Bluetooth Low Energy in Arduino 101

1st Edition

Tony Gaitatzis

BackupBrain Publishing, 2017

ISBN: 978-1-7751280-6-9

backupbrain.co

Bluetooth Low Energy in Arduino 101

by Tony Gaitatzis

Copyright © 2015 All Rights Reserved

(This page intentionally left blank)

Dedication

To Chris, for teaching me to be a magician

and Jared, for introducing me to Arduino

(This page intentionally left blank)

Preface

Thank you for buying this book. I'm excited to have written it and more excited that you are reading it.

I started with Bluetooth Low Energy in 2011 while making portable brain imaging technology. Later, while working on a friend's wearable electronics startup, I ended up working behind teh scenes on the TV show America's Greatest Makers in the Spring of 2016.

Coming from a web programming background, I found the mechanics and nomenclature of BLE confusing and cryptic. A er immersing myself in it for a period of time I acclimated to the di erences and began to appreciate the power behind this low-power technology.

Unlike other wireless technologies, BLE can be powered from a coin cell battery for months at a time - perfect for a wearable or Internet of Things (IoT) project! Because of its low power and short data transmissions, it is great for transmitting bite size information, but not great for streaming data such as sound or video.

Good luck and enjoy!

Conventions Used in This Book

Every developer has their own coding conventions. I personally believe that well-written code is self-explanatory. Moreover, consistent and organized coding conventions let developers step into each other's code much more easily, enabling them to reliably predict how the author has likely organized and implemented a feature, thereby making it easier to learn, collaborate, fix bugs and perform upgrades.

The coding conventions I used in this book is as follows:

Inline comments are as follows:

```
// inline comments
```

Multiline comments follow the Doxygen standard:

```
/**
 * This is a multiline comment
 * It features more than one line of comment
 * @parameter usage discription
 * @return type
 */
```

Constants and variables are written in camel case:

```
static const int constantName = 1;
int normalVariable = 1;
```

Function declarations are in Camel Case. In cases where there is not enough space for the whole function, parameters are written on another line:

```
void shortFunction() {
}
void superLongFunctionName(
  int parameterOne,
  int parameterTwo)
{
  ...
}
```

Long lines will be broken with a backslash (\ ) and the next line will be indented:

```
static const char[] someReallyLongVariableNameWillBeBroken = \
  "onto the next line";
```

Introduction

In this book you will learn the basics of how to program a Bluetooth Low Energy Peripheral on Arduino 101, culminating in three projects:

- An Echo Server and Client
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

- How Bluetooth Low Energy works,
- How data is sent and received
- Common paradigms for handling data

This book is an excellent read for anyone familiar with Arduino or C++ programming, who wants to build an Internet of Things device.

Overview

Bluetooth Low Energy (BLE) is a digital radio protocol. Very simply, it works by transmitting radio signals from one computer to another.

Bluetooth supports a hub-and-spoke model of connectivity. One device acts as a hub, or “Central” in Bluetooth terminology. Other devices act as “Peripherals.”

A Central may hold several simultaneous connections with a number of peripherals, but a peripheral may only hold one connection at a time (Figure 1-1). Hence the names Central and Peripheral.

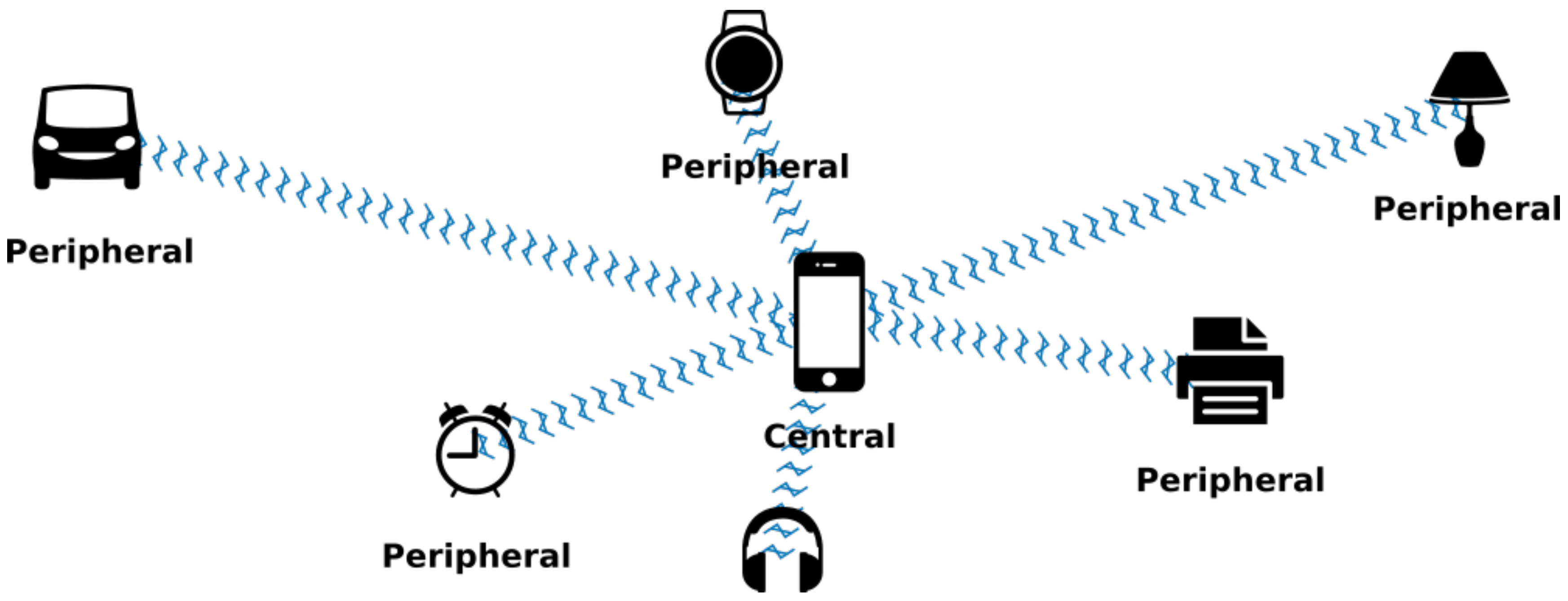


Figure 1-1. Bluetooth network topology

For example, your smartphone acts as a Central. It may connect to a Bluetooth speaker, lamp, smartwatch, and fitness tracker. Your fitness tracker and speaker, both Peripherals, can only be connected to one smartphone at a time.

The Central has two modes: scanning and connected. The Peripheral has two modes: advertising and connected. The Peripheral must be advertising for the Central to see it.

Advertising

A Peripheral advertises by advertising its device name and other information on one radio frequency, then on another in a process known as frequency hopping. In doing so, it reduces radio interference created from reflected signals or other devices.

Scanning

Similarly, the Central listens for a server's advertisement first on one radio frequency, then on another until it discovers an advertisement from a Peripheral. The process is not unlike that of trying to find a good show to watch on TV.

The time between radio frequency hops of the scanning Central happens at a different speed than the frequency hops of the advertising Peripheral. That way the scan and advertisement will eventually overlap so that the two can connect.

Each device has a unique media access control address (MAC address) that identifies it on the network. Peripherals advertise this MAC address along with other information about the Peripheral's settings.

Connecting

A Central may connect to a Peripheral after the Central has seen the Peripheral's advertisement. The connection involves some kind of handshaking which is handled by the devices at the hardware or firmware level.

While connected, the Peripheral may not connect to any other device.

Disconnecting

A Central may disconnect from a Peripheral at any time. The Peripheral is aware of the disconnection.

Communication

A Central may send and request data to a Peripheral through something called a "Characteristic." Characteristics are provided by the Peripheral for the Central to access. A Characteristic may have one or more properties, for example READ or WRITE. Each Characteristic belongs to a Service, which is like a container for Characteristics. This paradigm is called the Bluetooth Generic Attribute Profile (GATT).

The GATT paradigm is laid out as follows (Figure 1-2).

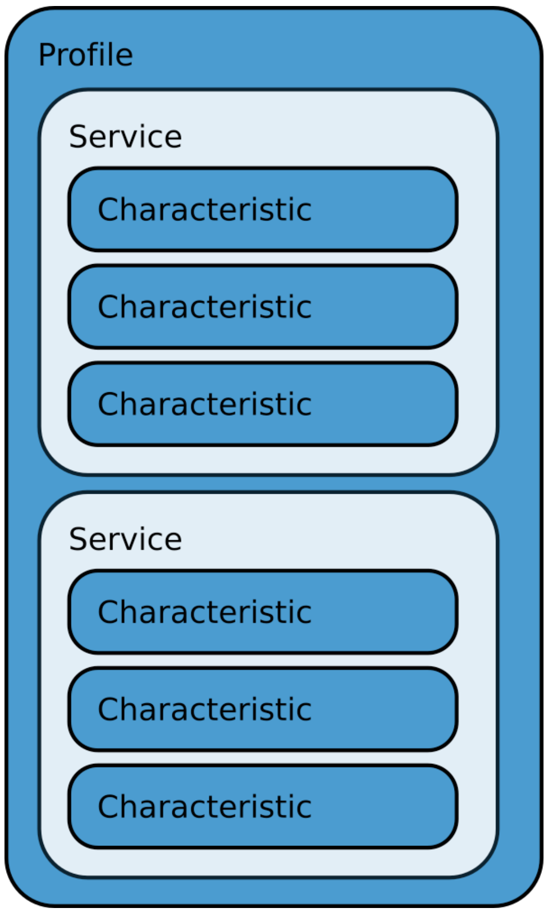


Figure 1-2. Example GATT Structure

To transmit or request data from a Characteristic, a Central must first connect to the Characteristic's Service.

For example, a heart rate monitor might have the following GATT profile, allowing a Central to read the beats per minute, name, and battery life of the server (Figure 1-3).

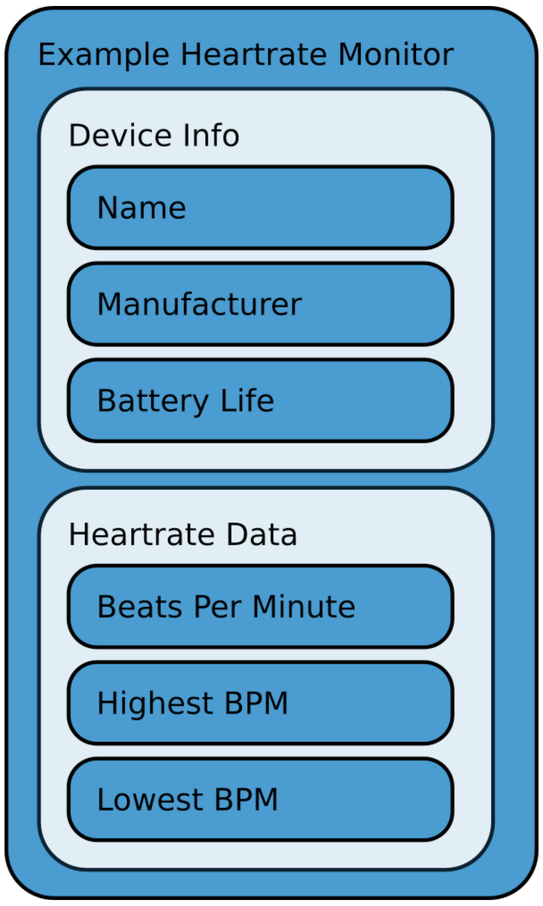


Figure 1-3. Example GATT structure for a heart monitor

In order to retrieve the battery life of the Characteristic, the Central must be connected also to the Peripheral's "Device Info" Service.

Because a Characteristic is provided by a Peripheral, the terminology refers to what can be done to the Characteristic. A "write" occurs when data is sent to the Characteristic and a "read" occurs when data is downloaded from the Characteristic.

To reiterate, a Characteristic is a field that can be written to or read from. A Service is a container that may hold one or more Characteristics. GATT is the layout of these Services and Characteristics. Characteristic can be written to or read from.

Byte Order

Bluetooth orders data in both Big-Endian and Little-Endian depending on the context.

During advertisement, data is transmitted in Big Endian, with the most significant bytes of a number at the end (Figure 1-4).

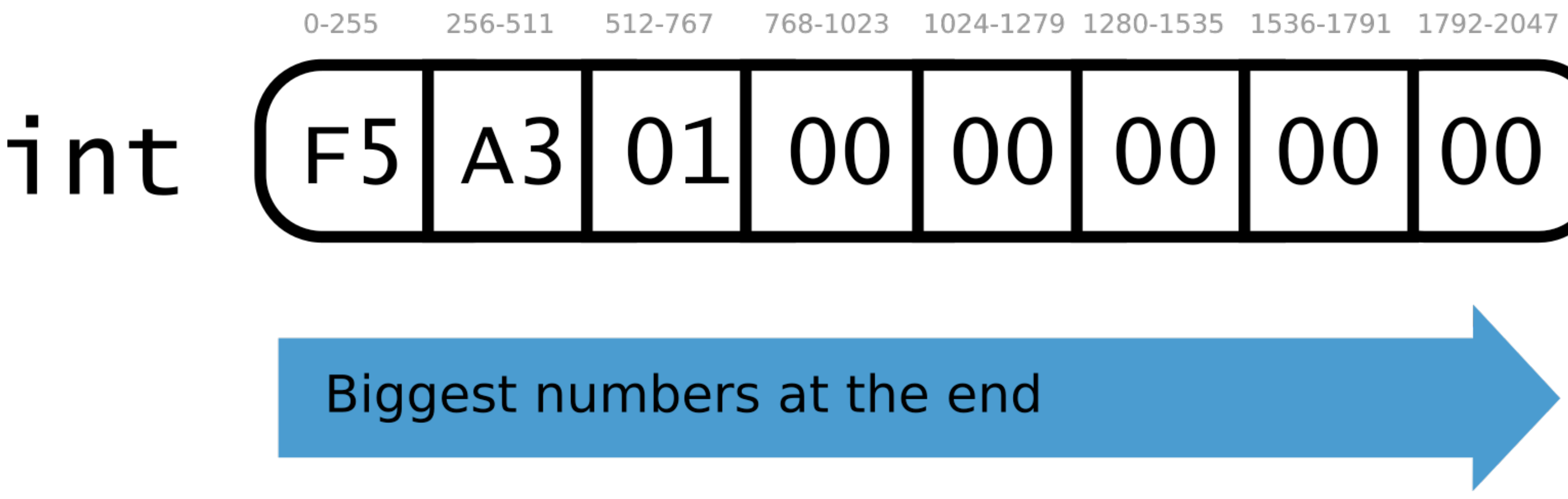


Figure 1-4. Big Endian byte order

Data transfers inside the GATT however are transmitted in Little Endian, with the least significant byte at the end (Figure 1-5).

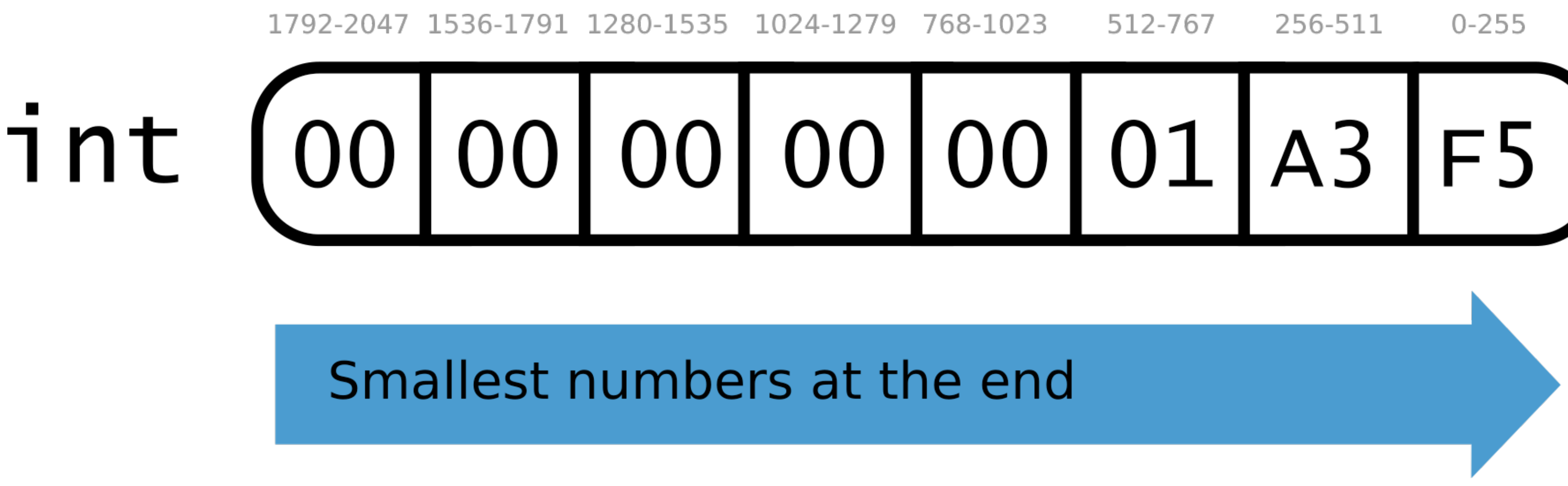


Figure 1-5. Little Endian byte order

Permissions

A Characteristic grants certain Permissions of the Central. These permissions include the ability to read and write data on the Characteristic, and to subscribe to Notifications.

Descriptors

Descriptors describe the configuration of a Characteristic. The only one that has been specified so far is the “Notification” flag, which lets a Central subscribe to Notifications.

UUIDs

A UUID, or Universally Unique IDentifier is a very long identifier that is likely to be unique, no matter when the UUID was created or who created it.

BLE uses UUIDs to label Services and Characteristics so that Services and Characteristics can be identified accurately even when switching devices or when several Characteristics share the same name.

For example, if a Peripheral has two “Temperature” Characteristics - one for Celsius and the other in Fahrenheit, UUIDs allow for the right data to be communicated.

UUIDs are usually 128-bit strings and look like this:

```
ca06ea56-9f42-4fc3-8b75-e31212c97123
```

But since BLE has very limited data transmission, 16-bit UUIDs are also supported and can look like this:

```
0x1815
```

Each Characteristic and each Service is identified by its own UUID. Certain UUIDs are reserved for specific purposes.

For example, UUID 0x180F is reserved for Services that contain battery reporting Characteristics.

Similarly, Characteristics have reserved UUIDs in the Bluetooth Specification.

For example, UUID 0x2A19 is reserved for Characteristics that report battery levels.

A list of UUIDs reserved for specific Services can be found in ***Appendix IV: Reserved GATT Services.***

A list of UUIDs reserved for specific Characteristics can be in ***Appendix V: Reserved GATT Characteristics.***

If you are unsure what UUIDs to use for a project, you are safe to choose an unassigned service (e.g. 0x180C) for a Service and generic Characteristic (0x2A56).

Although the possibility of two generated UUIDs being the same are extremely low, programmers are free to arbitrarily define UUIDs which may already exist. So long as the UUIDs defining the Services and Characteristics do not overlap in the a single GATT Profile, there is no issue in using UUIDs that exist in other contexts.

Bluetooth Hardware

All Bluetooth devices feature at least a processor and an antenna (Figure 1-6).

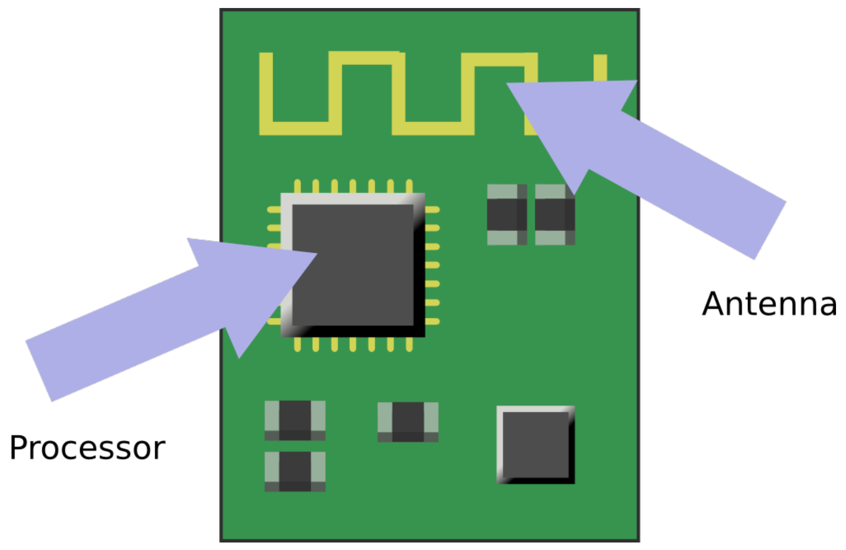


Figure 1-6. Parts of a Bluetooth device

The antenna transmits and receives radio signals. The processor responds to changes from the antenna and controls the antenna's tuning, the advertisement message, scanning, and data transmission of the BLE device.

Power and Range

BLE has 20x2 Mhz channels, with a maximum 10 mW transmission power, 20 byte packet size, and 1 Mbit/s speed.

As with any radio signal, the quality of the signal drops dramatically with distance, as shown below (Figure 1-7).

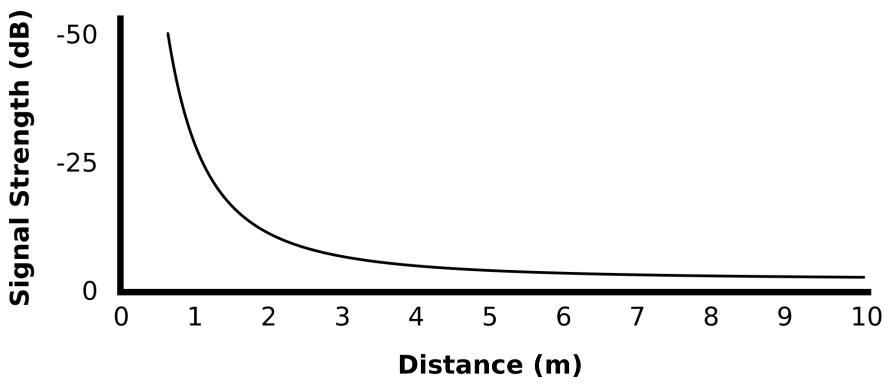


Figure 1-7. Distance versus Bluetooth Signal Strength

This signal quality is correlated the Received Signal Strength Indicator (RSSI).

If the RSSI is known when the Peripheral and Central are 1 meter apart (A), as well as the RSSI at the current distance (R) and the radio propagation constant (n). The distance betweeen the Central and the Peripheral in meters (d) can be approximated with this equation:

$$d \approx 10^{\frac{A - R}{10n}}$$

The radio propagation constant depends on the environment, but it is typically somewhere between 2.7 in a poor environment and 4.3 in an ideal environment.

Take for example a device with an RSSI of 75 at one meter, a current RSSI reading 35, with a propagation constant of 3.5:

$$d \approx 10^{\frac{75 - 35}{10 \times 3.5}}$$

$$d \approx 10^{\frac{40}{35}}$$

$$d \approx 14$$

Therefore the distance between the Peripheral and Central is approximately 14 meters.

Introducing Arduino

Arduino is an easy-to-program embedded device, making it an excellent platform for beginners.

As an embedded device, Arduino can be designed to work as a connected device that does everything from home automation to distributed weather monitoring, to wearable electronics.

We will be using Arduino to learn how to communicate between the Arduino and a smartphone using Bluetooth Low Energy (BLE). Although the examples in this book are trivial, the potential applications of this technology are profound. This is the technology that almost all wearable and Internet of Things (IoT) technologies employ at some level from configuration to data reporting. To program Arduino with BLE, you will need the Arduino board and the integrated development environment (IDE).

Arduino 101 (Board)

The Arduino 101 features the powerful Intel Curie processor that has onboard Bluetooth Low Energy and a variety of sensors, all packaged in a small, programmable circuit board that's easy to program and use (Figure 2-1):

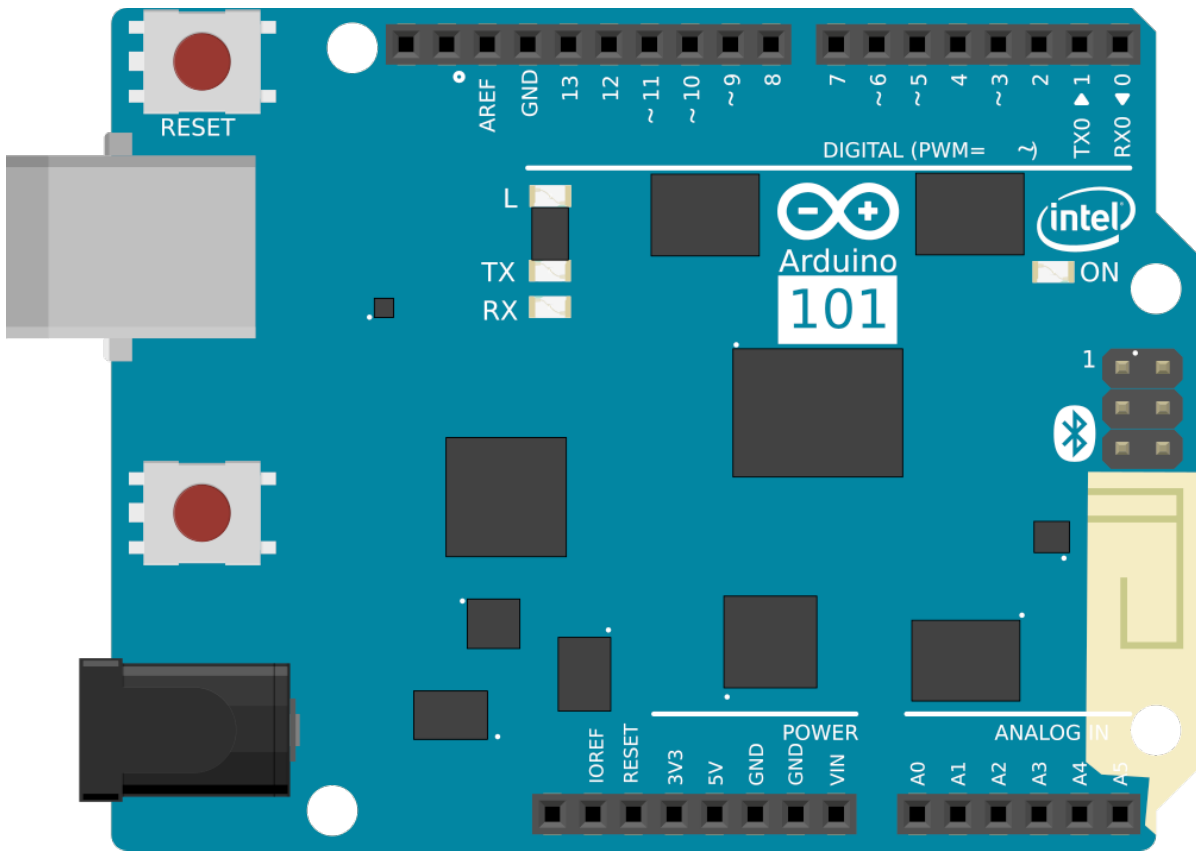


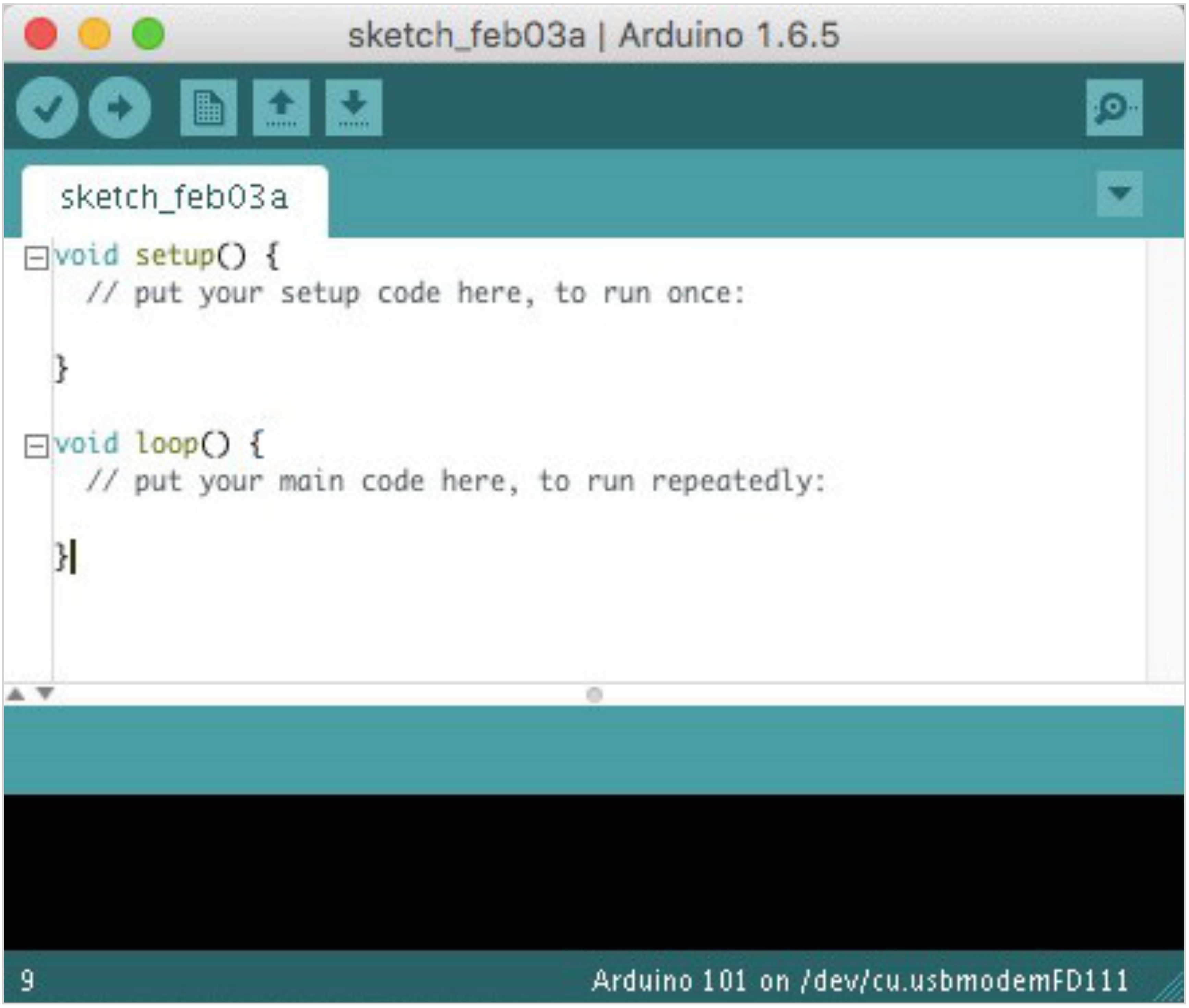
Figure 2-1. Arduino 101 board

The Arduino 101 board can be purchased from http://arduino.cc.

Arduino IDE

The Arduino IDE is a text editor that lets you program and interact with the Arduino board.

It looks like this (Figure 2-2).


sketch_feb03a | Arduino 1.6.5
sketch_feb03a

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

9
Arduino 101 on /dev/cu.usbmodemFD111

Figure 2-2. Arduino IDE

The Arduino IDE can be downloaded from http://arduino.cc. It's best to download the latest stable version for your operating system, and the Arduino 101 requires the Arduino IDE version 1.6.5 or higher.

Setup

Once you've downloaded, unpacked, and installed the Arduino IDE, you can run it.

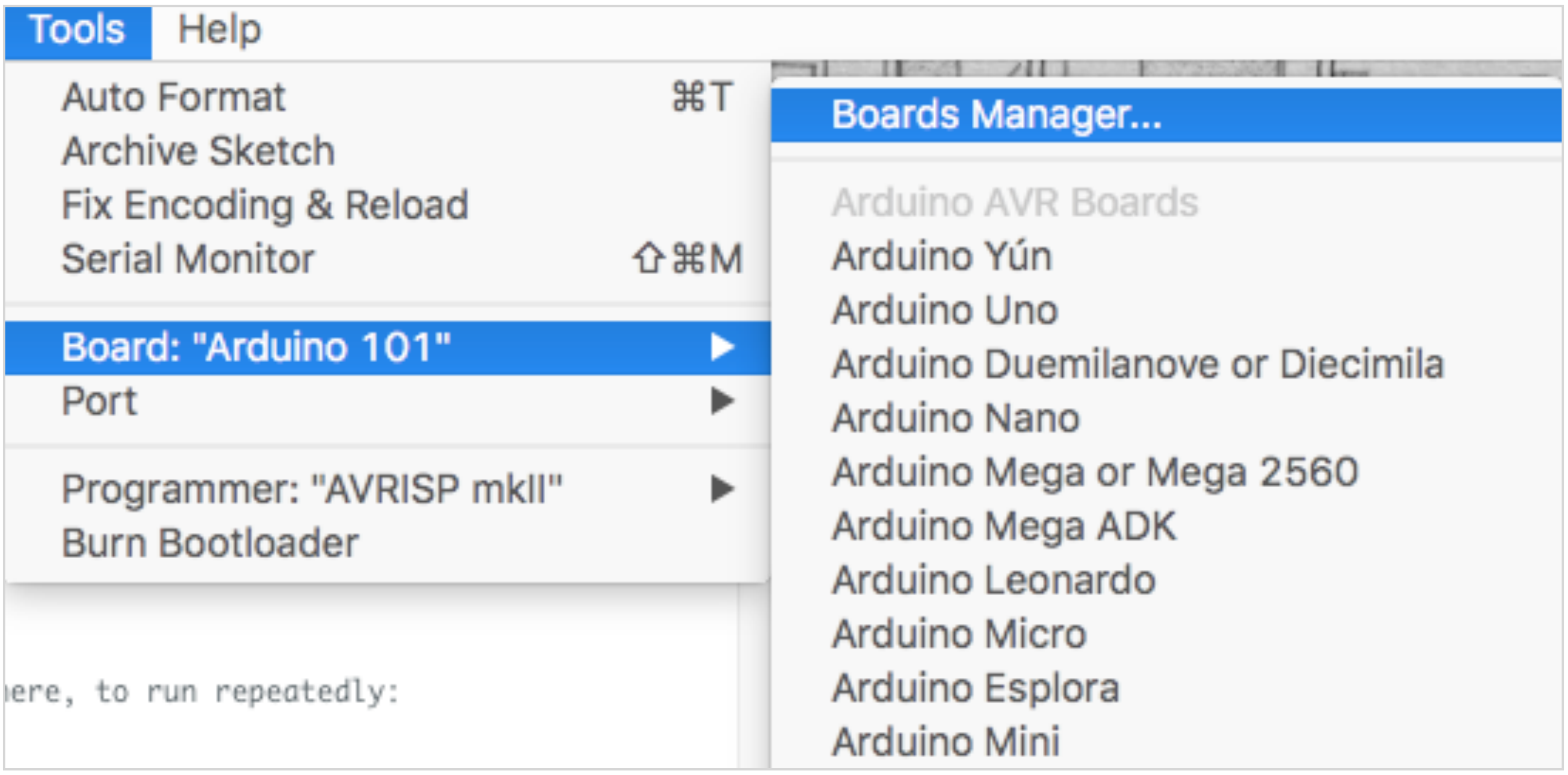
The Arduino IDE allows supports many boards in addition to Arduino boards, with its Board Manager. The Board Manager allows you to install and upgrade the software that programs the various boards you may want to use with the Arduino IDE.

In case your IDE does not natively support the Arduino 101 board, you must install support for it.

To check if you need to install support for the Arduino 101 board, go to Tools → Board in the menu and look for “Arduino 101” in the list of boards. If “Arduino 101” is in the list, you may begin programming! Otherwise, you must install support for it.

Install 101 (Curie) support

In the Arduino IDE menu, go to Tools → Board → Boards Manager… (Figure 2-3):

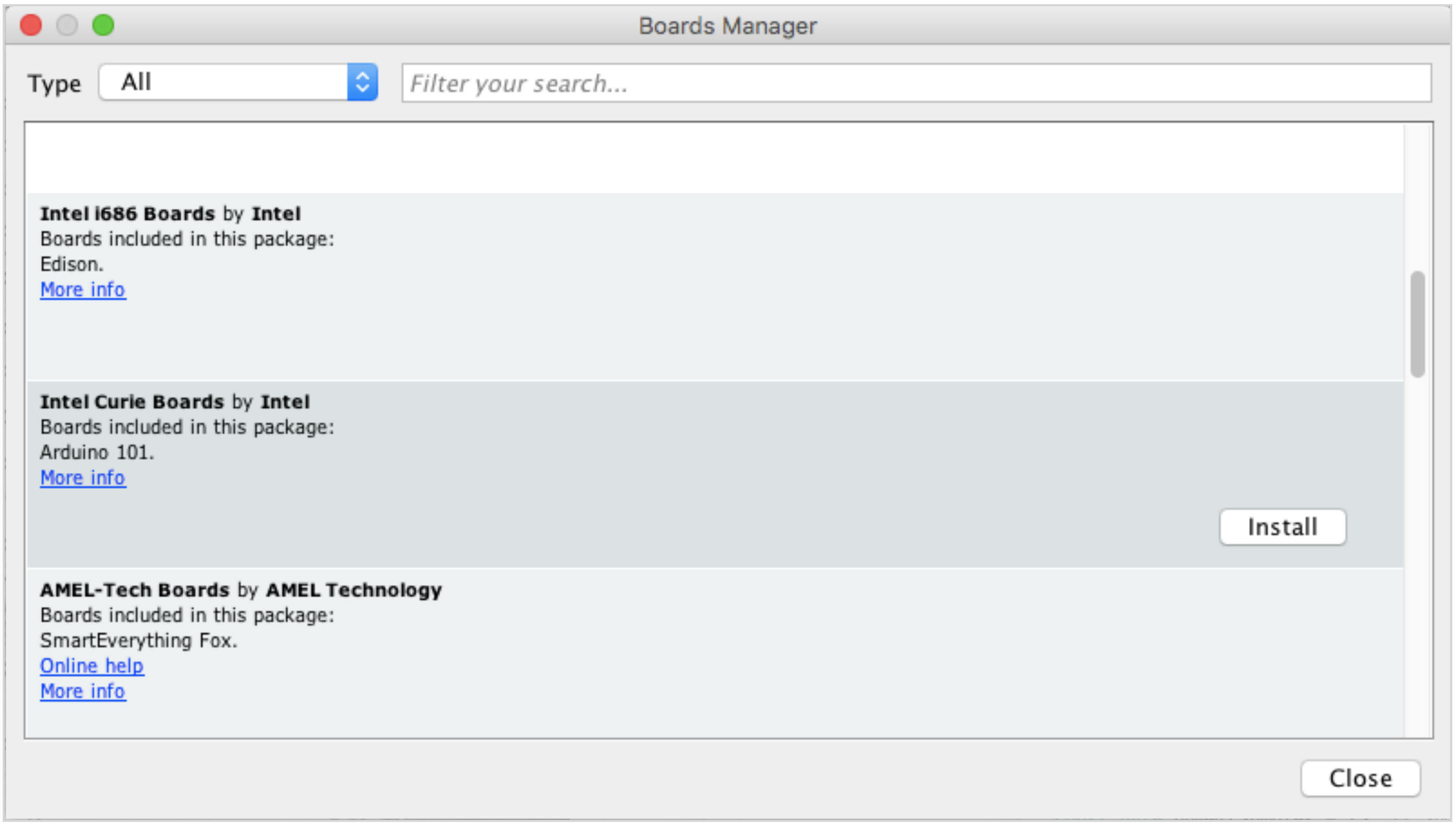


```
ere, to run repeatedly:
```

Figure 2-3. Arduino Boards Manager Menu

A screen will open where you can choose to install Intel Curie support for Arduino.

Click Install to support Arduino 101 (Figure 2-4).



Intel i686 Boards by **Intel**
Boards included in this package:
Edison.
More info

Intel Curie Boards by **Intel**
Boards included in this package:
Arduino 101.
More info

AMEL-Tech Boards by **AMEL Technology**
Boards included in this package:
SmartEverything Fox.
Online help
More info

Figure 2-4. Add board support in the Arduino Boards Manager

Connect your Arduino board to your computer, then tell the Arduino IDE that you will be programming an Arduino 101 by selecting it from the menu list under Tools → Board.

Finally, select the port that the Arduino 101 is connected to by selecting the Arduino 101 from Tools → Port

Now you may begin programming your Arduino 101

Bootstrapping

The first thing to do in any software project is to become familiar with the environment.

Because we are working with Bluetooth, it's important to learn how to initialize the Bluetooth radio and report what the program is doing.

Both the Central and Peripheral talk to the computer over USB when being programmed. That allows you to report errors and status messages to the computer when the programs are running. (Figure 3-1).

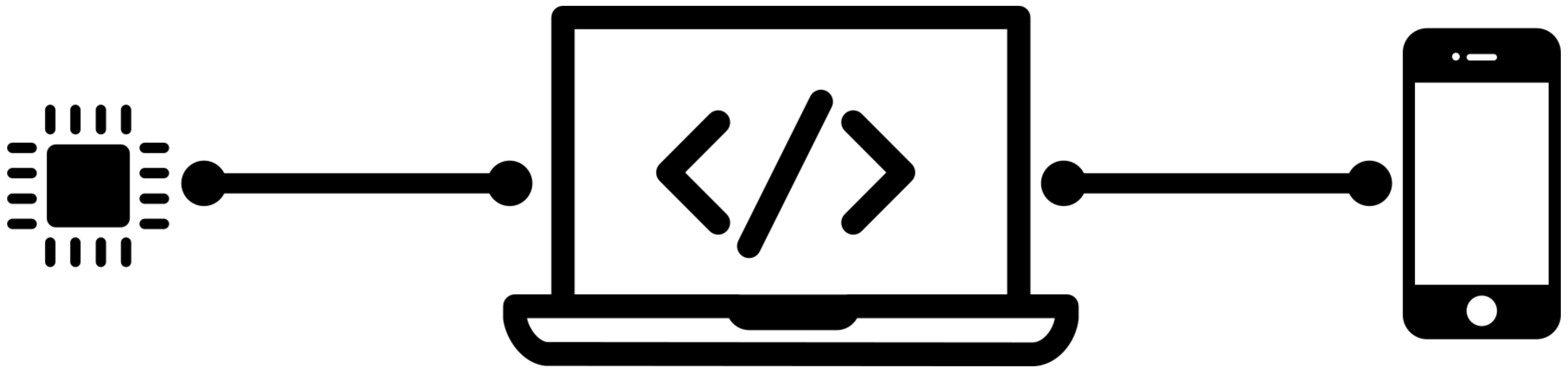


Figure 3-1. Programming configuration

Programming the Peripheral

This chapter details how to create a Central App that turns the Bluetooth Radio on You will need to print out what Arduino is thinking to help debugging later, so let's learn how to report data from the Arduino back to the computer you are programming on.

You will need to print out what Arduino is thinking to help debugging later, so let's learn how to report data from the Arduino back to the computer you are programming on.

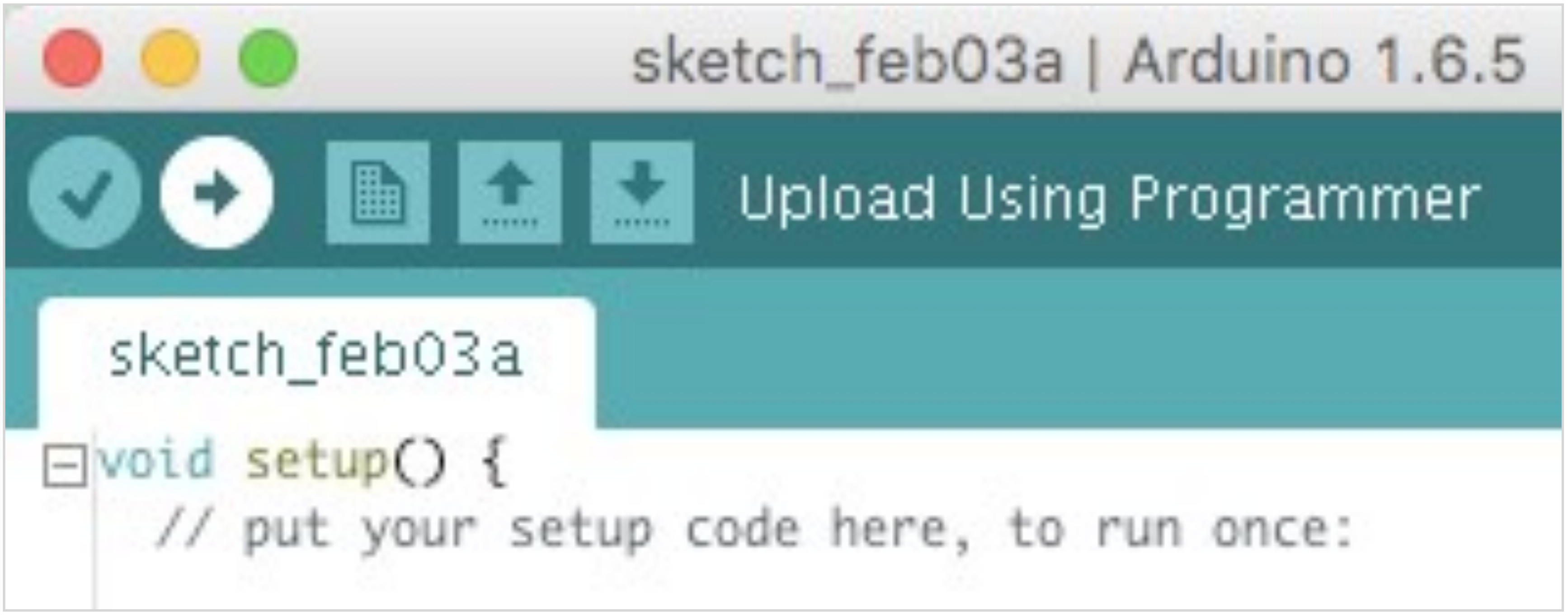
In your Arduino IDE create a new project titled ble_serial, and type the following code:

```
void setup() {
  Serial.begin(9600);
  Serial.println("Hello World!");
}

void loop() {
}
```

Run this program by clicking the Upload button on the Arduino IDE to compile the program and upload it to the Arduino.

The code structure should now look like this (Figure 3-2).



```
void setup() {
  // put your setup code here, to run once:
```

Figure 3-2. Upload sketch to the Arduino board

When it is done uploading, click the Serial Monitor button (Figure 3-3).

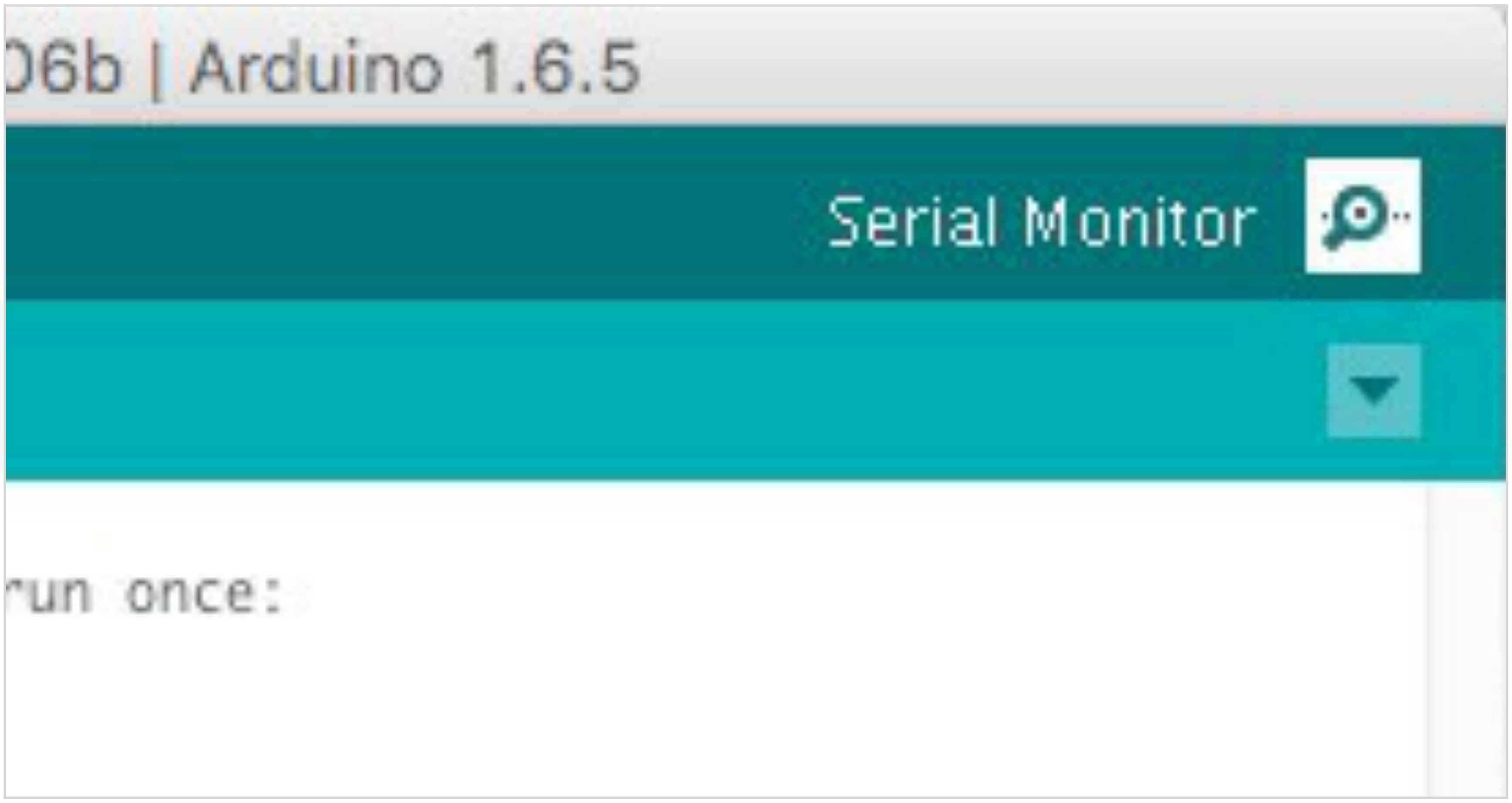


Figure 3-3. Serial Monitor Button

The resulting sketch will print “Hello World” in the Serial Monitor, like this (Figure 3-4):

```
Hello World!
```

Figure 3-4. Serial Monitor Output

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter03

Scanning and Advertising

The first step to any Bluetooth Low Energy interaction is for the Peripheral to make the Central aware of its existence, through a process called Advertising.

During the Advertising process, a Peripheral Advertises while a Central scans.

Bluetooth devices discover each other when they are tuned to the same radio frequency, also known as a Channel. There are three channels dedicated to device discovery in Bluetooth Low Energy: (Table 4-1):

Table 4-1. Bluetooth Low Energy Discovery Radio Channels

Channel	Radio Frequency
37	2402 Mhz
39	2426 Mhz
39	2480 Mhz

The peripheral will advertise its name and other data over one channel and then another. This is called frequency hopping (Figure 4-1).

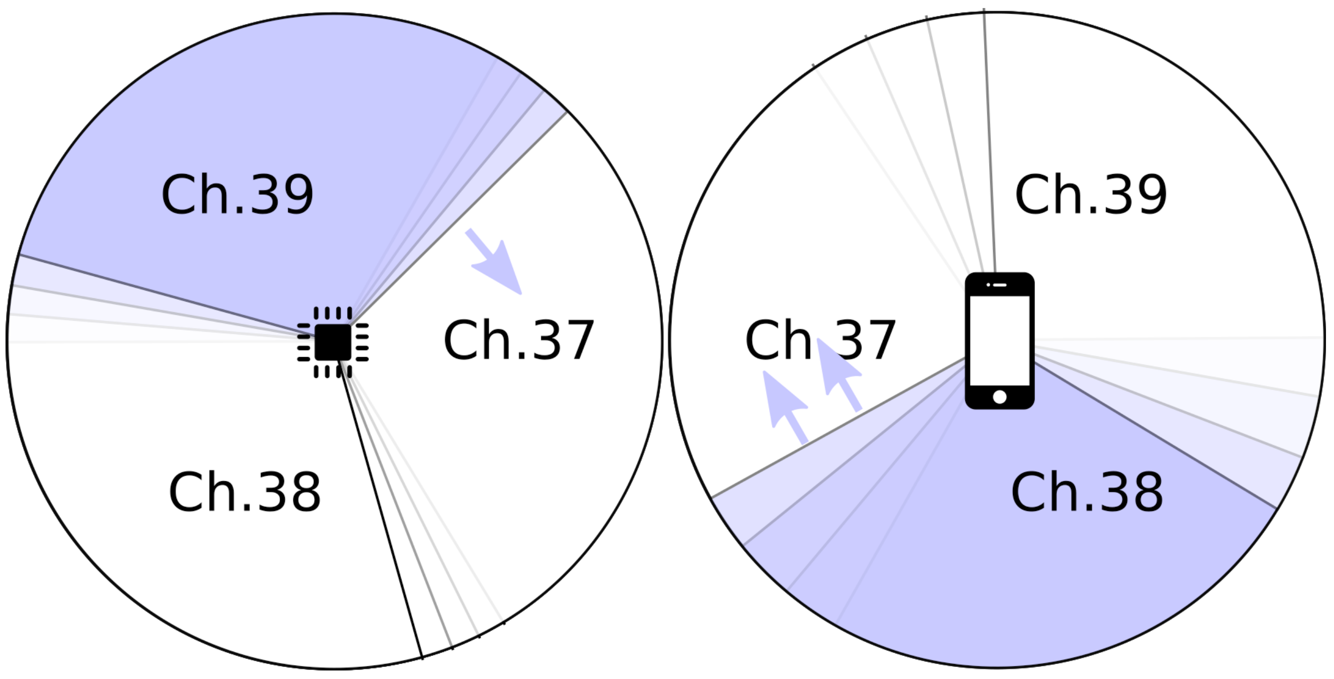


Figure 4-1. Advertise and scan processes

Similarly, the Central listens for advertisements first on one channel and then another. The Central hops frequencies faster than the Peripheral, so that the two are guaranteed to be on the same channel eventually.

Peripherals may advertise from 100ms to 100 seconds depending on their configuration, changing channels every 0.625ms (Figure 4-2).

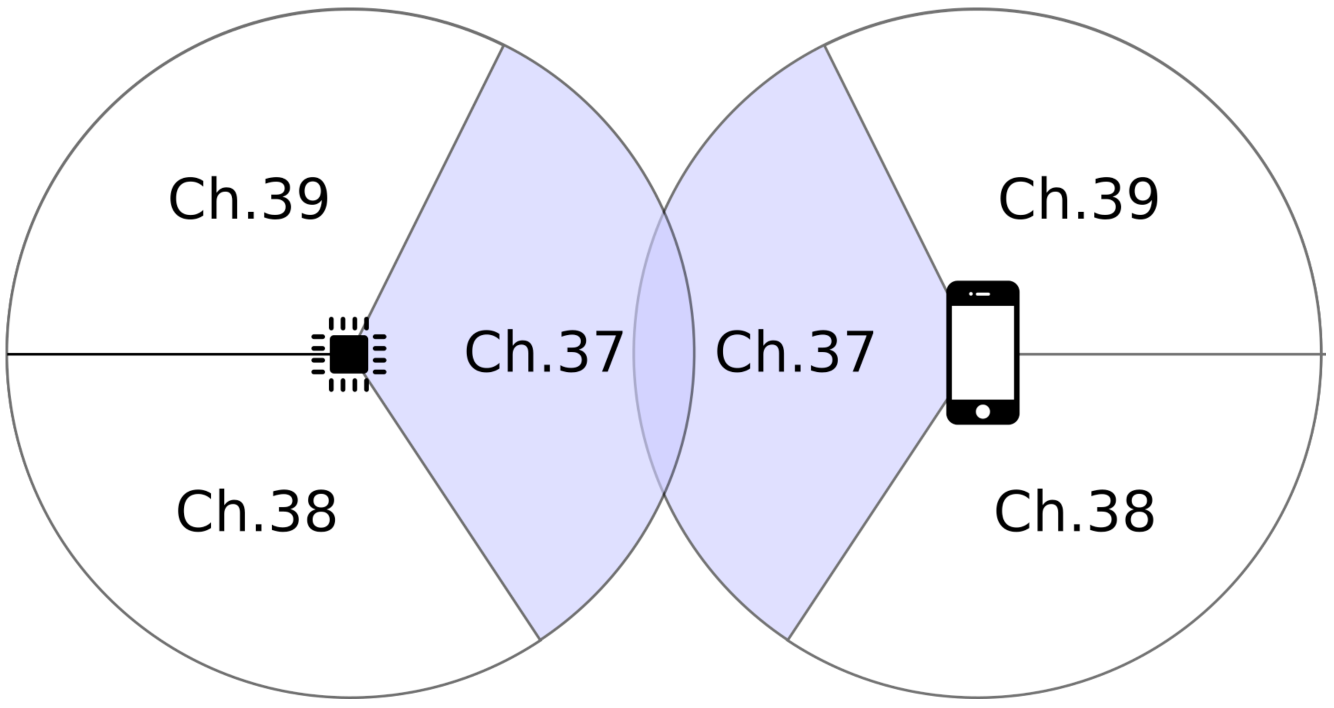


Figure 4-2. Scan finds Advertiser

Scanning settings vary wildly, for example scanning every 10ms for 100ms, or scanning for 1 second for 10 seconds.

Typically when a Central discovers advertising Peripheral, the Central requests a Scan Response from the Peripheral. In some cases, the Scan Response contains useful data. For example, iBeacons use Scan Response data to inform Centrals of each iBeacon's location without the Central needing to connect and download more data.

The first step to any Bluetooth Low Energy interaction is for the Peripheral to make the Central aware of its existence, through a process called Advertising.

Advertising reports the server name and other information one channel at a time until there are no more channels and the server repeats the process again at the first channel.

The Peripheral may start or stop advertising at any time.

Programming the Peripheral

There are several things you need to advertise over BLE. First, you must include Curie BLE support:

```
#include <CurieBle.h>
```

A BlePeripheral must be created, a name set, and then advertising started.

```
BlePeripheral blePeripheral;
setup() {
  blePeripheral.setLocalName("MyDevice");
  blePeripheral.begin();
}
```

Putting it All Together

Create a new sketch and save it as ble_advertise.

Here is how to create a simple BLE Advertise server:

Example 4-1. sketches/ble_advertise/ble_advertise.ino

```
#include <CurieBle.h>

static const char* bluetoothDeviceName = "MyDevice"; // name the device

BLEPeripheral blePeripheral; // initialize bluetooth

void setup() {
  blePeripheral.setLocalName(bluetoothDeviceName); // set the advertise name
  blePeripheral.begin(); // start advertising
}

void loop() {}
```

This sketch will create a Peripheral that advertises as “MyDevice.” Nearby Centrals will see this Peripheral when scanning.

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter04

Connecting

Once a Central has discovered a Peripheral, the central can attempt to connect. This must be done before data can be passed between the Central and Peripheral. A Central may hold several simultaneous connections with a number of peripherals, but a Peripheral may only hold one connection at a time. Hence the names Central and Peripheral (Figure 5-1).

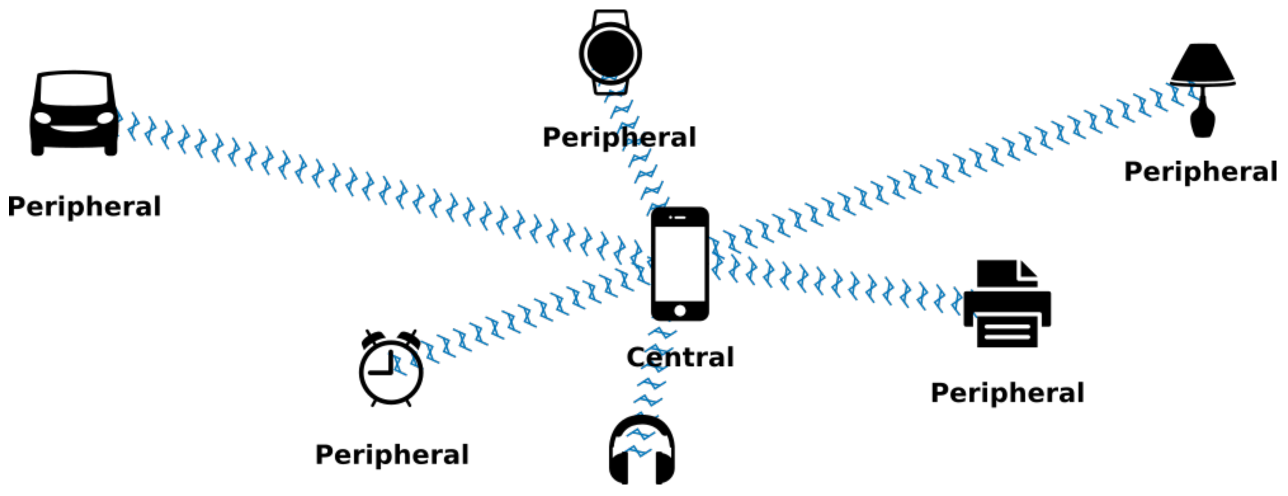


Figure 5-1. Bluetooth network topology

Bluetooth supports data 37 data channels ranging from 2404 MHz to 2478 MHz.

Once the connection is established, the Central and Peripheral negotiate which of these channels to begin communicating over. As part of this, a unique Media Access Control address (MAC) of the Central is sent to the Peripheral.

A MAC address is a 48-bit address given to every network device. It is typically represented in a hexadecimal format, similar to this:

```
08:00:27:0E:25:B8
```

Because the Peripheral can only hold one connection at a time, it must disconnect from the Central before a new connection can be made.

The connection and disconnection process works like this (Figure 5-2).

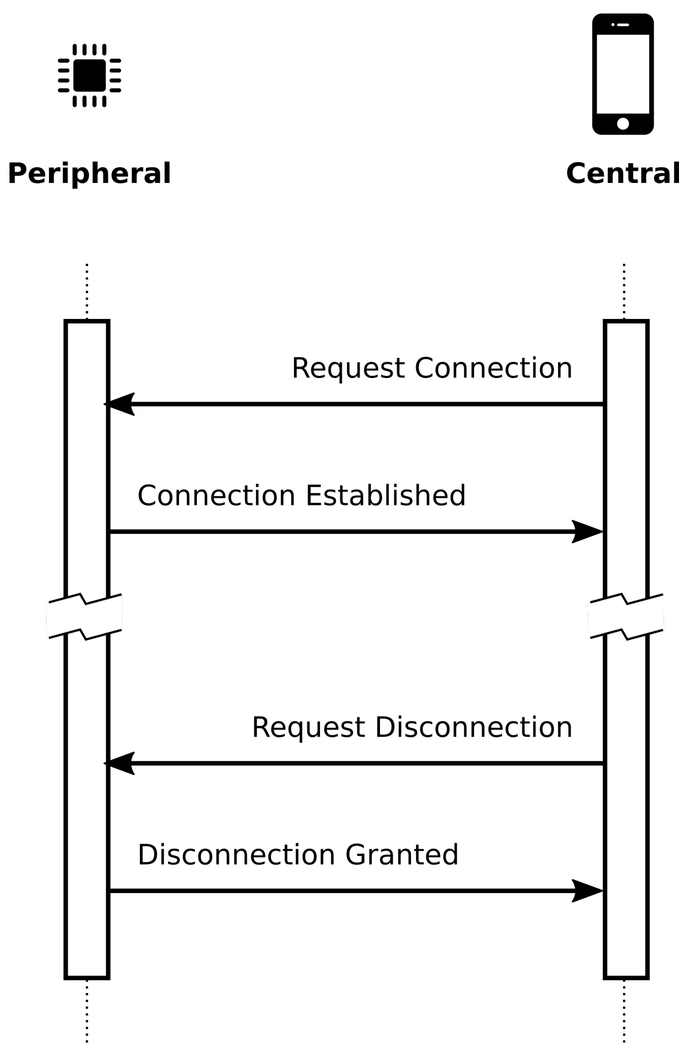


Figure 5-2. Connection and disconnection process

Programming the Peripheral

Once a Central has connected, its unique MAC address can be accessed like this:

```
...
central.address();
...
```

Callback functions attached to the BLEConnected and BLEDisconnected event enable the Peripheral to react to connection and disconnection events:

```
// bind a callback when a device connects
blePeripheral.setEventHandler(BLEConnected, onCentralConnected);
```

```
// attach callback when client disconnects
blePeripheral.setEventHandler(BLEDisconnected, onCentralDisconnected);
...
// Central connected. Print MAC address
void onCentralConnected(BLECentral& central) {
  Serial.print("Central connected: ");
  Serial.println(central.address());
}
// Central disconnected
void onCentralDisconnected(BLECentral& central) {
  Serial.print("Central disconnected");
}
...
```

Putting It All Together

Create a new sketch and save it as ble_connect.

This sketch will advertise as well as manage a connection to a Central. When a Central connects, the Peripheral will print the Central's MAC address into the Serial Monitor.

Example 5-1. sketches/ble_connect/ble_connect.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
BLEPeripheral blePeripheral;
// Central connected.  Print MAC address
void onCentralConnected(BLECentral& central) {
  Serial.print("Central connected: ");
  Serial.println(central.address());
}
// Central disconnected
```

```
void onCentralDisconnected(BLECentral& central) {
  Serial.print("Central disconnected: ");
  Serial.println(central.address());
}
void setup() {
  blePeripheral.setLocalName(bluetoothDeviceName);
  // attach callback when Central connects
  blePeripheral.setEventHandler(
    BLEConnected,
    onCentralConnected
  );
  // attach callback when Central disconnects
  blePeripheral.setEventHandler(
    BLEDisconnected,
    onCentralDisconnected
  );
  blePeripheral.begin();
}
void loop() {}
```

The output from the Serial monitor should resemble this when a Central connects, then disconnects from the Peripheral (Figure 5-3).

```
Central connected: 45:b5:7d:96:01:2f
Central disconnected
```

Figure 5-3. Serial Monitor Output

Your Peripheral can now handle connections and disconnections. You can modify the callback functions turn on an LED, print something to screen, or anything else.

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter05

Services and Characteristics

Before data can be transmitted back and forth between a Central and Peripheral, the Peripheral must host a GATT Profile. That is, the Peripheral must have Services and Characteristics.

Identifying Services and Characteristics

Each Service and Characteristic is identified by a Universally Unique Identifier (UUID). The UUID follows the pattern 0000XXXX-0000-1000-8000-00805f9b34fb, so that a 32-bit UUID 00002a56-0000-1000-8000-00805f9b34fb can be represented as 0x2a56.

Some UUIDs are reserved for specific use. For instance any Characteristic with the 16-bit UUID 0x2a35 (or the 32-bit UUID 00002a35-0000-1000-8000-00805f9b34fb) is implied to be a blood pressure reading.

For a list of reserved Service UUIDs, see ***Appendix IV: Reserved GATT Services***.

For a list of reserved Characteristic UUIDs, see ***Appendix V: Reserved GATT Characteristics***.

Generic Attribute Profile

Services and Characteristics describe a tree of data access points on the peripheral. The tree of Services and Characteristics is known as the Generic Attribute (GATT) Profile. It may be useful to think of the GATT as being similar to a folder and file tree (Figure 6-1).

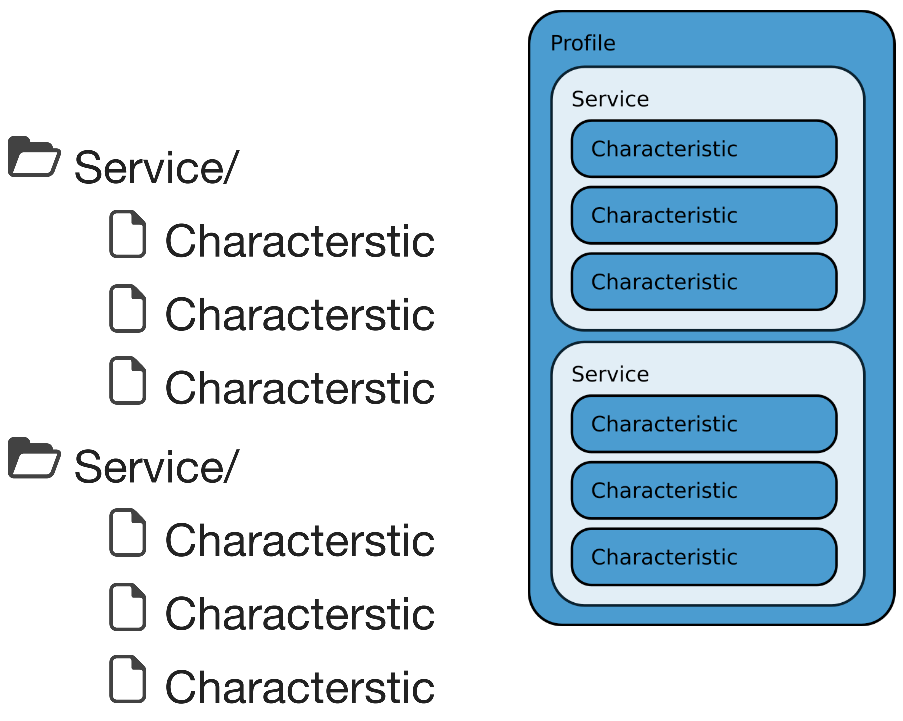


Figure 6-1. GATT Profile filesystem metaphor

Characteristics act as channels that can be communicated on, and Services act as containers for Characteristics. A top level Service is called a Primary service, and a Service that is within another Service is called a Secondary Service.

Permissions

Characteristics can be configured with the following attributes, which define what the Characteristic is capable of doing (Table 6-1):

Table 6-1. Characteristic Permissions

Descriptor	Description
Read	Central can read this Characteristic, Peripheral can set the value.
Write	Central can write to this Characteristic, Peripheral will be notified when the Characteristic value changes and Central will be notified when the write operation has occurred.
Notify	Central will be notified when Peripheral changes the value.

Because the GATT Profile is hosted on the Peripheral, the terms used to describe a Characteristic's permissions are relative to how the Peripheral accesses that Characteristic. Therefore, when a Central uploads data to the Peripheral, the Peripheral can "read" from the Characteristic. The Peripheral "writes" new data to the Characteristic, and can "notify" the Central that the data is altered.

Data Length and Speed

It is worth noting that Bluetooth Low Energy has a maximum data packet size of 20 bytes, with a 1 Mbit/s speed.

Programming the Peripheral

The Generic Attribute Profile is defined by setting the UUID and permissions of the Peripheral's Services and Characteristics.

Characteristics can be configured with the following permissions (Table 6-2):

Table 6-2. BLECharacteristic Permissions

Value	Permission	Description
BleRead	**Read**	Central can read data altered by the Peripheral
BleWrite	**Write**	Central can send data, Peripheral reads
BleNotify	**Notify**	Central is notified as a result of a change

Characteristics have a maximum length of 20 bytes. Since 16 bit and 8-bit data types are easy to pass around in C++, we will be using uint16_t (unsigned 16-bit integer) and uint8_t (unsigned 8-bit integer) values in the examples. Any data type including custom byte buffers can be transmitted and assembled over BLE.

Define a Service with UUID 180c (an unregistered generic UUID):

```
BLEService service("180C");
```

or

```
BLEService service("0000180C-000-1000-8000-00805f9-b34fb");
```

The first method lets the Peripheral automatically generate most of the UUID, and the second method forces the Peripheral to use a particular UUID. The first method is simpler but less precise. The second method is precise and useful for projects where there is a need to share the UUID with outside people or APIs.

Note: Certain UUIDs are unavailable for use. If a bad UUID is chosen, the Peripheral may crash without warning.

There are several types of Characteristic available in Arduino 101, depending on the type of data you need to transmit. Arrays, Integers, Floats, Booleans, and other data types have their own Characteristic constructors.

For instance, this 2-byte long Characteristic with UUID 1801 can be read by a Central and can notify the Central of changes:

```
BLECharacteristic readCharacteristic(
    “1801”, BLERead | BLENotify, 2
);
```

This 8-byte long Characteristic with UUID 2A56 (Digital Characteristic) can be written to by the Central:

```
BLECharacteristic writeCharacteristic(“2A56”, BLEWrite, 8);
```

Here are some examples of various data type specific Characteristics that can be created:

```
int properties = BLERead | BLEWrite | BLENotify;
BLEBoolCharacteristic \
    booleanCharacteristic(UUID, properties, maxLen);
BLEIntCharacteristic \
    integerDataCharacteristicName(UUID, properties, maxLen);
BLEUnsignedIntCharacteristic \
    yourCharacteristicName(UUID, properties, maxLen);
BLELongCharacteristic \
    yourCharacteristicName(UUID, properties, maxLen);
BLEUnsignedLongCharacteristic \
    yourCharacteristicName(UUID, properties, maxLen);
BLEFloatCharacteristic \
    yourCharacteristicName(UUID, properties, maxLen);
```

The Services and Characteristics are added to the GATT Profile via the BLEPeripheral. By adding the two Characteristics after the Service, they are assumed to be part of the same Service. This must happen before blePeripheral.begin().

```
...
```

```
BLEPeripheral blePeripheral;
...
blePeripheral.setAdvertisedServiceUuid(service.uuid());  // add service UUID
blePeripheral.addAttribute(service);   // Add the BLE Heart Rate service
blePeripheral.addAttribute(readCharacteristic); // add read characteristic
blePeripheral.addAttribute(writeCharacteristic); // add read characteristic
blePeripheral.begin();
...
```

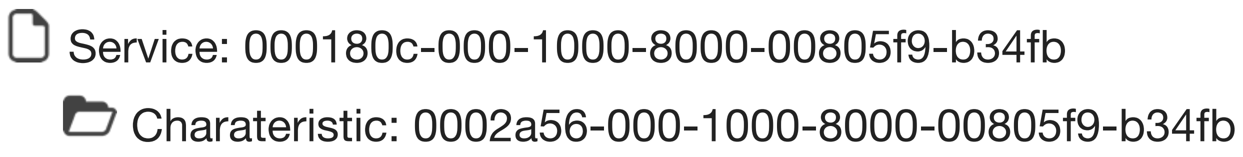
Putting It All Together

Create a new sketch named ble_characteristics and copy the following code.

Example 6-1. sketches/ble_characteristics/ble_characteristics.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
// Unregulated Service
static const char* serviceUuid = "180C";
// Unregulated Charactersitic
static const char* characteristicUuid = "2A56";
// 20 byte transmission
static const int   characteristicTransmissionLength = 20;
// create a service
BLEService service(serviceUuid);
// create a characteristic with Read and write attributes
BLECharacteristic characteristic(
  characteristicUuid,
  BLERead | BLEWrite,
  characteristicTransmissionLength
);
BLEPeripheral blePeripheral;
void setup() {
  blePeripheral.setLocalName(bluetoothDeviceName);
  Serial.println(bluetoothDeviceName);
  blePeripheral.setAdvertisedServiceUuid(service.uuid()); // attach service
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(characteristic); // attach characteristic
  blePeripheral.begin();
}
void loop() {}
```

When run, this sketch will create a Peripheral that advertises as “MyDevice” and will have a GATT profile featuring a single Characteristic with read and write permissions (Figure 6-2).



Service: 000180c-000-1000-8000-00805f9-b34fb

Charateristic: 0002a56-000-1000-8000-00805f9-b34fb

Figure 6-2. GATT Profile hosted on the Arduino 101

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter06

Reading Data from a Peripheral

The real value of Bluetooth Low Energy is the ability to transmit data wirelessly.

Bluetooth Peripherals are passive, so they don't push data to a connected Central. Instead, Centrals make a request to read data from a Characteristic. This can only happen if the Characteristic enables the Read Attribute.

This is called "reading a value from a Characteristic."

Therefore, if a Peripheral changes the value of a Characteristic, then later a Central downloads data from the Peripheral, the process looks like this (Figure 7-1):

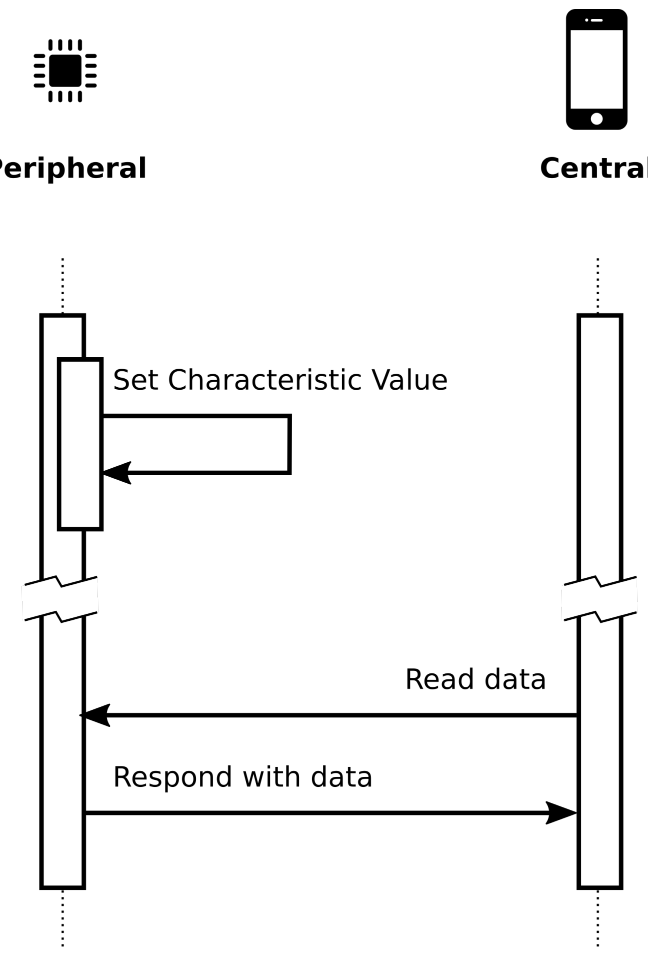


Figure 7-1. The process of a Central reading data from a Peripheral

A Central can read a Characteristic repeatedly, regardless if Characteristic's value has changed.

Programming the Peripheral

Create a Service and a Characteristic with read properties up to 20-bytes long. For example, a 16-byte Characteristic:

```
...
BLEService service(“180C”);
BLECharacteristic characteristic(
  “2A56”,
  BLERead, // readable from client's perspective
  16 // 16-byte long characteristic
);
setup() {
  ...
  // add service
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  // publish service
  blePeripheral.addAttribute(service);
  // add characteristic to service
  blePeripheral.addAttribute(characteristic);
  ...
}
```

Each Service and Characteristic must be uniquely identifiable with a UUID.

With that, the data in Characteristic can be set locally and a connected Central can read from it.

```
uint16_t someValue = 1024;
characteristic.setValue(someValue);
```

Putting It All Together

Create a new sketch called ble_send_data and copy the following code into the new sketch.

This sketch sets the value of the Characteristic to a random string every 5 seconds.

Example 7-1. sketches/ble_send_data/ble_send_data.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
static const char* serviceUuid = "180C";
static const char* characteristicUuid = "2A56";
static const int   characteristicTransmissionLength = 20;
char randomString[20] = {0};
BLEService service(serviceUuid);
BLECharacteristic characteristic(
  characteristicUuid,
  BLERead, // readable from client's perspective
  characteristicTransmissionLength
);
BLEPeripheral blePeripheral;
unsigned long lastBleCharacteristicUpdateTime_ms = 0;
unsigned long updateTimeout_ms = 5000; // update every 5 seconds
void generateRandomString(const int stringLength, char* outputString) {
  static const char alphanum[] =
    "0123456789"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  int alphanumLength = 36;
  outputString[stringLength];
  for (int i = 0; i < stringLength; i++) {
    outputString[i] = alphanum[random(0, alphanumLength)];
  }
  outputString[stringLength] = '\0';
}
void setBleCharacteristicValue(char* output, int length) {
  characteristic.setValue((const unsigned char*) output, length);
}
```

```
void setup() {
  blePeripheral.setLocalName(bluetoothDeviceName);
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(characteristic);
  blePeripheral.begin();
  randomSeed(analogRead(0)); // initialize random numbers
  lastBleCharacteristicUpdateTime_ms = millis();
}
void loop() {
  unsigned long currentTime_ms = millis();
  if ((currentTime_ms - lastBleCharacteristicUpdateTime_ms) > \
      updateTimeout_ms)
  {
    lastBleCharacteristicUpdateTime_ms = currentTime_ms;
    int randomStringLength = random(1, characteristicTransmissionLength);
    generateRandomString(randomStringLength, randomString);
    setBleCharacteristicValue(randomString, randomStringLength);
  }
}
```

When the sketch is run, the Serial Monitor output should resemble this (Figure 7-2):

```
Setting characteristic to: RY0G5WSWG5YTh
Setting characteristic to: HFVTh
Setting characteristic to: 88L9J25245ZUh
Setting characteristic to: A6D7SZZFYQ8Sh
```

Figure 7-2. Serial Monitor Output

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter07

Writing Data to a Peripheral

Data is sent from the Central to a Peripheral when the Central writes a value in a Characteristic hosted on the Peripheral, presuming that Characteristic has write permissions.

The process looks like this (Figure 8-1):

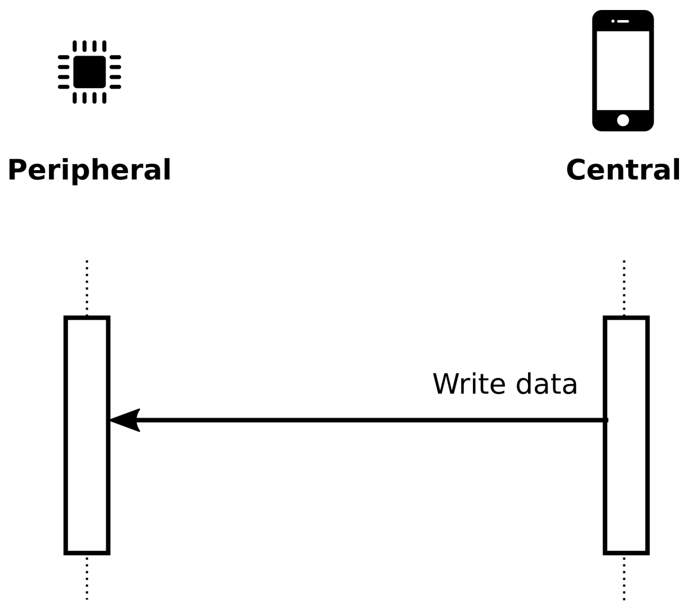


Figure 8-1. The process of a Central writing data to a Peripheral

Programming the Peripheral

To allow the Peripheral to receive data from a Central, it must have a Characteristic that gives permission to be written.

For example, this 8-byte characteristic supports “write” operations:

```
BLECharacteristic characteristic("2A56", BLEWrite, 8);
```

The Characteristic must have a callback to describe how to process incoming data. The callback comes back with a connected central device and the characteristic that was written to. The characteristic contains the details of the transmission, including the length and the value of the transmission.

For example, it is possible to extract the incoming 8-bit data value and print it to Serial.

```
void onBleCharacteristicWritten(
    BLECentral& central,
    BLECharacteristic &characteristic) {
  uint8_t value = characteristic.value();
  Serial.print("Data written to characteristic: ");
  Serial.println(characteristic.uuid());
  Serial.println(value);
}
```

The callback must be bound to the Characteristic like this:

```
characteristic.setEventHandler(
  BLEWritten,
  onBleCharacteristicWritten
);
```

The callback is handled in an interrupt, outside normal operation. It is a good practice not to implement a lot of functionality inside the callback. A best practice is to set a boolean flag, then look for that flag inside the main loop.

For example, it is possible to store the newly written Characteristic value inside a struct while in the callback, then print that data to Serial in the main loop:

```
...
// store details about transmission here
struct BleTransmission {
  char data[characteristicTransmissionLength];
  unsigned int length;
  const char* uuid;
};
BleTransmission bleTransmissionData; // structure contains BLE transmission
bool bleDataWritten = false; // true if data has been received
...
void onBleCharacteristicWritten(
    BLECentral& central,
    BLECharacteristic &characteristic) {
  bleDataWritten = true;
  bleTransmissionData.uuid = characteristic.uuid();
  bleTransmissionData.length = characteristic.valueLength();
  // Since we are playing with strings, we must use strncpy
  strncpy(bleTransmissionData.data,
    (char*) characteristic.value(), characteristic.valueLength());
}
...
void loop() {
  // if the bleDataWritten flag has been set, print out the incoming data
  if (bleDataWritten) {
    bleDataWritten = false; // ensure only happens once
    Serial.print(bleTransmissionData.length);
    Serial.print(" bytes sent to characteristic ");
    Serial.print(bleTransmissionData.uuid);
    Serial.print(": ");
    Serial.println(bleTransmissionData.data);
  }
}
```

Putting It All Together

Create a new sketch named ble_receive_data with the following code.

Example 8-1. sketches/ble_receive_data/ble_receive_data.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
static const char* serviceUuid = "180C";
static const char* characteristicUuid = "2A56";
static const int   characteristicTransmissionLength = 20;
// store details about transmission here
struct BleTransmission {
  char data[characteristicTransmissionLength];
  unsigned int length;
  const char* uuid;
};
BleTransmission bleTransmissionData;
bool bleDataWritten = false; // true if data has been received
BLEService service(serviceUuid);
BLECharacteristic characteristic(
  characteristicUuid,
  BLEWrite, // writable from client's perspective
  characteristicTransmissionLength
);
BLEPeripheral blePeripheral;
// When data is sent from the client, it is processed here inside a callback
// it is best to handle the result of this inside the main loop
void onBleCharacteristicWritten(BLECentral& central,
    BLECharacteristic &characteristic) {
  bleDataWritten = true;
  bleTransmissionData.uuid = characteristic.uuid();
  bleTransmissionData.length = characteristic.valueLength();
  // Since we are playing with strings, we must use strncpy
  strncpy(bleTransmissionData.data,
    (char*) characteristic.value(), characteristic.valueLength());
}
void setup() {
  Serial.begin(9600);
  while (!Serial); // wait for Serial console to start
  blePeripheral.setLocalName(bluetoothDeviceName);
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(characteristic);
  // trigger onbleCharacteristicWritten when data is sent to the characteris-
tic
  characteristic.setEventHandler(
    BLEWritten,
    onBleCharacteristicWritten
  );
  blePeripheral.begin();
}
void loop() {
  // if the bleDataWritten flag has been set, print out the incoming data
  if (bleDataWritten) {
    bleDataWritten = false; // ensure only happens once
    Serial.print(bleTransmissionData.length);
    Serial.print(" bytes sent to characteristic ");
    Serial.print(bleTransmissionData.uuid);
    Serial.print(": ");
    Serial.println(bleTransmissionData.data);
  }
}
```

The output from the Serial monitor when a connected Central writes data looks like this (Figure 8-2):

```
5 bytes sent to characteristic 2A56: hello
5 bytes sent to characteristic 2A56: world
16 bytes sent to characteristic 2A56: Bluetooth works!
```

Figure 8-2. Serial Monitor Output

Example code

The code for this chapter is available online
at: https://github.com/BluetoothLowEnergyInArduino101/Chapter08

Using Notifications

Being able to read from the Central has limited value if the Central does not know when new data is available.

Notifications solve this problem. A Characteristic can issue a notification when it's value has changed. A Central that subscribes to these notifications will know when the Characteristic's value has changed, but not what that new value is. The Central can then read the latest data from the Characteristic.

The whole process looks something like this (Figure 9-1):

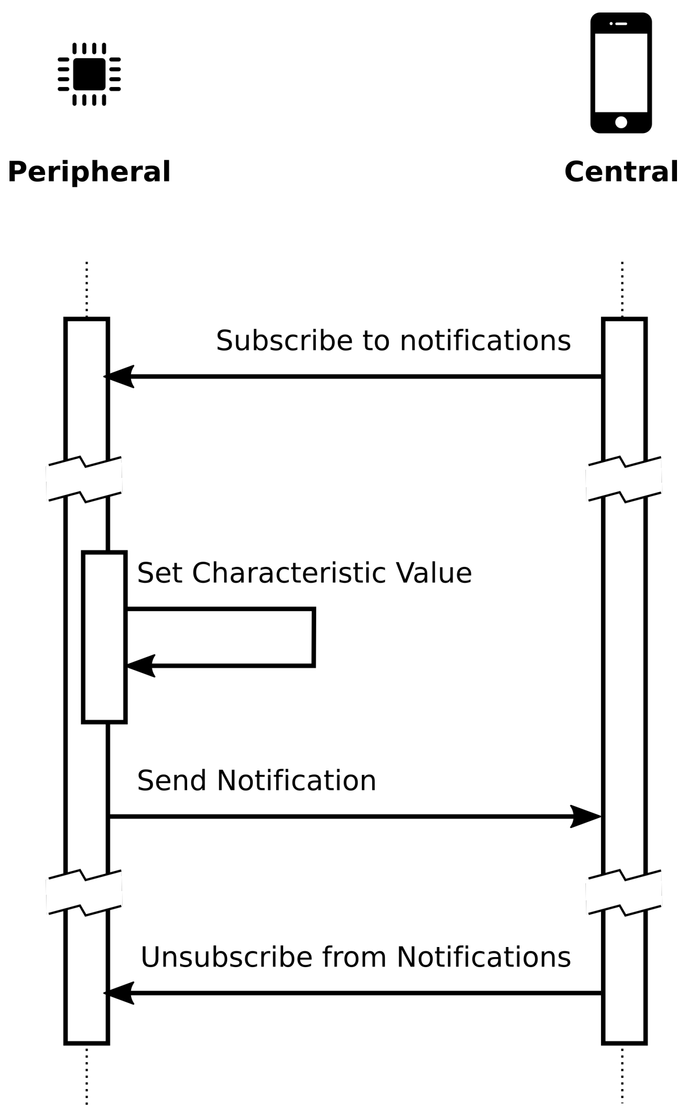


Figure 9-1. The process of a Peripheral notifying a connected Central of changes to a Characteristic

Programming the Peripheral

Often, battery life is at a premium on Bluetooth Peripherals. For this reason, it is useful to notify a connected Central when a Characteristic's value has changed, but not send the new data to the Central. Waking up the Bluetooth radio to send one byte consumes less battery than sending 20 or more bytes.

Notifications can be enabled in a Characteristic by setting the BLENotify flag.

For example, this Characteristic supports both read access and notifications.

```
BLECharacteristic characteristic(
  “2803”,
  BLERead | BLENotify, // allow client to subscribe to notifications
  20 // 20-byte Characteristic
);
```

When the value of the Characteristic is changed, a notification will be automatically sent to the connected Central.

```
characteristic.setValue("Hello World”, sizeof(“Hello World"));
```

Putting It All Together

Create a new sketch named ble_notifications with the following code.

Example 9-1. sketches/ble_notifications/ble_notifications.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
static const char* serviceUuid = "1800";
static const char* characteristicUuid = "2803";
static const int   characteristicTransmissionLength = 20;
char randomString[20] = {0};
BLEService service(serviceUuid);
```

```
BLECharacteristic characteristic(
  characteristicUuid,
  BLERead | BLENotify, // allow client to subscribe to notifications
  characteristicTransmissionLength
);
BLEPeripheral blePeripheral;
unsigned long bleCharacteristicLastUpdateTime_ms = 0;
unsigned long updateTimeout_ms = 5000;
bool bleCharacteristicSubscribed = false; // true when a client subscribes
void generateRandomString(const int stringLength, char* outputString) {
  static const char alphanum[] =
    "0123456789"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  int alphanumLength = 36;
  outputString[stringLength];
  for (int i = 0; i < stringLength; i++) {
    outputString[i] = alphanum[random(0, alphanumLength)];
  }
  outputString[stringLength] = '\0';
}
void setBleCharacteristicValue(char* output, int length) {
  characteristic.setValue((const unsigned char*) output, length);
}
void onBleCharacteristicSubscribed(BLECentral& central,
    BLECharacteristic &characteristic) {
  bleCharacteristicSubscribed = true;
  Serial.print("Characteristic ");
  Serial.print(characteristic.uuid());
  Serial.println(" subscribed to");
}
void onBleCharacteristicUnsubscribed(BLECentral& central,
    BLECharacteristic &characteristic) {
  bleCharacteristicSubscribed = false;
  Serial.print("Characteristic ");
  Serial.print(characteristic.uuid());
  Serial.println(" unsubscribed from");
}
void setup() {
  blePeripheral.setLocalName(bluetoothDeviceName);
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(characteristic);
  // notify us when a client subscribes to the characteristic
  characteristic.setEventHandler(
    BLESubscribed,
    onBleCharacteristicSubscribed
  );
  characteristic.setEventHandler(
    BLEUnsubscribed,
    onBleCharacteristicUnsubscribed
  );
  blePeripheral.begin();
  randomSeed(analogRead(0)); // initialize random numbers
  bleCharacteristicLastUpdateTime_ms = millis();
}
void loop() {
  unsigned long currentTime_ms = millis();
  if ((currentTime_ms - bleCharacteristicLastUpdateTime_ms) > updateTime-
out_ms) {
    bleCharacteristicLastUpdateTime_ms = currentTime_ms; // reset timer
    if (bleCharacteristicSubscribed) {
      int randomStringLength = random(1, characteristicTransmissionLength);
      generateRandomString(randomStringLength, randomString);
      setBleCharacteristicValue(randomString, randomStringLength);
    }
  }
}
```

When the sketch is run, the output of the Serial Monitor will resemble this (Figure 9-1):

```
Setting characteristic to: FMZAp
Setting characteristic to: GG0RPFNU7KYJCNJKp
Setting characteristic to: G665p
```

Figure 9-2. Serial Monitor Output

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter09

Streaming Data

The maximum packet size you can send over Bluetooth Low Energy is 20 bytes. More data can be sent by dividing a message into packets of 20 bytes or smaller, and sending them one at a time

These packets can be sent at a certain speed.

Bluetooth Low Energy transmits at 1 Mb/s. Between the data transmission time and the time it may take for a Peripheral to process incoming data, there is a time delay between when one packet is sent and when the next one is ready to be sent.

To send several packets of data, a queue/notification system must be employed, which alerts the Central when the Peripheral is ready to receive the next packet.

There are many ways to do this. One way is to set up a Characteristic with read, write, and notify permissions, and to flag the Characteristic as “ready” after a write has been processed by the Peripheral. This sends a notification to the Central, which sends the next packet. That way, only one Characteristic is required for a single data transmission.

This process can be visualized like this (Figure 10-1).

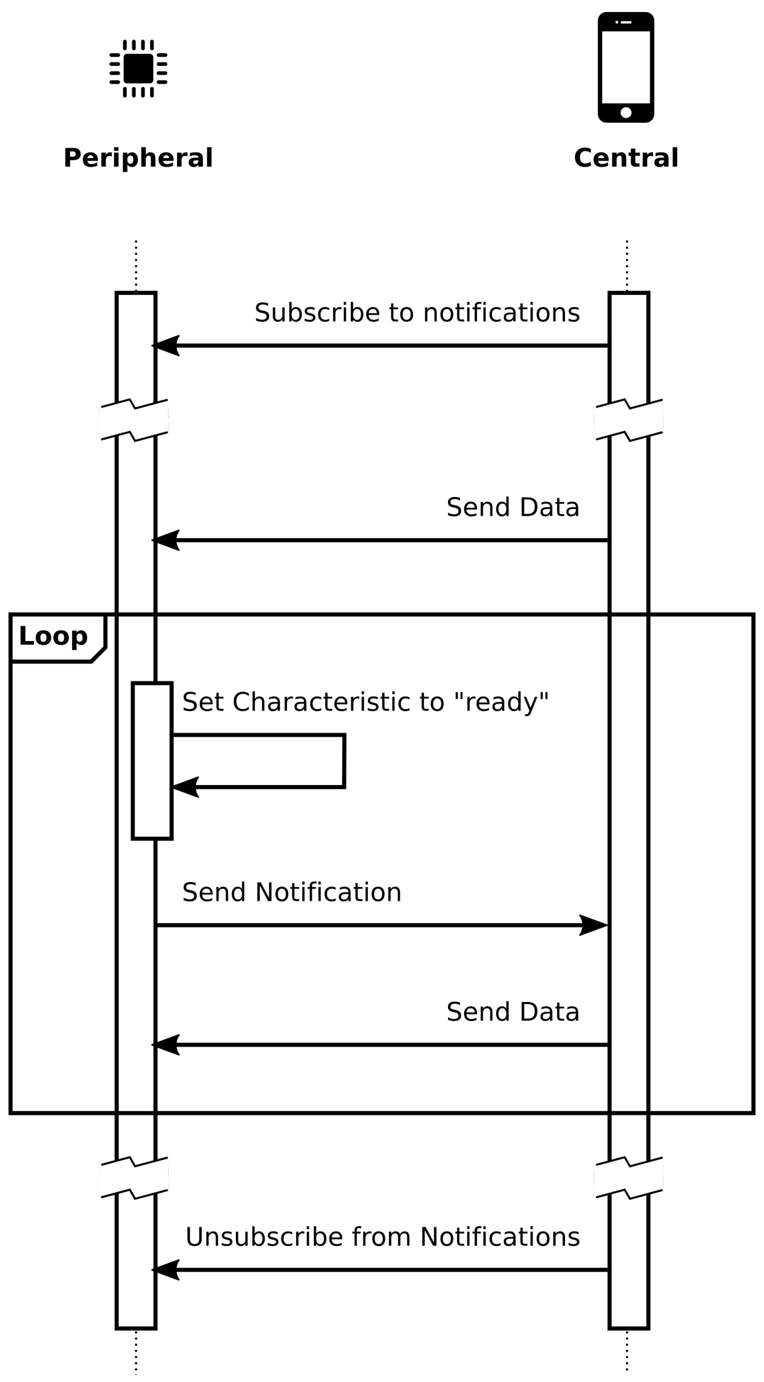

Peripheral
Central
Subscribe to notifications
Send Data
Loop
Set Characteristic to "ready"
Send Notification
Send Data
Unsubscribe from Notifications

Figure 10-1. The process of using notifications to handle flow control on a multi-packed data transfer

Programming the Peripheral

This method of implementing a queueing Characteristic requires a Characteristic that supports read, write, and notify permissions.

In this example, a 16-bit long Characteristic:

```
...
BLECharacteristic characteristic(
  "2803",
  BLERead | BLENotify | BLEWrite, // read, write, notify
  16
);
...
```

When this Characteristic is written to and the data is processed, the Characteristic’s value will be switched to a flow control message; in this example, “ready.”

Because onCharacteristicWritten is handled like an interrupt, it is a best practice to avoid processing the data inside that function. Instead, it’s better to set a flag and process the data in the main loop.

```
...
static const char* bleFlowControlMessage = "ready";
static const int   bleFlowControlMessageLength = 5;
bool bleDataWritten = false; // true if data has been received
char bleCharacteristicValue[16] = {0};
int bleCharactersiticValueLength;
char bleCharacteristicUUID[32];
void onCharacteristicWritten(
    BLECentral& central,
    BLECharacteristic &characteristic) {
  bleDataWritten = true; // alert that data has been written
  bleCharacteristicUUID = characteristic.uuid();
  bleCharactersiticValueLength = characteristic.valueLength();
  `
  // Since we are playing with strings, use strcpy
  strcpy(bleCharacteristicValue, (char*) characteristic.value());
}
// setup
void loop() {
  if (bleDataWritten) { // has data been written?
    bleDataWritten = false; // clear the flag
    // send out flow control message
    setCharacteristicValue(
      (char*) bleFlowControlMessage,
      bleFlowControlMessageLength);
  }
}
```

Putting It All Together

Create a new sketch named ble_flowcontrol and copy the following code.

Example 10-1. sketches/ble_flowcontrol/ble_flowcontrol.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "MyDevice";
static const int   characteristicTransmissionLength = 20;
static const char* bleReadReceiptMessage = "ready";
static const int   bleReadReceiptMessageLength = 5;
// store details about transmission here
struct BleTransmission {
  char data[characteristicTransmissionLength];
  unsigned int length;
  char uuid[32];
};
BleTransmission bleTransmissionData;
bool bleDataWritten = false; // true if data has been received
BLEService service("180C");
BLECharacteristic characteristic(
  "2A56",
  BLERead | BLENotify | BLEWrite, // read, write, notify
  characteristicTransmissionLength
);
BLEPeripheral blePeripheral;
bool bleCharacteristicSubscribed = false; // true when a client subscribes
// When data is sent from the client, it is processed here inside a callback
// it is best to handle the result of this inside the main loop
```

```
void onBleCharacteristicWritten(BLECentral& central,
    BLECharacteristic &characteristic) {
  bleDataWritten = true;
  bleTransmissionData.uuid = characteristic.uuid();
  bleTransmissionData.length = characteristic.valueLength();
  // Since we are playing with strings, we must use strncpy
  strncpy(bleTransmissionData.data,
    (char*) characteristic.value(), characteristic.valueLength());
}
void setBleCharacteristicValue(char* output, int length) {
  characteristic.setValue((const unsigned char*) output, length);
}
void setup() {
  Serial.begin(9600); // open a Serial connection
  while (!Serial); // wait for Serial console to open
  blePeripheral.setLocalName(bluetoothDeviceName);
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(characteristic);
  // trigger onbleCharacteristicWritten when data is sent to the characteris-
tic
  characteristic.setEventHandler(
    BLEWritten,
    onBleCharacteristicWritten
  );
  blePeripheral.begin();
}
void loop() {
  // if the bleDataWritten flag has been set, print out the incoming data
  if (bleDataWritten) {
    bleDataWritten = false; // ensure only happens once
    Serial.print(bleTransmissionData.length);
    Serial.print(" bytes sent to characteristic ");
    Serial.print(bleTransmissionData.uuid);
    Serial.print(": ");
    Serial.println(bleTransmissionData.data);
    // send out flow control message
    setBleCharacteristicValue((char*) bleReadReceiptMessage,
        bleReadReceiptMessageLength);
  }
}
```

When the sketch is run, the Peripheral prints incoming data into the Serial Monitor, then notifies the connected Central that it is ready to receive more data (Figure 10-2).

```
16 bytes sent to characteristic 2A56: this is a super
Ready for more data
16 bytes sent to characteristic 2A56: long message
Ready for more data
```

Figure 10-2. Serial Monitor Output

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter10

Project: Echo Server

An Echo Server is the “Hello World” of network programming. It has the minimum features required to transmit, store, and respond to data on a network - the core features required for any network application.

And yet it must support all the features you’ve learned so far in this book - advertising, reads, writes, notifications, segmented data transfer, and encryption. It’s a sophisticated program!

The Echo Server works like this:

In this example, the Peripheral acts as a server, the “Echo Server” and the Central acts as a client (Figure 11-1).

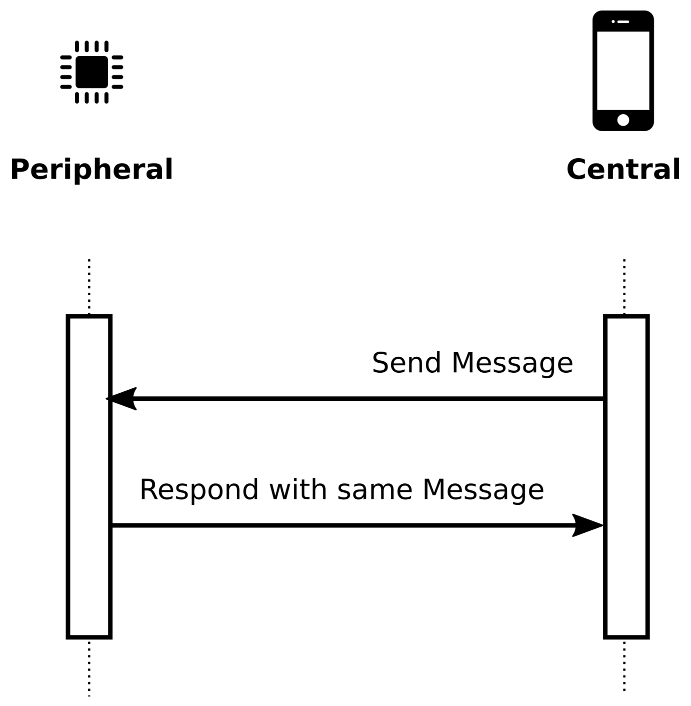


Figure 11-1. How an Echo Server works

This project is based heavily on code seen up until Chapter 10, so there shouldn’t be any surprises.

Programming the Peripheral

The Echo Server will handle incoming writes on one Characteristic (0x2a57) and echo back messages on another Characteristic (0x2a56).

Advertising and GATT Profile

The Peripheral must advertise and host a GATT Profile. The Peripheral will host a read-only, notifiable Characteristic on UUID 0x2a56 and a write-only Characteristic on UUID 0x2a57:

```
...
static const int characteristicTransmissionLength = 20;
BLEService service("180C");
BLECharacteristic readCharacteristic(
  "2A56",
  BLERead | BLENotify,
  characteristicTransmissionLength
);

BLECharacteristic writeCharacteristic(
  "2A57",
  BLEWrite,
  characteristicTransmissionLength
);
...
void setup() {
  ...
  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(readCharacteristic);
  blePeripheral.addAttribute(writeCharacteristic);
  ...
}
```

```
```

It will advertise as "EchoServer" to be discoverable by the corresponding Central:

```
...
static const char* bluetoothDeviceName = "EchoServer";
...
void setup() {
...
  blePeripheral.setLocalName(bluetoothDeviceName);
...
}
```

Handling Reads and Writes

When the Characteristic is written to, the Peripheral stores it for later use, and sets a flag to alert the main loop that there is new message:

```
void onCharacteristicWritten(
  BLECentral& central,
  BLECharacteristic &characteristic) {
  bleDataWritten = true;
  uuid = characteristic.uuid();
  bleMessageLength = characteristic.valueLength();
  strcpy((char*) bleMessage, (const char*) characteristic.value());
}
void setup() {
  ...
  characteristic.setEventHandler(
    BLEWritten,
    onCharacteristicWritten
  );
  ...
}
```

In the main loop, the Echo Server looks for a new message. If one exists, it is sent back to the Central using the same Characteristic.

```
void loop() {
  if (bleDataWritten) {
    bleDataWritten = false; // ensures only happens once
    sendBleMessage((unsigned char*)bleMessage);
  }
}
```

The message is sent like this:

```
void sendBleMessage(unsigned char* bleMessage) {
  Serial.print("Sending message: ");
  Serial.println((char*) bleMessage);
  readCharacteristic.setValue(
    (const unsigned char*) bleMessage,
    bleMessageLength
  );
}
```

Putting It All Together

The following sketch will create a Peripheral that can receive a message through a Characteristic, print that message into the Serial Monitor, and send the message back though the Characteristic to be read by a connected Central.

Create a new sketch called ble_echo_server, and copy the following code into your sketch.

Example 11-1. sketches/ble_echo_server/ble_echo_server.ino

```
#include "CurieBle.h"
```

```
static const char* bluetoothDeviceName = "EchoServer";
static const int   characteristicTransmissionLength = 20;

char bleMessage[characteristicTransmissionLength];
int bleMessageLength;
const char* uuid;
bool bleDataWritten = false;

BLEService service("180C");
BLECharacteristic readCharacteristic(
  "2A56",
  BLERead | BLENotify,
  characteristicTransmissionLength
);

BLECharacteristic writeCharacteristic(
  "2A57",
  BLEWrite,
  characteristicTransmissionLength
);

BLEPeripheral blePeripheral;

void onCharacteristicWritten(
  BLECentral& central,
  BLECharacteristic
  &characteristic)
{
  bleDataWritten = true;
  uuid = characteristic.uuid();
  bleMessageLength = characteristic.valueLength();
  strcpy((char*) bleMessage, (const char*) characteristic.value());
}

// Central connected.  Print MAC address
void onCentralConnected(BLECentral& central) {
  Serial.print("Central connected: ");
  Serial.println(central.address());
}

// Central disconnected
void onCentralDisconnected(BLECentral& central) {
  Serial.println("Central disconnected");
}

void sendBleMessage(unsigned char* bleMessage) {
  Serial.print("Sending message: ");
  Serial.println((char*) bleMessage);
  readCharacteristic.setValue(
    (const unsigned char*) bleMessage,
    bleMessageLength
  );
}

void setup() {
  Serial.begin(9600);
  while (!Serial) {;}
  blePeripheral.setLocalName(bluetoothDeviceName);

  // attach callback when central connects
  blePeripheral.setEventHandler(
    BLEConnected,
    onCentralConnected
  );
  // attach callback when central disconnects
  blePeripheral.setEventHandler(
    BLEDisconnected,
    onCentralDisconnected
  );

  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(readCharacteristic);
  blePeripheral.addAttribute(writeCharacteristic);

  writeCharacteristic.setEventHandler(
    BLEWritten,
    onCharacteristicWritten
  );

  Serial.print("Starting ");
  Serial.println(bluetoothDeviceName);
  blePeripheral.begin();
}

void loop() {
  if (bleDataWritten) {
    bleDataWritten = false; // ensures only happens once

    Serial.print("Incoming message found: ");
    Serial.println((char*) bleMessage);
    sendBleMessage((unsigned char*)bleMessage);
  }
}
```

Here is the output from the Serial monitor when a Central connects to the running Peripheral, then sends the message “Hello” to the Characteristic (Figure 11-2).

```
Starting EchoServer
Device connected: 6d:65:b7:4d:c7:83
Incoming message found: hello
Sending message: hello
```

Figure 11-2. Serial Monitor Output

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter11

Project: Remote Control LED

So far, this book has worked a lot with text data rather than binary data, because it's easy to text without using specialized tools such as oscilloscopes or logic analyzers.

Most real-world projects transmit binary instead of text. Binary is much more efficient in transmitting information.

Because binary data it is the language of computers, it is easier to work with than text. There is no need to worry about character sets, null characters, or cut-off words.

This project will show how to remotely control an LED on a Peripheral using software on a Central.

The LED Remote works like this (Figure 12-1).

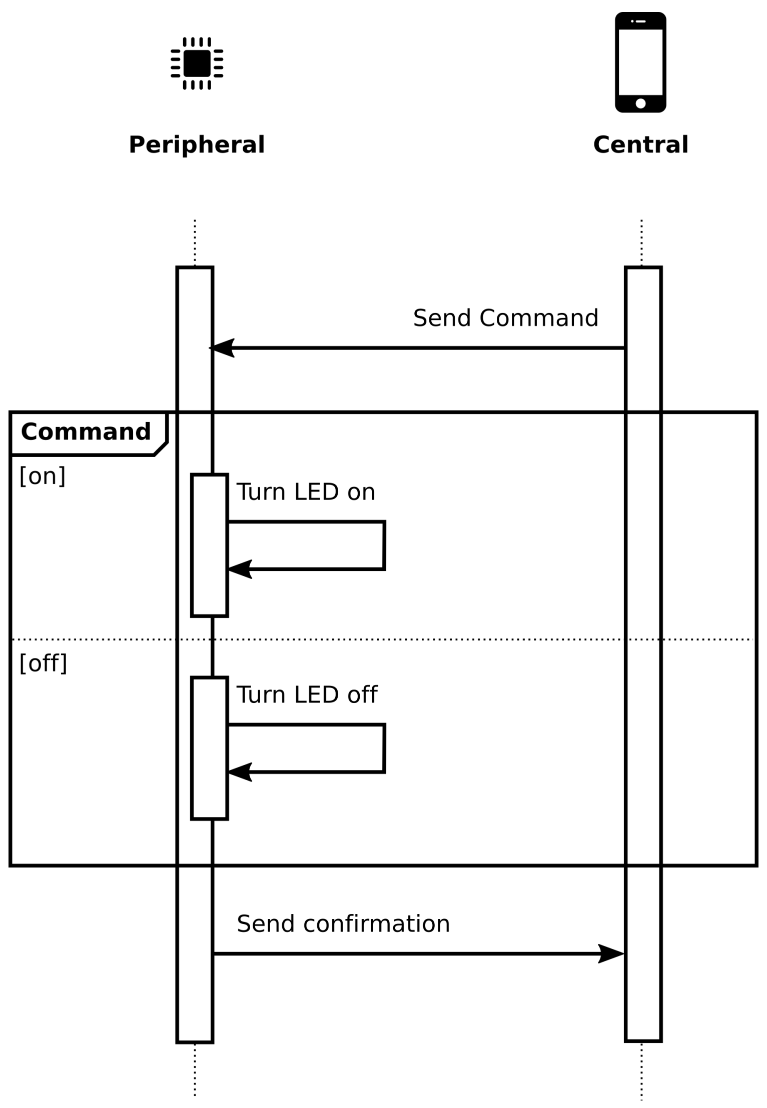


Figure 12-1. How a Remote Control LED works

In all the other examples, text was being sent between Central and Peripheral.

In order for the Central and Peripheral to understand each other, they need shared language between them. In this case, a data packet format.

Sending Commands to Peripheral

When the Central sends a message, it should be able to specify if it is sending a command or an error. We can do this in two bytes, like this (Figure 12-2).

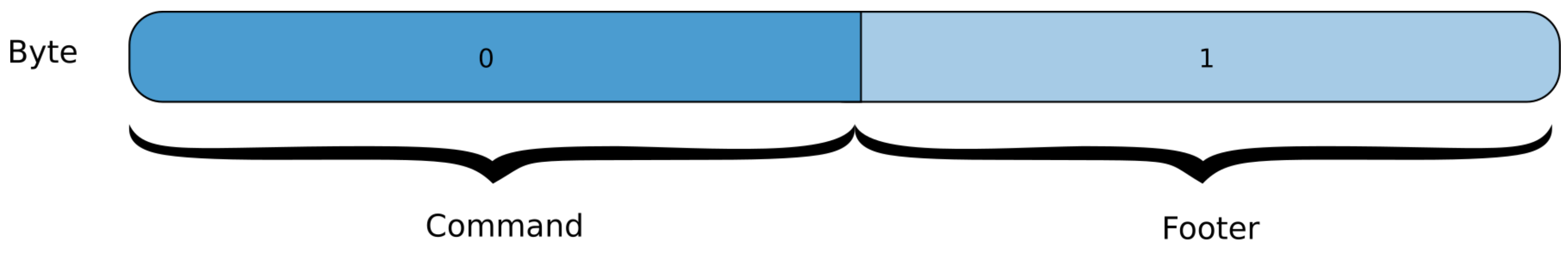


Figure 12-2. Packet structure for commands

The Peripheral reads the footer byte of the incoming message to determine the type of message, i.e., an error or a command. For example, define the message types as:

Table 12-1. Footer Values

Name	Value	Description
bleResponseError	0	The Central is sending an error
bleResponseConfirmation	1	The Central is sending a confirmation
bleResponseCommand	2	The Central is sending a command

The Peripheral reads the first byte to determine the type of error or command. For example, define the commands as:

Table 13-2. Command Values

Name	Value	Description
bleCommandLedOff	1	Turn off the Peripheral's LED
bleCommandLedOn	2	Turn on the Peripheral's LED

The Peripheral then responds to the Central with a status message regarding the success or failure to execute the command. This can also be expressed as two bytes (Figure 12-3).

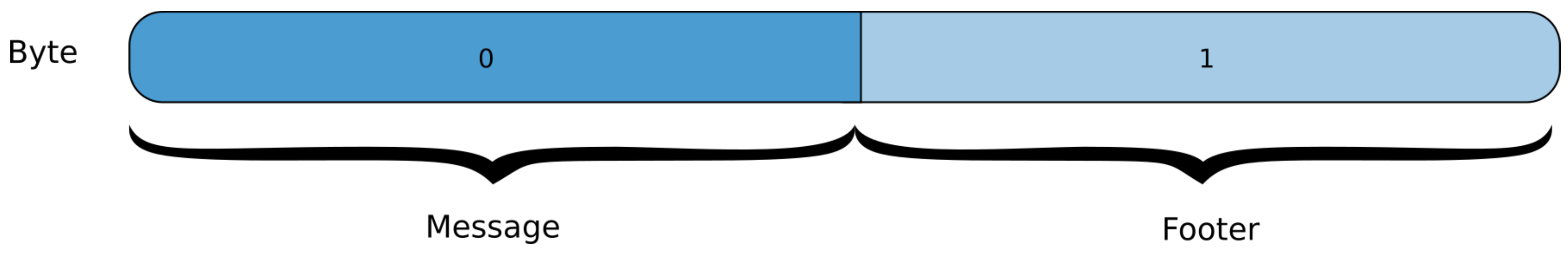


Figure 12-3. Packet structure for responses

If the Peripheral sends a confirmation that the LED state has changed, then the Central inspects the first byte of the message to determine what the current state of the Peripheral's LED is:

Table 13-2. Confirmation Values

Name	Value	Description
ledStateOff	1	The Peripheral's LED is off
ledStateOn	2	The Peripheral's LED is on

In this way, a common language is established between the Central and the Peripheral.

Gatt Profile

The Bluetooth Low Energy specification provides a special Service, the Automation IO Service (0x1815), specifically for remote control devices such as this.

It is a best practice to use each Characteristic for a single purpose. For this reason, Characteristic 0x2a56 will be used for sending commands to the Peripheral and Characteristic 0x2a57 will be used for responses from the Peripheral:

Table 12-4. Characteristic Usages

UUID	Use
0x2a56	Send commands from Central to Peripheral
0x2a57	Send responses from Peripheral to Central

Programming the Peripheral

This project is relatively simple since the Central and Peripheral both speak in their native language, binary. As a result, fewer steps are needed to process the data than with text.

Advertising and GATT Profile

There must be two Characteristics: one for writing commands and one for reading responses, under the Automation IO Service (0x1815). The response Characteristic will support notifications:

```
static const int characteristicTransmissionLength = 2;
BLEService service("1815");
BLECharacteristic commandCharacteristic(
  "2A56",
  BLEWrite,
  characteristicTransmissionLength
);
BLECharacteristic responseCharacteristic(
  "2A57",
  BLERead | BLENotify,
  characteristicTransmissionLength
);
```

The Characteristic needs to be able to access data written by the Central, like this:

```
boolean bleCommandReceived = false;
char bleMessage[characteristicTransmissionLength];
const char* uuid;
void onCharacteristicWritten(
  BLECentral& central, BLECharacteristic &characteristic) {
  bleCommandReceived = true; // let system know that a message was recevied
  uuid = characteristic.uuid();
  memcpy((char*) bleMessage,
    (const char*) characteristic.value(),
    characteristic.valueLength());
}
```

Data Formatting

Message types must be defined as being footers, commands, and responses:

```
// Commands
static const uint8_t bleCommandFooterPosition = 1;
static const uint8_t bleCommandDataPosition = 0;
static const uint8_t bleCommandFooter = 1;
static const uint8_t bleCommandLedOn = 1;
static const uint8_t bleCommandLedOff = 2;

// Responses
static const uint8_t bleResponseFooterPosition = 1;
static const uint8_t bleResponseDataPosition = 0;
static const uint8_t bleResponseErrorFooter = 0;
static const uint8_t bleResponseConfirmationFooter = 1;
static const uint8_t bleResponseLedError = 0;
static const uint8_t bleResponseLedOn = 1;
static const uint8_t bleResponseLedOff = 2;
```

Processing Commands

This program has two commands: one to turn the LED on and one to turn the LED off.

```
static const unsigned int ledOn = 1;
static const unsigned int ledOff = 2;
Responses to the Central must represent the state of the LED.
void sendBleCommandResponse(int ledState) {
  byte confirmation[characteristicTransmissionLength] = {0x0};
  confirmation[bleResponseDataPosition] = (byte)ledState;
  confirmation[bleResponseFooterPosition] = \
    (byte)bleResponseConfirmationFooter;
  responseCharacteristic.setValue(
    (const unsigned char*) confirmation,
    characteristicTransmissionLength
  );
}
```

Once a command has been received, the Peripheral needs to change the LED state, then notify the Central that the command has been processed.

```
void loop() {
  if (bleCommandReceived) {
    bleCommandReceived = false; // ensures only executed once
    // incoming command is one byte
    unsigned int command = bleMessage[bleMessageDataPosition];
    if (command == bleCommandLedOn) {
      Serial.println("Turning LED on");
      ledState = HIGH;
      sendBleCommandResponse(ledState);
    } else {
      Serial.println("Turning LED off");
      ledState = LOW;
      sendBleCommandResponse(ledState);
    }
    digitalWrite(ledPin, ledState);
  }
}
```

Putting It All Together

The following sketch will create a Peripheral that can receive pre-programmed commands to turn the onboard LED on and off.

Create a new project called led_remote, and copy the following code:

Example 12-1. sketches/led_remote/led_remote.ino

```
#include "CurieBle.h"
static const char* bluetoothDeviceName = "LedRemote";
static const int   characteristicTransmissionLength = 2;

// Commands
static const uint8_t bleCommandFooterPosition = 1;
static const uint8_t bleCommandDataPosition = 0;
static const uint8_t bleCommandFooter = 1;
static const uint8_t bleCommandLedOn = 1;
static const uint8_t bleCommandLedOff = 2;

// Responses
static const uint8_t bleResponseFooterPosition = 1;
static const uint8_t bleResponseDataPosition = 0;
static const uint8_t bleResponseErrorFooter = 0;
static const uint8_t bleResponseConfirmationFooter = 1;
static const uint8_t bleResponseLedError = 0;
static const uint8_t bleResponseLedOn = 1;
static const uint8_t bleResponseLedOff = 2;

// Peripheral Properties
static const byte ledPin = 13;
static const unsigned int ledError = 0;
static const unsigned int ledOn = 1;
static const unsigned int ledOff = 2;
int ledState = ledOff;

// internal state
char bleMessage[characteristicTransmissionLength];
const char* uuid;
bool bleCommandReceived = false;

BLEService service("180C");
BLECharacteristic commandCharacteristic(
  "2A56",
  BLEWrite,
  characteristicTransmissionLength
);

BLECharacteristic responseCharacteristic(
  "2A57",
  BLERead | BLENotify,
  characteristicTransmissionLength
);

BLEPeripheral blePeripheral;

void onCharacteristicWritten(
  BLECentral& central,
  BLECharacteristic &characteristic)
{
  bleCommandReceived = true;
  uuid = characteristic.uuid();
  strcpy((char*) bleMessage, (const char*) characteristic.value());
}

void sendBleCommandResponse(int ledState) {
  byte confirmation[characteristicTransmissionLength] = {0x0};
  confirmation[bleResponseDataPosition] = (byte)ledState;
  confirmation[bleResponseFooterPosition] = \
    (byte)bleResponseConfirmationFooter;
  responseCharacteristic.setValue(
    (const unsigned char*) confirmation,
    characteristicTransmissionLength
  );
}

// Central connected.  Print MAC address
void onCentralConnected(BLECentral& central) {
  Serial.print("Central connected: ");
  Serial.println(central.address());
}

// Central disconnected
void onCentralDisconnected(BLECentral& central) {
  Serial.println("Central disconnected");
}



void setup() {
  Serial.begin(9600);
  while (!Serial) {;} // wait for Serial to connect
  digitalWrite(ledPin, LOW); // start with LED off
  blePeripheral.setLocalName(bluetoothDeviceName);

  // attach callback when central connects
  blePeripheral.setEventHandler(
    BLEConnected,
    onCentralConnected
  );
  // attach callback when centlal disconnects
  blePeripheral.setEventHandler(
    BLEDisconnected,
    onCentralDisconnected
  );

  blePeripheral.setAdvertisedServiceUuid(service.uuid());
  blePeripheral.addAttribute(service);
  blePeripheral.addAttribute(commandCharacteristic);
  blePeripheral.addAttribute(responseCharacteristic);
  commandCharacteristic.setEventHandler(
    BLEWritten,
    onCharacteristicWritten
  );
  Serial.print("Starting ");
  Serial.println(bluetoothDeviceName);
  blePeripheral.begin();
}

void loop() {
  if (bleCommandReceived) {
    bleCommandReceived = false; // ensures only executed once

    // incoming command is one byte
    unsigned int command = bleMessage[bleCommandDataPosition];
    if (command == bleCommandLedOn) {
      Serial.println("Turning LED on");
      ledState = HIGH;
      sendBleCommandResponse(ledState);
    } else {
      Serial.println("Turning LED off");
      ledState = LOW;
      sendBleCommandResponse(ledState);
    }
    digitalWrite(ledPin, ledState);
  }
}
```

The resulting Peripheral turns an LED on and off when instructed, and reports that it's processed the command.

When the Central sends the "Light On" command, the Serial Monitor should resemble this (Figure 12-4):

```
Turning LED off
Starting LedRemote
Device connected: 71:29:f1:8c:0a:d7
Turning LED on
```

Figure 12-4. Serial Monitor Output

Additionally, the Arduino should respond lighting this LED (Figure 12-5).

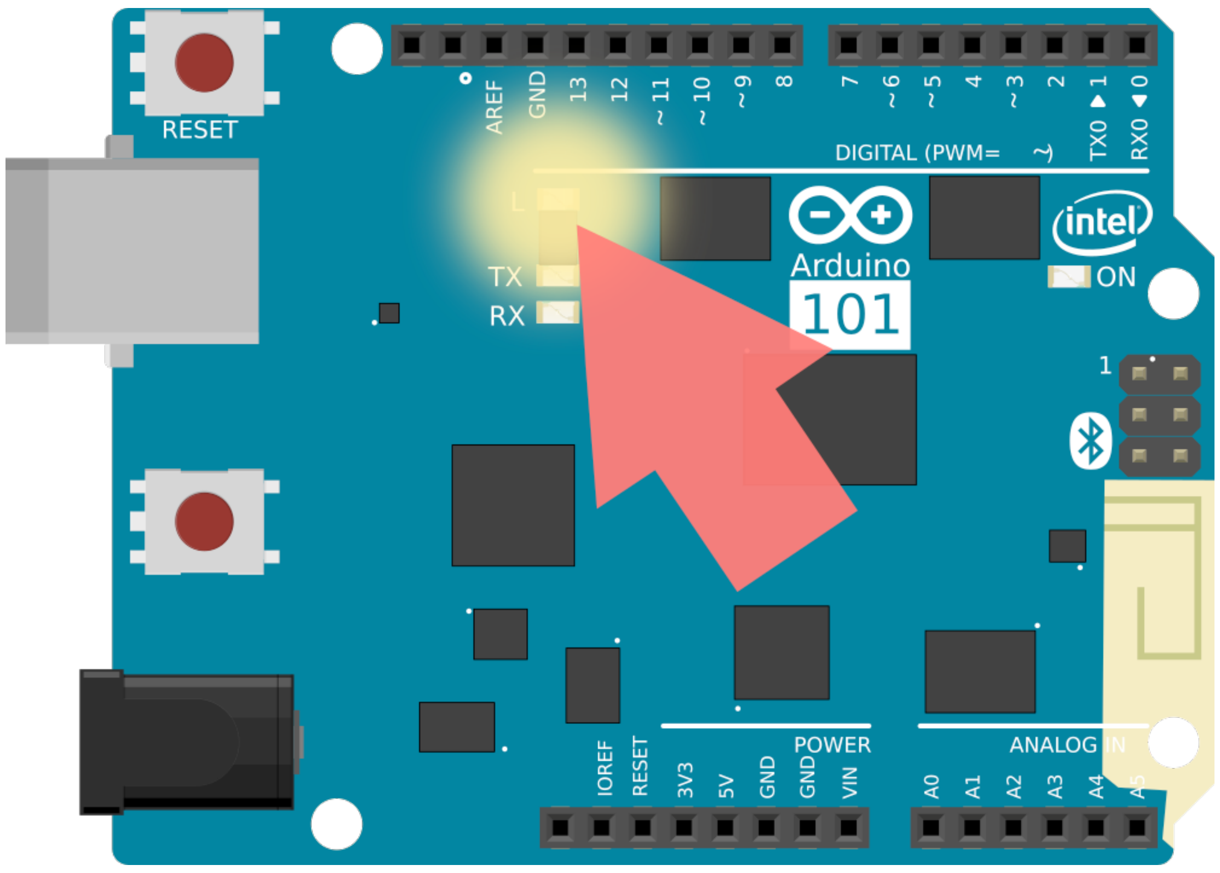


Figure 12-5. Arduino LED turns on and off

Example code

The code for this chapter is available online at: https://github.com/BluetoothLowEnergyInArduino101/Chapter12

Appendix

For reference, the following are properties of the Bluetooth Low Energy network and hardware.

Range	100 m (330 ft)
Data Rate	1M bit/s
Application Throughput	0.27 Mbit/s
Security	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)
Robustness	Adaptive Frequency Hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check
Range	100 m (330 ft)
Data Rate	1M bit/s
Application Throughput	0.27 Mbit/s
Security	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)
Peak Current Consumption	< 15 mA
Byte-Order in Broadcast	Big Endian (most significant bit at end)
Range	100 m (330 ft)
Data Rate	1M bit/s
Application Throughput	0.27 Mbit/s
Security	128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)

Source: Wikipedia: Bluetooth_Low_Energy
Retrieved from https://en.wikipedia.org/wiki/Bluetooth_low_energy

Appendix II: UUID Format

Bluetooth Low Energy has tight space requirements. Therefore it is preferred to transmit 16-bit UUIDs instead of 32-bit UUIDs. UUIDs can be converted between 16-bit and 32-bit with the standard Bluetooth Low Energy UUID format:

Table II-1. 16-bit to 32-bit UUID Conversion Standard

UUID Format	uuid16	Resulting uuid32
0000*0000*-0000-1000-8000-00805f9b34fb	0x2A56	0000*2A56*-0000-1000-8000-00805f9b34fb

Appendix III: Minimal Recommended GATT

As a best practice, it is good to host a standard set of Services and Characteristics in a Peripheral's GATT Profile. These Characteristics allow connected Centrals to get the make and model number of the device, and the battery level if the Peripheral is battery-powered:

Table III-1. Minimal GATT Profile

GATT Type	Name	Data Type	UUID
Service	Device Information Service		0x180a
Characteristic	Device Name	char array	0x2a00
Characteristic	Model Number	char array	0x2a24
Characteristic	Serial Number	char array	0x2a04
Service	Battery Level		0x180f
Characteristic	Battery Level	integer	0x2a19

Appendix IV: Reserved GATT Services

Services act as a container for Characteristics or other Services, providing a tree-like structure for organizing Bluetooth I/O.

These Services UUIDs have been reserved for special contexts, such as Device Information (0x180A) Which may contain Characteristics that communicate information about the Peripheral's name, version number, or settings.

Note: All Bluetooth Peripherals should have a Battery Service (0x180F) Service containing a Battery Level (0x2A19) Characteristic.

Table IV-1. Reserved GATT Services

Specification Name	UUID	Specification Type
Alert Notification Service	0x1811	org.bluetooth.service.alert_notification
Automation IO	0x1815	org.bluetooth.service.automation_io
Battery Service	0x180F	org.bluetooth.service.battery_service
Blood Pressure	0x1810	org.bluetooth.service.blood_pressure
Body Composition	0x181B	org.bluetooth.service.body_composition
Bond Management	0x181E	org.bluetooth.service.bond_management
Continuous Glucose Monitoring	0x181F	org.bluetooth.service.continuous_glucose_monitoring

Specification Name	UUID	Specification Type
Current Time Service	0x1805	org.bluetooth.service.current_time
Cycling Power	0x1818	org.bluetooth.service.cycling_power
Cycling Speed and Cadence	0x1816	org.bluetooth.service.cycling_speed_and_cadence
Device Information	0x180A	org.bluetooth.service.device_information
Environmental Sensing	0x181A	org.bluetooth.service.environmental_sensing
Generic Access	0x1800	org.bluetooth.service.generic_access
Generic Attribute	0x1801	org.bluetooth.service.generic_attribute
Glucose	0x1808	org.bluetooth.service.glucose
Health Thermometer	0x1809	org.bluetooth.service.health_thermometer
Heart Rate	0x180D	org.bluetooth.service.heart_rate
HTTP Proxy	0x1823	org.bluetooth.service.http_proxy
Human Interface Device	0x1812	org.bluetooth.service.human_interface_device
Immediate Alert	0x1802	org.bluetooth.service.immediate_alert
Indoor Positioning	0x1821	org.bluetooth.service.indoor_positioning
Internet Protocol Support	0x1820	org.bluetooth.service.internet_protocol_support

Specification Name	UUID	Specification Type
Link Loss	0x1803	org.bluetooth.service.link_loss
Location and Navigation	0x1819	org.bluetooth.service.location_and_navigation
Next DST Change Service	0x1807	org.bluetooth.service.next_dst_change
Object Transfer	0x1825	org.bluetooth.service.object_transfer
Phone Alert Status Service	0x180E	org.bluetooth.service.phone_alert_status
Pulse Oximeter	0x1822	org.bluetooth.service.pulse_oximeter
Reference Time Update Service	0x1806	org.bluetooth.service.reference_time_update
Running Speed and Cadence	0x1814	org.bluetooth.service.running_speed_and_cadence
Scan Parameters	0x1813	org.bluetooth.service.scan_parameters
Transport Discovery	0x1824	org.bluetooth.service.transport_discovery
Tx Power	0x1804	org.bluetooth.service.tx_power
User Data	0x181C	org.bluetooth.service.user_data
Weight Scale	0x181D	org.bluetooth.service.weight_scale

Source: Bluetooth SIG: GATT Services
Retrieved from https://www.bluetooth.com/specifications/gatt/services

Appendix V: Reserved GATT Characteristics

Characteristics act a data port that can be read from or written to.

These Characteristic UUIDs have been reserved for specific types of data, such as Device Name (0x2A00) which may read the Peripheral's current battery level.

Note: All Bluetooth Peripherals should have a Battery Level (0x2A19) Characteristic, contained inside a Battery Service (0x180F) Service.

Table V-1. Reserved GATT Characteristics

Specification Name	UUID	Specification Type
Aerobic Heart Rate Lower Limit	0x2A7E	org.bluetooth.characteristic.aerobic_heart_rate_lower_limit
Aerobic Heart Rate Upper Limit	0x2A84	org.bluetooth.characteristic.aerobic_heart_rate_upper_limit
Aerobic Threshold	0x2A7F	org.bluetooth.characteristic.aerobic_threshold
Age	0x2A80	org.bluetooth.characteristic.age
Aggregate	0x2A5A	org.bluetooth.characteristic.aggregate
Alert Category ID	0x2A43	org.bluetooth.characteristic.alert_category_id
Alert Category ID Bit Mask	0x2A42	org.bluetooth.characteristic.alert_category_id_bit_mask
Alert Level	0x2A06	org.bluetooth.characteristic.alert_level
Alert Notification Control Point	0x2A44	org.bluetooth.characteristic.alert_notification_control_point

Specification Name	UUID	Specification Type
Alert Status	0x2A3F	org.bluetooth.characteristic.alert_status
Altitude	0x2AB3	org.bluetooth.characteristic.altitude
Anaerobic Heart Rate Lower Limit	0x2A81	org.bluetooth.characteristic.anaerobic_heart_rate_lower_limit
Anaerobic Heart Rate Upper Limit	0x2A82	org.bluetooth.characteristic.anaerobic_heart_rate_upper_limit
Anaerobic Threshold	0x2A83	org.bluetooth.characteristic.anaerobic_threshold
Analog	0x2A58	org.bluetooth.characteristic.analog
Apparent Wind Direction	0x2A73	org.bluetooth.characteristic.apparent_wind_direction
Apparent Wind Speed	0x2A72	org.bluetooth.characteristic.apparent_wind_speed
Appearance	0x2A01	org.bluetooth.characteristic.gap.appearance
Barometric Pressure Trend	0x2AA3	org.bluetooth.characteristic.barometric_pressure_trend
Battery Level	0x2A19	org.bluetooth.characteristic.battery_level
Blood Pressure Feature	0x2A49	org.bluetooth.characteristic.blood_pressure_feature
Blood Pressure Measurement	0x2A35	org.bluetooth.characteristic.blood_pressure_measurement
Body Composition Feature	0x2A9B	org.bluetooth.characteristic.body_composition_feature
Body Composition Measurement	0x2A9C	org.bluetooth.characteristic.body_composition_measurement
Body Sensor Location	0x2A38	org.bluetooth.characteristic.body_sensor_location

Specification Name	UUID	Specification Type
Bond Management Control Point	0x2AA4	org.bluetooth.characteristic.bond_management_control_point
Bond Management Feature	0x2AA5	org.bluetooth.characteristic.bond_management_feature
Boot Keyboard Input Report	0x2A22	org.bluetooth.characteristic.boot_keyboard_input_report
Boot Keyboard Output Report	0x2A32	org.bluetooth.characteristic.boot_keyboard_output_report
Boot Mouse Input Report	0x2A33	org.bluetooth.characteristic.boot_mouse_input_report
Central Address Resolution	0x2AA6	org.bluetooth.characteristic.gap.central_address_resolution_support
CGM Feature	0x2AA8	org.bluetooth.characteristic.cgm_feature
CGM Measurement	0x2AA7	org.bluetooth.characteristic.cgm_measurement
CGM Session Run Time	0x2AAB	org.bluetooth.characteristic.cgm_session_run_time
CGM Session Start Time	0x2AAA	org.bluetooth.characteristic.cgm_session_start_time
CGM Specific Ops Control Point	0x2AAC	org.bluetooth.characteristic.cgm_specific_ops_control_point
CGM Status	0x2AA9	org.bluetooth.characteristic.cgm_status
CSC Feature	0x2A5C	org.bluetooth.characteristic.csc_feature
CSC Measurement	0x2A5B	org.bluetooth.characteristic.csc_measurement
Current Time	0x2A2B	org.bluetooth.characteristic.current_time
Cycling Power Control Point	0x2A66	org.bluetooth.characteristic.cycling_power_control_point

Specification Name	UUID	Specification Type
Cycling Power Feature	0x2A65	org.bluetooth.characteristic.cycling_power_feature
Cycling Power Measurement	0x2A63	org.bluetooth.characteristic.cycling_power_measurement
Cycling Power Vector	0x2A64	org.bluetooth.characteristic.cycling_power_vector
Database Change Increment	0x2A99	org.bluetooth.characteristic.database_change_increment
Date of Birth	0x2A85	org.bluetooth.characteristic.date_of_birth
Date of Threshold Assessment	0x2A86	org.bluetooth.characteristic.date_of_threshold_assessment
Date Time	0x2A08	org.bluetooth.characteristic.date_time
Day Date Time	0x2A0A	org.bluetooth.characteristic.day_date_time
Day of Week	0x2A09	org.bluetooth.characteristic.day_of_week
Descriptor Value Changed	0x2A7D	org.bluetooth.characteristic.descriptor_value_changed
Device Name	0x2A00	org.bluetooth.characteristic.gap.device_name
Dew Point	0x2A7B	org.bluetooth.characteristic.dew_point
Digital	0x2A56	org.bluetooth.characteristic.digital
DST Offset	0x2A0D	org.bluetooth.characteristic.dst_offset
Elevation	0x2A6C	org.bluetooth.characteristic.elevation
Email Address	0x2A87	org.bluetooth.characteristic.email_address

Specification Name	UUID	Specification Type
Exact Time 256	0x2A0C	org.bluetooth.characteristic.exact_time_256
Fat Burn Heart Rate Lower Limit	0x2A88	org.bluetooth.characteristic.fat_burn_heart_rate_lower_limit
Fat Burn Heart Rate Upper Limit	0x2A89	org.bluetooth.characteristic.fat_burn_heart_rate_upper_limit
Firmware Revision String	0x2A26	org.bluetooth.characteristic.firmware_revision_string
First Name	0x2A8A	org.bluetooth.characteristic.first_name
Five Zone Heart Rate Limits	0x2A8B	org.bluetooth.characteristic.five_zone_heart_rate_limits
Floor Number	0x2AB2	org.bluetooth.characteristic.floor_number
Gender	0x2A8C	org.bluetooth.characteristic.gender
Glucose Feature	0x2A51	org.bluetooth.characteristic.glucose_feature
Glucose Measurement	0x2A18	org.bluetooth.characteristic.glucose_measurement
Glucose Measurement Context	0x2A34	org.bluetooth.characteristic.glucose_measurement_context
Gust Factor	0x2A74	org.bluetooth.characteristic.gust_factor
Hardware Revision String	0x2A27	org.bluetooth.characteristic.hardware_revision_string
Heart Rate Control Point	0x2A39	org.bluetooth.characteristic.heart_rate_control_point
Heart Rate Max	0x2A8D	org.bluetooth.characteristic.heart_rate_max
Heart Rate Measurement	0x2A37	org.bluetooth.characteristic.heart_rate_measurement

Specification Name	UUID	Specification Type
Heat Index	0x2A7A	org.bluetooth.characteristic.heat_index
Height	0x2A8E	org.bluetooth.characteristic.height
HID Control Point	0x2A4C	org.bluetooth.characteristic.hid_control_point
HID Information	0x2A4A	org.bluetooth.characteristic.hid_information
Hip Circumference	0x2A8F	org.bluetooth.characteristic.hip_circumference
HTTP Control Point	0x2ABA	org.bluetooth.characteristic.http_control_point
HTTP Entity Body	0x2AB9	org.bluetooth.characteristic.http_entity_body
HTTP Headers	0x2AB7	org.bluetooth.characteristic.http_headers
HTTP Status Code	0x2AB8	org.bluetooth.characteristic.http_status_code
HTTPS Security	0x2ABB	org.bluetooth.characteristic.https_security
Humidity	0x2A6F	org.bluetooth.characteristic.humidity
IEEE 11073-20601 Regulatory Certification Data List	0x2A2A	org.bluetooth.characteristic.ieee_11073-20601_regulatory_certification_data_list
Indoor Positioning Configuration	0x2AAD	org.bluetooth.characteristic.indoor_positioning_configuration
Intermediate Cuff Pressure	0x2A36	org.bluetooth.characteristic.intermediate_cuff_pressure
Intermediate Temperature	0x2A1E	org.bluetooth.characteristic.intermediate_temperature
Irradiance	0x2A77	org.bluetooth.characteristic.irradiance

Specification Name	UUID	Specification Type
Language	0x2AA2	org.bluetooth.characteristic.language
Last Name	0x2A90	org.bluetooth.characteristic.last_name
Latitude	0x2AAE	org.bluetooth.characteristic.latitude
LN Control Point	0x2A6B	org.bluetooth.characteristic.ln_control_point
LN Feature	0x2A6A	org.bluetooth.characteristic.ln_feature
Local East Coordinate	0x2AB1	org.bluetooth.characteristic.local_east_coordinate
Local North Coordinate	0x2AB0	org.bluetooth.characteristic.local_north_coordinate
Local Time Information	0x2A0F	org.bluetooth.characteristic.local_time_information
Location and Speed	0x2A67	org.bluetooth.characteristic.location_and_speed
Location Name	0x2AB5	org.bluetooth.characteristic.location_name
Longitude	0x2AAF	org.bluetooth.characteristic.longitude
Magnetic Declination	0x2A2C	org.bluetooth.characteristic.magnetic_declination
Magnetic Flux Density - 2D	0x2AA0	org.bluetooth.characteristic.magnetic_flux_density_2D
Magnetic Flux Density - 3D	0x2AA1	org.bluetooth.characteristic.magnetic_flux_density_3D
Manufacturer Name String	0x2A29	org.bluetooth.characteristic.manufacturer_name_string
Maximum Recommended Heart Rate	0x2A91	org.bluetooth.characteristic.maximum_recommended_heart_rate

Specification Name	UUID	Specification Type
Measurement Interval	0x2A21	org.bluetooth.characteristic.measurement_interval
Model Number String	0x2A24	org.bluetooth.characteristic.model_number_string
Navigation	0x2A68	org.bluetooth.characteristic.navigation
New Alert	0x2A46	org.bluetooth.characteristic.new_alert
Object Action Control Point	0x2AC5	org.bluetooth.characteristic.object_action_control_point
Object Changed	0x2AC8	org.bluetooth.characteristic.object_changed
Object First-Created	0x2AC1	org.bluetooth.characteristic.object_first_created
Object ID	0x2AC3	org.bluetooth.characteristic.object_id
Object Last-Modified	0x2AC2	org.bluetooth.characteristic.object_last_modified
Object List Control Point	0x2AC6	org.bluetooth.characteristic.object_list_control_point
Object List Filter	0x2AC7	org.bluetooth.characteristic.object_list_filter
Object Name	0x2ABE	org.bluetooth.characteristic.object_name
Object Properties	0x2AC4	org.bluetooth.characteristic.object_properties
Object Size	0x2AC0	org.bluetooth.characteristic.object_size
Object Type	0x2ABF	org.bluetooth.characteristic.object_type
OTS Feature	0x2ABD	org.bluetooth.characteristic.ots_feature

Specification Name	UUID	Specification Type
Peripheral Preferred Connection Parameters	0x2A04	org.bluetooth.characteristic.gap.peripheral_preferred_connection_parameters
Peripheral Privacy Flag	0x2A02	org.bluetooth.characteristic.gap.peripheral_privacy_flag
PLX Continuous Measurement	0x2A5F	org.bluetooth.characteristic.plx_continuous_measurement
PLX Features	0x2A60	org.bluetooth.characteristic.plx_features
PLX Spot-Check Measurement	0x2A5E	org.bluetooth.characteristic.plx_spot_check_measurement
PnP ID	0x2A50	org.bluetooth.characteristic.pnp_id
Pollen Concentration	0x2A75	org.bluetooth.characteristic.pollen_concentration
Position Quality	0x2A69	org.bluetooth.characteristic.position_quality
Pressure	0x2A6D	org.bluetooth.characteristic.pressure
Protocol Mode	0x2A4E	org.bluetooth.characteristic.protocol_mode
Rainfall	0x2A78	org.bluetooth.characteristic.rainfall
Reconnection Address	0x2A03	org.bluetooth.characteristic.gap.reconnection_address
Record Access Control Point	0x2A52	org.bluetooth.characteristic.record_access_control_point
Reference Time Information	0x2A14	org.bluetooth.characteristic.reference_time_information
Report	0x2A4D	org.bluetooth.characteristic.report
Report Map	0x2A4B	org.bluetooth.characteristic.report_map

Specification Name	UUID	Specification Type
Resolvable Private Address Only	0x2AC9	org.bluetooth.characteristic.resolvable_private_address_only
Resting Heart Rate	0x2A92	org.bluetooth.characteristic.resting_heart_rate
Ringer Control Point	0x2A40	org.bluetooth.characteristic.ringer_control_point
Ringer Setting	0x2A41	org.bluetooth.characteristic.ringer_setting
RSC Feature	0x2A54	org.bluetooth.characteristic.rsc_feature
RSC Measurement	0x2A53	org.bluetooth.characteristic.rsc_measurement
SC Control Point	0x2A55	org.bluetooth.characteristic.sc_control_point
Scan Interval Window	0x2A4F	org.bluetooth.characteristic.scan_interval_window
Scan Refresh	0x2A31	org.bluetooth.characteristic.scan_refresh
Sensor Location	0x2A5D	org.blueooth.characteristic.sensor_location
Serial Number String	0x2A25	org.bluetooth.characteristic.serial_number_string
Service Changed	0x2A05	org.bluetooth.characteristic.gatt.service_changed
Software Revision String	0x2A28	org.bluetooth.characteristic.software_revision_string
Sport Type for Aerobic and Anaerobic Thresholds	0x2A93	org.bluetooth.characteristic.sport_type_for_aerobic_and_anaerobic_thresholds
Supported New Alert Category	0x2A47	org.bluetooth.characteristic.supported_new_alert_category
Supported Unread Alert Category	0x2A48	org.bluetooth.characteristic.supported_unread_alert_category

Specification Name	UUID	Specification Type
System ID	0x2A23	org.bluetooth.characteristic.system_id
TDS Control Point	0x2ABC	org.bluetooth.characteristic.tds_control_point
Temperature	0x2A6E	org.bluetooth.characteristic.temperature
Temperature Measurement	0x2A1C	org.bluetooth.characteristic.temperature_measurement
Temperature Type	0x2A1D	org.bluetooth.characteristic.temperature_type
Three Zone Heart Rate Limits	0x2A94	org.bluetooth.characteristic.three_zone_heart_rate_limits
Time Accuracy	0x2A12	org.bluetooth.characteristic.time_accuracy
Time Source	0x2A13	org.bluetooth.characteristic.time_source
Time Update Control Point	0x2A16	org.bluetooth.characteristic.time_update_control_point
Time Update State	0x2A17	org.bluetooth.characteristic.time_update_state
Time with DST	0x2A11	org.bluetooth.characteristic.time_with_dst
Time Zone	0x2A0E	org.bluetooth.characteristic.time_zone
True Wind Direction	0x2A71	org.bluetooth.characteristic.true_wind_direction
True Wind Speed	0x2A70	org.bluetooth.characteristic.true_wind_speed
Two Zone Heart Rate Limit	0x2A95	org.bluetooth.characteristic.two_zone_heart_rate_limit
Tx Power Level	0x2A07	org.bluetooth.characteristic.tx_power_level

Specification Name	UUID	Specification Type
Uncertainty	0x2AB4	org.bluetooth.characteristic.uncertainty
Unread Alert Status	0x2A45	org.bluetooth.characteristic.unread_alert_status
URI	0x2AB6	org.bluetooth.characteristic.uri
User Control Point	0x2A9F	org.bluetooth.characteristic.user_control_point
User Index	0x2A9A	org.bluetooth.characteristic.user_index
UV Index	0x2A76	org.bluetooth.characteristic.uv_index
VO2 Max	0x2A96	org.bluetooth.characteristic.vo2_max
Waist Circumference	0x2A97	org.bluetooth.characteristic.waist_circumference
Weight	0x2A98	org.bluetooth.characteristic.weight
Weight Measurement	0x2A9D	org.bluetooth.characteristic.weight_measurement
Weight Scale Feature	0x2A9E	org.bluetooth.characteristic.weight_scale_feature
Wind Chill	0x2A79	org.bluetooth.characteristic.wind_chill

Source: Bluetooth SIG: GATT Characteristics

Retrieved from https://www.bluetooth.com/specifications/gatt/characteristics

Appendix VI: GATT Descriptors

The following GATT Descriptor UUIDs have been reserved for specific uses.

GATT Descriptors describe features within a Characteristic that can be altered, for instance, the Client Characteristic Configuration (0x2902) which can be flagged to allow a connected Central to subscribe to notifications on a Characteristic.

Table VI-1. Reserved GATT Descriptors

Specification Name	UUID	Specification Type
Characteristic Aggregate Format	0x2905	org.bluetooth.descriptor.gatt.characteristic_aggregate_format
Characteristic Extended Properties	0x2900	org.bluetooth.descriptor.gatt.characteristic_extended_properties
Characteristic Presentation Format	0x2904	org.bluetooth.descriptor.gatt.characteristic_presentation_format
Characteristic User Description	0x2901	org.bluetooth.descriptor.gatt.characteristic_user_description
Client Characteristic Configuration	0x2902	org.bluetooth.descriptor.gatt.client_characteristic_configuration
Environmental Sensing Configuration	0x290B	org.bluetooth.descriptor.es_configuration
Environmental Sensing Measurement	0x290C	org.bluetooth.descriptor.es_measurement
Environmental Sensing Trigger Setting	0x290D	org.bluetooth.descriptor.es_trigger_setting
External Report Reference	0x2907	org.bluetooth.descriptor.external_report_reference

Specification Name	UUID	Specification Type
Number of Digitals	0x2909	org.bluetooth.descriptor.number_of_digitals
Report Reference	0x2908	org.bluetooth.descriptor.report_reference
Server Characteristic Configuration	0x2903	org.bluetooth.descriptor.gatt.server_characteristic_configuration
Time Trigger Setting	0x290E	org.bluetooth.descriptor.time_trigger_setting
Valid Range	0x2906	org.bluetooth.descriptor.valid_range
Value Trigger Setting	0x290A	org.bluetooth.descriptor.value_trigger_setting

Source: Bluetooth SIG: GATT Descriptors

Retrieved from https://www.bluetooth.com/specifications/gatt/descriptors

Appendix VII: Company Identifiers

The following companies have specific Manufacturer Identifiers, which identify Bluetooth devices in the Generic Access Profile (GAP). Peripherals with no specific manufacturer use ID 65535 (0xffff). All other IDs are reserved, even if not yet assigned.

This is a non-exhaustive list of companies. A full list and updated can be found on the Bluetooth SIG website.

Table VII-1. Company Identifiers

Decimal	Hexadecimal	Company
0	0x0000	Ericsson Technology Licensing
1	0x0001	Nokia Mobile Phones
2	0x0002	Intel Corp.
3	0x0003	IBM Corp.
4	0x0004	Toshiba Corp.
5	0x0005	3Com
6	0x0006	Microsoft
7	0x0007	Lucent

Decimal	Hexadecimal	Company
8	0x0008	Motorola
13	0x000D	Texas Instruments Inc.
19	0x0013	Atmel Corporation
29	0x001D	Qualcomm
36	0x0024	Alcatel
37	0x0025	NXP Semiconductors (formerly Philips Semiconductors)
60	0x003C	BlackBerry Limited (formerly Research In Motion)
76	0x004C	Apple, Inc.
86	0x0056	Sony Ericsson Mobile Communications
89	0x0059	Nordic Semiconductor ASA
92	0x005C	Belkin International, Inc.
93	0x005D	Realtek Semiconductor Corporation
101	0x0065	Hewlett-Packard Company
104	0x0068	General Motors
117	0x0075	Samsung Electronics Co. Ltd.

Decimal	Hexadecimal	Company
120	0x0078	Nike, Inc.
135	0x0087	Garmin International, Inc.
138	0x008A	Jawbone
184	0x00B8	Qualcomm Innovation Center, Inc. (QuIC)
215	0x00D7	Qualcomm Technologies, Inc.
216	0x00D8	Qualcomm Connected Experiences, Inc.
220	0x00DC	Procter & Gamble
224	0x00E0	Google
359	0x0167	Bayer HealthCare
367	0x016F	Podo Labs, Inc
369	0x0171	Amazon Fulfillment Service
387	0x0183	Walt Disney
398	0x018E	Fitbit, Inc.
425	0x01A9	Canon Inc.
427	0x01AB	Facebook, Inc.

Decimal	Hexadecimal	Company
474	0x01DA	Logitech International SA
558	0x022E	Siemens AG
605	0x025D	Lexmark International Inc.
637	0x027D	HUAWEI Technologies Co., Ltd. ( )
720	0x02D0	3M
876	0x036C	Zipcar
897	0x0381	Sharp Corporation
921	0x0399	Nikon Corporation
1117	0x045D	Boston Scientific Corporation
65535	0xFFFF	No Device ID

Source: Bluetooth SIG: Company Identifiers
Retrieved from
https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers

Glossary

The following is a list of Bluetooth Low Energy terms and their meanings.

Attribute - An unit of a GATT Profile which can be accessed by a Central, such as a Service or a Characteristic.

Arduino - An open source computer hardware and software company, project, and user community that designs and manufactures single-board microcontrollers and microcontroller kits for building digital devices and interactive objects that can sense and control objects in the physical world.

Beacon - A Bluetooth Low Energy Peripheral which continually Broadcasts so that Centrals can discern their location from information gleaned from the properties of the broadcast.

Bluetooth Low Energy (BLE) - A low power, short range wireless protocol used on micro electronics.

Broadcast - A feature of Bluetooth Low Energy where a Peripheral outputs a name and other specific data about a itself

Central - A Bluetooth Low Energy device that can connect to several Peripherals.

Channel - A finely-tuned radio frequency used for Broadcasting or data transmission.

Characteristic - A port or data endpoint where data can be read or written.

Descriptor - A feature of a Characteristic that allows for some sort of data interaction, such as Read, Write, or Notify.

E0 - The encryption algorithm built into Bluetooth Low Energy.

Generic Attribute (GATT) Profile - A list of Services and Characteristics which are unique to a Peripheral and describe how data is served from the Peripheral. GATT profiles are hosted by a Peripheral

iBeacon - An Apple compatible Beacon which allows a Central to download a specific packet of data to inform the Central of its absolute location and other properties.

Intel® Curie™ Module - The Intel® module that powers the Arduino 101 and contains the Bluetooth chipset.

Notify - An operation where a Peripheral alerts a Central of a change in data.

Peripheral - A Bluetooth Low Energy device that can connect to a single Central. Peripherals host a Generic Attribute (GATT) profile.

Read - An operation where a Central downloads data from a Characteristic.

Scan - The process of a Central searching for Broadcasting Peripherals.

Scan Response - A feature of Bluetooth Low Energy which allows Centrals to download a small packet of data without connecting.

Service - A container structure used to organize data endpoints. Services are hosted by a Peripheral.

Universally Unique Identifier (UUID) - A long, randomly generated alphanumeric sting that is unique regardless of where it's used. UUIDs are designed to avoid name collisions that may happen when countless programs are interacting with each other.

Write - An operation where a Central alters data on a Characteristic.

(This page intentionally left blank)

About the Author

Tony's infinite curiosity compels him to want to open up and learn about everything he touches, and his excitement compels him to share what he learns with others.

He has two true passions: branding and inventing.

His passion for branding led him to start a company that did branding and marketing in 4 countries for firms such as Apple, Intel, and Sony BMG. He loves weaving the elements of design, writing, product, and strategy into an essential truth that defines a company.

His passion for inventing led him to start a company that uses brain imaging to quantify meditation and to predict seizures, a company acquired $1.5m in funding and was incubated in San Francisco where he currently resides.

Those same passions have led him on some adventures as well, including living in a Greek monastery with orthodox monks and to tagging along with a gypsy in Spain to learn to play flamenco guitar.

(This page intentionally left blank)

About this Book

This book is a practical guide to programming Bluetooth Low Energy for Arduino 101.

In this book, you will learn the basics of how to program an Arduino 101 to communicate with any Central or Peripheral device over Bluetooth Low Energy. Each chapter of the book builds on the previous one, culminating in three projects:

- A Beacon and Scanner
- An Echo Server and Client
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

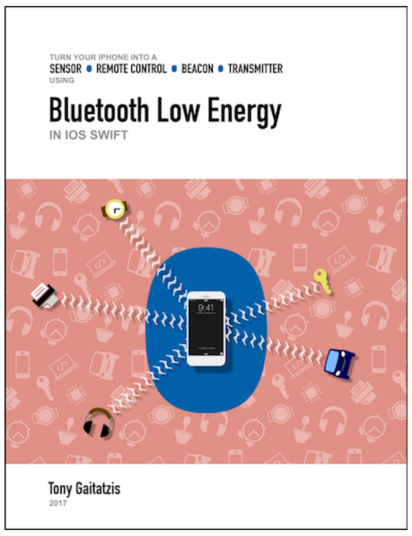
- How Bluetooth Low Energy works
- How data is sent and received
- Common paradigms for handling data

Skill Level

This book is excellent for anyone who has basic or advanced knowledge of Arduino programming or C++.

Other Books in this Series

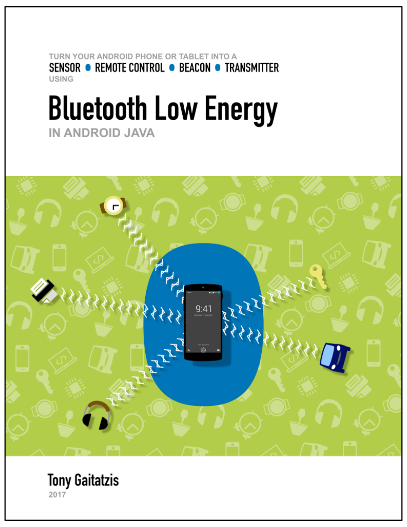
If you are interested in programming other Bluetooth Low Energy Devices, please check out the other books in this series or visit bluetoothlowenergybooks.com:



Bluetooth Low Energy in iOS Swift

Tony Gaitatzis, 2017

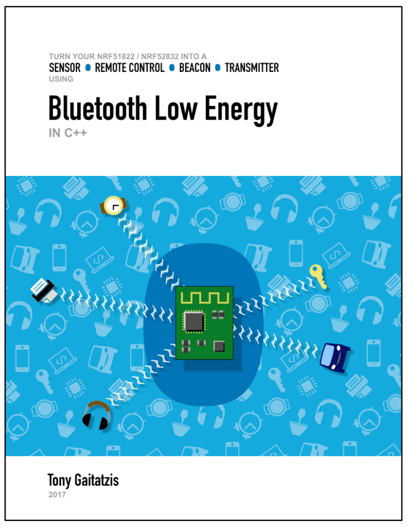
ISBN: 978-1-7751280-5-2


TURN YOUR ANDROID PHONE OR TABLET INTO A
SENSOR • REMOTE CONTROL • BEACON • TRANSMITTER
USING
Bluetooth Low Energy
IN ANDROID JAVA
Tony Gaitatzis
2017

Bluetooth Low Energy in Android Java

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-4-5


TURN YOUR NRF51822 / NRF52832 INTO A
SENSOR • REMOTE CONTROL • BEACON • TRANSMITTER
USING
Bluetooth Low Energy
IN C++
Tony Gaitatzis
2017

Bluetooth Low Energy in C++ for nRF Microcontrollers

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-7-6

(This page intentionally left blank)