGHTS_ON)
```

Finally, I update the current beacon count and build the graph for display on RasPiConnect.

```
# build beacon count graph
iBeaconChart.iBeacondetect(rollingiBeaconRSSI)
else:
# lost position
myPosition = [-myPosition[0], -myPosition[1]]
```

That is the entire main program for BeaconAir.

Here is the full listing of the main program of BeaconAir:

```
#!/usr/bin/python3

#!/usr/bin/python3

# BeaconAir - Reads iBeacons and controls HUE lights
# SwitchDoc Labs December 2020
#
#
from __future__ import division
from __future__ import print_function
from past.builtins import cmp
from future import standard_library
standard_library.install_aliases()
from builtins import str
from past.utils import old_div
import sys
import time
import utils

sys.path.append('./ble')
sys.path.append('./config')

# if conflocal.py is not found, import default conf.py

# Check for user imports
try:
        import conflocal as conf
except ImportError:
        import conf
import bleThread

import lights
import webmap
import bubblelog
import iBeaconChart

from threading import Thread
from queue import Queue

# State Variables

currentiBeaconRSSI=[]
rollingiBeaconRSSI=[]
currentiBeaconTimeStamp=[]

# Light State Variables

currentLightState= []

LIGHT_BRIGHTNESS_SENSITIVITY = 2.0
LIGHT_DISTANCE_SENSITIVITY = 2.0
BEACON_ON = True
DISPLAY_BEACON_ON = True
DISPLAY_LIGHTS_ON = True
# init state variables
for beacon in conf.BeaconList:
    currentiBeaconRSSI.append(0)
    rollingiBeaconRSSI.append(0)
    currentiBeaconTimeStamp.append(time.time())

lights.initializeHue('192.168.1.23')

# init light state variables
for light in conf.LightList:
    print(type(light[7]))
    print(light[7])
    currentLightState.append(lights.getInitialLightState
    (light[7]))

# recieve commands from command file

def completeCommand():

        f = open("/home/pi/SDL_Pi_BeaconAir/state/
BeaconAirCommand.txt", "w")
        f.write("DONE")
        f.close()

def processCommand():
        global LIGHT_BRIGHTNESS_SENSITIVITY
        global LIGHT_DISTANCE_SENSITIVITY
        global BEACON_ON
        global DISPLAY_BEACON_ON
        global DISPLAY_LIGHTS_ON
        global currentLightState

        f = open("/home/pi/SDL_Pi_BeaconAir/state/
        BeaconAirCommand.txt", "r")
        command = f.read()
        f.close()

        if (command == "") or (command == "DONE"):
                # Nothing to do
                return False

        # Check for our commands

        print("Processing Command: ", command)
        if (command == "BEACONON"):
                BEACON_ON = True
                completeCommand()
                return True

        if (command == "BEACONOFF"):
                BEACON_ON = False
                completeCommand()
                return True

        if (command == "ALLLIGHTSON"):
                lights.allLights(True, currentLightState )
                completeCommand()
                return True

        if (command == "BEACONON"):
                BEACON_ON = True
                completeCommand()
                return True

        if (command == "BEACONOFF"):
                BEACON_ON = False
                completeCommand()
                return True

        if (command == "DISPLAYBEACONON"):
                DISPLAY_BEACON_ON = True
                completeCommand()
                return True

        if (command == "DISPLAYBEACONOFF"):
                DISPLAY_BEACON_ON = False
                completeCommand()
                return True
```

```
if (command == "DISPLAYLIGHTSON"):
        DISPLAY_LIGHTS_ON = True
        completeCommand()
        return True

if (command == "DISPLAYLIGHTSOFF"):
        DISPLAY_LIGHTS_ON = False
        completeCommand()
        return True

if (command == "UPDATESENSITIVITIES"):

    try:
            f = open("/home/pi/SDL_Pi_BeaconAir/state/
            distanceSensitivity.txt", "r")
            commandresponse = f.read()
            LIGHT_DISTANCE_SENSITIVITY =
            float(commandresponse)
            f.close()
    except:
            LIGHT_DISTANCE_SENSITIVITY = 2.0

    try:
            f = open("/home/pi/SDL_Pi_BeaconAir/state/
            brightnessSensitivity.txt", "r")
            commandresponse = f.read()
            f.close()
            LIGHT_BRIGHTNESS_SENSITIVITY =
            float(commandresponse)
    except:
            LIGHT_BRIGHTNESS_SENSITIVITY = 2.0
            print("LIGHT_DISTANCE_SENSITIVITY, LIGHT_
            BRIGHTNESS_SENSITIVITY= ", LIGHT_DISTANCE_
            SENSITIVITY, LIGHT_BRIGHTNESS_SENSITIVITY)
            completeCommand()
            return True

        completeCommand()
        return True

# build configuration Table

# set up BLE thread
# set up a communication queue

queueBLE = Queue()
BLEThread = Thread(target=bleThread.bleDetect, args=
(__name__,10,queueBLE,))
BLEThread.daemon = True
BLEThread.start()

bubblelog.writeToBubbleLog("BeaconAir Started")

# the main loop of BeaconAir
myPosition = [0,0]
lastPosition = [1,1]
beacons = []
while True:
    if (BEACON_ON == True):
        # check for iBeacon Updates
        print("Queue Length =", queueBLE.qsize())
        if (queueBLE.empty() == False):
            result = queueBLE.get(False)
            print("------")
```

```
utils.processiBeaconList(result,currentiBeaconRSSI,
currentiBeaconTimeStamp,rollingiBeaconRSSI)
utils.clearOldValues(10,currentiBeaconRSSI, current
iBeaconTimeStamp,rollingiBeaconRSSI)
for beacon in conf.BeaconList:
    utils.printBeaconDistance(beacon,
    currentiBeaconRSSI, currentiBeaconTimeStamp,rol
    lingiBeaconRSSI)
# update position
if (utils.haveThreeGoodBeacons(rollingiBeaconRSSI)
>= 3):
    oldbeacons = beacons
    beacons = utils.get3ClosestBeacons(rollingi
    BeaconRSSI)
    print("beacons=", beacons)
    if (cmp(oldbeacons, beacons) != 0):
        bubblelog.writeToBubbleLog("closebeacon
        s:%i,%i,%i" % (beacons[0], beacons[1],
        beacons[2]))

    # setup for Kludge
    #rollingiBeaconRSSI[7] = rollingiBeaconRSSI[6]

    myPosition = utils.getXYFrom3Beacons(beacons[0]
    ,beacons[1],beacons[2], rollingiBeaconRSSI)
    print("myPosition1 = %3.2f,%3.2f" %
    (myPosition[0], myPosition[1]))
    #bubblelog.writeToBubbleLog("position
    updated:%3.2f,%3.2f" % (myPosition[0],
    myPosition[1]))

    # calculate jitter in position
```

```
jitter = ((old_div((lastPosition[0] - my
Position[0]),lastPosition[0])) + (old_
div((lastPosition[1] - myPosition[1]),lastPosit
ion[1])))/2.0
jitter = jitter * 100.0   # to get to percent
lastPosition = myPosition
print("jitter=", jitter)

f = open("/home/pi/SDL_Pi_BeaconAir/state/
distancejitter.txt", "w")

f.write(str(jitter))
f.close()

for light in conf.LightList:
    lightdistance = utils.distanceBetweenTwoPoi
    nts([light[2],light[3]], myPosition)
    print("distance to light %i : %3.2f" %
    (light[0], lightdistance))
print("LIGHT_DISTANCE_SENSITIVITY, LIGHT_
BRIGHTNESS_SENSITIVITY= ", LIGHT_DISTANCE_
SENSITIVITY, LIGHT_BRIGHTNESS_SENSITIVITY)
lights.checkForLightTrigger(myPosition,
LIGHT_DISTANCE_SENSITIVITY, LIGHT_BRIGHTNESS_
SENSITIVITY, currentLightState)
print("DISPLAY_BEACON_ON, DISPLAY_LIGHTS_ON",
DISPLAY_BEACON_ON, DISPLAY_LIGHTS_ON)
# build webpage
webmap.buildWebMapToFile(myPosition,
rollingiBeaconRSSI, currentLightState, DISPLAY_
BEACON_ON, DISPLAY_LIGHTS_ON)

# build beacon count graph
iBeaconChart.iBeacondetect(rollingiBeaconRSSI)
```

```
            else:
                # lost position
                myPosition = [-myPosition[0], -myPosition[1]]

        #print currentiBeaconRSSI
        #print currentiBeaconTimeStamp

    # end of BEACON_ON - always process commands
    else:
        if (queueBLE.empty() == False):
            result = queueBLE.get(False)
        print("------")
        print("Beacon Disabled")
        # process commands from command file

        processCommand()

    time.sleep(0.25)
```

One very interesting part of BeaconAir is building the HTML-based map showing position, beacons, and lights.

As I started this project, I felt that building this live map was going to be the biggest problem. I looked at building a live map with Matplotlib on the Pi, but it was computationally expensive and complicated. Then I looked at HTML drawing solutions, and I found that it was almost trivial to do so. Figure 4-7 shows the completed HTML map.

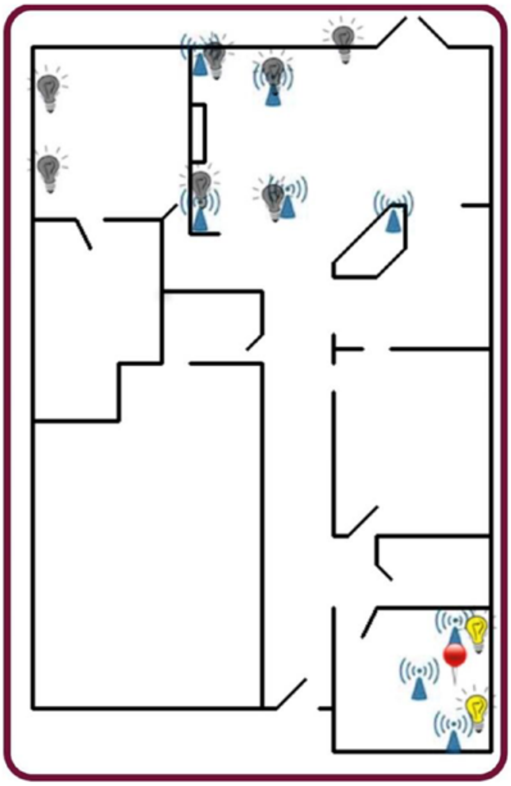


Figure 4-7. *HTML House Map*

To make this HTML map work, follow these steps:

1. Build a JPEG with the plan of your office or house. I took a picture of the house plans and then used GIMP [www.gimp.org] to draw the walls on the JPEG and then remove the JPEG layer. Worked like a champ. Then I had to measure a wall in meters and use GIMP to measure the same wall in pixels, and I had my meters to pixels constant (0.0375 m/px in my case). I put the 0,0 point at the top of the JPEG and then x is positive going to the right and y is positive down the left side.

2. In the configuration file, figure out the positions x,y for each of the lights and the beacon and put in the BeaconAir configuration file, replacing the default Beacon Values.

When you run the BeaconAir software, you will find the web page under the state directory called "beacon.html". The resulting HTML code looks like the following (this is for a 3 beacon test):

```
<html>
  <head>
    <script>

    </script>
  </head>
  <body>
  <div>
  <img src='mainplanfull.png' style='position: relative;
  top: 0; left: 0;'/>

  </div>
<img src='iBeacon.png' style='position: absolute; top: 490px;
left: 299px;'/>
<img src='iBeacon.png' style='position: absolute; top: 19px;
left: 122px;'/>
<img src='iBeacon.png' style='position: absolute; top: 19px;
left: 122px;'/>
<img src='OnLightBulb.png' style='position: absolute; top:
-17px; left: -15px;'/
>
<img src='red-pin.png' style='position: absolute; top: 26px;
left: 11px;'/>

  </body>
</html>
```

This works like a champ. I made my icons with transparent backgrounds (again using GIMP).

BeaconAir Configuration File

The BeaconAir configuration file needs to be set up before the system can be used. It is located under config in the main directory. The three major parts of the configuration file are as follows:

```
Scaling Factors
Beacon Configuration
Light Configuration
```

The following declarations show two of the scaling factors used in the project:

```
def pixelConv(pixels):
        return pixels * 0.0375    # in meters

def meterToPixel(meters):
        return int(meters / 0.0375)    # in pixels
```

These two constants set how close you should be to the light (in meters) before you turn it on. Try the following numbers before changing them:

```
LIGHT_BRIGHTNESS_SENSITIVITY = 2.0
LIGHT_DISTANCE_SENSITIVITY = 2.0
```

Our beacon configuration is accomplished as follows:

```
# configuration for house
# list of iBeacons with x,y coordinates.  Top left corner of
  image is 0,0
# list of lists
# Beacon format:
```

```
#     BeaconNumber, LocalName, x, y, MAC, Major, Minor,
      Measured Power (from spec), x in px, y in px
# BeaconNumber is incremental from 0 up.  Don't skip a number

BeaconList=[]
BeaconCount = 0

Beacon = [BeaconCount,"Estimote #0 Beacon", pixelConv(314),
pixelConv(507),  "43:ed:21:ac:77:ed", 64507, 5414, -64, 314,
507]
BeaconList.append(Beacon)
BeaconCount += 1
```

Finally, there is our light configuration:

```
#list of lights
#Light Format
#     LightNumber, LocalName, x, y, pixel x, pixel y, light on/
      off (1/0), huelightnumber

LightList=[]
LightCount = 0
Light = [LightCount, "Lab Left", pixelConv(330),
pixelConv(435),330, 435,0, 2]
LightList.append(Light)
LightCount += 1
```

The iBeacon MAC number is printed at the back of the BlueCharm iBeacons. You can also read it from a number of different scanners or their excellent configuration app, KBeacon. Other iBeacon MAC addresses can be found by using testblescan.py and looking for the UDID of a device and then entering the MAC address (the first entry on the scanned beacon) into the configuration file. Putting ":" between each two digits is optional and will be filtered out.

You can get the Hue light number from the Philips app under Light overview, or you can write a short program (look at the phue examples) to get the dictionary from the Philips Hue Hub. I provided a short Hue program to give the light numbers for all of the Hue lights in the house. It is called listHue.py. If I run this at my house, here is what I get:

```
pi@SwitchDocLabs:~/SDL_Pi_BeaconAir $ sudo python3 listHue.py
id=6 name=Living room front
id=7 name=Living room left
id=8 name=Middle living room
id=23 name=East kitchen counter
id=24 name=West kitchen counter
id=25 name=Lab desk
id=26 name=Right bedroom
id=27 name=Middle Kitchen Counter
id=28 name=One kitchen
id=29 name=Kitchen three
id=30 name=Kitchen two
id=33 name=Kitchen five
id=36 name=Kitchen four
id=38 name=Living room top right
id=39 name=Living room top left
id=40 name=Bar  Back  Right
id=41 name=Bar Front Left
id=42 name=Bar Back Left
id=43 name=Bat Front Middle
id=44 name=Bar Front Left
id=45 name=Dining room #1
id=46 name=Dining room #2
id=47 name=Dining room #3
id=48 name=Dining room #4
id=49 name=Dining room #5
id=50 name=Dining room #6
```

```
id=51 name=Dining room #7
id=52 name=Dining room #8
id=53 name=Dining room #9
id=54 name=Theater 1
id=55 name=Theater 4
id=56 name=Theater 3
id=57 name=Theater 2
id=58 name=Left Bedroom
pi@SwitchDocLabs:~/SDL_Pi_BeaconAir $
```

Wow! I have a lot of Hue lights. And more on the way.

iBeacon Software

The iBeacons are problematic. They aren't very accurate, and lots of different environmental factors affect the RSSI (received power). If you have your BeaconAir sensitivities set high, you can sit in one place and watch the lights grow brighter and dimmer as the received signals vary. It's kind of a visual map of the electromagnetic spectrum. Setting your brightness sensitivities lower will increase the sensitivity, while the light range higher clears this up.

There are two functions used by BeaconAir to determine position. First is the calculation of distance from RSSI. We use a smoothing function on the received RSSI values to reduce the jitter, for example:

```
def calculateDistanceWithRSSI(rssi,beaconnumber):

        beacon = conf.BeaconList[beaconnumber];
        txPower = beacon[7]
        ratio_db = txPower - rssi;
        ratio_linear = pow(10, ratio_db / 10);
        r = pow(ratio_linear, .5);
        return r
```

The result from this function is already scaled in meters.

Trilateralization

The second key piece is the calculation of position by using trilateration. Trilateration is the method of determining the position of a point, given the distance to three control points [citation: `en.wikipedia.org/wiki/Trilateration`].

SwitchDoc Note When you hear someone talking about detecting where something is by measuring distance from points, they usually say "triangulation" where they really mean "trilateration." What is the difference? You use triangulation when you know the angle to an object from two different sources. Then you can locate the object on a plane.

In the case of iBeacons, all we know is the distance. We don't know anything about direction. We can use three distances (or iBeacons in this case) and fix the location of the Raspberry Pi on a plane. It occurs to me that we could put a directional Bluetooth antenna on a stepper motor and possibly use that to use triangulation, which means we would only need two iBeacons to find our position, rather than three as in trilateration.

The purpose of the following getXYFrom3Beacons function is to take the information from the three selected iBeacons found (a,b,c) and calculate the XY coordinates of your BeaconAir device:

```
def getXYFrom3Beacons(beaconnumbera, beaconnumberb,
beaconnumberc, rollingRSSIArray):

        beacona = conf.BeaconList[beaconnumbera];
        beaconb = conf.BeaconList[beaconnumberb];
        beaconc = conf.BeaconList[beaconnumberc];
```

```
xa = float(beacona[2])
ya = float(beacona[3])
xb = float(beaconb[2])
yb = float(beaconb[3])
xc = float(beaconc[2])
yc = float(beaconc[3])

ra = float(calculateDistanceWithRSSI(rollingRSSIArray
[beaconnumbera],beaconnumbera ))
rb = float(calculateDistanceWithRSSI(rollingRSSIArray
[beaconnumberb],beaconnumberb ))
rc = float(calculateDistanceWithRSSI(rollingRSSIArray
[beaconnumberc],beaconnumberc ))

S = (pow(xc, 2.) - pow(xb, 2.) + pow(yc, 2.)
- pow(yb, 2.) + pow(rb, 2.) - pow(rc, 2.)) / 2.0
T = (pow(xa, 2.) - pow(xb, 2.) + pow(ya, 2.)
- pow(yb, 2.) + pow(rb, 2.) - pow(ra, 2.)) / 2.0

try:
        y = ((T * (xb - xc)) - (S * (xb - xa))) /
        (((ya - yb) * (xb - xc)) - ((yc - yb) *
        (xb - xa)))
        x = ((y * (ya - yb)) - T) / (xb - xa)

except ZeroDivisionError as detail:
        print 'Handling run-time error:', detail
        return [-1,-1]

point = [x, y]
return point
```

Issuing Commands to LightSwarm

Basically, the idea is that external software sends a command to the BeaconAir software that is running on a different thread in the same system. Remember that the Raspberry Pi Linux-based system is multitasking and you can run many different programs at once.

In Table 4-2, we show the various text commands that can be placed in the text file BeaconAirCommand.txt that will be picked up by BeaconAir.py and executed by the processcommand() function.

Table 4-2. *BeaconAir Command List*

Command	Syntax	Description
Beacon On	BEACONON	Turns on iBeacon sensing
Beacon Off	BEACONOFF	Turns off iBeacon sensing
Turn all Lights on	ALLLIGHTSON	Turns on all lights under BeaconAir control
Turn all Lights off	ALLLIGHTSOFF	Turns off all lights under BeaconAir control
Display Beacons on	DISPLAYBEACONON	Turns on display of iBeacons on house map html file
Display Beacons off	DISPLAYBEACONOFF	Turns off display of iBeacons on house map html file
Display Lights on	DISPLAYLIGHTSON	Turns on display of lights on house map html file
Display Lights Off	DISPLAYLIGHTSOFF	Turns off display of lights on house map html file

To use these commands is simple. Under the SDL_Pi_BeaconAir/state subdirectory, edit the BeaconAirCommand.txt and enter one of the preceding commands. Shortly the main loop in BeaconAir.py will pick up the command and execute the command.

Installing BlueZ and Phue on the Raspberry Pi

Now it's time to check the installation of blueZ and install phue on the Raspberry Pi.

BlueZ

BlueZ is included in the buster and newer versions of Raspbian (now called the Raspberry Pi OS). Verify this by typing in a terminal window:

```
bluetoothctl -v
```

You should get the following back:

```
bluetoothctl: 5.50
```

Now you have blueZ verified and running on your Raspberry Pi.

Now you can look for the Bluetooth device using hciconfig:

```
pi@SwitchDocLabs:~/SDL_Pi_iBeaconScanner $ hciconfig
hci0:   Type: Primary  Bus: UART
        BD Address: DC:A6:32:02:EE:50  ACL MTU: 1021:8
        SCO MTU: 64:1
        UP RUNNING
        RX bytes:1542 acl:0 sco:0 events:94 errors:0
        TX bytes:2565 acl:0 sco:0 commands:94 errors:0
```

Finally, turn on the device:

```
pi@BeaconAir ~/BeaconAir/ble $ sudo hciconfig hci0 up
```

Now let's install the Beacon Scanning software. Type the following into the command line:

```
cd
sudo apt-get install python-bluez
sudo pip3 install PyBluez
git clone https://github.com/switchdoclabs/SDL_Pi_
iBeaconScanner
cd SDL_Pi_iBeaconScanner
```

And now run the blescanner command to see what iBeacons might be around you. If you don't have an iBeacon, you can simulate it on with either your iPhone or Android phone with any number of apps on the App Store.

```
cd /home/pi/SDL_Pi_iBeaconScanner
sudo python3 testblescan.py
```

```
pi@SwitchDocLabs:~/SDL_Pi_iBeaconScanner $ sudo python3
testblescan.py
ble thread started
----------
43:bc:c4:82:59:1d:01:00,1d5982c4bc430e0201060aff
4c001005,17692,42096,-26,-54
43:bc:c4:82:59:1d:01:04,0104011d59,33476,48195,0,-54
dc:56:e7:49:39:f3:00:00,e756dc1102011a020a0c0aff
4c001005,2324,22049,24,-87
dc:56:e7:49:39:f3:00:04,010400f339,18919,22236,0,-88
7b:03:5f:01:4b:0c:01:00,037b1202011a020a0c0bff4c001006
0d,7585,3364,24,-26
7b:03:5f:01:4b:0c:01:04,0104010c4b,351,891,0,-26
```

```
73:df:a1:ac:0e:34:01:00,0613ff4c000c0e00e070ca209cd1f4
0e,62240,22431,40,-49
73:df:a1:ac:0e:34:01:04,010401340e,44193,57203,0,-51
c8:69:cd:63:df:0f:00:00,020a0c11ff4c000f08c00a99b7
fe0044,2832,513,4,-86
----------
f8:04:2e:bf:d2:05:00:03,80a0f8042ebfd205fa042eb
fd2040100,0,0,0,-61
0a:62:b6:b2:a1:e9:01:03,fd0925f23cb3236420d2fe9f7dd48c
ad,38805,14355,-81,-50
4e:a5:00:3e:a2:64:01:00,64a23e00a54e0e0201060aff
4c001005,17692,42096,-26,-55
4e:a5:00:3e:a2:64:01:04,01040164a2,15872,42318,0,-56
43:bc:c4:82:59:1d:01:00,1d5982c4bc430e0201060aff
4c001005,17692,42096,-26,-51
43:bc:c4:82:59:1d:01:04,0104011d59,33476,48195,0,-51
dc:56:e7:49:39:f3:00:00,e756dc1102011a020a0c0aff
4c001005,2324,22049,24,-89
dc:56:e7:49:39:f3:00:04,010400f339,18919,22236,0,-88
0b:a7:c3:bb:5c:3a:01:03,5cbbc3a70b0f02011a0bff
4c00090603,6592,43009,9,-75
        c8:69:cd:63:df:0f:00:00,020a0c11ff4c000f08c00a99b7
        fe0044,2832,513,4,-84
```

The content of each preceding line is this:

```
example:
```

```
dc:56:e7:49:39:f3:00:00,e756dc1102011a020a0c0aff
4c001005,2324,22049,24,-87
```

```
Beacon MAC Address,iBeacon UDID, iBeacon Major Number, iBeacon
Minor Number, TX Power at 1m, RSSI
```

Note that there are some odd and unknown devices in the preceding content that are *not* my BlueCharm iBeacons. The odd devices have larger numbers or actually numbers that vary. Interesting information. I found over 100 Bluetooth devices in the range of my scanner. I only know what about 1/3 of those devices are. There is an article to be written about this!

Phue

The installation of phue on your Pi is simple.

```
sudo pip3 install phue
```

Note that after the first time you run BeaconAir on your Raspberry Pi, the phue library will quit telling you that you need to push the Philips Hue Hub button and rerun the software. The BeaconAir Raspberry Pi will then be paired with the Philips Hue Hub.

Startup Procedure

To start up BeaconAir, you need to do three things:

1. Turn on the Bluetooth dongle as follows:

   ```
   sudo hciconfig hci0 up
   ```

2. Change to the BeaconAir directory. Start up the BeaconAir Python programming by invoking either

   ```
   sudo python3 BeaconAir.py (in its own
   terminal window)
   ```

   or

   ```
   sudo nohup python3 BeaconAir.py & (from the
   command line)
   ```

If you use nohup, you can close the terminal window and the program keeps running in the background until you reboot or kill the program. All of the debug data goes into a file nohup.out in the start directory. If you want to watch what is going on using nohup, go to the program directory and type sudo *tail -f nohup.out* on the command line.

Making BeaconAir Start on Bootup

You can make BeaconAir start when the Pi boots by editing /etc/rc.local. The /etc/rc.local script is a script on the Raspberry Pi that runs when Linux first boots. To edit it, you will need root privileges. Invoking the editor via sudo is the way to go here:

```
sudo nano /etc/rc.local
```

Then add the following lines to the rc.local file. Place the newly added lines before the exit 0 statement.

```
sudo hciconfig hci0 up

date >> /var/log/RasPiConnectServer.log
echo "Starting RasPiConnectServer…" >> /var/log/
RasPiConnectServer.log
nohup /home/pi/RasPiConnectServer/startserver.sh >>/var/log/
RasPiConnectServer.log 2>&1 &

date >> /var/log/BeaconAir.log
echo "Starting BeaconAir.." >> /var/log/BeaconAir.log
cd /home/pi/SDL_Pi_BeaconAir/
nohup sudo python3 BeaconAir.py >>/var/log/RasPiConnectServer.
log 2>&1 &
```

How It Works in Practice

BeaconAir does work. As I walk around with the BeaconAir Portable Pi in a fanny pack, the lights come on and off.

However, the iBeacons are not very reliable and will vary significantly just sitting in one spot. You can watch the little red pin bounce around on the control panel. A lot could be done to smooth this out by doing a little signal processing at the cost of response time. It certainly makes an interesting demo to show people.

Things to Do

You can use the MQTT techniques of Chapters 5 and 6 to connect the BeaconAir to an external IoT server or tweet your location as in Chapter 3.

One idea I had for another project revolving around iBeacons was to reverse the system. Carry an iBeacon and build a mesh network of Raspberry Pis all listening to the iBeacon via my BLE Scanner program and then communicating the RSSI information to a central Pi that would figure out the location of the iBeacon and report it to the control panel. Now granted, it's more expensive than buying a bunch of iBeacons and one Pi, but it would have some really interesting dataflows and the response time could be excellent. It would be a lot better for the user as they would just have to carry a small iBeacon in their pocket! Better for the fashion conscious than a fanny pack. Figure 4-8 shows BeaconAir installed in a very fashion-conscious fanny pack.

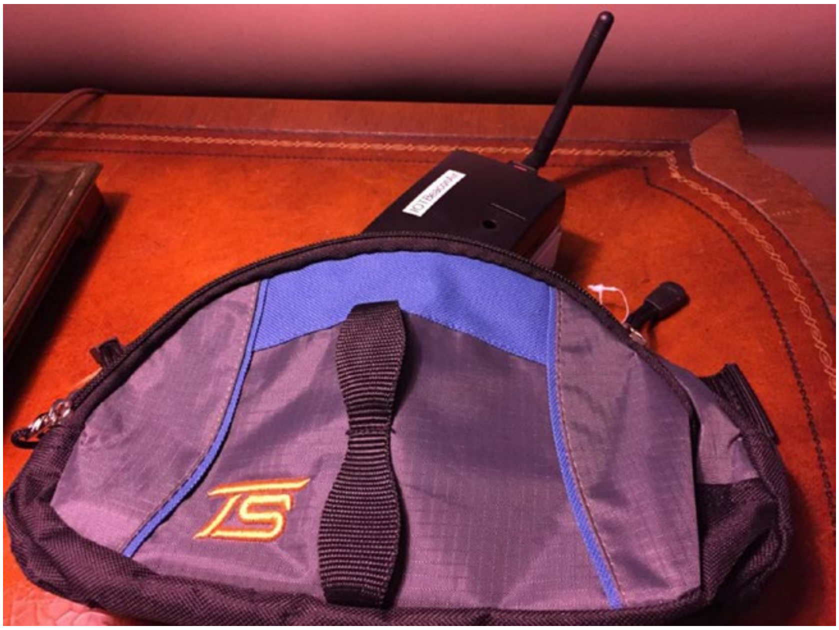


Figure 4-8. *BeaconAir Ready to Walk*

The first thing I would add to this unit is the ability for the Raspberry Pi to sense the light levels in the environment. Why turn the lights on if it is already bright in the room?

The Classic Distributed System Problems

Now I will talk about the classic problem with groups of IoT devices. What do you do if two IoT units are telling your lights two different things? Who wins?

If we were to build a bunch of these devices and put them on other members of the household (other people or the cat), we now need to build an arbitration system. What do I mean by that? We have to "arbitrate" the information coming in from the IoT devices. In computer speak, a group of BeaconAir units with no central coordinator becomes a distributed system.

From the Wikipedia definition of distributed systems [`en.wikipedia.org/wiki/Distributed_computing`], "A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components." You can see from this that any group of IoT devices fit within this definition.

Since there is no global server for BeaconAir, there is no mechanism currently defined to handle arbitration of actions based on detection. In general, it is easier to resolve these kinds of issues by using a global server. But, this makes your system more susceptible to failure of the server. Group self-organization and group arbitration is a better way to go, but it is much more complicated to design, implement, and test.

Here are some of the questions to arbitrate once you have multiple IoT units of BeaconAir:

- Which BeaconAir turns the lights on?
- Which BeaconAir turns the lights off?
- How do you handle one person leaving the room?
- How do you rank the BeaconAir units to one another? Is the cat on the bottom or top of the priority list?
- How do you handle people turning lights on or off by hand?
- How do you detect a person sitting down to read vs. walking through the house?

And the list goes on. Any time you put more than one IoT device in control of anything physical at all, things can get very complicated.

Conclusion

In Chapter 3, we built an IoT weather device that generated and supplied information to the rest of the Internet. In this chapter, you saw how BeaconAir reverses that. It gathers information from the IoT devices (iBeacons) around itself and then modifies the environment without going out to the Internet (or the wider IoT) at all. This is a much more localized IoT application.

In Chapter 6, we'll extend this idea further by building an RFID (Radio Frequency IDentification) unit that will gather information from passive units (that are only activated when they are being interrogated for information) and then do both major functions. Send data to the Internet and modify the local environment.

For all of the software, see the following:

```
github/switchdoclabs/SDL_Pi_BeaconAir

github/switchdoclabs/SDL_Pi_iBeaconScanner
```

For more on BeaconAir and discussion: `www.switchdoc.com`.

CHAPTER 5

Connecting an IoT Device to a Cloud Server: IoTPulse

Chapter Goal: Build a Portable IoT Device for Reading Your Pulse

Topics Covered in This Chapter:

- IoT on a global network
- The cloud and the Internet of Things
- The IoTPulse design
- Building the IoTPulse device
- Connecting and testing the hardware and software for IoTPulse
- Setting up the IBM Cloud and connecting IoTPulse
- Examining results and advanced features

This chapter will show you how to connect an IoT device to a cloud server, the IBM Cloud. The complexity in this chapter is not building the device, but rather navigating the setup process for adding your device to the cloud. To avoid this complexity, we will use the IBM Cloud Quickstart server.

A major part of designing devices that will connect with the IoT is to determine what to do with the data after you have gathered it. A huge amount of data and no way to store it or analyze it is not very useful. An IoT project device generally is very CPU limited and also only has a limited amount of storage available. You generally have to send the data collected by a sensor-filled IoT device to a bigger computer and storage cloud to perform complex analysis and determine actions on that data.

In this chapter, we are building a device to collect the pulse rate from a person and then periodically (every 10 seconds) send it to a cloud-based storage and analysis system called the IBM Cloud IoT Foundation.

Before we selected the IBM Cloud for this chapter, we also looked at three other providers. We determined that the Amazon IoT solution was too "heavy" for these small devices because of the protocol and encryption required. The two other smaller vendors were rejected because of length of time in existence and limited analysis functionality. It should be pointed out that the smaller vendors had distinctly superior display capabilities out of the box from either of the two majors.

The IBM Cloud system brought a good mix of flexible protocols, a "light" method for sending data, and an amazing collection of analytical tools. Ramifications of the IBM Cloud methodology on encryption vs. the Amazon IoT solution will be discussed in Chapter 7.

IoT Characterization of This Project

As we discussed in Chapter 1, the first thing to do to understand an IoT project is to look at our six different aspects of IoT shown in Table 5-1. IoTPulse is a more complex project than LightSwarm and is much closer to a production IoT device.

Table 5-1. *IoTPulse Characterization (CPLPFC)*

Aspect	Rating	Comments
Communications	9	WiFi connection to the Internet – can do ad hoc mesh-type communication
Processor power	7	80MHz XTensa Harvard Architecture CPU, ~80KB data RAM/~35KB of instruction RAM/200K ROM
Local storage	8	4MB Flash (or 3MB file system!)
Power consumption	8	~200mA transmitting, ~60mA receiving, no WiFi ~15mA, standby ~1mA
Functionality	7	Partial Arduino support (limited GPIO/analog inputs)
Cost	9	< $10 and getting cheaper

Ratings are from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications.

This gives us a CPLPFC rating of 8.0. Great for learning and experimenting, and it could be deployed for some market applications.

The ESP8266 provides a WiFi transmitter/receiver, a TCP/IP stack, and firmware to support direction connections to a local WiFi access point, which then can connect to the Internet. The data generated by the IoTPulse design will be sent to IBM's Cloud Cloud IoT site.

The Internet of Things on the Global Network

As was stated in the introduction, what do you do with all this data you are gathering? Whether it is the age of your milk in the refrigerator, your current blood sugar or pulse rate, or your home security system, the data needs to be stored and analyzed; or what is the point of gathering it?

Sometimes you don't need to store the data. Sometimes the local devices can do the necessary actions (like in the LightSwarm project in Chapter 2), but there are many applications that require more computing resources than might be available among the local IoT devices.

This is the situation that has led to the development of server-based (often on a computing cloud) support software for the upcoming IoT explosion. In Chapter 2, we used a Raspberry Pi to gather the data from the LightSwarm and log the rather complex behavior of the cooperative swarm itself for later analysis. If you have a thousand devices, you most likely are not going to be able to host your server on a Raspberry Pi, which has limited CPU power and storage. Another major problem with basing your system on physical servers that you control (like Raspberry Pis, but also rack mount servers with much larger CPU and memory capabilities) is that of scalability. If suddenly you have 10,000 devices connecting to your servers instead of 1000 devices, you need to be able to scale quickly and effectively.

It is this realization that IoT systems need to scale quickly and efficiently that has moved the IoT back-end applications into the realm of cloud computing. For an IoT application, there are three main areas that need to be addressed. They are cloud computing, application builders, and report and generation software.

Cloud Computing

Cloud computing is evolving quickly to morph into more services, ubiquitous back ends, high-speed network connectivity, and multiple languages and solutions. The National Institute of Science and Technology (NIST) provides a document that defines cloud computing for government purposes [http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf]. While parts of it are already outdated, it does give a good base definition of cloud computing.

NIST defines cloud computing as this: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

The essential characteristics of cloud computing are as follows:

1. On-demand – User can provision computing services as needed without requiring interaction with service provider.

2. High Internet connectivity – Cloud capabilities are available over the network and accessed through mechanisms that promote multiple platform usage (not just desktops, but many other types of computers and devices).

3. Ability to pool resources – Resources such as memory, CPU power, and storage are pooled to support multiple users and customers according to user demand.

4. Rapid scaling – Capabilities and resources can be provisioned and released automatically or under user control. With the proper system and software, the cloud can look to be virtually unlimited to the user or application.

5. Ability to limit (or meter) resources – A cloud will be able to limit resources and optimize resource use at several levels of abstraction.

6. Ability to charge for service – A cloud service is able to account for resource usage, be able to monitor those resources, and report usage in a transparent manner for both the supplier and the user of the system.

Application Builders

There is no way that a cloud provider can anticipate every application or use for data that an IoT provider will need. A good cloud provider will support major types of databases and a variety of standard software application programming interfaces (APIs). It will provide analytical tools for the data and a variety of different services for manipulating that data. A cloud application (or cloud app) is a program that runs in a cloud environment. It has some aspects of a pure local app (for an iPad or a desktop) and some aspects of a pure web-based app. A local app will reside entirely on your local computer, while a web app is generally stored entirely on a remote computer and is delivered via a browser interface. There are web apps that are a hybrid (such as WebEx) that have a very light client on the browser side (still an app, but a small one) that connects via an API to a much more complex program on the server side. The application builders in the IBM Cloud have a heavier client (running the graphics and editing part of the application) and then deliver the user design to the servers via an API.

In the IoT world, an application often has both aspects, where the local computer contains part of the application (say, sensor data gathering) and the cloud contains the back-end heavy metal processing for the full application. The IoTPulse project is one of these mixed types of applications.

Display and Report Generation

Displaying pertinent information to the user about the user's IoT devices and what actions they are performing on the user's behalf is an important aspect that will lead to the adoption of widespread IoT applications. It is argued by some that it is important not to overwhelm the user with data about what is going on. However, I believe that the "overwhelming" part of the IoT can be handled with hierarchical interfaces. The top-most level of the user interface may only display the very basic information needed by the user (say, temperature and furnace status). I attended an early meeting on what was to become the IoT at Microsoft in Redmond, Washington, and the speaker called the most basic mode "Grandma Mode" because he felt there was a need to support substantially every type of user that was going to be using the product with a simple interface.

I felt this was true as far as it went, but with the IoT, the ability for a device and software interface to show what is going on with the data being gathered is paramount. A method and interface needs to be provided for the user to really drill down and look at what is happening inside his network and device at several different levels. Not as part of the "Grandma Mode" interface, but another set of interfaces that are accessible to any user if the user should wish.

Doing this hierarchical interface is a key part of establishing the necessary trust between the user and the IoT device.

A well-designed cloud-based application and report generation system needs to provide the application developer many different ways of analyzing the data, acting on the data, and displaying the data. The cloud developer will never be able to supply all the possible build interfaces that the app developer will need for every application.

The IBM Cloud Internet of Things Solution

The IBM Cloud (Figure 5-1) includes an IoT-focused implementation of IBM's Open Cloud Architecture that enables you to create, deploy, and manage cloud applications for the Internet of Things.

Figure 5-1. *The IBM Cloud Logo*

There is a growing ecosystem of runtime frameworks and services, including non-IBM third-party solutions and applications. The IBM Cloud provides a dashboard for you to create, view, and manage your applications, IoT devices, and desired services; the IBM Cloud dashboards also provide the ability to manage organizations, spaces, and (very importantly!) user access.

We did find that the “dashboard” paradigm in the IBM Cloud system is somewhat of a misnomer and a bit confusing. Instead of having one dashboard, you have dashboards for every service that you attach. It's easy to get lost in the sequence. But with some perseverance, we got through the learning curve.

The IBM Cloud provides access to a wide variety of services that can be incorporated into an application from multiple vendors.

For more complex applications than the IoTPulse device, you have a set of application libraries that you can build your applications connecting to a variety of database functions and frameworks. Here are some of the more common ones:

- Node.js
- PHP
- Python
- Ruby

All in all, the IBM Cloud is a good implementation of a cloud-based IoT platform to build upon.

Note that the IBM Cloud is an evolving platform, and there will be changes going forward in their product maturation process.

There are many features that are free for prototypes (such as IoTPulse), and all of the features mentioned in this chapter are free for use on the IBM Cloud. Looking through the paid services and rates, it seems that they are reasonable for both small and large applications.

The IoTPulse Design

The IoTPulse design consists of three blocks. The first block is the ESP8266 and the 9V battery, which contains the computer; the WiFi interface; and a single analog-to-digital converter (ADC) pin, called A on the ESP8266 board and A0 inside the Arduino IDE software. The ESP8266 is responsible for doing the signal processing and translation of the incoming analog heartbeat signal to a digital heartbeats per minute (BPM) and then will periodically send the latest BPM to the IBM IoT cloud.

The second block is the Pulse Sensor Amped, which is an open source hardware design by www.pulsesensor.com. Figure 5-2 shows the schematic for the Pulse Sensor. The key thing to take away from the Pulse Sensor is that it is designed to get an approximate measure of heart rate through the skin in a non-invasive way. It works by shining a bright green LED into your skin and then detecting the relative changes in light intensity to the sensor. With a small amount of signal shaping, you can detect pulse rate via an analog output signal.

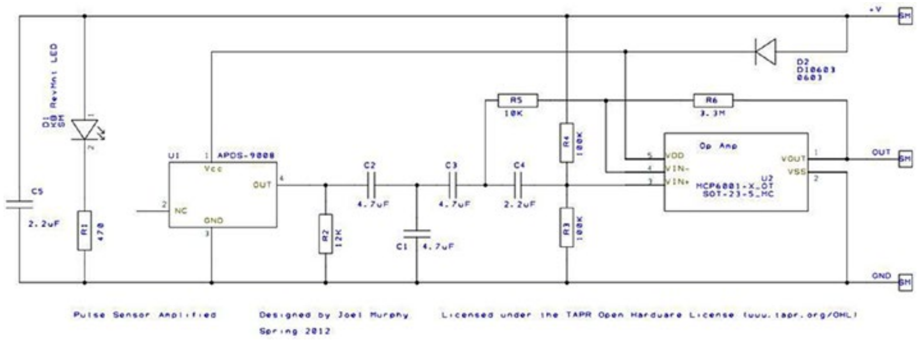


Figure 5-2. *Schematic for the Pulse Sensor*

For more information on the technique, see photoplethysmogram [https://en.wikipedia.org/wiki/Photoplethysmogram].

The third block is a resistor-based voltage divider. The issue that we are solving with this voltage divider is that the signal output from the Pulse Sensor is about 1.5V and the ADC input on the ESP8266 only goes up to 1V. For a full description on how a voltage divider works, check out the Wikipedia article [https://en.wikipedia.org/wiki/Voltage_divider].

We need to take 1.5V down to about 0.6V for the ESP8266 and the software to work. The Pulse Sensor provides a signal that on the average is about 1.5V, which is too high for the ESP8266 ADC to work correctly. With the choice of a 26K ohm resistor for R1 and an 18K ohm resistor for R2 in Figure 5-3, we get the following ratio of input to output voltage:

```
Vout = Vin * R2/(R1 + R2)

Vout = 1.5V * 18K/(26K + 18K) = 0.6V
```

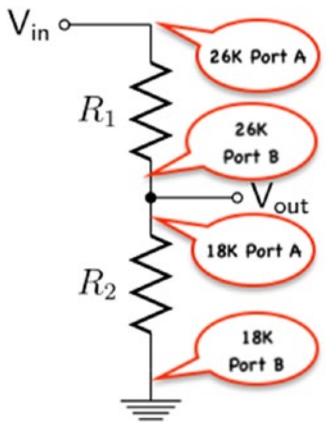


Figure 5-3. *Simple Voltage Divider*

The 0.6V is comfortably in the 0.0–1.0V range of the ESP8266 ADC and works well. There is virtually no current drain through this voltage divider because of the high resistance of the voltage divider (44K ohms).

Why does this work? It is because the software in the ESP8266 is calculating the heartbeat by looking at relative changes in the light intensity and not the absolute voltage. As long as the input value is changing, the ESP8266 will pick up the pulse.

If we did not put the voltage divider on the output of the Pulse Sensor, then the ADC in the ESP8266 would always be pegged to the highest level (1.0V) and the sensor software would not be able to pick up a pulse.

Because the ESP8266 ADC is so limited, there are better solutions out there for an ADC. Two of the easier to use boards are these:

- Seeedstudio Grove I2C ADC – One channel, 12 bits (requires 5V – needs some help to work with the ESP8266) [`www.seeedstudio.com/wiki/Grove_-_I2C_ADC`]
- SwitchDoc Labs Grove I2C ADC – Four channels, 16 bits 3.3V/5V compatible [`www.switchdoc.com/grove-4-channel-16-bit-adc-based-ads1115/`]

Figure 5-4 shows the block diagram, including the voltage divider, of the IoTPulse project. The completed IoTPulse device, including the 3D printed case, is shown in Figure 5-5.

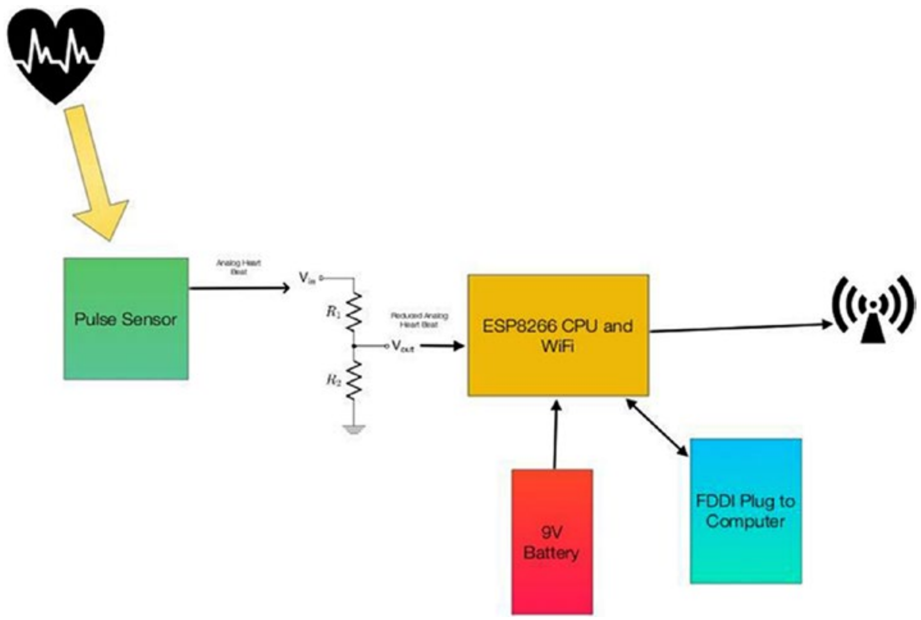


Figure 5-4. *Block Diagram of IoTPulse*

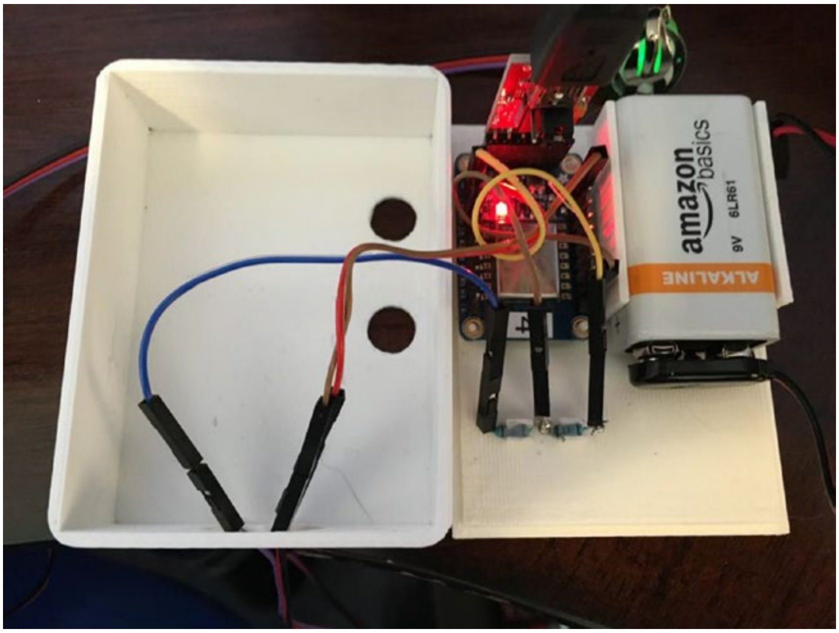


Figure 5-5. *IoTPulse in Case*

Building the IoTPulse

The parts list is shown in Table 5-2, followed by the wiring list. The parts for the IoTPulse device are readily available, and only the voltage divider will require any soldering.

Table 5-2. *Parts List*

Part Number	Count	Description	Approximate Cost per Board	Source
ESP8266 Huzzah board	1	CPU/WiFi board	$10	www.adafruit.com/products/2471
Pulse Sensor Amped	1	I2C light sensor	$25	www.Pulsesensor.com
FTDI cable	1	Cable for programming the ESP8266 from PC/Mac	$11	www.switchdoc.com/inexpensive-ftdi-cable-for-arduino-esp8266-includes-usb-cable/
26K ohm resistor	1	1/4 watt	$17 for a large mix of resistors (Joe Knows Electronics)	http://amzn.to/1QCASLB
18K ohm resistor	1	1/4 watt	$17 for a large mix of resistors (Joe Knows Electronics)	http://amzn.to/1QCASLB

Plugging the FTDI Cable into the ESP8266

An FTDI cable is plugged into the end of the Adafruit Huzzah ESP8266. Make sure you align the GND pin on the FTDI cable with the GND pin on the ESP8266 breakout board as shown in Figure 5-6.

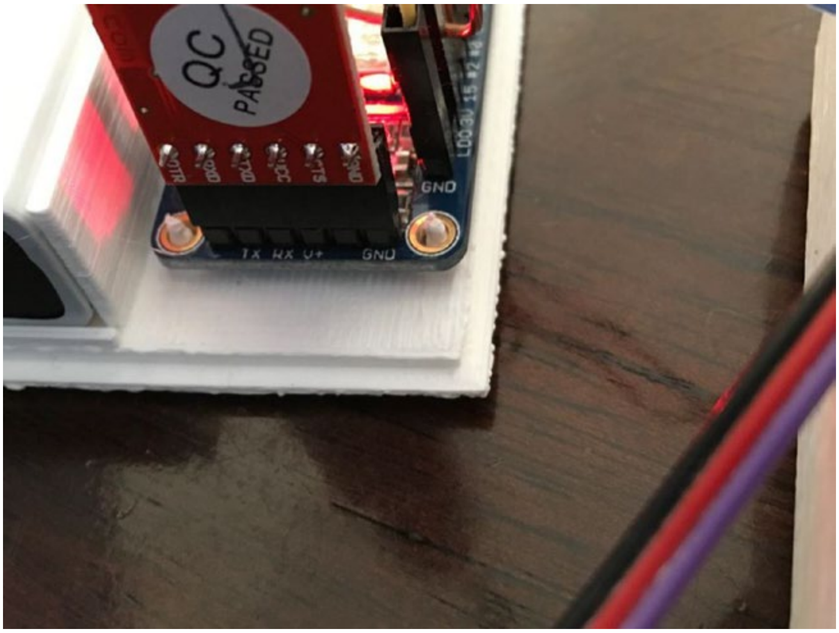


Figure 5-6. *FTDI Cable Plugged into the Huzzah Board. Note GND Connection*

Table 5-3 contains the wiring list for the IoTPulse project.

Table 5-3. *IoTPulse Wiring List*

From	To	Description
ESP8266/GND	Pulse Sensor/GND (black wire)	Ground for Pulse Sensor
ESP8266/3V	Pulse Sensor/3.3V (red wire)	3.3V power for Pulse Sensor
Pulse Sensor/output (purple wire)	26K/Port A	Top of the resistor divider
26K/Port B	18K/Port A	Middle of the resistor divider
18K/Port B	ESP8266/GND	Ground for bottom of resistor divider
26K/Port B	ESP8266/A	A0 – analog-to-digital converter input on ESP8266

3D Printing Files for the IoT Case

Finding a case for a custom project can always be a problem. All of the projects in the chapters have suggested using 3D printing to build cases that fit the project, and the IoTPulse project is no different. The most interesting part of this case is the way we designed a bevel that goes around the base so the top case will snap onto the bottom case (Figure 5-7).

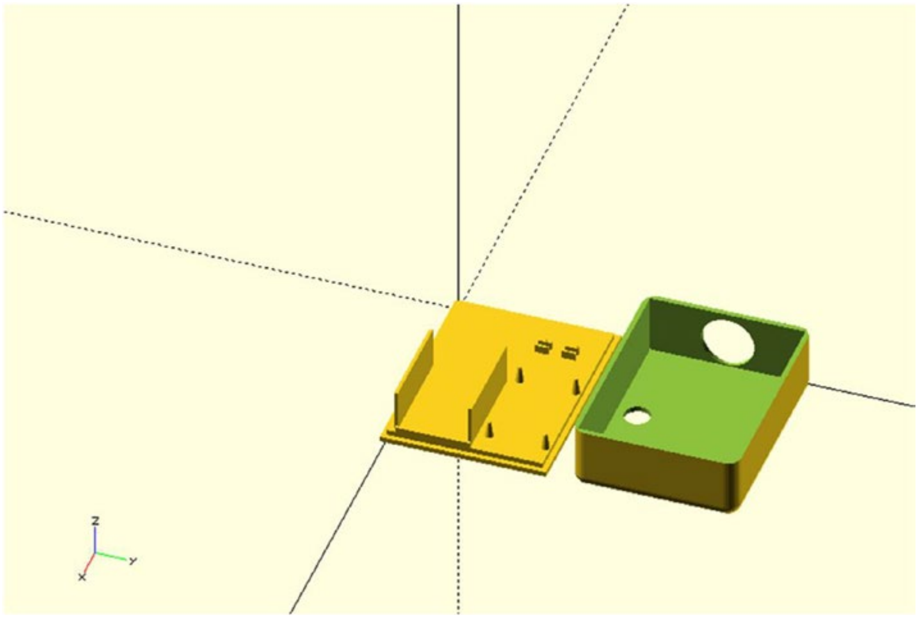


Figure 5-7. *IoT Case*

We have introduced openSCAD, a free code-based 3D modeler in previous chapters. There are hundreds of example models on Thingverse.com and the OpenSCAD website to choose from and to learn from. Listing 5-1 contains the code that builds the case shown in Figure 5-7.

Listing 5-1. OpenSCAD Code for the IoT Case

```
//
// IOTPulse Case
//
// SwitchDoc Labs
// August 2015
//
```

```
include <RoundedRect.scad>

module rcube(size=[30, 20, 10], radius=[3, 2, 1], center=true)
    hull() {
        translate( center ? [0,0,0] : size/2 ) {
            cube(size-2*radius+[2*radius[0],0,0],center=true);
            cube(size-2*radius+[0,2*radius[1],0],center=true);
            cube(size-2*radius+[0,0,2*radius[2]],center=true);

            for(x = [-0.5,0.5], y = [-0.5,0.5], z = [-0.5,0.5])
                translate([x * ( size[0] - 2*radius[0]),
                           y * ( size[1] - 2*radius[1]),
                           z * ( size[2] - 2*radius[2])])
                    scale([radius[0], radius[1], radius[2]])
 //                       sphere(1.0,$fn=4*4);
                      sphere(1.0,$fn=6*6);
        }
    }

module IOTPulseTopCase()
{

    difference()
    {
        //cube([82,62,25]);

        rcube(size=[82,62,30],radius=[5,5,5],center=false);

        #translate([2,2,2])
        cube([78,58,25]);

        #translate([-10,-10,25])
        cube( [100,100,50]);
        // hole for pulse meter 18mm

        translate([-5,62/2,12.5])
        #rotate([0,90,0])
        cylinder(h=10,r=10,10);

        // hole for seeing light
        #translate([82-20,10,-2])
        cylinder(h=10,r=5,10);

        // hole for seeing wifi light
        #translate([82-43,10,-2])
        cylinder(h=10,r=5,10);

    }

}

union()
{

    cube([78,58,4]);
    translate([-2,-2,0])
    cube([82,62,2]);

    // Mount for Battery

    translate([40-2,2,0])
    cube([40-2,1.35,20]);
    translate([40-2,26.10+3.3,0])
    cube([40-2,1.5,20]);

    // lips for battery
    translate([79-2,3,0])
    cube([1,28,6]);
    // pylons for ESP8266

    translate([70-1.0,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-1.0,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,35,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
    translate([70-34,56,0])
    cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);

    // pylons for resistors for divider


    // gap of 2.3mm

    translate([15,35,2])
    cube([1,5,5]);

    translate([15+3,35,2])
    cube([1,5,5]);

   translate([15,45,2])
    cube([1,5,5]);

    translate([15+3,45,2])
    cube([1,5,5]);

   // top case


    translate([0,70,0])
    IOTPulseTopCase();
}
```

The code for the IoTPulse case is broken into two main sections. The main program builds the lower part of the case, while the top of the case is located in the function IOTPulseTopCase(). When building models with openSCAD, it is a good idea to break different parts into functions even if you are only calling them once. It simplifies the code. Now, let us look at the software for IoTPulse.

Software Needed

For you to program the IoTPulse board, you will need a Mac or PC to program the ESP8266 via the Arduino IDE. See Chapter 2 for how to set up the Arduino IDE with the ESP8266 libraries. You will also need to install the Arduino library "pubsubclient". You can see how to install this library here:

`www.arduino.cc/reference/en/libraries/pubsubclient/`

The IoTPulse Code

There are three files that are part of the build for the IoTPulse software. The IOTPulse.ino is the main Arduino IDE file (in the C language), and the other two files (Interrupt.h and AllSerialHandling.h) are included in the main file. While the code is fairly straightforward, comments need to be made for each module.

IOTPulse.ino

IOTPulse.ino is the main program for IoTPulse (Listing 5-2). It consists of two major functions, setup() and loop(). The setup() function initializes the ESP8266 WiFi code, reads an IP address using DHCP from the local wireless access point, and initializes variables for containing micros(), which is a function returning the number of microseconds since the reboot of the ESP8266 micros() is a timekeeping function. Note that the ESP8266 does not have a real-time clock on board and really doesn't know what time it is. It only knows how long it has been running. If you wish to set the time of day in the ESP8266, please check out the NTP protocol in this Instructable [`www.instructables.com/id/Internet-time-syncronized-clock-for-Arduino/`].

In the IoTPulse project, we really don't care what time it is. When we send a sample to the IBM Cloud IoT Platform, our friend IBM will timestamp the reception of the data. If we cared about our time, we could build an NTP receiver and synchronize it to network time at a specific NTP server. We are much more interested in intervals of time (specifically 2 milliseconds and 10 seconds' intervals) rather than absolute time.

The loop() function contains the code for ongoing operation of the IoTPulse data-gathering operation. We basically do four things in the loop() function:

- We check the value of QS. If QS is true, then the software in Interrupt.h has found a BPM (Beats Per Minute) and IBI (Interval Between beats) value for the current pulse rate for our Pulse Sensor connected to your ear.
- Next, if 10 seconds have elapsed (10,000,000 microseconds), we send the current value of BPM and IBI to the IBM Cloud IoT cloud. We then reset the time to wait for the next 10-second interval, which is stored in the oldIOTTime variable. The actual time for the next interval is oldIOTTime + 10 seconds (10,000,000 microseconds).
- If 2 milliseconds have elapsed since oldPulseTime variable (initialized in loop()), we call the timerCallback() function defined in Interrupt.h. We then reset the oldPulseTime variable and wait for another 2ms.

Note at the end of the loop() function, there is a yield() function. This *must* be called periodically for the ESP8266 software to continue running the WiFi interface and other housekeeping functions. Remember, unlike a regular Arduino, there is a great deal going on in the background to keep this chip running and connected to the network.

One limitation of the current ESP8266 software is that you can't use the timers [www.switchdoc.com/2015/10/iot-esp8266-timer-tutorial-arduino-ide/] to generate interrupts without causing the WiFi to fail and stop connecting to the local WiFi access point and the Internet.

This makes it very difficult to generate an interrupt to the computer every 2ms as is called for by the Pulse Sensor. You could generate an interrupt every 2ms by using an external hardware timer and then connecting the output of the hardware timer to one of the GPIO pins on the ESP8266, which can all be programmed to generate an interrupt to the CPU.

To solve this issue, we use a scheduling technique that is more CPU intensive (read uses more power) than the interrupt scheme. We check the time periodically and run the Pulse Sensor software every 2ms and also connect to the IBM Cloud every 10 seconds.

The biggest disadvantage of this is that we cannot "sleep" the processor very easily. We are looking for an alternative architecture for this problem and are watching the ESP8266 development websites for solutions.

However, with the exception of power consumption, the current solution works well.

You will need to modify the code to connect to your local WiFi using the name of the WiFi and your password. Change the following lines:

```
const char* ssid = "YOURSSID";
const char* password = "YOURPASSWORD";
```

You can download the IOTPulse ESP8266 code from the SDL repository on github.com:

```
github.com/switchdoclabs/SDL_ESP8266_IOTPulse
```

Listing 5-2. SDL_ESP8266_IOTPulse.ino

```
/*
    SwitchDoc Labs Code for IOT Pulse
    Connects Pulse detector to the IBM Cloud IoT using a
    ESP8266 processor
    based on pulsecounting code from www.pulsesensor.com
    January 2021

  ----------------------  Notes ------------  -----------------
  This code:
  1) Blinks an LED to User's Live Heartbeat   PIN 0
  2) Fades an LED to User's Live HeartBeat
  3) Determines BPM
  4) Prints All of the Above to Serial
*/

extern "C" {
#include "user_interface.h"
}
#include <ESP8266WiFi.h>
#include <PubSubClient.h> // https://github.com/knolleary/
pubsubclient/releases/tag/v2.3

//--------------------------------------------------------------
//Local WiFi Variables

const char* ssid = "YOURSSID";
const char* password = "YOURPASSWORD";

#define IOTPULSEVERSION 005
```

```
// IBM Cloud IOT Foundation Data

#define ORG "quickstart" // your organization or "quickstart"
#define TOKEN "xxxxxxxxxxxxxxx" // not used with "quickstart"
#define DEVICE_TYPE "ESP8266"  // not used with quickstart or
customize to your registered device type
#define DEVICE_ID "IOTPulse-01" // use anything for quickstart
or customize to your registered device id
#define EVENT "myEvent" // use this default or customize to
your event type

char server[] = "quickstart.messaging.internetofthings.
ibmcloud.com";
char topic[] = "iot-2/evt/status/fmt/json";
char authMethod[] = "use-token-auth";
char token[] = TOKEN;
char clientId[] = "d:" ORG ":" DEVICE_TYPE ":" DEVICE_ID;

//-------------------------------------------------------------

//  Variables
int pulsePin = A0;                 // Pulse Sensor purple wire
                                      connected to analog pin 0
int blinkPin = 0;                // pin to blink led at each
                                    beat
int fadePin = 5;                  // pin to do fancy classy
                                     fading blink at each beat
int fadeRate = 0;                 // used to fade LED on with
                                     PWM on fadePin

// Volatile Variables, used in the interrupt service routine!
volatile int BPM;                  // int that holds raw
                                      Analog in 0. updated
                                      every 2ms
```

```
volatile int Signal;                // holds the incoming raw
                                       data
volatile int IBI = 600;             // int that holds the time
                                       interval between beats!
                                       Must be seeded!
volatile boolean Pulse = false;     // "True" when User's live
                                       heartbeat is detected.
                                       "False" when not a "live
                                       beat".
volatile boolean QS = false;        // becomes true when
                                       Arduino finds a beat.

// Regards Serial Output  -- Set This Up to your needs
static boolean serialData = true;   // Set to 'false' by
                                       Default.  Re-set to
                                       'true' to see Arduino
                                       Serial Monitor data

#include "AllSerialHandling.h"
#include "Interrupt.h"

void callback(char* topic, byte* payload, unsigned int length)
{
  Serial.println("callback invoked from IOT IBM Cloud");
}

WiFiClient wifiClient;
PubSubClient client(server, 1883, callback, wifiClient);

unsigned long oldPulseTime;
unsigned long oldIOTTime;

void setup() {

  pinMode(blinkPin, OUTPUT);        // pin that will blink to
                                       your heartbeat!

  Serial.begin(115200);             // we agree to talk fast!
  Serial.println("----------------");
  Serial.println("IOTPulse IBM Cloud IOT");
  Serial.println("----------------");

  Serial.print("Connecting to ");
  Serial.print(ssid);
  if (strcmp (WiFi.SSID().c_str(), ssid) != 0) {
    WiFi.begin(ssid, password);
  }
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");

  Serial.print("Local WiFi connected, IP address: ");
  Serial.println(WiFi.localIP());

  // interruptSetup();                // sets up to read Pulse
                                         Sensor signal every 2ms
  // Note:  Interrupts based on os_timer seems to break the
     ESP8266 WiFi.  Moving to micros() polling methodology
  oldPulseTime = micros();
  oldIOTTime = micros();
}

int sampleCount = 0;
int beatCount = 0;

int beatValue = 0;
unsigned long newPulseDeltaTime;
unsigned long newIOTDeltaTime;

//  Where the Magic Happens
void loop() {
```

```
//serialOutput() ;

if (QS == true) {    // A Heartbeat Was Found
  // BPM and IBI have been Determined
  // Quantified Self "QS" true when arduino finds a heartbeat
  digitalWrite(blinkPin, LOW);    // Blink LED, we got a
  beat.
  beatCount++;

  serialOutputWhenBeatHappens();   // A Beat Happened, Output
                                      that to serial.
  QS = false;                      // reset the Quantified
                                      Self flag for next time
}

newPulseDeltaTime = micros() - oldPulseTime; // doing this
handles the 71 second rollover because of unsigned arithmetic

newIOTDeltaTime = micros() - oldIOTTime; // doing this
handles the 71 second rollover because of unsigned arithmetic

// do this every ten seconds
if (newIOTDeltaTime > 10000000)  // check for 10sec work to
                                    be done
{

  Serial.print("IOT Delta time =");
  Serial.println(newIOTDeltaTime);
  sampleCount++;

  // Sending payload: {"d":{"IOTPulse":"IP1","VER":2"SC":0,"B
     PM":235,"IBI":252}}
```

```
String payload = "{\"d\":{\"IOTPulse\":\"IP1\",";
payload += "\"BPM\":";
payload += BPM;
payload += ",\"VER\":";
payload += IOTPULSEVERSION;
payload += ",\"SC\":";
payload += sampleCount;

payload += ",\"IBI\":";
payload += IBI;
payload += ",\"BC\":";
payload += beatCount;
payload += "}}";

if (!!!client.connected()) {
  Serial.print("Reconnecting client to ");
  Serial.println(server);
  while (!!!client.connect(clientId)) {
    Serial.print(".");
    delay(500);
  }
  Serial.println();
}

Serial.print("Sending IOTPulse payload: ");
Serial.println(payload);

if (client.publish(topic, (char*) payload.c_str())) {
  Serial.println("IBM Cloud IOT Publish ok");
} else {
  Serial.println("IBM Cloud IOT Publish failed");
}
oldIOTTime = micros();
```

```
    // restart the pulse counter
    restartPulse();
  }

  //Serial.print("micros()=");
  //Serial.println(micros());

  if (newPulseDeltaTime > 2000)  // check for 2ms work to be done
  {

    //Serial.print("Pulse Delta time =");
    //Serial.println(newPulseDeltaTime);
    // do the work for pulse calculation
    timerCallback(NULL);

    oldPulseTime = micros();
  }

  yield(); //  take a break
}
```

AllserialHandling.h (Listing 5-3) contains the serial output debugging routines that are useful when modifying the code.

Listing 5-3. AllSerialHandling.h

```
//////////
/////////  All Serial Handling Code,
/////////  It's Changeable with the 'serialVisual' variable
/////////  Set it to 'true' or 'false' when it's declared at
           start of code.
/////////

void sendDataToSerial(char symbol, int data );

void serialOutput() {  // Decide How To Output Serial.
```

```
    sendDataToSerial('S', Signal);     // goes to
                                          sendDataToSerial
                                          function
}

//  Decides How To OutPut BPM and IBI Data
void serialOutputWhenBeatHappens() {
  if (serialData == true) {           //  Code to Make the
                                          Serial Monitor
                                          Visualizer Work
    Serial.print("*** Heart-Beat Happened *** ");  //ASCII Art
                                                     Madness
    Serial.print("BPM: ");
    Serial.print(BPM);
    Serial.print(" IBI: ");
    Serial.print(IBI);
    Serial.println("  ");
  }
}

//  Sends Data to Pulse Sensor Processing App, Native Mac App,
    or Third-party Serial Readers.
void sendDataToSerial(char symbol, int data ) {
  Serial.print(symbol);

  Serial.println(data);
}
```

Interrupt.h (Listing 5-4) is called by this name because the original Pulse Sensor software used an Arduino timer to generate a 2ms interrupt. As explained earlier, the ESP8266 is not currently capable of doing the same thing while maintaining a functional WiFi interface. The function timerCallback() in Interrupt.h is now called at a scheduled time by the loop() function in the IOTPulse.ino file.

Basically, timerCallback() acts as a detector and averaging filter that detects the upswing on the Pulse Sensor output and then builds a set of times for IBI (Interval Between beats) and provides an average. It adjusts to the recorded levels coming from the Pulse Sensor because the actual level is not as important as the time interval between adjusted pulses from the Pulse Sensor. The software acts as a peak and trough detector with an average system that removes high-frequency noise.

Listing 5-4. Interrupt.h

```
volatile int rate[10];                    // array to hold last
                                             ten IBI values
volatile unsigned long sampleCounter = 0;    // used to determine
                                                pulse timing
volatile unsigned long lastBeatTime = 0;      // used to find
                                                 IBI
volatile int P = 680;                       // used to find peak in
                                               pulse wave, seeded 512
volatile int T = 680;                         // used to find
                                                 trough in pulse
                                                 wave, seeded 512
volatile int thresh = 700;                  // used to find instant
                                               moment of heart beat,
                                               seeded 525
volatile int amp = 100;                       // used to hold
                                                 amplitude of pulse
                                                 waveform, seeded
volatile boolean firstBeat = true;            // used to seed
                                                 rate array so
                                                 we startup with
                                                 reasonable BPM
```

```
volatile boolean secondBeat = false;      // used to seed
                                             rate array so
                                             we startup with
                                             reasonable BPM

void restartPulse()
{

  sampleCounter = 0;          // used to determine pulse timing
  lastBeatTime = 0;           // used to find IBI
  P = 680;                    // used to find peak in pulse
                                 wave, seeded 512
  T = 680;                    // used to find trough in pulse
                                 wave, seeded 512
  thresh = 700;               // used to find instant moment of
                                 heart beat, seeded 525
  amp = 100;                  // used to hold amplitude of
                                 pulse waveform, seeded
  firstBeat = true;           // used to seed rate array so we
                                 startup with reasonable BPM
  secondBeat = false;

}
void timerCallback(void *pArg);

// Timer  makes sure that we take a reading every 2
milliseconds
void timerCallback(void *pArg) {         // triggered on
                                            interrupts
  //cli();                               // disable interrupts
                                            while we do this
  Signal = analogRead(pulsePin);         // read the Pulse
                                            Sensor
  //Serial.print("Signal-AO-:");
  //Serial.println(Signal);

  sampleCounter += 2;                       // keep track of the
                                               time in ms with
                                               this variable
  int N = sampleCounter - lastBeatTime;       // monitor the
                                                 time since the
                                                 last beat to
                                                 avoid noise

  //  find the peak and trough of the pulse wave
  if (Signal < thresh && N > (IBI / 5) * 3) { // avoid dicrotic
  noise by waiting 3/5 of last IBI
    if (Signal < T) {                         // T is the trough
      T = Signal;                             // keep track of
                                                 lowest point
                                                 in pulse wave
    }
  }

  if (Signal > thresh && Signal > P) {       // thresh condition
                                                helps avoid noise
    P = Signal;                               // P is the peak
  }                                           // keep track of
                                                 highest point
                                                 in pulse wave

  //  NOW IT'S TIME TO LOOK FOR THE HEART BEAT
  // signal surges up in value every time there is a pulse
  if (N > 250) {                             // avoid high
                                                frequency noise
```

```
if ( (Signal > thresh) && (Pulse == false) && (N >
(IBI / 5) * 3) ) {
  Pulse = true;                           // set the Pulse
                                             flag when we
                                             think there is
                                             a pulse
  digitalWrite(blinkPin, LOW);            // turn on  LED
  IBI = sampleCounter - lastBeatTime;     // measure time
                                             between beats
                                             in ms
  lastBeatTime = sampleCounter;          // keep track of time
                                            for next pulse

  if (secondBeat) {                      // if this is the
                                            second beat, if
                                            secondBeat == TRUE
    secondBeat = false;                   // clear
                                             secondBeat flag
    for (int i = 0; i <= 9; i++) {       // seed the running
                                            total to get a
                                            realistic BPM at
                                            startup
      rate[i] = IBI;
    }
  }

  if (firstBeat) {                       // if it's the first
                                            time we found a
                                            beat, if firstBeat
                                            == TRUE
    firstBeat = false;                    // clear firstBeat
                                             flag
    secondBeat = true;                   // set the second
                                            beat flag
   //sei();                            // enable interrupts
                                          again
    return;                              // IBI value is
                                            unreliable so
                                            discard it
  }

  // keep a running total of the last 10 IBI values
  word runningTotal = 0;                 // clear the
                                            runningTotal
                                            variable

  for (int i = 0; i <= 8; i++) {         // shift data in
                                            the rate array
    rate[i] = rate[i + 1];               // and drop the
                                            oldest IBI
                                            value
   runningTotal += rate[i];            // add up the 9
                                          oldest IBI values
  }

  rate[9] = IBI;                         // add the latest
                                            IBI to the rate
                                            array
  runningTotal += rate[9];               // add the
                                            latest IBI to
                                            runningTotal
  runningTotal /= 10;                    // average the
                                            last 10 IBI
                                            values
    BPM = 60000 / runningTotal;           // how many beats can
                                             fit into a minute?
                                             that's BPM!
    // reduce to 83% based on comparison
    BPM= (BPM * 83)/100;
    QS = true;                              // set Quantified
                                               Self flag
    // QS FLAG IS NOT CLEARED INSIDE THIS ISR
  }
}

if (Signal < thresh && Pulse == true) {  // when the values
                                            are going down,
                                            the beat is over
  digitalWrite(blinkPin, HIGH);          // turn off pin 13 LED
  Pulse = false;                          // reset the Pulse
                                            flag so we can do
                                            it again
  amp = P - T;                            // get amplitude of
                                            the pulse wave
  thresh = amp / 2 + T;                   // set thresh at 50%
                                            of the amplitude
  P = thresh;                             // reset these for
                                            next time
  T = thresh;
}

if (N > 2500) {                           // if 2.5 seconds go
                                            by without a beat
  thresh = 512;                          // set thresh default
  P = 512;                                // set P default
  T = 512;                                // set T default
```

```
    lastBeatTime = sampleCounter;          // bring the lastBeatTime
                                              up to date
    firstBeat = true;                         // set these to
                                                 avoid noise
    secondBeat = false;                       // when we get the
                                                 heartbeat back
  }
  //sei();                                    // enable interrupts
                                                 when you're done!
}// end isr
```

Reviewing the Arduino IDE Serial Monitor Results

Listing 5-5 shows the output from the Serial Monitor on the Arduino IDE as you run the software. Note that this code was run after joining the IBM Cloud IoT Quickstart as shown later in this chapter. If you haven't set up the IBM Quickstart yet, then you will still see that your data is published to Quickstart as there is no authentication required to contact the IBM Cloud Quickstart site.

Listing 5-5. Serial Monitor Output

```
IOTPulse IBM Cloud IOT
----------------
Connecting to gracie....
Local WiFi connected, IP address: 192.168.1.38
IOT Delta time =10000007
Reconnecting client to quickstart.messaging.internetofthings.
ibmcloud.com
.......
Sending IOTPulse payload: {"d":{"IOTPulse":"IP1","BPM":69,"VER"
:5,"SC":14,"IBI":1126,"BC":14}}
IBM Cloud IOT Publish ok
*** Heart-Beat Happened *** BPM: 113 IBI: 436
*** Heart-Beat Happened *** BPM: 115 IBI: 368
*** Heart-Beat Happened *** BPM: 117 IBI: 358
*** Heart-Beat Happened *** BPM: 122 IBI: 252
*** Heart-Beat Happened *** BPM: 128 IBI: 274
*** Heart-Beat Happened *** BPM: 126 IBI: 476
*** Heart-Beat Happened *** BPM: 118 IBI: 706
*** Heart-Beat Happened *** BPM: 119 IBI: 424
*** Heart-Beat Happened *** BPM: 115 IBI: 564
*** Heart-Beat Happened *** BPM: 107 IBI: 776
*** Heart-Beat Happened *** BPM: 106 IBI: 466
*** Heart-Beat Happened *** BPM: 86 IBI: 1430
*** Heart-Beat Happened *** BPM: 79 IBI: 860
IOT Delta time =10000001
Reconnecting client to quickstart.messaging.internetofthings.
ibmcloud.com
.......
Sending IOTPulse payload: {"d":{"IOTPulse":"IP1","BPM":79,"VER"
:5,"SC":15,"IBI":860,"BC":27}}
```

```
IBM Cloud IOT Publish ok
IOT Delta time =10000005
Sending IOTPulse payload: {"d":{"IOTPulse":"IP1","BPM":79,"VER"
:5,"SC":16,"IBI":860,"BC":27}}
IBM Cloud IOT Publish ok
IOT Delta time =10000009
Sending IOTPulse payload: {"d":{"IOTPulse":"IP1","BPM":79,"VER"
:5,"SC":17,"IBI":860,"BC":27}}
```

Figure 5-8 shows the completed IoTPulse device in the case. Note the holes on the top to view the LEDs on the ESP8266 board. Figure 5-9 shows the Pulse Sensor clipped on an ear. Notice the green LED used for sensing the pulse rate in the ear.

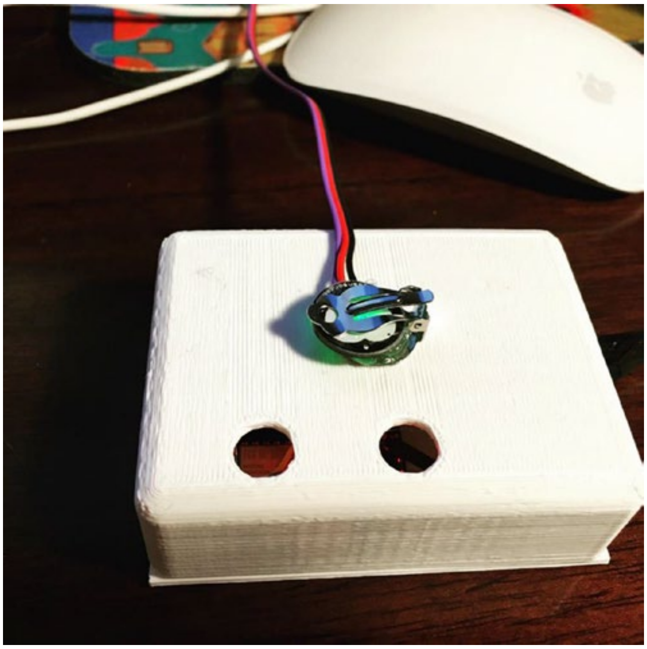


Figure 5-8. *IoTPulse in Case*

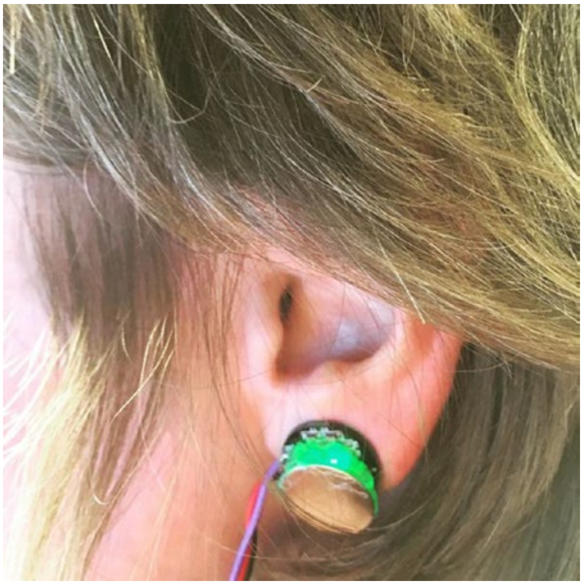


Figure 5-9. *IoTPulse Sensor on Ear*

Joining IBM Cloud and the IoT Foundation

The IBM Cloud and the IoT Foundation are the two key IoT cloud services we will be using to connect the IoTPulse device to the cloud. The tasks that need to be done to connect mostly have to do with setting up user accounts and putting billing information into the system. You can use the free levels of the IBM Cloud system, but IBM still requires billing information. Figure 5-10 describes the overall architecture of the Cloud/IoT Foundation cloud application.

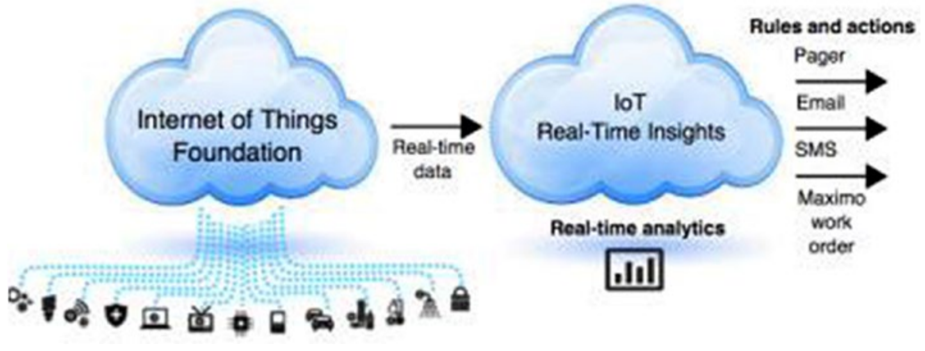


Figure 5-10. *IBM Cloud Block Diagram*

While I would recommend joining the free level of the IBM Cloud to explore the features available (and they are many and *awesome*!), we will use the IBM Cloud IoT Quickstart service.

Sending Your Data to Cloud

First of all, what do we send to the cloud from the ESP8266? It turns out the data protocols are pretty straightforward. There are two major protocols that require descriptions: MQTT and JSON.

MQTT

MQTT is a publish-subscribe-based "light weight" messaging protocol for use on top of the TCP/IP protocol, such as the WiFi packets that we are using in this project. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. Both of these conditions are met with an ESP8266 IoT design, so it makes sense to use. There is also an excellent library available for MQTT for the Arduino IDE [`https://github.com/knolleary/pubsubclient`]. The publish-subscribe messaging pattern requires a message broker. The broker is responsible for distributing messages to interested clients based on the topic of a message (Figure 5-11).

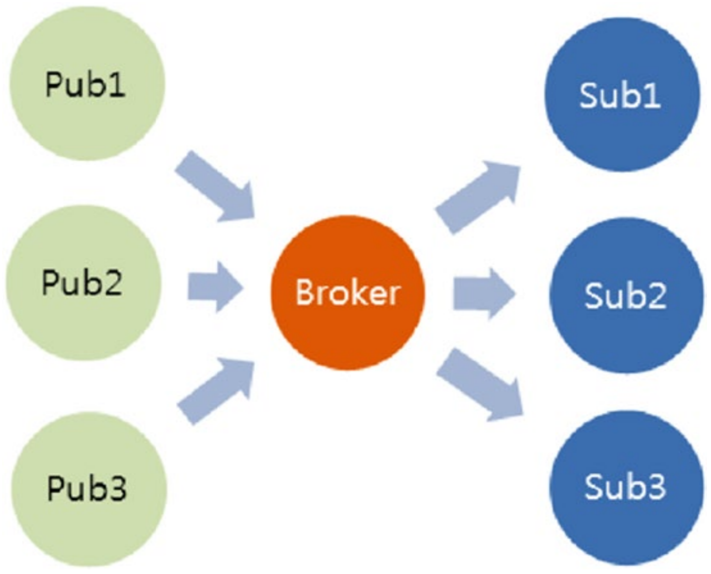


Figure 5-11. *MQTT Publish-Subscribe Protocol*

Publish-subscribe is a pattern where senders of messages, called publishers (in this case, our ESP8266 is the publisher), don't program the messages to be sent directly to subscribers, but instead characterize message payloads into classes without the specific knowledge of which subscribers the messages are sent to. Similarly, subscribers (the IBM Cloud IoT in this case) will only receive messages that are of interest without specific knowledge of which publishers there are. The IBM Cloud operates as the broker in this system and routes the published data to the appropriate subscribers inside of the IBM Cloud.

JSON Data Payload

JSON is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs. It is the primary data format used for asynchronous browser/server communication, largely replacing XML. XML is a "heavier" protocol that is also hierarchical in nature, but with a great deal more redundancy than JSON. Yes, there are class wars going on for people that advocate JSON over XML, but in today's world of higher-speed communication, it rarely matters. You can make the argument that the higher data density of JSON is a better choice for IoT applications.

Here is an example of the data packet we are using in the ESP8266 code in JSON for the LightSwarm data payload:

```
{"d":
      {
         "LightSwarm IOT":"LS1",
         "sampleCount":2118,
         "lightValue":383
      }
}
```

The IoTPulse JSON payload follows:

```
{"d":
{
      "IOTPulse":"IP1",
      "BPM":79,
      "VER":5,
      "SC":15,
      "IBI":860,
      "BC":27}
}
```

Note the BPM (Beats Per Minute) field in the preceding JSON payload. This is what we will be plotting on the IBM Cloud.

Examining Real-Time Data on the IBM Cloud IoT Platform

Now we are going to look at the data that will be sent by our application to the cloud. Building a graphical dashboard interface for this data on the IBM Cloud is possible, but it is beyond the scope of this chapter. See Figure 5-12 for an example of the kind of dashboard that can be built.

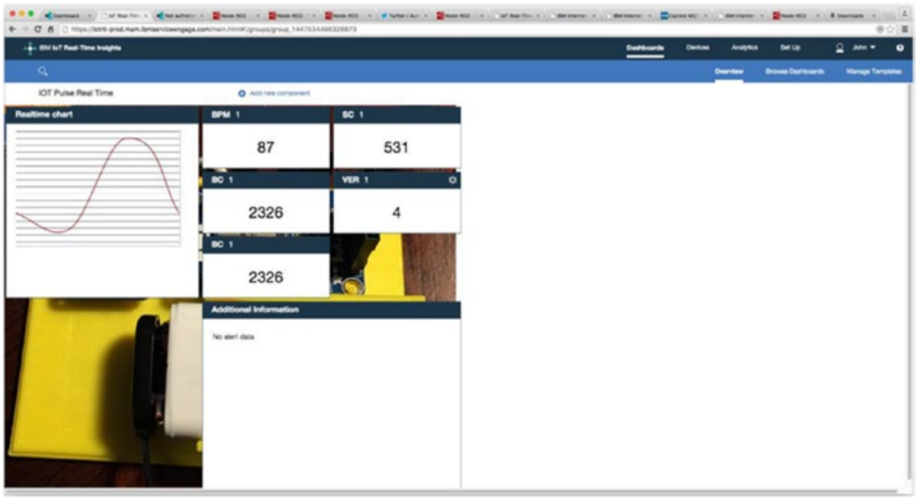
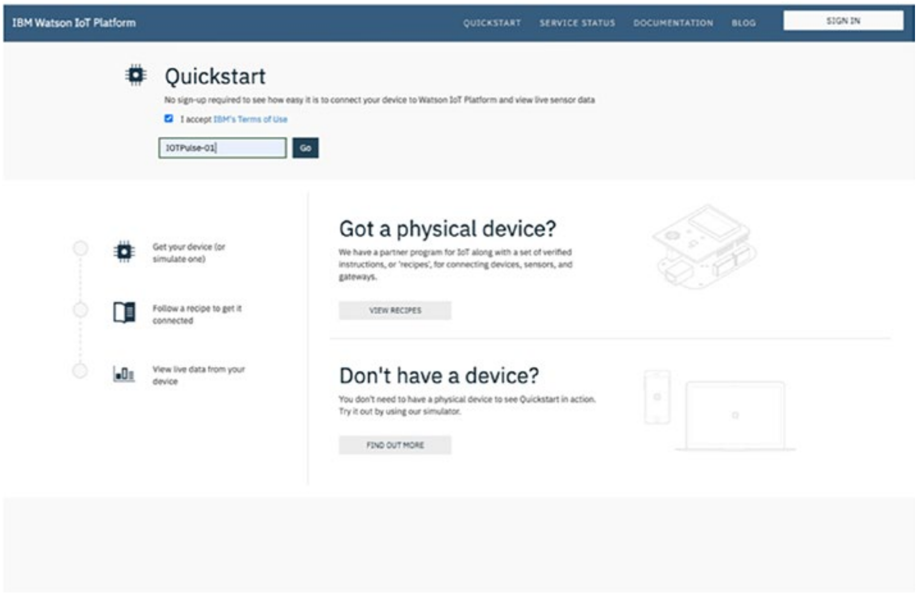


Figure 5-12. *IBM Cloud IoT Real-Time Dashboard Example*

You can set up and use a full IoT IBM Cloud account in “lite mode” for free. I would suggest you try this after you have gone through this entire chapter. However, for simplicity sake, I am going to use the “Quickstart” method of the IBM Cloud IoT Platform. It is much simpler to use and requires no authentication. You really can’t do much with it but see a plot of the first data variable in your JSON payload, which by design happens to be the heartbeat number we are looking for, the variable BPM.

1. Go to the URL: `https://quickstart.internetofthings.ibmcloud.com/`.

2. Type "IOTPulse-01" into the device ID, check the accept terms box, and click the Go button (see Figure 5-13).



Quickstart

No sign-up required to see how easy it is to connect your device to Watson IoT Platform and view live sensor data

☑ I accept IBM's Terms of Use

IOTPulse-01

Get your device (or simulate one)

Follow a recipe to get it connected

View live data from your device

Got a physical device?

We have a partner program for IoT along with a set of verified instructions, or 'recipes', for connecting devices, sensors, and gateways.

Don't have a device?

You don't need to have a physical device to see Quickstart in action. Try it out by using our simulator.

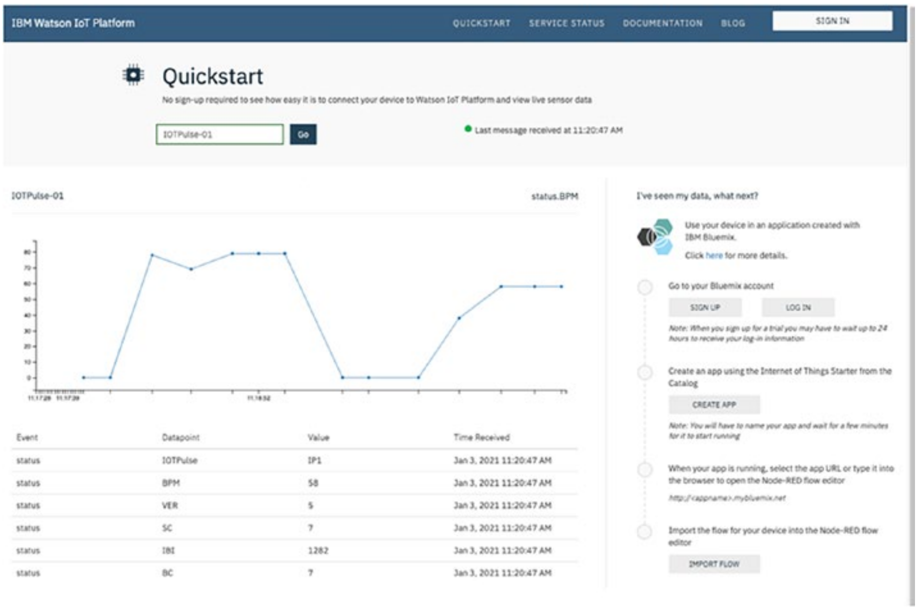
Figure 5-13. *IBM Cloud IoT Quickstart Site*

Make sure your IoTPulse program is running (the program has already been set up to connect to the IBM Cloud IoT Quickstart server).

Now Click Go! After a few minutes, you will see a chart like this of the Beats Per Minute (BPM) being sent to the IBM Cloud. You are connected. Figure 5-14 shows you the results.

Advanced Topics

The IBM Cloud offerings have literally hundreds of options and possibilities for bringing your IoT data to the cloud. In the following sections, we show a few of these "apps" available on the IBM Cloud system. We will show how it is possible to graph your historical IoT data and a brief introduction to a complex, but very powerful, programming system, Node-RED.



Quickstart

No sign-up required to see how easy it is to connect your device to Watson IoT Platform and view live sensor data

IOTPulse-01

Last message received at 11:20:47 AM

IOTPulse-01 status.BPM

Event	Datapoint	Value	Time Received
status	IOTPulse	IP1	Jan 3, 2021 11:20:47 AM
status	BPM	58	Jan 3, 2021 11:20:47 AM
status	VER	5	Jan 3, 2021 11:20:47 AM
status	SC	7	Jan 3, 2021 11:20:47 AM
status	IBI	1282	Jan 3, 2021 11:20:47 AM
status	BC	7	Jan 3, 2021 11:20:47 AM

I've seen my data, what next?

Use your device in an application created with IBM Bluemix.

Click here for more details.

Go to your Bluemix account

Note: When you sign up for a trial you may have to wait up to 24 hours to receive your log-in information

Create an app using the Internet of Things Starter from the Catalog

Note: You will have to name your app and wait for a few minutes for it to start running

When your app is running, select the app URL or type it into the browser to open the Node-RED flow editor

http://<appname>.mybluemix.net

Import the flow for your device into the Node-RED flow editor

Figure 5-14. *IoTPulse Data on the IBM Cloud*

Historical Data

Non-real-time historical data can also be viewed in the IBM Cloud (Figure 5-15). This requires you to build an app (not as complex as you might think) and then deploy the app. A good tutorial for this is located on the IBM Cloud website [`www.ng.Cloud.net/docs/starters/install_cli.html`]. It does require you to install a command-line interface for Cloud Foundry on your computer.

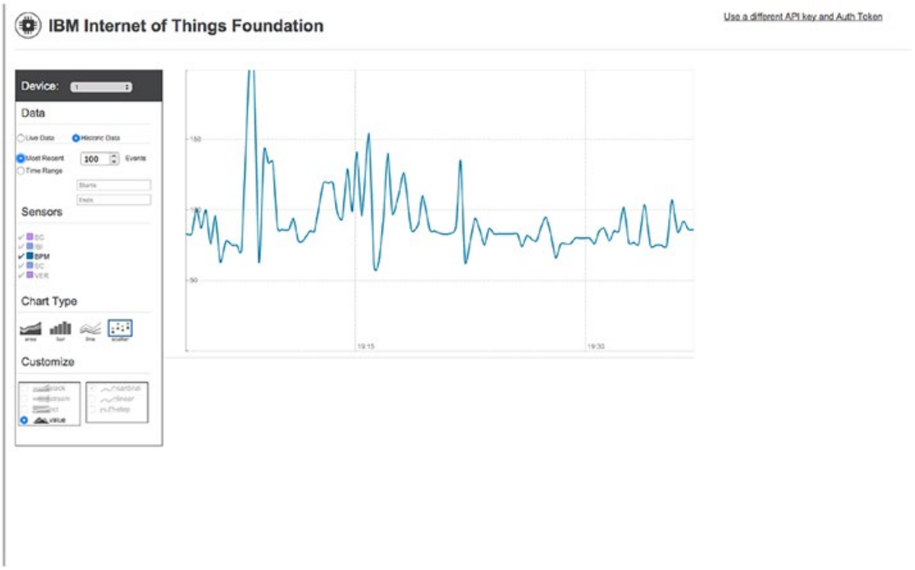


Figure 5-15. *Historical IoTPulse Data*

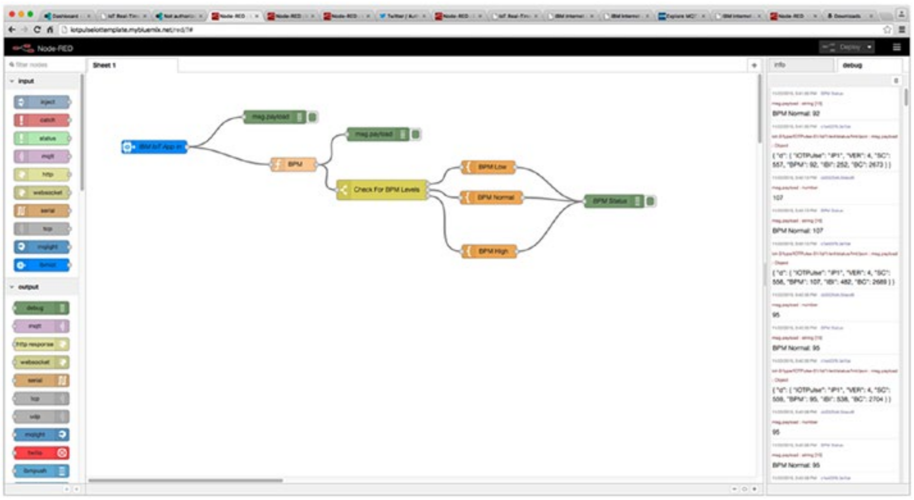
One thing this historical data graph of our pulse rate does show is that the data from IoTPulse could be improved by more filtering. The average of the heart rate looks good, but the jumps up and down look like noise, especially earlier in the graph.

Node-RED Applications

Node-RED is a graphical tool for wiring together devices, external APIs, and online services. You use a browser-based flow editor that makes it simple to wire together flows from an extensive library. And it is easy to add your own nodes.

Node-RED is fully supported by the IBM Cloud. It is primarily an event-based software generation system. If an event happens (such as IoTPulse sending data), you can take that data and process it in various ways, send it to other applications, and store it in databases. Other inputs can cause displays to be updated, queries to be sent to databases, and many other events.

For an example, I built a small Node-RED application for IoTPulse IoT device shown in Figure 5-16.



```
BPM Normal: 92
{ "d": { "IOTPulse": "IP1", "VER": 4, "SC": 557, "BPM": 92, "IBI": 252, "BC": 2673 } }
107
BPM Normal: 107
{ "d": { "IOTPulse": "IP1", "VER": 4, "SC": 558, "BPM": 107, "IBI": 482, "BC": 2689 } }
95
BPM Normal: 95
{ "d": { "IOTPulse": "IP1", "VER": 4, "SC": 558, "BPM": 95, "IBI": 538, "BC": 2704 } }
95
BPM Normal: 95
```

Figure 5-16. *Node-RED Application Flow for IoTPulse*

Watson Applications

One of the more exciting class of applications that can be built in the IBM Cloud are those utilizing the IBM Watson software. All of the Watson services on the IBM Cloud Platform are accessed via HTTP transfers known as REST services. REST stands for Representational State Transfer and is a common way of accessing web-based services.

Watson supports a number of very sophisticated services all available to your device and applications in the IBM Cloud:

- Speech to text
- Text to speech
- Language translation
- Support for user dialogs

- Determining the “tone” for text
- Visual recognition
- Image analysis – Objects, people, and text

Conclusion

The future of the IoT is data: how to gather it, how to transmit it, how to analyze and display it, and how to act on it. The IoTPulse project shows how to gather data and how to send it to a cloud computing system for analysis and actions.

With a proper sensor, the IoTPulse project could be a commercial product. The sensor and analysis of the heart rate are definitely not medical grade, but still illustrate the principles of the IoT device and the device relationship with the IoT cloud.

CHAPTER 6

Using IoT for RFID and MQTT and the Raspberry Pi

Chapter Goal: Build an IoT Device for Reading RFID Tags on Various Packages

Topics Covered in This Chapter:

- Introduction to RFID
- Introduction to MQTT publish/subscribe systems and servers
- Building a Raspberry Pi–based MQTT server
- Building an RFID Reader to connect to the IoT
- Connecting your RFID Reader to your MQTT Raspberry Pi server
- What to do with the RFID data on the server

The world of IoT is dominated by small computers having small amounts of resources and power to communicate information with the rest of the world. MQTT is a publish-subscribe, lightweight protocol for

IoT devices to communicate with the server and through the server, with each other. MQTT is also available for “mesh”-type networks, like Zigbee, that allow device-to-device communication. RFID (Radio Frequency IDentification) is an inexpensive way of reading uniquely numbered cards and tags via radio waves from short distances. RFID and its cousin, NFC (Near Field Communications), are showing up in more and more applications.

In this chapter, we build an RFID device and then connect through an MQTT publisher to an MQTT broker as in Chapter 5, but with the difference that we are replacing the IBM Bluemix MQTT broker with a Mosquitto broker running on a $20 Raspberry Pi.

IoT Characterization of This Project

As we discussed in Chapter 1, the first thing to do to understand an IoT project is to look at our six different aspects of IoT. IoTRFID is a simpler project than our other four projects, but the dataflow is much more complex.

Table 6-1 characterizes different aspects of the project. Ratings are from 1 to 10, 1 being the least suitable for IoT and 10 being the most suitable for IoT applications. This gives us a CPLPFC rating of 8.3. Great for learning and experimenting, and it could be deployed for some market applications.

***Table 6-1.** IoT Characterization of the Project*

Aspect	Rating	Comments
Communications	9	WiFi connection to the Internet – can do ad hoc mesh-type communication
Processor power	9	80MHz XTensa Harvard Architecture CPU, ~80KB data RAM/~35KB of instruction RAM/200K ROM
Local storage	8	4MB Flash (or 3MB file system!)
Power consumption	8	~200mA transmitting, ~60mA receiving, no WiFi ~15mA, standby ~1mA
Functionality	7	Partial Arduino support (limited GPIO/analog inputs)
Cost	9	< $12 and getting cheaper

Note that the power consumption in this application could be dramatically reduced by adding a power down mode, say, after 5 minutes of not using the RFID Reader.

What Is RFID Technology?

Radio Frequency IDentification (RFID) is the use of radio waves to transfer data, specifically in the area of identifying and tracking tags attached to objects, people, and animals. Often the RFID Tag has no power supply and is activated and powered by radio waves beamed at the RFID Tag. This technology goes back to the 1940s and was first demonstrated by Theremin in the Soviet Union. Interestingly enough, he used this technology to put a covert listening device in the US Embassy conference room in 1946. It was used for six years when, in 1952, it was discovered that the seal contained a microphone and a resonant cavity that could be stimulated from an outside radio signal. While this used a resonant cavity that changed its shape when stimulated by sound waves, and hence modulated a reflected radio wave beamed at the unit, it is considered a predecessor to RFID technology.

`[en.wikipedia.org/wiki/The_Thing_(listening_device)]`

RFID tags can be active, passive (no battery power – we are using this technology in this chapter), and battery-assisted passive. A passive tag contains no battery, instead using the radio energy transmitted by the RFID Reader. To operate a passive tag, it needs to be hit with a power level roughly 1000 times greater than the resulting signal transmissions.

When an RFID Tag is hit by a beam of radio waves from a transmitter, the passive tag is powered up and transmits back the tag identification number and other information. This may be a unique serial number, a stock or lot number, or other specific information. Since tags have unique numbers, the RFID system can read multiple tags simultaneously, with some clever programming of the receiver and transmitter.

There are a number of different RFID standards in common use. There are three standards for putting ID chips in pets, which of course are not compatible with each other and require different readers. The short-range tags we are using in this chapter operated at the low frequency of 125kHz and can go up to 100mm (Figure 6-1). Other tags can be picked up (with active RFID technology) up to 100 meters.

Figure 6-1. *125kHz Card-Shaped RFID Tag*

While RFID and NFC technologies are designed for short distance use, the addition of a large power transmitter and a large antenna can change the meaning of "short distance."

What Is MQTT?

In Chapter 5, we used an MQTT publisher module to talk to the IBM MQTT broker at IBM Bluemix.

To refresh the reader's memory, MQTT is a publish-subscribe-based "light weight" messaging protocol for use on top of the TCP/IP protocol, such as the WiFi packets that we are using in this project. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.

Publish-subscribe is a pattern where senders of messages, called publishers (in this case, our project IoTRFID is the publisher), don't program the messages to be sent directly to subscribers, but instead characterize message payloads into classes.

You can think of it as writing stories for a newspaper where you don't know who will be subscribing to the article.

Hardware Used for IoTRFID

There are three major parts of the hardware from the IoTRFID (Figure 6-2). The first is the ESP8266 Adafruit Huzzah that we have seen in previous chapters. The second piece is the RFID reader and Antenna, while the third is the 125kHz RFID tag itself.

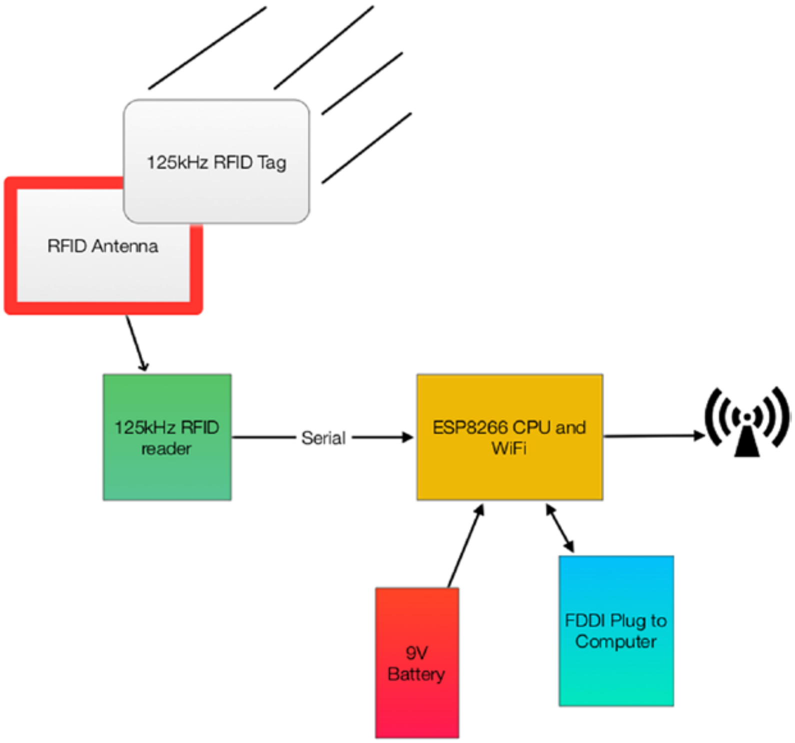


Figure 6-2. *Block Diagram of IoTRFID*

The RFID reading circuit is a small inexpensive board from Seeedstudio in China. The key reason this board was chosen was the fact it works on 3.3V (which is the same as the ESP8266) and requires no external components (well, yes, it does require an antenna that is included).

We also purchased a 125kHz RFID Tag from SparkFun. Be careful of the RFID Tag you buy. Not all are compatible.

Building an MQTT Server on a Raspberry Pi

As described in Chapter 5, MQTT requires a message broker. The design pattern is this:

1. IoTRFID publishes a payload.
2. Raspberry Pi MQTT broker receives the payload.
3. Raspberry Pi MQTT broker disseminates to all subscribing objects (which may be different processes in the same machine – as in our case – or even different machines entirely).

In Chapter 5, we used the IBM Bluemix as the message broker. In the IoTRFID project, we are going to build a message broker on the Raspberry Pi.

Figure 6-3 shows the dataflow in the entire application – including the Raspberry Pi. The dataflow in this application is basically one way: from the RFID Tag to the Raspberry Pi Mosquitto server. Note, however, there are two channels available over the WiFi, and you could send commands back to the IoTRFID project telling the user information about the part being inventoried. For example, you could display the destination for the box or the expiration date.

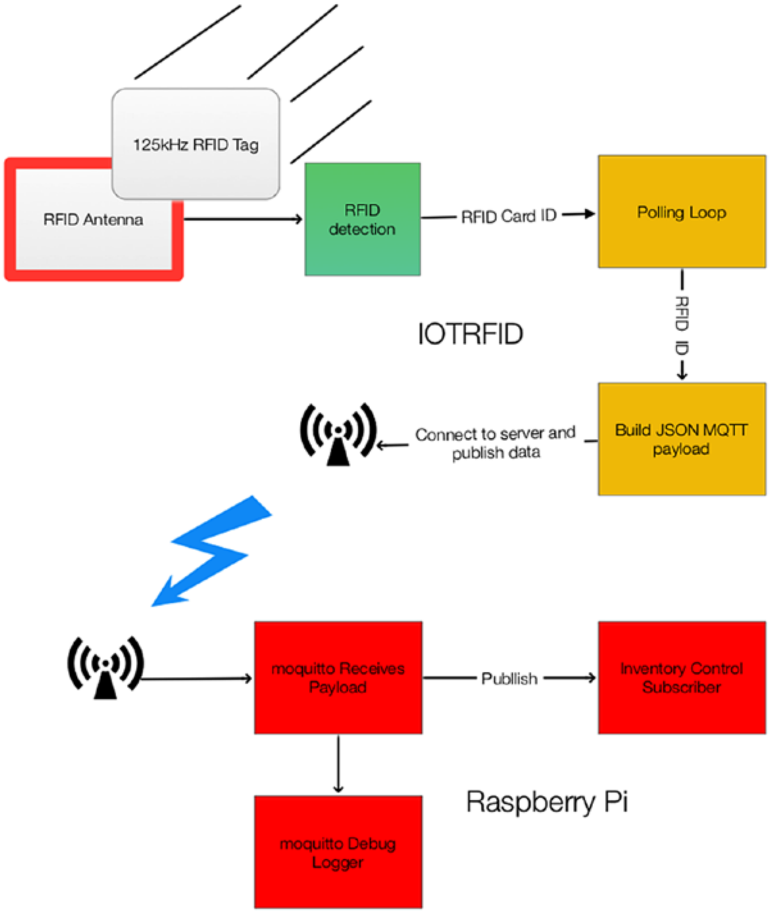


Figure 6-3. *Dataflow of Complete System*

There are a number of MQTT brokers available for different machines. For this project, we have selected one of the most popular and stable brokers, “Mosquitto.” Note the two “t”’s in Mosquitto. The bane of spell checkers everywhere.

Mosquitto supports MQTT v3.1/3.1.1 and is easily installed on the Raspberry Pi and somewhat less easy to configure. Next we step through installing and configuring the Mosquitto broker.

The Software on the Raspberry Pi

We start by setting up the software on the Raspberry Pi. We do this so our IoTRFID has something to talk to when we set turn the IoTRFID on.

Installing the MQTT "Mosquitto"

Unfortunately, the Raspberry Pi normal "apt-get" archives do not contain the latest version of the Mosquitto software. If you don't install the latest version of the broker, you will get odd errors (because of version compatibility errors) and it will not work. So, the first thing to do is to open a terminal window (or log in using ssh to your Raspberry Pi) and type the following:

```
sudo wget http://repo.mosquitto.org/debian/mosquitto-repo.gpg.
key
sudo apt-key add mosquitto-repo.gpg.key
cd /etc/apt/sources.list.d/
sudo wget http://repo.mosquitto.org/debian/mosquitto-wheezy.
list
sudo apt-get update
sudo apt-get install mosquitto
```

Next we can install the three parts of Mosquitto proper:

- mosquitto – The MQTT broker (or, in other words, a server)
- mosquitto-clients – Command-line clients, very useful in debugging
- python-mosquitto – The Python language bindings

Execute the following command to install these three parts:

```
sudo apt-get install mosquitto mosquitto-clients python-
mosquitto
sudo apt-get install python-pip
sudo pip install paho-mqtt
```

As is the case with most packages from Debian, the broker is immediately started. Since we have to configure it first, stop it:

```
sudo /etc/init.d/mosquitto stop
```

Configuring and Starting the Mosquitto Server

Before using Mosquitto, we need to set up the configuration file. The configuration file is located at /etc/mosquitto.

Open the file as follows:

```
sudo nano /etc/mosquitto/mosquitto.conf
```

You should see the following:

```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d
```

Change the "log_dest" line to the following:

```
log_dest topic
```

This puts the logging information as a "topic" so we can subscribe to it later on to see what is going on in our IoTRFID system.

Next add the following six lines:

```
log_type error
log_type warning
log_type notice
log_type information

connection_messages true
log_timestamp true
```

Now your /etc/mosquitto.conf files should look like this:

```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest topic
log_type error
log_type warning
log_type notice
log_type information

connection_messages true
log_timestamp true

include_dir /etc/mosquitto/conf.d
```

Starting the Mosquitto Server

Now start the Mosquitto server:

```
sudo /etc/init.d/mosquitto start
```

The server should start, and you are ready to move on to testing.

Testing the Mosquitto Server

Open up two more terminal windows.

In Terminal window 1, type the following:

```
mosquitto_sub -d -t hello/world
```

In Terminal window 2, type the following:

```
mosquitto_pub -d -t hello/world -m "Hello from Terminal window 2!"
```

When you have done the second statement, you should see this in the Terminal 1 window:

```
~ $ sudo mosquitto_sub -d -t hello/world
Client mosqsub/3014-LightSwarm sending CONNECT
Client mosqsub/3014-LightSwarm received CONNACK
Client mosqsub/3014-LightSwarm sending SUBSCRIBE (Mid: 1,
Topic: hello/world, QoS: 0)
Client mosqsub/3014-LightSwarm received SUBACK
Subscribed (mid: 1): 0
```

```
Client mosqsub/3014-LightSwarm received PUBLISH (d0, q0, r0,
m0, 'hello/world', ... (32 bytes))
Greetings from Terminal window 2
```

Now you are running the Mosquitto broker successfully.

Next, let's build the IoTRFID device.

Building the IoTRFID

The IoTRFID project consists of four major parts:

- ESP8266
- RFID Reader
- Software for the ESP8266
- Software for the Raspberry Pi

The purpose of this project is to prototype an inventory control system that uses RFID tags. The ESP8266 controls the RFID Reader and reports the RFID Tag to the Raspberry Pi server. We then use MQTT through the WiFi interface on the ESP8266 to send the inventory information to the Raspberry Pi. The Raspberry Pi could then use the RFID information to work with a database, alert the end customer, and so on.

The Parts Needed

The IoTRFID can be assembled for about $35 from the sources in Table 6-2.

Table 6-2. *Parts List for IoTRFID*

Part Number	Count	Description	Approximate Cost per Board	Source
ESP8266 Huzzah board	1	CPU/WiFi board	$18	`https://amzn.to/34VEgxJ`
Mini 125kHz RFID Reader	1	125kHz RFID Serial Reader	$11	`www.seeedstudio.com/depot/Mini-125Khz-RFID-Module-External-LEDBuzzer-Port-70mm-Reading-Distance-p-1724.html`
125kHz RFID tags	1	RFID Tag compatible with Mini 125kHz RFID Reader	$2	`www.sparkfun.com/products/8310`
FTDI cable	1	Cable for programming the ESP8266 from PC/Mac	$20	`https://amzn.to/3pOvxp9`

Installing Arduino Support on the PC or Mac

Once again, the key to making this project work is the development software. While there are many ways of programming the ESP8266 (MicroPython [`https://micropython.org`], NodeMCU Lua interpreter [`http://nodemcu.com`], and the Arduino IDE (Integrated Development Environment) [`www.arduino.cc/en/Main/Software`]), we chose the Arduino IDE for its flexibility and the large number of sensors and device libraries available.

To install the Arduino IDE, you need to do the following:

1. Download the Arduino IDE package for your computer and install the software [`www.arduino.cc/en/Guide/HomePage`].
2. Download the ESP libraries so you can use the Arduino IDE with the ESP breakout board. Adafruit has an excellent tutorial for installing the ESP8266 support for the Arduino IDE [`https://learn.adafruit.com/adafruit-huzzah-esp8266-breakout/using-arduino-ide`].

The Hardware

The main pieces of hardware in the swarm device are these:

- ESP8266 – CPU/WiFi interface
- Mini 125kHz RFID Reader
- 9V battery – Power
- FTDI cable – Programming and power

The ESP8266 communicates with the Raspberry Pi by using the WiFi interface. The ESP8266 uses a serial interface to communicate with the light sensor. The WiFi is a standard that is very common. The serial interface used is one wire (Tx line on the RFID Reader to Rx on the ESP8266 – Pin GPIO #4 on the ESP8266). In a serial interface, one bit at a time is sent along with stop and start bits for each byte. The data is sent at 9600 baud, which is an old way of saying about 9600 bits per second.

What Is This Sensor We Are Using?

The Mini 125kHz RFID Reader is a simple, inexpensive 125kHz RFID Tag reader with a range of about 70mm. Note that the range for a reader like this is set by a combination of the output power of the reader and the antenna design.

If you have a bigger antenna and more power, you can read RFID tags at a much larger distance. A quick check on the Web showed the record for passive 125kHz RFID tags is around 10 or 15 meters. Better antennas that are focused on a beam could do even better than this. Don't assume that someone cannot read the cards in your wallet! It's just a matter of antenna size. Tin foil will block RF signals if you want to wrap your cards.

3D Printed Case

The 3D printed case for this project has a holder for a 9V battery, pylons to hold the antenna in place, a slot to mount the RFID Reader board, and pylons for the ESP8266. The code is for OpenSCAD, a free 3D CAD system that appeals to programmers. Figure 6-4 shows the 3D printed case with pylons for mounting the boards.

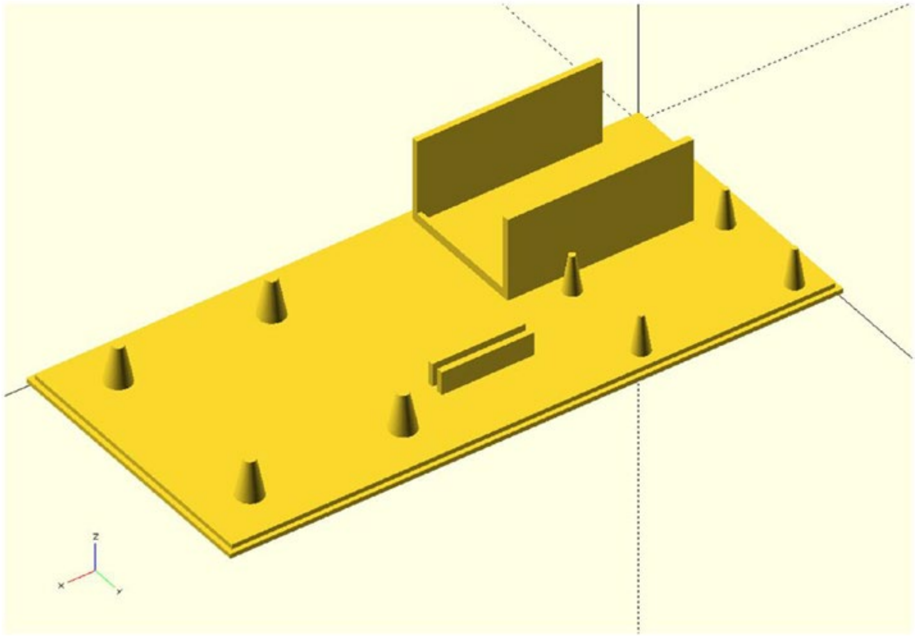


Figure 6-4. *3D Printed Case for the IoTRFID Project*

The following OpenSCAD code builds the mounting base step by step by merging basic shapes such as cubes and cones. We also build a stand to keep the battery from sliding across the base.

```
//
// IOT IOTRFID Mounting Base
//
// SwitchDoc Labs
// February 2016
//

union()
{

    cube([130,60,2]);
    translate([-1,-1,0])
    cube([132,62,1]);
```

```
// Mount for Battery

translate([-30,0,0])
union ()
{
translate([40,2,0])
cube([40,1.35,20]);
translate([40,26.10+3.3,0])
cube([40,1.5,20]);

// lips for battery
translate([79,2,0])
cube([1,28,4]);

// pylons for ESP8266

translate([70-1.0,35,0])
cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
translate([70-1.0,56,0])
cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
translate([70-34,35,0])
cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
translate([70-34,56,0])
cylinder(h=10,r1=2.2, r2=1.35/2, $fn=100);
}

// stand for board

translate([15,40,0])
union ()
{
translate([40,2,0])
cube([20,1.35,7]);
translate([40,3.55+1.35,0])
```

```
    cube([20,1.35,7]);
    }
    // pylons for RFID  board

    translate([50,0,0])
    union()
    {
    translate([33+36.0,10,0])
    cylinder(h=10,r1=3.2, r2=2.40/2, $fn=100);
    translate([33+36.0,50,0])
    cylinder(h=10,r1=3.2, r2=2.40/2, $fn=100);
    translate([36,10,0])
    cylinder(h=10,r1=3.2, r2=2.40/2, $fn=100);
    translate([36,50,0])
    cylinder(h=10,r1=3.2, r2=2.40/2, $fn=100);
    }

}
```

These files and the STL file are available at github.com/switchdoclabs/SDL_STL_IOTRFID.

Type the following in a terminal window on your Raspberry Pi:

```
git clone https://github.com/switchdoclabs/SDL_STL_IOTRFID.git
```

The Full Wiring List

Figure 6-5 shows the back of the Mini 125kHz RFID Reader to clearly display the pin labels on the back of the board.

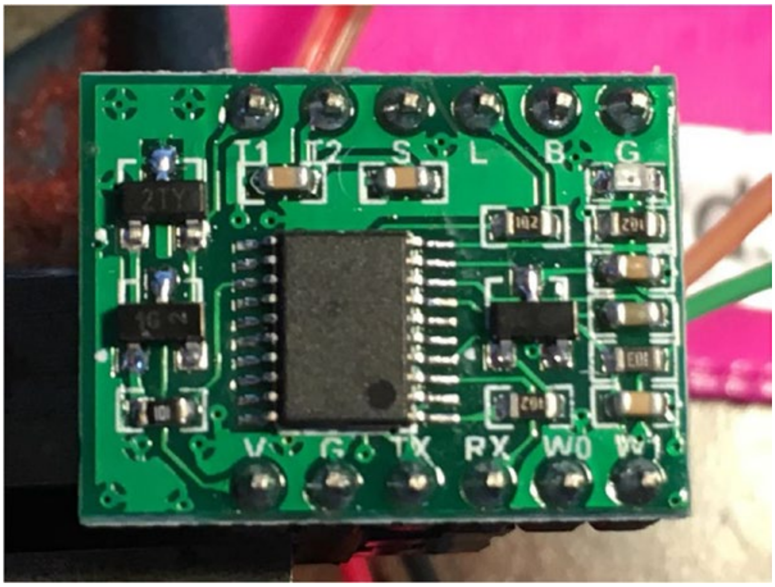


Figure 6-5. *Closeup of Mini RFID Board*

The following is the complete wiring list for the IoTRFID project. As you wire it, check off each wire for accuracy.

Table 6-3 contains all the wiring information for building the IoTRFID. The key for Table 6-3 is given in the following. Follow it closely to make your project work the first time. This table contains all of the individual wiring connections to complete the project.

ESP8266 Huzzah board:	**ESP8266**
Mini 125kHz RFID Reader:	**RFIDBoard**
9V battery:	**9VBat**

***Table 6-3.** Wiring List for the IoTRFID Project*

ESP8266 Huzzah board (ESP8266)		
From	**To**	**Description**
ESP8266/GND	RFIDBoard/G	Ground for RFID Reader board
ESP8266/3V	RFIDBoard/V	3.3V power for RFID Board
ESP8266/#5	RFIDBoard/Rx	RFID Board Receiver (not used in this project)
ESP8266/#4	RFIDBoard/Tx	RFID Board Serial Transmitter
ESP8266/GND	9VBat/“-” terminal (minus terminal)	Ground for battery
ESP8266/VBat	9VBat/“+” terminal (plus 9V)	9V from battery
RFIDBoard/T1	RFID Antenna JST2 Plug	Lead from antenna – push plug over T1 and T2 – order doesn’t matter
RFIDBoard/T2	RFID Antenna JST2 Plug	Lead from antenna – push plug over T1 and T2 – order doesn’t matter

Figure 6-6 shows the JST2 Plug pushed over the two pins on the RFID Board connecting the RFID Board with the antenna.

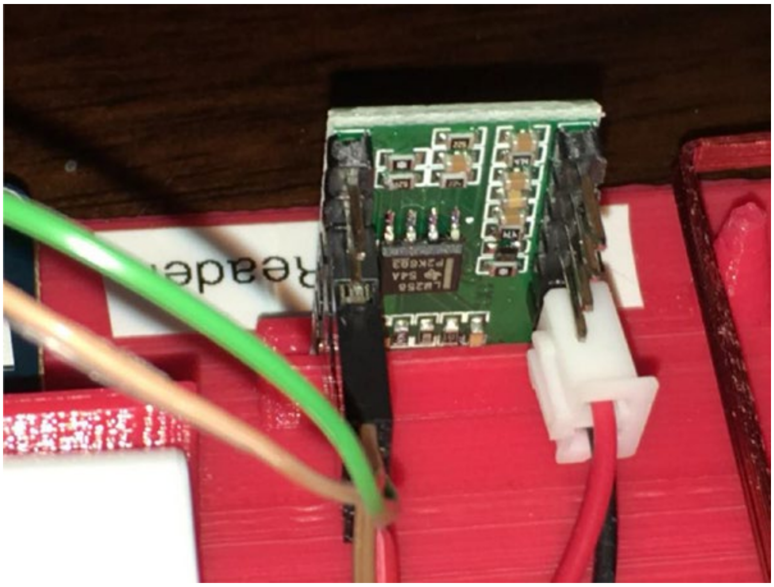


Figure 6-6. *Closeup of RFID Antenna JST2 Plug*

Your completed IoTRFID project should look similar to Figure 6-7.

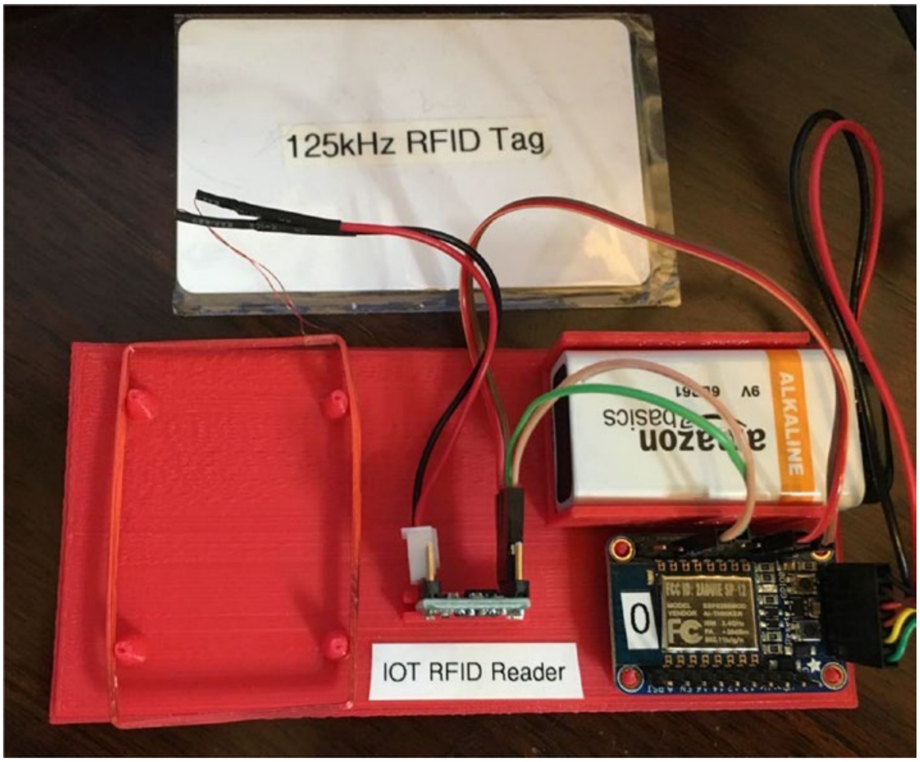


Figure 6-7. *The Completed IoTRFID Project*

The Software for the IoTRFID Project

No computer-based project is complete without the software to make the computer and board perform the functions designed. The main code IoTRFID is very short and makes good use of existing libraries.

The Libraries

In this project, we are using two libraries. Because we didn't have to modify the libraries to make them work with the ESP8266, we have not reproduced them here. The two libraries are the following:

> seeedRFID – The library for interfacing the ESP8266 to the RFID Reader. Basically a simple shell around the SoftSerial Arduino libraries [`https://github.com/Seeed-Studio/RFID_Library`].

> PubSubClient – A simple client for MQTT. This is a very usable and well-documented library. We use this library to talk to the MQTT server (Mosquitto) on the Raspberry Pi [`http://pubsubclient.knolleary.net`].

The Main Software

The ESP8266 Arduino IDE software for this project is relatively straightforward. You will see some similarities to the last chapter. The main difference is that you are now doing both ends of the MQTT connections. Remember to put your own WiFi access point and password in the code as well as enter the IP address of your Raspberry Pi.

The general flow of the software consists of first initializing the WiFi connection and the RFID hardware and then going into a loop, checking for RFID Tag events.

You can download the IoTRFID from github.com/switchdoclabs/SDL_ESP8266_IOTRFID by using this command in a terminal window:

```
git clone https://github.com/switchdoclabs/SDL_ESP8266_IOTRFID.
git

/*
    SwitchDoc Labs Code for IOT RFID
    IOT RFID uses publish subscribe to communicate to
    Raspberry Pi
    January 2020
*/

// BOF preprocessor bug prevent - insert on top of your
arduino-code
#if 1
__asm volatile ("nop");
#endif

// Board options

#pragma GCC diagnostic ignored "-write-strings"

extern "C" {
#include "user_interface.h"
}
#include <ESP8266WiFi.h>
#include "PubSubClient.h"

#include "seeedRFID.h"

#define RFID_RX_PIN 4
#define RFID_TX_PIN 5

#undef TEST

SeeedRFID RFID(RFID_RX_PIN, RFID_TX_PIN);
RFIDdata tag;
```

```
int count = 0;   // counter for buffer array

//  Variables


int blinkPin = 0;                // pin to blink led at each
                                    reception of RFID code

#include "Utils.h"

//--------------------------------------------------------------
//Local WiFi Variables

const char* ssid = "YOURWIFIACCESSPOINT";
const char* password = "YOURWIFIPASSWORD";

#define IOTRFIDVERSION 005
// Raspberry Pi Information

#define ORG "switchdoc"
#define DEVICE_TYPE "IOTRFID-01"
#define DEVICE_ID "1"
#define TOKEN "ul!fjH!y8y0gDREmsA"

// setup for IOT Raspberry Pi

char server[] = "192.168.1.40";  // Replace with YOUR RASPBERRY
                                    IP Number
char topic[] = "IOTRFID";
char authMethod[] = "use-token-auth";
char token[] = TOKEN;
char clientId[] = "IOTRFID";

void callback(char* topic, byte* payload, unsigned int length)
{
  Serial.println("callback invoked from IOT RFID");
}
```

```
WiFiClient wifiClient;
PubSubClient client(server, 1883, callback, wifiClient);

void setup() {
  // put your setup code here, to run once:

  pinMode(0, OUTPUT);

  Serial.begin(9600);             // we agree to talk fast!

  Serial.println("----------------");
  Serial.println("IOTRFID publish/subscribe Inventory");
  Serial.println("----------------");

  // signal start of code - three quick blinks
  blinkLED(3, 250);

  Serial.print("Connecting to WiFi ");

  if (strcmp (WiFi.SSID().c_str(), ssid) != 0) {
    WiFi.begin(ssid, password);
  }
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");

  Serial.print("Local WiFi connected, IP address: ");
  Serial.println(WiFi.localIP());

  blinkLED(5, 500);
}
```

```
void loop() {
  // put your main code here, to run repeatedly:

  count = 0;

  if (!!!client.connected()) {
    Serial.print("Reconnecting client to ");
    Serial.println(server);
while (!!!client.connect(clientId)) {
      Serial.print(".");
      delay(500);
    }
    Serial.println();
  }

  // Check for RFID available

  String payload;
  if (RFID.isAvailable())
  {
    tag = RFID.data();
    Serial.print("RFID card number read: ");
    Serial.println(RFID.cardNumber());
#ifdef TEST
    Serial.print("RFID raw data: ");
    for (int i = 0; i < tag.dataLen; i++) {
      Serial.print(tag.raw[i], HEX);
      Serial.print('\t');
    }
#endif

    // Sending payload

    payload = "{\"d\":{\"IOTRFID\":\"IR1\",";
    payload += "\"VER\":\"";
    payload += IOTRFIDVERSION;
    payload += "\",\"RFID_ID\":\"";
    payload += String(RFID.cardNumber());
    payload += "\"";
    payload += "}}";

    // check for message

    Serial.println(payload.length());

    count = 0;

    if (payload.length() >= 53) // good message
    {
      Serial.print("Sending IOTRFID payload: ");
      Serial.println(payload);

      if (client.publish(topic, (char*) payload.c_str())) {
        Serial.println("IOTRFID Publish ok");
        blinkLED(1, 500);
      } else {
        Serial.println("IOTRFID Publish failed");
        blinkLED(2, 500);
      }
    }
    else
    {
      delay(500);
    }
  }

  yield();   // This is necessary for the ESP8266 to do the
                 background tasks
}
```

Testing the IoTRFID System

Using the Arduino IDE, flash the ESP8266 with the preceding IoTRFID software.

After it has been flashed, you should see the following on the Arduino Serial Monitor:

```
----------------
IOTRFID publish/subscribe Inventory
----------------
Connecting to WiFi ......
Local WiFi connected, IP address: 192.168.1.42
Reconnecting client to 192.168.1.40
```

Leave the IoTRFID running for now.

Setting Up the Mosquitto Debug Window

Go back into Terminal window 1 from earlier in the chapter and hold down the control key and click “c” (control-c) to kill the running process from the previous step. If you had closed it, open up another terminal window and follow the following instructions.

You can download the Python3 Software for the Raspberry Pi from github.com/switchdoclabs/SDL_Pi_IOTRFID by typing the following command in a terminal window:

```
git clone https://github.com/switchdoclabs/SDL_Pi_IOTRFID.git
```

Build a Python3 file for our debug and logging subscription to the Mosquitto broker:

```
nano IOTRFIDLogSubscribe.py
```

Enter the following code:

```
#
# SwitchDoc Labs
#
# Display logging subscription
#
# January 2020
#

import paho.mqtt.client as mqtt

# The callback for when the client receives a CONNACK response
from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))

    # Subscribing in on_connect() means that if we lose the
      connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("$SYS/broker/log/#");
# The callback for when a PUBLISH message is received from the
server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

# Blocking call that processes network traffic, dispatches
  callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded
  interface and a
# manual interface.
client.loop_forever()
```

Then run the code by typing this:

```
sudo python3 IOTRFIDLogSubscribe.py
```

We use "sudo" to make sure that Python is running with root privileges to avoid any potential permission issues.

If you left your IoTRFID running, you will see this:

```
Connected with result code 0
SYS/broker/log/N 1454535157: Client IOTRFID has exceeded
timeout, disconnecting.
$SYS/broker/log/N 1454535157: Socket error on client IOTRFID,
disconnecting.
$SYS/broker/log/N 1454535157: New connection from 192.168.1.135
on port 1883.
$SYS/broker/log/N 1454535157: New client connected from
192.168.1.135 as IOTRFID (c1, k15).
```

This means that your IoTRFID is connected to the Mosquitto MQTT broker.

Next, let's go to another terminal window and set up a subscriber on the Raspberry Pi to our IoTRFID device.

Set Up a Subscriber on the Raspberry Pi

In another terminal window on the Raspberry Pi, build another Python file by typing the following:

```
sudo nano IOTRFIDSubscriber.py
```

Enter the following code:

```
#
#
#
# IOTRFID Subscribe Module
#
# Receives inventory message from IOTRFID
# SwitchDoc Labs December 2020
#

import json
import paho.mqtt.client as mqtt

def filter_non_printable(str):

  return ''.join([c for c in str if ord(c) > 31 or ord(c) == 9])

#  The callback for when the client receives a CONNACK response
   from the server.
def on_connect(client, userdata, flags, rc):

    print("Connected with result code "+str(rc))

    # Subscribing in on_connect() means that if we lose the
      connection and

    # reconnect then subscriptions will be renewed.

    client.subscribe("IOTRFID/#")

# he callback for when a PUBLISH message is received from the
  server.

def on_message(client, userdata, msg):
   print(msg.topic+" "+str(msg.payload.decode()))
   result = json.loads(msg.payload.decode())  # result is now a
   dict / filter start and stop characters
   print(result)
   InventoryRFID = result['d']['RFID_ID']

   print()
   print("IOTRFID Inventory Number Received From
   ID#"+result['d']['IOTRFID'])
   print("Inventory Item = " + InventoryRFID)
   print

# main program

client = mqtt.Client()

client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

# Blocking call that processes network traffic, dispatches
  callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded
  interface and a
# manual interface.

client.loop_forever()
```

Execute the code by typing this:

```
sudo python3 IOTRFIDSubscriber.py
```

You will see the following:

```
Connected with result code 0
```

Next, we will run the full system test.

Testing the Entire IoTRFID System

Take your RFID Tag card and wave it over the IoTRFID antenna as shown in Figure 6-8.

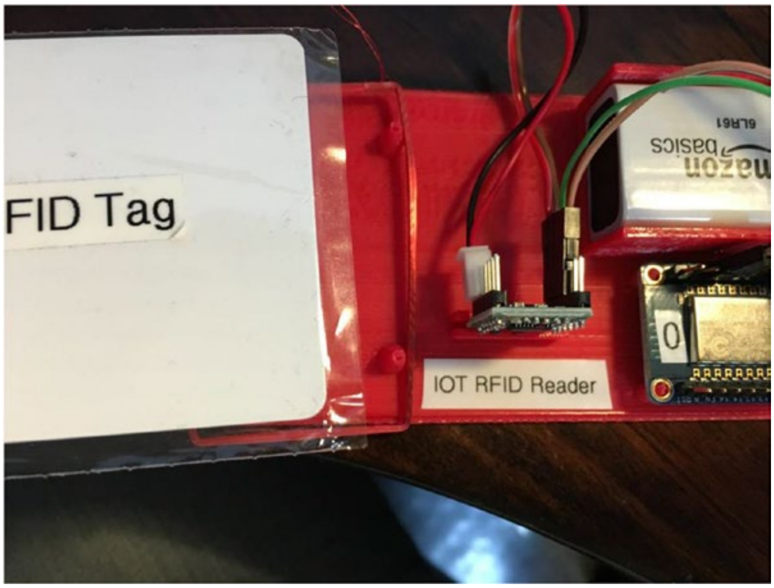


Figure 6-8. *RFID Tag Card over the IoTRFID Device*

If your IoTRFID is running, you should see something similar to the following in the Arduino IDE Serial Window:

```
Reconnecting client to 192.168.1.40

RFID card number read: 2413943
53
Sending IOTRFID payload: {"d":{"IOTRFID":"IR1","VER":"5","RF
ID_ID":"2413943"}}
IOTRFID Publish ok
```

And over in Terminal 2, you should see something similar to this:

```
IOTRFID {"d":{"IOTRFID":"IR1","VER":"5","RFID_ID":"2413943"}}
IOTRFID Inventory Number Recieved From ID#IR1
Inventory Item = 2413943
```

If you look at the debug window in Terminal 1, you should see something like this if you have no errors:

```
$SYS/broker/log/N 1454536590: New client connected from
192.168.1.135 as IOTRFID (c1, k15).
$SYS/broker/log/N 1454536615: Client IOTRFID has exceeded
timeout, disconnecting.
$SYS/broker/log/N 1454536615: Socket error on client IOTRFID,
disconnecting.
$SYS/broker/log/N 1454536615: New connection from 192.168.1.135
on port 1883.
$SYS/broker/log/N 1454536615: New client connected from
192.168.1.135 as IOTRFID (c1, k15).
```

You just successfully published an MQTT message from IoTRFID to the Mosquitto MQTT broker on the Raspberry Pi and received it on a MQTT subscriber in Terminal 2.

You have connected your IoT device into your own MQTT broker just as you did with IBM Bluemix in the previous chapter.

What to Do with the RFID Data on the Server

In our subscriber example, we have a very small piece of code that is handling the RFID inventory information coming in from IoTRFID. This code is the “on_message” function shown as follows:

```
# The callback for when a PUBLISH message is received from the
  server.
def on_message(client, userdata, msg):
   print(msg.topic+" "+str(msg.payload))
   result = json.loads(filter_non_printable(msg.payload))
# result is now a dict / filter start and stop characters
   InventoryRFID = result['d']['RFID_ID']

   Print()
   print("IOTRFID Inventory Number Received From
ID#"+result['d']['IOTRFID'])
   print("Inventory Item = " + InventoryRFID)
   print()
```

While we are just printing it out to the screen, if you were building an actual inventory system, this function is where you would call the database, action programs, and other publish-subscribe nodes that need to be notified that this piece of inventory has been scanned.

Conclusion

It should be somewhat surprising that we can duplicate a significant amount of the functionality of IBM Bluemix with less than only $50 worth of hardware. In truth, we have done that with this project. However, there are lots of things missing in our implementation vs. the IBM Bluemix system. Our system has no redundancy, no way to scale it up to tens of thousands of devices, is lacking a real administrative control panel, and has not addressed the very important issue of computer security and hackability.

The concepts we have explored in this chapter should give you some very solid insights into the technology and turn a black box (like IBM Bluemix) into something that is easier to understand.

As far as computer security goes for IoT devices, this is an excellent segue into our next chapter, Chapter 7.

CHAPTER 7

Computer Security and the IoT

Chapter Goal: Understand the Basics of IoT Computer Security

Topics Covered in This Chapter:

- The top five things to worry about
- What computer security is
- Computer security for communications
- Computer security for hardware devices
- Protecting your users

Why are we worried about computer security on the IoT?

Hackers are everywhere. They attack our banks. They attack our stores (think Target). They are trying to attack our infrastructure (power, water, sewer, etc.). Now, with the advent of IoT, they have a gateway into your home.

Guess what? They are going to attack your IoT device. You may think you are safe because you are not very important (see number 2 below – not being important is not a good defense); they will attack.

Understand one very important fact. Hackers rarely attack something specific. They use programs to attack thousands and tens of thousands of things at a time. Then they just wait for the results. If you are exposed on the Internet, they *will* find you. Your job is to make sure they can't do anything important about it.

Who are the hackers? There are the black hat people trying to make money from attacks or ransomware. There are nation-states doing this (and not just the United States, China, and Russia, either. North Korea and many other nations have programs). It's just not that expensive to do and the Internet gives you access to anywhere.

There are white hat hackers that are trying to find and fix security issues in programs and on the Internet. These people are trying to actually help the problem.

Oh, and then there is a whole other category of people that are somewhere in the middle. Gray hats if you like. These are people that are not really sure what they are doing. People can download hacking programs and run them without having any idea what is going on. These people are called "Script Kiddies" and can cause all sorts of damage on purpose or not.

IoT: Top Five Things to Know About IoT Computer Security

I have run a computer security company and have taught computer security and information warfare at the undergraduate and graduate levels at several universities. In this chapter, I am taking a different look at computer security than most technical books. While I am discussing methods of encryption and authentication, I am also taking a top-level view of the problem and a realistic view of what can and can't be done with the small computers that make up the IoT.

With that, let's start with my thoughts about the top five things to know about IoT computer security.

Number 1: This Is *Important*. You Can Prove Your Application *Is Insecure*, but You Can't Prove Your Application *Is Secure*

"What? That doesn't make any sense. My application only has 200 lines of code in it and I can see that it is secure!"

There are two things to consider here. First of all is that those 200 lines have been compiled by a compiler that has 100,000+ lines of code. The operating system you are running on has at least 25,000 lines of code (yes, even an Arduino) and millions of lines of code in a Raspberry Pi or Windows machine. Your 200 lines of code interact with tens of thousands of lines of code. You don't know how big your own program is. You don't know about the compiler. You don't know about the operating system. Yes, some micro-controllers allow you to set up everything, but in today's development systems, this is the exception, not the rule.

The second thing to consider is a proven theorem from Matt Bishop's excellent book *Computer Security: Art and Science* on computer security: "It is undecidable whether a given state of a given protection system is safe for a given generic right."

What does this mean? It means that "You can tell if your computer program is insecure, but you can't know if it is secure."

Ouch.

Number 2: Security Through Obscurity Is Not Security

An IoT system that relies on secrecy of the implementations or components of the system is not security. Obscurity can be part of a defense in-depth strategy but should not be relied on to provide security. Yes, someone can take your design and reverse engineer it and find out everything about it. Using a different port for SSH doesn't even slow down hackers these days. People can snoop on what you are sending and figure it out. Your system needs to rely on the key to your system and not the structure of the lock.

Number 3: Always Connected? Always Vulnerable

Every moment that your IoT device is connected to the Internet or the network is a moment that it can be attacked. Keep your device off the network as much as possible. This saves power, too, which is often a defining design criteria.

Number 4: Focus on What Is *Important* to Be Secure in Your IoT Application

Does a hacker care that the temperature in your aquarium is 85 degrees? Probably not. Do you want them to be able to change the temperature in your aquarium? Probably not. Do you want your front door lock (that is connected to the Internet?) to be secure? Yes, all the time. And no, you don't want hackers to be able to tell if the door is locked or unlocked. Just remember all the encryption in the world doesn't matter if a person has the key. In this case, either a physical key or a cryptographic key. Both can open your door. In our door lock IoT application, we must keep the key safe.

Number 5: Computer Security Rests on Three Main Aspects: Confidentiality, Integrity, and Availability

Confidentiality is defined as the concealment of information or resources. Keeping things secret (like keys) so the hackers can't use them.

Integrity is defined as the trustworthiness of the data or resources. Making sure that a hacker can't forge the directives to open your house door or car doors. Oh, that happens. Not good when it does.

Availability refers to the ability to access information or resources when required. If someone is doing a Denial of Service on your house or your Internet provider, you can't get to your door lock. Yes, even with an Internet-connected door lock, you should take your physical key along. And don't set the access code to your birthday.

In this chapter, I am going to talk about some IoT applicable techniques for addressing all three parts of the Triad. The CIA Triad is shown in Figure 7-1.

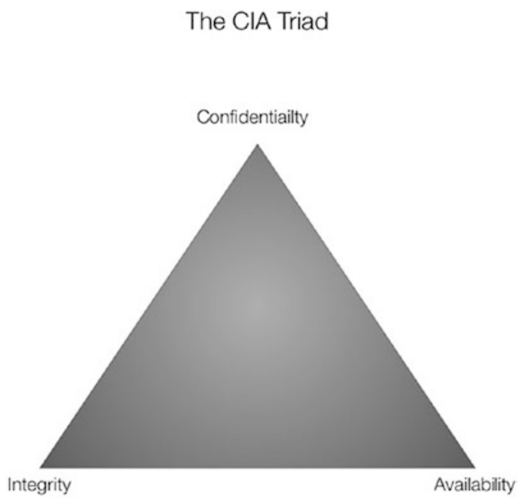


Figure 7-1. *The CIA Triad*

What Are the Dangers?

The dangers of computer insecurity depend on your information type and actuators of your IoT device. First, let's talk about information types.

The danger of hacking your device to gain information completely depends on the "value" of the information, both in a local sense and in a temporal sense. For example, a sensor reading the temperature in your closet probably isn't worth very much at any one time, and knowing the temperature of your closet ten hours ago (temporal value) is probably even worth less.

Going the other direction, transmitting your password for your WiFi in clear text from an IoT device is a valuable piece of information; and in most systems, it is still valuable from ten hours in the past.

So, the value of information has two main components. Is the information valuable at the time of production, and does it continue to remain valuable as time advances?

Assigning Value to Information

In a real sense, the ultimate risk of data disclosure can be described as the value of the information disclosed. In the recent Target store information loss, it was expensive. The value of the information lost (credit card and customer information) was high, not only in fraud but in fraud prevention (replacing all those cards), and eventually cost the CEO of Target his job. Now, note, because of these remedial actions, the time value of the information lost declined rapidly after the disclosure. What was the value of the information lost to Target? Somewhere around $10 or $15 million. While that might seem like a lot of money, it was less than 1% of the company's revenues.

After reimbursement from insurance and minus tax deductions, it was probably even less. The startling conclusion after looking at the reports is that big data breaches don't cost all that much to these companies. Similar numbers and costs apply to the Sony and the Home Depot data breaches. The costs just aren't material to these large companies.

All the consumers inconvenienced by getting new credit cards, or having fraud on their card, or their identity stolen would probably feel differently.

On a much smaller scale, I have experienced this on a more personal level. In 2000, I started a bank (yes, a real FDIC-insured actual bank) and have served on the board of directors since then. Naturally, I have served as Chairman of the Technology Committee. It's been interesting.

In the mid-2000s, our Visa payment provider was compromised, and we had a small fraud problem with the compromised cards (around $2000) that the regulations say we were responsible for. We also had about $10,000 of cost in issuing new cards. We got a small reimbursement from our cyber-insurance, but we had to go after the payment provider. After six months that went nowhere (they didn't want to pay anything even if it was clearly their fault), we gave up. So who really pays in this case? You, the consumer, with higher costs passed on to you.

Most academic papers focus on the cost of data loss and exposure in large- and medium-size enterprises. There is surprisingly little work about the cost of information loss and exposure in IoT.

So, how do you assign value to information in the IoT? You look at what it means to the user to have the data lost or compromised and estimate a financial (or personal) harm to the user. The preceding companies focus on cost to themselves. In the IoT, you need to look at the cost to the end user.

What happens if someone hacks your front wireless door lock? How about your TV? According to reports, the Samsung Smart TV listens to you even when you aren't talking to the TV. Alexa and Google Nest listen too. They warn you not to discuss sensitive information around your TV. Good grief. When it was disclosed that it was possible to hack into the Samsung Smart TV camera on the top of the set, they suggested putting black tape over it or pushing it into the set with a pencil. Samsung is a $180 billion a year company. You would think they would put a little more thought into computer security.

The Gartner company forecasts [www.gartner.com/newsroom/id/3165317] that by 2022 there will be 20 billion "things" connected to the Internet. Using the rough numbers of the amount Gartner forecasts that will be spent on the global IoT security market (about $7 billion in 2015 and increasing to $30 billion in 2020), we come up with an interesting number. The amount that will be spent on securing the IoT will be about $1 per IoT device. Now this is something we can use in our budgeting for IoT development. My gut feel is that is about right.

Back to the value of information. In 2020, 10 million IoT devices get connected every day. These devices being connected include cars, kitchen appliances, smart TVs, wristwatches, toys, medical instruments, and so on.

When an IoT device gets connected, it needs to be protected.

Building the Three Basic Security Components for IoT Computers

Computer security is a complex topic with lots of interactions between different systems and subsystems in even a simple transaction from an IoT device to a server across the Internet. We are going to now return to the CIA Triad (confidentiality, integrity, and availability) and see what we can do with each of these concepts in practice on these small computers.

Confidentiality – Cryptography

Confidentiality is defined as concealment of information. As was mentioned earlier, you can almost always get information out of device if you have physical access to the device. What we are focusing on in this section is protecting the information transmission channel rather than the physical IoT device itself.

Note from number 2 above, obscurity doesn't count as far as computer security goes. If we are transmitting valuable information, then we need to encrypt it using cryptographic techniques. What is cryptography? Cryptographic techniques include uses for all of the CIA Triad, but here we are going to focus on encryption that is a subset of these techniques.

Discussing cryptography and encryption techniques can easily fill an entire book so I will stay away from the math and focus on the application. The one thing we do need to discuss is that modern cryptography is very heavily based on math and computer science methodologies. All of the following techniques can be broken based on the theoretical considerations of the algorithms, but it is impractical to do so by any known techniques. That makes these techniques "computationally secure." Someday in the future, they may be able to read your thermometer, but will it be worth it? Depends on the value of the information.

There is one important thing to realize about encryption and cryptography. These algorithms tend to be computationally expensive in terms of amount of CPU clocks and RAM memory to do these calculations. Remember that IoT devices don't have a lot of CPU clocks or a lot of RAM memory. Therefore, use the minimum amount of these techniques to protect your data. You need to be smart to save battery and RAM space.

Cryptography on the Arduino

There are basically two ways for Arduino to communicate with external devices. You either have a network connection (Ethernet, Bluetooth, WiFi, Zigbee, or other networks), or you have a serial connector (USB, straight serial). While the techniques described in this section are applicable to both kinds of connections, we will focus on the network-type techniques. For the Arduino, I am going to focus on using AES. While this next example is run on Arduino Mega2560, it can also be run on an ESP8266 with minimal changes.

AES is what they call a symmetric block cipher used by the US government to protect classified information. It's pretty good encryption and can't be broken at the writing of this chapter by brute force attacks. However, as Bruce Shinier has said, "Attacks always get better; they never get worse."

Here are just some of the AES libraries available for the Arduino:

- `github.com/DavyLandman/AESLib`
- `spaniakos.github.io/AES/index.html`
- `utter.chaos.org.uk:/~markt/AES-library.zip`

We are using an Arduino Mega2560 (shown in Figure 7-2) instead of an Arduino Uno because the Mega2560 has 8K bytes of SRAM available vs. 2K bytes in the Uno.

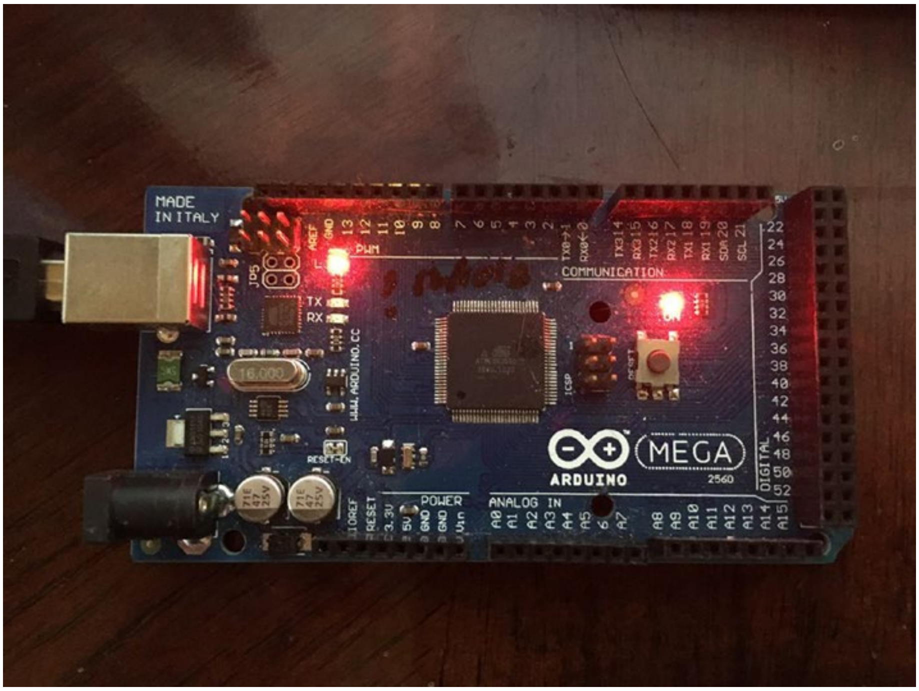


Figure 7-2. *Arduino Mega2560*

We are going to use the Davy Landman library for our examples. To install into your Arduino library, follow the directions in the README.md file. The following is an example of how to use the AES library in an Arduino program. You can substitute any Arduino for this example. However, it does not compile under the ESP8266 Arduino IDE unless changes are made to the AES library itself.

```
//
//
// AES Library Test
// SwitchDoc Labs
// February 2016
//

#include <AESLib.h>

// 256 bit key for AES
uint8_t key[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31};

void setup() {

  Serial.begin(115200);
}

void loop() {
  // put your main code here, to run repeatedly:

  // 16 bytes
  char data[] = "01234567890Hello";

  int sizeOfArray;
  int i;
  sizeOfArray = sizeof(data) / sizeof(char);

  Serial.println();

  Serial.print("data to be encrypted (ASCII format):\t ");
  Serial.println(data);

  Serial.print("data to be encrypted (Hex format):\t ");


  for (i = 0; i < sizeOfArray - 1; i++)
  {
    if (uint8_t(data[i]) < 0x10)
      Serial.print("0");
    Serial.print(uint8_t(data[i]), HEX);

  }
  Serial.println();
  // Now we encrypt the data using a 256 byte AES key

  aes256_enc_single(key, data);

  // print the encrypted data out in Hex

  Serial.print("encrypted (in Hex):\t\t\t ");

  for (i = 0; i < sizeOfArray - 1; i++)
  {
    if (uint8_t(data[i]) < 0x10)
      Serial.print("0");
    Serial.print(uint8_t(data[i]), HEX);

  }
  Serial.println();
  // Now decrypt the data and print it out

  aes256_dec_single(key, data);
  Serial.print("decrypted:\t\t\t\t ");
  Serial.println(data);
}
```

Note the key given at the beginning of the program. This 256-bit key is what you need both to encrypt and decrypt the data. Granted, this is not a very good key in our example.

The following is an example of the results of the program:

```
data to be encrypted (ASCII format):   01234567890Hello
data to be encrypted (Hex
format):     303132333435363738393048656C6C6F
encrypted (in
Hex):                        438693C632D91AF2283651F416BBA61E
decrypted:                                 01234567890Hello
```

This is the single piece of secret information. Both ends of the communication channel need to have this key for AES to work.

Key management (how to send these keys around) is a key topic of IoT computer security design and will be discussed later in this chapter.

Looking at this program, you could see how you could decompile this code and retrieve the secret key. This is an example of why physical access to the device can often lead to the ability to hack the device and then the network of devices.

How fast are these libraries on an Arduino? Based on a 16MHz ATMega328, a 128-bit AES key can generate about 27,000 bytes per second. Using a 256-bit key gets you about 20,000 bytes per second.

Now that is full bore CPU usage, so if you have other things to do with the processor (as is almost certainly the case), you won't be able to make these data rates continuously.

Note that AES libraries are already built into the firmware for the ESP8266, but as of the writing of this chapter, there really are no complete examples of how to use them and the documentation is very sparse.

Cryptography on the Raspberry Pi

The Raspberry Pi has a far richer set of cryptography libraries available in the operating system or as easily available downloads. For example, if you are communicating using web requests (between Raspberry Pis or Linux-based computers), you can use SSL to encrypt the traffic between the two nodes. It is highly standardized, and if properly configured is strongly secure. Why don't we use SSL on an Arduino or ESP8266-based system? Two reasons: first, memory/processing power limitations; and second is the problem of distribution of the certificates necessary for secure SSL links. This is another variant of the key distribution problem.

The same AES256 functions are available on the Raspberry Pi so they could be used in communicating with an Arduino. However, this method is a long way from the full SSL implementation used by modern web browsers and computer (think `https:// instead of http://`).

The Spaniakos AES encryption library as cited earlier under the "Cryptography on the Arduino" section is also available for the Raspberry Pi.

Note that there are hardware implementations of AES starting to appear for Arduinos and Raspberry Pi.

Integrity – Authentication

Integrity is defined as the trustworthiness of the data or resources. Note the difference between integrity and encryption. When I speak of integrity, I am speaking of making sure that the directives or data sent to my IoT device has not been changed en route and that I know for sure who sent it. The directives or data may or may not be encrypted.

Looking at this from a value-of-information perspective, this means that we don't really care if a hacker looks at the data (I am turning up the heat in my house), but we use integrity to make sure that a hacker cannot forge the directive or data and that the hacker cannot change the data itself.

There are many ways of using cryptographic algorithms for establishing the integrity of the contents of the message and establishing who sent the message.

Establishing that the message has not been changed en route can be done by using what is called a cryptographic hash on the message. In concept, you take each byte of the message, and by using a cryptographic algorithm, you can determine that the message has not been changed – with a very, very high degree of certainty.

Cryptographic Hashes on the Arduino/Raspberry Pi

In order to prove that the message has not been altered on transit, you will need to send the message + the hash across the network. Much like the preceding encryption example, you need to take your message and cryptographically hash the message to generate the hash codes. We will use the fork from Cathedrow library [https://github.com/maniacbug/Cryptosuite] for this example.

However, the libraries required some tweaking to make them work in the newer versions of the Arduino IDE, so I have included them in the download of this software as additional tabs. This will work with an Arduino board and an ESP8266 board.

```
//
// SHA256 Hash Test
//
// For both the Arduino and ESP8266
// SwitchDoc Labs
//
```

```
// February 2020

#include "sha256.h"

void printHash(uint8_t* hash) {
  int i;
  for (i = 0; i < 32; i++) {
    Serial.print("0123456789abcdef"[hash[i] >> 4]);
    Serial.print("0123456789abcdef"[hash[i] & 0xf]);
  }
  Serial.println();
}

void setup() {

  Serial.begin(115200);
  Serial.println("SHA256 Hash Test");
    Serial.println();

}

void loop() {

  uint8_t *hash;
  Sha256.init();
  String hashMessage;
  int i;

  while (1)
  {

    hashMessage = "This is a message to hash-";
    hashMessage = hashMessage + String(i % 10);
    Serial.print("Hashing Message: ");
    Serial.println(hashMessage);
    Sha256.print(hashMessage);
    hash = Sha256.result();

    printHash(hash);
    Serial.println();
    delay(5000);

    i++;

  }

}
```

The results are as follows:

```
SHA256 Hash Test
Hashing Message: This is a message to hash-2
f2a47cefff87600aeb9089cf1f11a51b833ccdd91b808df75b7238fc78ba53f2

Hashing Message: This is a message to hash-3
5d135bc35a33004cad4c4ed37fc0011c1475f09c6ddd7dec01c348734e575f41

Hashing Message: This is a message to hash-4
fe1aa898ab0e041d1ce4539b388439c75a221efab0672896fc5e14b699c01492
```

If the hashed message is changed, the SHA256 hash will change. You can say that the message has not been changed en route to your device. Note we have no cryptographic key to exchange. SHA256 will work the same everywhere.

SwitchDoc Note Proving something has not been altered in transit will tell you it is good. However, what if we just have a slightly corrupted message? This could be caused by noise in the line (think satellite communication). If you want to correct errors in a message like this, a whole other set of algorithms is used. Reed-Solomon error correction codes are a great example of these. You can correct the data and then decrypt it. Obviously, you need to encrypt the message first and then put these codes over the top of the encrypted data to make your communication channel more reliable. Reversing the order (encrypting the Reed-Solomon code) makes no sense and will not provide error correction.

The Raspberry Pi code is very similar.

So, this seems pretty straightforward. What are we missing? We have the proof that the message has not been changed. Great. Proof of data integrity.

However, we have not proved that the data has not been forged using a man-in-the-middle attack. Now we have to look at the last part of data integrity. Proof where it is coming from, or in other words, non-repudiation.

Non-repudiation is usually based upon some variation of public/private key encryption.

It works like this. Bob has a private key, only known to him. He also has a public key that he gives to anybody who wants to communicate with him. When Bob wants to send a message to Alice, he encodes the message with his private key and sends it to Alice. Alice uses the public key she got from Bob to decrypt the message. If it successfully decrypts, then Alice knows that the message came from Bob.

So, using this public/private key method, we can encrypt the message and prove it came from Bob and is unaltered. Reversing the procedure, we can send a message to Bob from Alice safely.

OK, looks perfect. What are the complications? Once again, we are back to key exchanges. Both our IoT server and the IoT device have to have private keys and publish public keys to the server. This does sound like a lot of work. And it is. Those more-sophisticated readers will notice that this sounds a lot like SSH/HTTPS. Correct. And as noted previously in this chapter, those are big libraries for small computers; although the Raspberry Pi can handle it, the Arduino class of processors really can't. The ESP8266 is barely able to do this (and does it for their over-the-air update option), but they are not distributing keys. They update a one-way communication. For bidirectional communication links, you need keys for both directions.

There is a whole additional level of complexity laid on top of this. Certificates must be traceable to a known trusted authority to prove the system hasn't been compromised.

Showing examples of the public/private key software is well beyond the scope of this book. Some keywords to look for are "Digital Signature, Diffie-Hellman, and RSA Public Key Encryption" when searching for this technology. A good example of a system showing this kind of a system between an Arduino and a Windows PC is available at this link: `github.com/arpitchauhan/cryptographic-protocols-arduino-and-PC/`.

The overall methodology used is as follows:

- Public keys are exchanged securely between the two devices. (Note: Both have their own private key.)
- After the public keys are exchanged, you can transfer a key for AES encryption.
- Now the devices can communicate securely and safely using AES as in the preceding example.

An important thing to remember is public/private key encryption is much more computationally expensive than AES encryption. This is why you exchange your AES keys with this scheme and then just use AES to send the data in normal operation.

Availability – Handling DOS/Loss of Server/Watchdogs

There really isn't a lot of difference between an attack to deny you service (DOS), the Internet going down, your wireless router rebooting, or your microwave oven going crazy. All of these things disrupt your communication path to your IoT device.

One thing that commercial wireless IoT devices are especially vulnerable to is the inability to talk to the server because of electrical noise in the environment whether generated by error or on purpose. When you are planning your IoT device, you need to make sure you design it so your device can handle a disruption of communication service, whether it is done on purpose or by equipment failure. There needs to be some method for telling the user or consumer that you have a fault.

How does the military get around this? It uses complex schemes like frequency-hopping radios, large amounts of radiated radio power to "burn through" the noise, and big antennas. These are not techniques that are cheaply available to our IoT devices.

Key Management

Key management is a hard problem. Configuration management of IoT devices is difficult. From the sections earlier in this chapter, we need to have a shared key to do AES256 encryption to keep our communication confidential. We also need keys to prove that the message has come from the correct person or server.

Remember that *any* time your IoT device is communicating with the Internet or the local area network, there is a possibility of compromise.

There are two approaches to key management on IoT devices. The first is by using hardware key management.

Companies like Infineon build chips that include private and public keys that can be used to build networks of IoT devices. You can establish trusted certificates (linking to a trusted certificate authority) at the time of manufacture and then establish a set of software protocols for using these keys after deployment. As the process and chips become less expensive, I believe we will see more and more of these types of systems deployed. At the writing of this book, these chips are not cheap. Using one will roughly double the price of your IoT device in many cases. What about physical access to these devices? Yes, you can compromise a device by taking it and tearing it apart (and I don't just mean taking the screws out. I'm talking about removing the plastic from the chips and going through the chip with a microscope. Remember our discussion about the value of information?) It's probably not worth doing all of this to compromise your thermostat or toaster.

The second method is to use software to distribute the keys during configuration (or even during manufacturing). These methods in general are called PKI (Public Key Infrastructure).

Generally, many IoT devices are more vulnerable to attacks than normal servers, PCs, and smartphones because they operate with embedded software (firmware) that has been developed without online security in mind. There are many examples of really bad software flaws that can include back-end management with default passwords that are easily found on the Internet and non-encrypted communications. People are actively hacking into things like routers and other connected devices and using them in DOS (Denial of Service) attacks. Think of our LightSwarm devices in a previous chapter. You could program them all to start sending messages as quickly as possible and create your own DOS attack. During the development of the LightSwarm, I inadvertently did that very thing. I overwhelmed the Raspberry Pi server by sending messages all at once, as quickly as possible.

Another real issue with IoT devices is that the internal IoT device firmware may not be updated for years (or the company may go out of business) and the infrastructure for updating these devices just doesn't exist or not.

The use of current PKI methods is just impractical to scale to the upcoming multiple billions of IoT devices projected to be connected to the network. Because of this, many manufacturers take shortcuts to secure their devices. For example, all of the IoT devices may share a single key (either an AES encryption key or a single public/private key pair). This method does supply a significant amount of security, but if the single key is compromised (as has happened many times in the computer industry, think Sony and Microsoft), then the entire network is open to hackers.

The industry has the mathematic and technical tools to protect the IoT. What we don't have is a simple and secure method for distributing the keys to all of the IoT.

This is a problem that has not been solved yet and probably won't be before we deploy another few billion computers.

Update Management

According to one pundit (actually me), the two biggest problems for keeping IoT devices safe are the following:

- Lack of remote updating capability
- Remote updating capability

Being able to update your firmware is key to avoiding obsolescence and to recover from flaws that come to light after deployment of devices. This is one very important channel of information that needs to be cryptographically secured, both from a data integrity point of view and to make sure it is only being updated from where it should be updated from (non-repudiation). The preceding techniques can be used to do these very things.

If anybody can figure out how to fool your updating system, the person can reprogram your entire system.

Conclusion

Why would someone want to break into your thermostat? To provide an entrance to your network. Remember the Target data breach I talked about earlier? That was initiated via a cooling system maintenance application. They got in the network via the air conditioner.

The IoT represents a huge opportunity to improve people's lives and experiences. With great power comes great responsibility. When you design your Internet of Things killer application and device, design security in from the beginning. Don't tack it on at the end.

Watch the news. In the near future, you will see articles about people (and programs) hacking into cars – oh wait, they are already doing that. You will see articles about people hacking into hotel door locks – oh wait, they are already doing that. And you will see people hacking into your home thermostat – oh wait, they are already doing that too.

Here is an example from an Amazon review:

> The man wrote that his wife had left him for another man and then moved her new man into their old home that they had shared, which had a Honeywell WiFi thermostat. The ex-lover could still control the thermostat through the mobile app installed on his smartphone, so he used it to change the environment for the couple now living in his old house:
>
> "Since this past Ohio winter has been so cold I've been messing with the temp while the new lovebirds are sleeping. Doesn't everyone want to wake up at 7 a.m. to a 40 degree house? When they are away

> on their weekend getaways, I crank the heat up to 80 degrees and back down to 40 before they arrive home. I can only imagine what their electricity bills might be. It makes me smile. I know this won't last forever, but I can't help but smile every time I log in and see that it still works. I also can't wait for warmer weather when I can crank the heat up to 80 degrees while the lovebirds are sleeping. After all, who doesn't want to wake up to an 80 degree home in the middle of June?"

Now granted, that isn't really a technical hack – more of a social engineering experience, but it shows what can be done with these little systems.

Your Nest Thermostat (insert your own IoT example here) can be a gateway into your home. In many systems, like the Nest Thermostat, you are only one password from being hacked. Makes a good argument for good passwords on *any* system that is exposed to the Internet.

Oh and did you know that the Nest Thermostat has a microphone? Google says that it is not currently used, but they never disclosed the existence. And remember that all devices are vulnerable to being hacked with physical access. So make sure you keep your "real" metal keys safe too.

My house is filled with IoT devices. Over 100 at last count. But one thing I do not have is my door locks on IoT devices. I still like metal keys for locking and unlocking my doors.

APPENDIX A

Suggestions for Further Work

The IoT is an area of active research and now active deployment. Nobody in the industry has yet found the “killer application” that will take the IoT to mass consumer deployment, but it is just a matter of time. If you took the time to build the IoT projects in the previous chapters, you now have a basic set of skills to prototype and innovate in the IoT space. There is no better way of learning a new set of abilities than actually building things. That is the mantra of the Maker Movement.

The following are some of the IoT projects that I am currently working on. They are good suggestions for further work on your own, if you are so inclined.

- OurWeather – A no-soldering required IoT Weather Station kit for ages 8 and up
- Air quality IoT device
- Low-power RF links for IoT devices
- SunRover – Solar-powered IoT connector robot
- SunGrassHopper – IoT connected small solar-powered robot

- Cloud cover detecting IoT sensor
- Building IoT devices using Grove connectors for easy and fast prototyping

Another idea is to regularly scour the following websites for new devices and new sensors:

- `www.switchdoc.com`
- `www.adafruit.com`
- `www.seeedstudio.com`
- `www.tinysine.com`

For more technical users:

- `www.digkey.com`
- `www.mouser.com`
- `www.texasinstruments.com`

When I see a new sensor that is interesting, I jump on it, build a project, and publish it in either magazines or on my blog. There are new IoT-based devices (like the Adafruit Huzzah) coming online every day. If you see one and you are interested, buy it and build something. Come visit my blog and join the conversation at `www.switchdoc.com`.

Parting Words…

The IoT is going to be big. Just how big, nobody knows. The technologies that are now exploding in the marketplace are accessible to the reasonably technically minded person. As always, I feel the best learning is by doing. If you have built the five projects in this book, then you have a really good idea of what is going on with the devices in your household or workplace. IoT probably doesn't mean "black box" to you anymore.

Have you ever heard of Metcalfe's law? It states that "the value of a telecommunications network is proportional to the square of the number of connected users of the system."

Substitute "connected IoT devices" for "connected users" and read it again. The world is at the cusp of something. It might just be bigger than the Internet. Oh wait. It will be the Internet. The Internet of Things will be much larger than the Internet of connected people.

Keep reading the blogs and building things. Stay current with the technology and the future won't be nearly as mysterious as it was when you started this book.

Index