Bluetooth with Raspberry Pi and bleno

E-book
made by Skyrise.tech

skyrise.tech

Bluetooth configuration as peripheral device can be overwhelming, especially when done for the first time with unknown technology. Marcin Budny and Konrad Roj will guide you through the process, so you can seamlessly and independently implement the technology for your own purposes.

Jarosław Pilarczyk (Founder & CEO of skyrise.tech)

What is the easiest way to build the software for a Linux-based Bluetooth peripheral device? The answer is really simple! It's by using the right tools and having knowledge of the protocol. With this e-book you'll create your first application in less than one hour.

What you'll find inside:

- How to setup the Linux environment and the bleno library
- How to build an application that broadcasts using iBeacon protocol
- What the GATT protocol is and how to implement services and characteristics
- How the notifications work
- How to connect to your peripheral device from an iOS device

Enjoy!

What's inside

This ebook is a continuation of a blog post **Bluetooth with Raspberry Pi and bleno – part 1: iBeacon** where we demonstrate how easy it is to write software for Bluetooth peripheral device with bleno library by Sandeep Mistry.

1. Introduction
2. About the Authors
3. Chapter 1: Bluetooth with Raspberry Pi and bleno – **iBeacon**
4. Chapter 2: Bluetooth with Raspberry Pi and bleno – **GATT**
5. Chapter 3: Bluetooth with Raspberry Pi and bleno – **Notifications**
6. Chapter 4: Bluetooth with Raspberry Pi and bleno – **iOS client**
7. Contact with experts from Skyrise.tech!

About the Authors

Marcin Budny
Head of R&D at skyrise.tech

An IT architect and developer for over 10 years. He works with intelligent transport systems, researches new technology and explores ways to apply it. He is passionate about building software and is always looking to learn something new. He is primarily focused on .NET, yet he is curious about the rest of the dev world.

Konrad Roy
iOS Developer at skyrise.tech

A Software Engineer who has been focused on Apple platforms and their technological innovations for over five years. He is a connoisseur of good UX and is fascinated by the development of adaptive and intelligent software.

Chapter 1

Bluetooth with Raspberry Pi and bleno – iBeacon

Easy bleno setup for Linux Os. Prepare environment and create your first beacon app!

by Marcin Budny

Introduction

Working directly with BlueZ, the Linux Bluetooth protocol stack, can be overwhelming. The learning curve is steep and prohibitive, especially when all you want to do is to spike new technology. Fortunately, bleno provides a very nice node.js wrapper on top of BlueZ, allowing you to work with Bluetooth 4.0+ and GATT protocol. bleno allows you to implement a peripheral device (e.g. a sensor you connect to in order to get some data). If you are interested in implementing a central role (e.g. a smartphone), there is another library called noble.

In this first chapter my goal is to demonstrate how to setup bleno on your Pi (or any Linux machine for that matter). We're not going to dive into GATT just yet. We'll create a simple app that broadcasts as an iBeacon. iBeacon is a protocol on top of Bluetooth 4.0 that makes it possible for a device to broadcast its identifiers to nearby receivers.

Prepare a development environment

You'll need a Linux machine with a Bluetooth 4.0+ adapter. This can be a Raspberry Pi 3 with integrated Bluetooth module, or a laptop running Linux (with Bluetooth built-in or added as a USB dongle).

You can verify available Bluetooth adapters with:

```
hciconfig
```

bleno will use `hci0` by default. If you want to use another adapter, specify which one with an environment variable:

```
export NOBLE_HCI_DEVICE_ID=1 # uses hci1
```

As a prerequisite, you'll also need node.js and npm installed and able to build native modules. You'll also need following libraries:

```
sudo apt install bluetooth bluez libbluetooth-dev libudev-dev
```

The bluetooth system service needs to be disabled for bleno to work, otherwise some operations will just fail silently. This is quite easy to miss.

```
sudo service bluetooth stop
sudo hciconfig hci0 up # reactivate hci0 or another hciX you want to use
```

The beacon app

Bleno lets you create an iBeacon app with just a few lines of code:

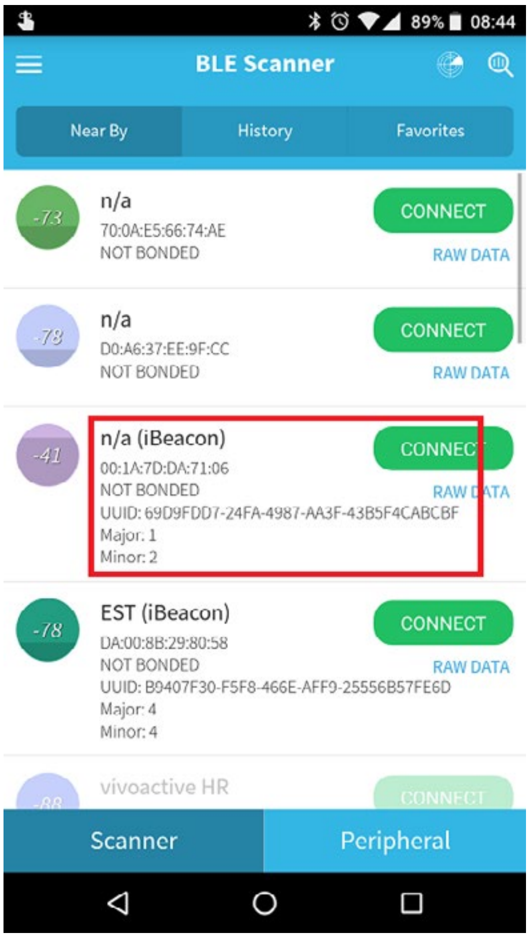
```
npm install bleno --save
```

```
const bleno = require(„bleno”);

const UUID = „69d9fdd724fa4987aa3f43b5f4cabcbf”; // set your own value
const MINOR = 2; // set your own value
const MAJOR = 1; // set your own value
const TX_POWER = -60; // just declare transmit power in dBm

console.log(„Starting bleno...”);

bleno.on(„stateChange”, state => {

    if (state === ‚poweredOn’) {
        console.log(„Starting broadcast...”);

        bleno.startAdvertisingIBeacon(UUID, MAJOR, MINOR, TX_POWER, err => {
            if(err) {
                console.error(err);
            } else {
                console.log(`Broadcasting as iBeacon uuid:${UUID}, major:
${MAJOR}, minor: ${MINOR}`);
            }
        });
    } else {
        console.log(„Stopping broadcast...”);
        bleno.stopAdvertising();
    }
});
```

You always need to handle the stateChangeevent so that you only start using other bleno functionality, once the Bluetooth adapter has been properly initialized. The broadcast itself is started with single method call. The tx power param does not actually change the power of the transmission. It is declared and used by receivers to estimate the range from the beacon. Run the application (see bleno docs if you want to skip `sudo`).

```
sudo node app.js
```

Now you should be able to see the beacon in any Bluetooth scanner app. Here's an example:



-73
n/a
70:0A:E5:66:74:AE
NOT BONDED

-78
n/a
D0:A6:37:EE:9F:CC
NOT BONDED

-41
n/a (iBeacon)
00:1A:7D:DA:71:06
NOT BONDED
UUID: 69D9FDD7-24FA-4987-AA3F-43B5F4CABCBF
Major: 1
Minor: 2

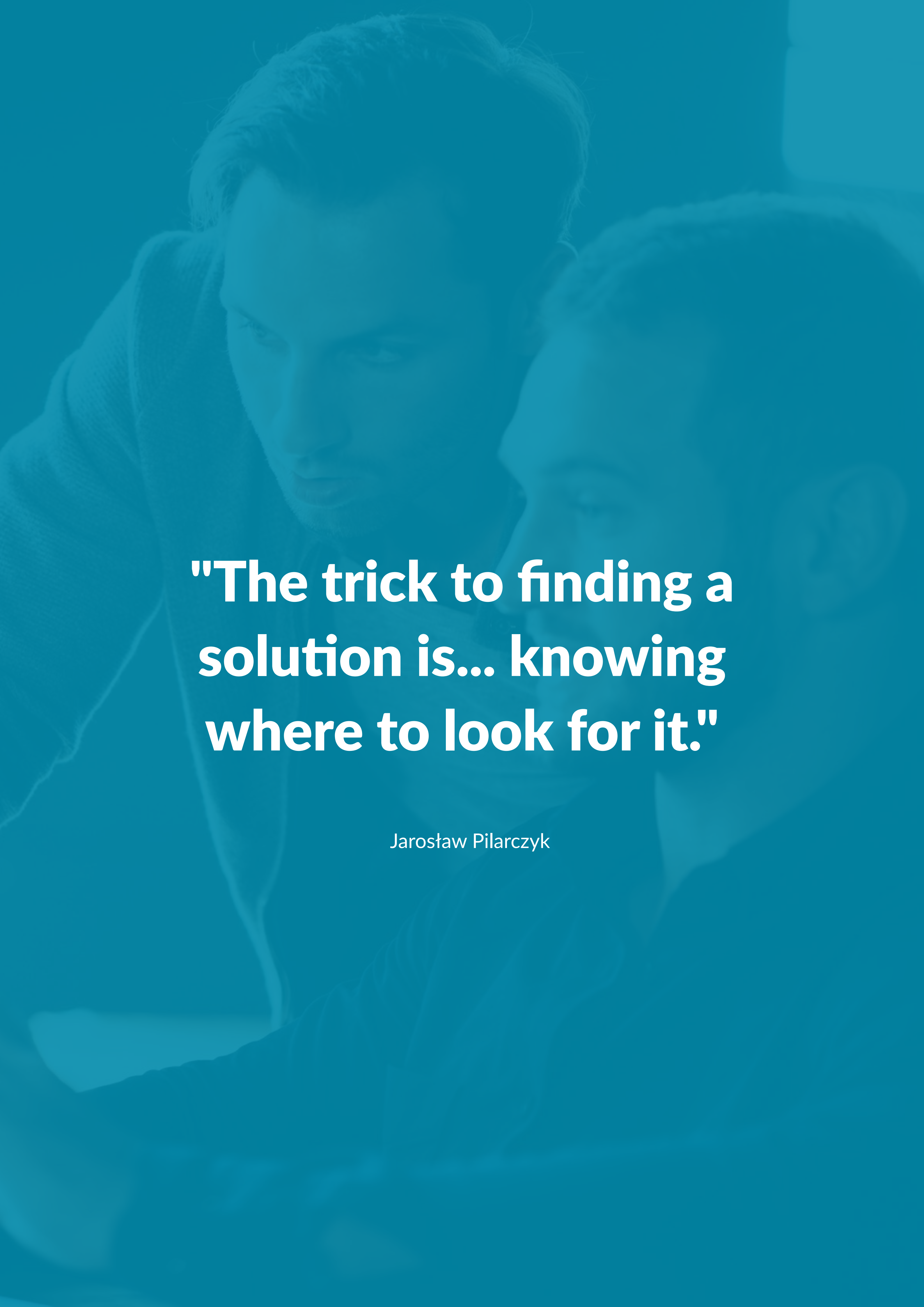
-78
EST (iBeacon)
DA:00:8B:29:80:58
NOT BONDED
UUID: B9407F30-F5F8-466E-AFF9-25556B57FE6D
Major: 4
Minor: 4

-88
vivoactive HR

Summary

In this chapter we showed you how to quickly implement app broadcasting as an iBeacon with bleno. In the next chapter, we'll dig deeper and work with GATT.

Chapter 2: Bluetooth with Raspberry Pi and bleno – **GATT**
Chapter 3: Bluetooth with Raspberry Pi and bleno – **Notifications**
Chapter 4: Bluetooth with Raspberry Pi and bleno – **iOS client**



"The trick to finding a solution is... knowing where to look for it."

Jarosław Pilarczyk

Chapter 2

skyrise.tech

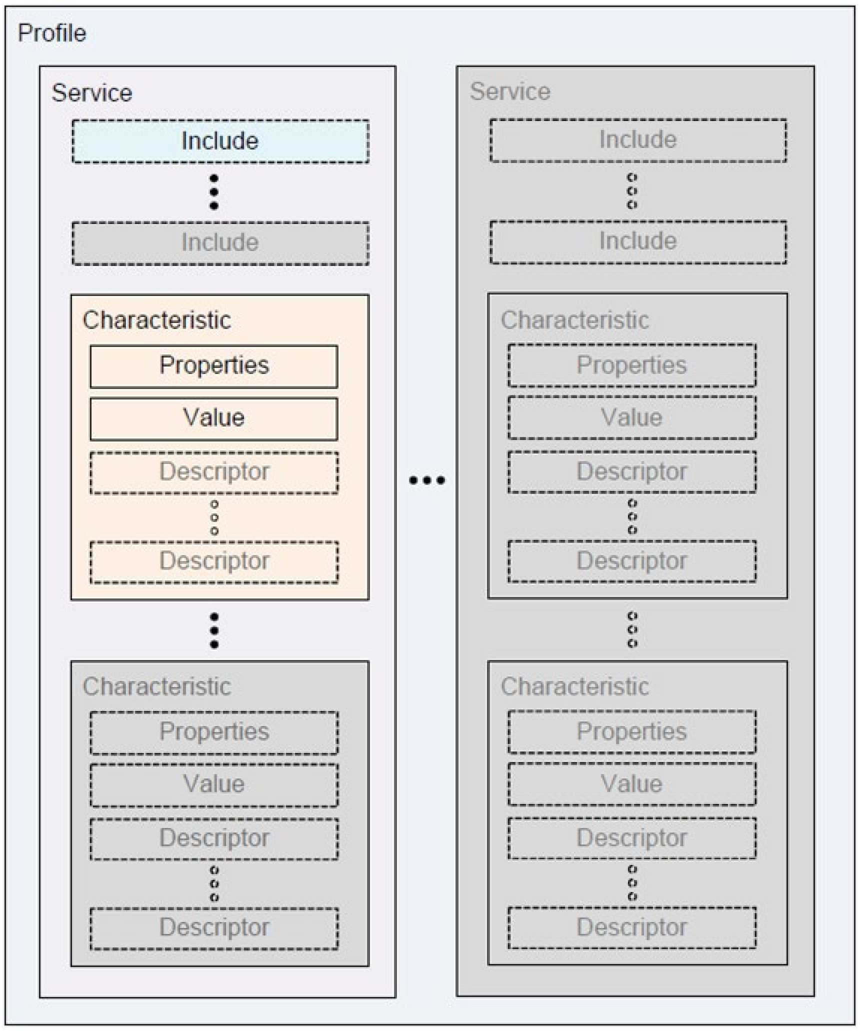
Bluetooth with Raspberry Pi and bleno – GATT

Using GATT: characteristic creation, service declaration, testing. Learn how to implement a simple calculator

by Marcin Budny

GATT

GATT (Generic Attribute Profile) specifies a hierarchical data structure, that can be used by a GATT client and GATT server to communicate with each other.



The structure consists of one or more services. Each service has its unique UUID and contains a set of characteristics. Each characteristic also has its own UUID, a value, information about supported operations (like reading or writing) and other metadata. An analogy could be dictionaries (services) containing key – value entries (characteristics).

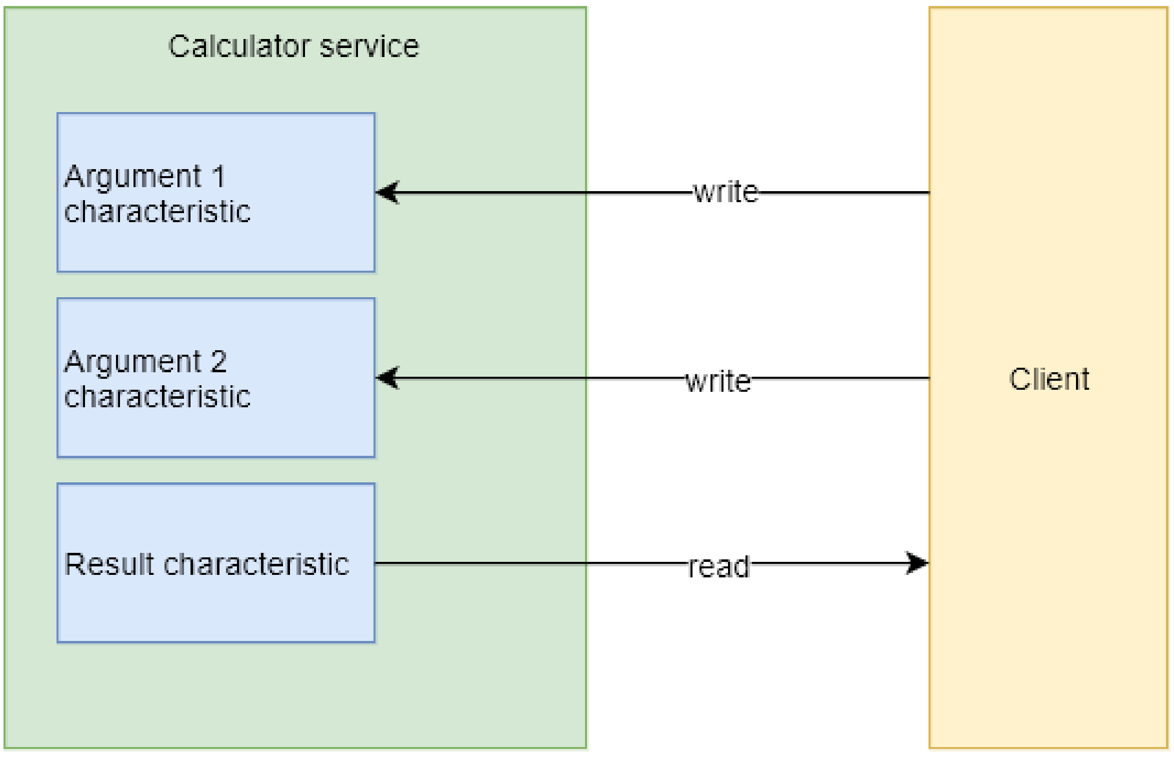
The GATT defines the procedures for a client to discover services and characteristics hosted on a server. The client can then read, write or subscribe to selected characteristics.

Contrary to what may feel intuitive, the GATT server is usually a Bluetooth peripheral device, like a heartbeat monitor. The client on the other hand is a central device, e.g. a smartphone.

There are also some predefined services and characteristics that unify the way of exchaning common types of information. For example, a device may implement the Battery Service which has a mandatory Battery Level characteristic. This way every client can easily discover battery level reporting functionality in a device, regardless of its type and manufacturer.

A Bluetooth calculator

We'll implement a simple calculator that adds two numbers written to its input characteristics and lets the client read the result. We're not going to use predefined services or characteristics, but rather use our own custom ones.



Selecting UUIDs

A base Bluetooth UUID is `00000000-0000-1000-8000-00805F9B34FB` .All services and attributes using short 16 and 32 bit uuids are actually converted from 128 bit `XXXXYYYY-0000-1000-8000-00805F9B34FB` format. When selecting a custom UUID, make sure it is not in that range. Remember that you have no guarantee on the uniqueness of the UUID – somebody else might be also using it.

The code

I'll assume you know how to setup bleno and write the boilerplate code from the previous chapter. First, let's declare the UUIDs.

```
const CALCULATOR_SERVICE_UUID = „00010000-89BD-43C8-9231-40F6E305F96D”;
const ARGUMENT_1_UUID = „00010001-89BD-43C8-9231-40F6E305F96D”;
const ARGUMENT_2_UUID = „00010002-89BD-43C8-9231-40F6E305F96D”;
const RESULT_UUID = „00010010-89BD-43C8-9231-40F6E305F96D”;
```

Now we need to create a characteristic representing an argument. We'll have a generic one for both arguments.

```
class ArgumentCharacteristic extends bleno.Characteristic {
    constructor(uuid, name) {
        super({
            uuid: uuid,
            properties: [„write”],
            value: null,
            descriptors: [
                new bleno.Descriptor({
                    uuid: „2901”,
                    value: name
                  })
            ]
        });

        this.argument = 0;
        this.name = name;
    }

    onWriteRequest(data, offset, withoutResponse, callback) {
        try {
            if(data.length != 1) {
                callback(this.RESULT_INVALID_ATTRIBUTE_LENGTH);
                return;
            }

            this.argument = data.readUInt8();
            console.log(`Argument ${this.name} is now ${this.argument}`);
            callback(this.RESULT_SUCCESS);

        } catch (err) {
            console.error(err);
            callback(this.RESULT_UNLIKELY_ERROR);
        }
    }
}
```

In the constructor, we're calling the base class and specifying the characteristic's UUID, access mode (write) and name descriptor (this is optional). Then we need to implement the onWriteRequestmethod that will be called by bleno framework when the client attempts to write a value. The value is available as a Buffer so we need to parse it.
For simplicity's sake, we just assume the data is an 8 bit unsigned integer, but it can effectively be anything.

*Note: There is a limit on the data size in the characteristic. The maximum is 512 bytes, however this data will not be sent all at once, but rather in chunks. The default chunk size is 23 bytes. In many cases, splitting data into chunks will be handled automatically for you by the Bluetooth programming framework you are using. However, sending a lot of small packets has negative impact on throughput, so you might want to increase the chunk size by performing MTU (Max Transfer Unit) negotiation. Some client libraries, like the Android one, allow you to do that. Others, like the iOS one, handle this themselves. Read more on MTU here.

Similar to argument, we also need the result characteristic:

```
class ResultCharacteristic extends bleno.Characteristic {
    constructor(calcResultFunc) {
        super({
            uuid: RESULT_UUID,
            properties: [„read”],
            value: null,
            descriptors: [
                new bleno.Descriptor({
                    uuid: „2901”,
                    value: „Calculation result”
                  })
            ]
        });

        this.calcResultFunc = calcResultFunc;
    }

    onReadRequest(offset, callback) {
        try {
            const result = this.calcResultFunc();
            console.log(`Returning result: ${result}`);

            let data = new Buffer(1);
            data.writeUInt8(result, 0);
            callback(this.RESULT_SUCCESS, data);
        } catch (err) {
            console.error(err);
            callback(this.RESULT_UNLIKELY_ERROR);
        }
    }
}
```

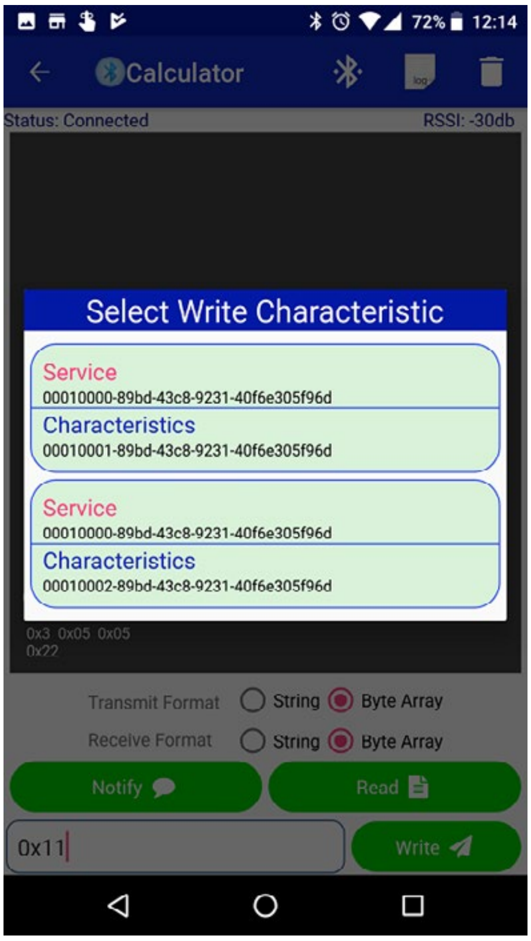
Here, the client needs to read data, so we need to implement `onReadRequest` method. Again, the data needs to be serialized into a `Buffer`.

Now, we need to declare a service:

```
bleno.on(„advertisingStart", err => {

    console.log(„Configuring services...");

    if(err) {
        console.error(err);
        return;
    }

    let argument1 = new ArgumentCharacteristic(ARGUMENT_1_UUID, „Argument 1");
    let argument2 = new ArgumentCharacteristic(ARGUMENT_2_UUID, „Argument 2");
    let result = new ResultCharacteristic(() => argument1.argument + argument2.
argument);

    let calculator = new bleno.PrimaryService({
        uuid: CALCULATOR_SERVICE_UUID,
        characteristics: [
            argument1,
            argument2,
            result
        ]
    });

    bleno.setServices([calculator], err => {
        if(err)
            console.log(err);
        else
            console.log(„Services configured");
    });
});

bleno.on(„stateChange", state => {

    if (state === „poweredOn") {
        bleno.startAdvertising(„Calculator”, [CALCULATOR_SERVICE_UUID], err
=> {
            if (err) console.log(err);
        });

    } else {
        console.log(„Stopping...”);
        bleno.stopAdvertising();
    }
});
```

The full source code is available here.

When you run the app with `sudo node app.js`, you can then test it with a generic BLE client app like this one. It's not ideal, but allows you to do some testing without actually writing client code. Alternatively, you can use `hcitool` and `gatttool` from another Linux machine.



Status: Connected RSSI: -30db

Select Write Characteristic

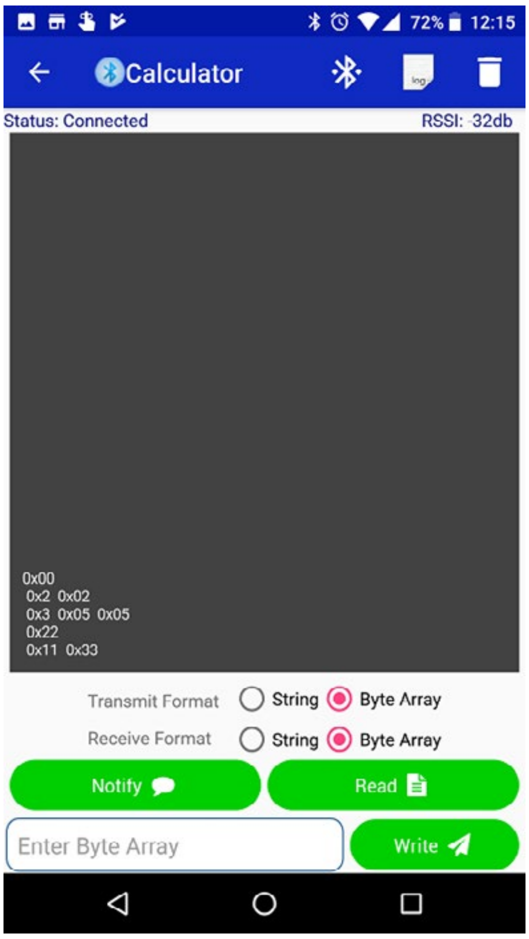
Service
00010000-89bd-43c8-9231-40f6e305f96d
Characteristics
00010001-89bd-43c8-9231-40f6e305f96d

Service
00010000-89bd-43c8-9231-40f6e305f96d
Characteristics
00010002-89bd-43c8-9231-40f6e305f96d

```
0x3  0x05  0x05
0x22
```

Transmit Format String [✓ Byte Array]
Receive Format String [✓ Byte Array]

0x11



Status: Connected RSSI: -32db

```
0x00
 0x2  0x02
 0x3  0x05  0x05
 0x22
 0x11  0x33
```

Transmit Format String [✓ Byte Array]
Receive Format String [✓ Byte Array]

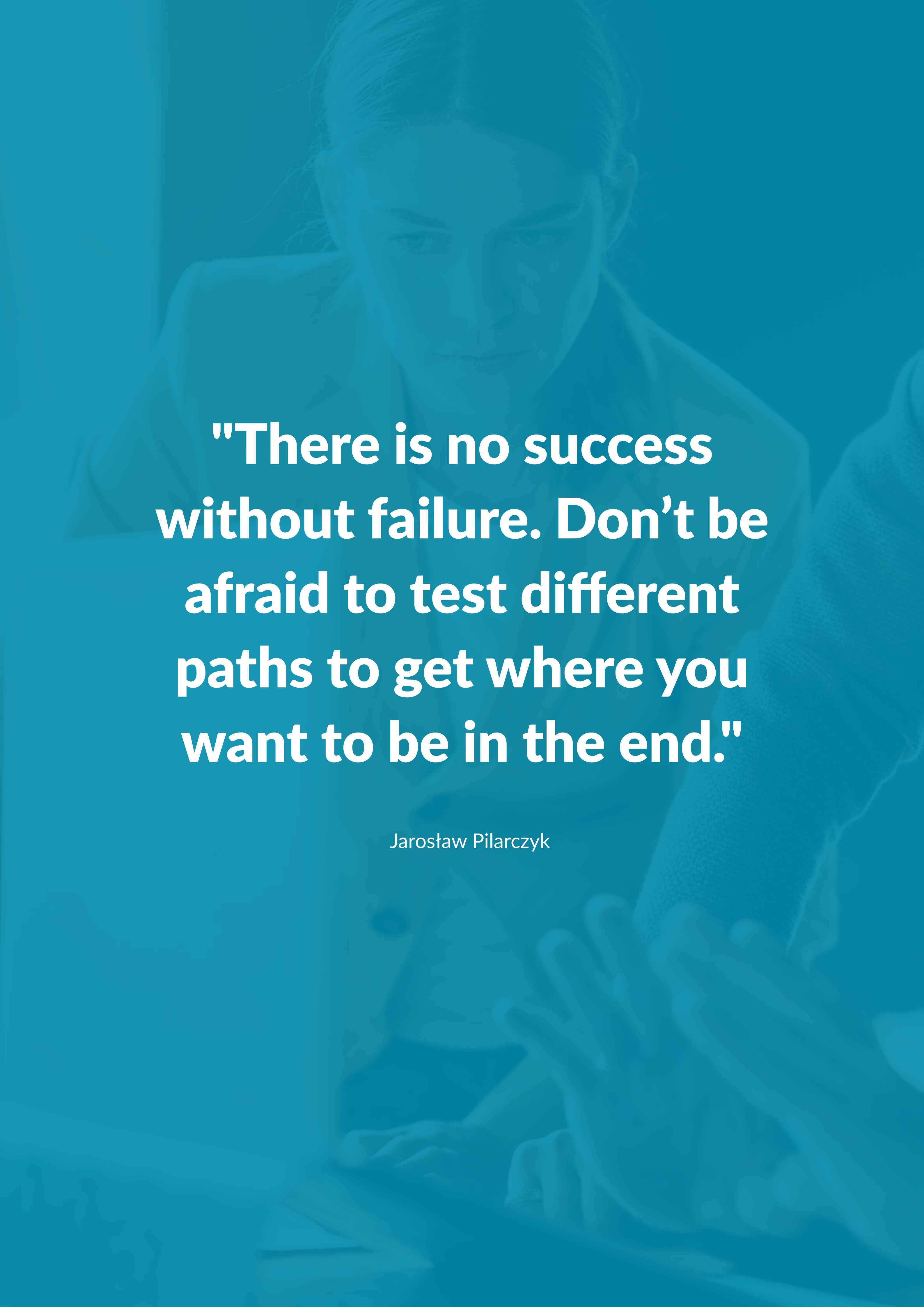
Here's a sample app's output. The disconnect in the middle is required due to poor design of the client app – you can't select another characteristic to write without disconnecting first.

```
Starting bleno...
Bleno: Adapter changed state to poweredOn
Configuring services...
Bleno: servicesSet
Services configured
Bleno: advertisingStart
Bleno: accept 59:99:a8:5d:d7:d6
Argument Argument 2 is now 34
Bleno: disconnect 59:99:a8:5d:d7:d6
Bleno: accept 59:99:a8:5d:d7:d6
Argument Argument 1 is now 17
Returning result: 51
Bleno: disconnect 59:99:a8:5d:d7:d6
```

Summary

In this chapter the GATT protocol was explained and it was shown how to implement a simple Bluetooth service with bleno. In the next chapter, we'll explain how to notify the client about changes in the data.

Chapter 1: Bluetooth with Raspberry Pi and bleno – **iBeacon**
Chapter 3: Bluetooth with Raspberry Pi and bleno – **Notifications**
Chapter 4: Bluetooth with Raspberry Pi and bleno – **iOS client**



"There is no success without failure. Don't be afraid to test different paths to get where you want to be in the end."

Jarosław Pilarczyk

Chapter 3

skyrise.tech

Bluetooth with Raspberry Pi and bleno – Notifications

Dealing with notifications initiated by the peripheral Bluetooth device.

by Marcin Budny

Notifications

Sometimes it is useful to push information to a connected client instead of waiting for it to pull it. An example could be a heartbeat monitor that pushes information about the reading after each detected change. Fortunately, GATT provides us with a way to implement such scenario. A characteristic may support `notify` access mode, so that a client can subscribe to it.

Counter

I'll use a simple counter as an example in this chapter. The counter will increase its value once per second. The client will be able to subscribe to it and observe the values as they change.

Let's start with some UUIDs.

```
const COUNTER_SERVICE_UUID = „00010000-9FAB-43C8-9231-40F6E305F96D”;
const COUNTER_CHAR_UUID = „00010001-9FAB-43C8-9231-40F6E305F96D”;
```

The first step is to declare the counter characteristic:

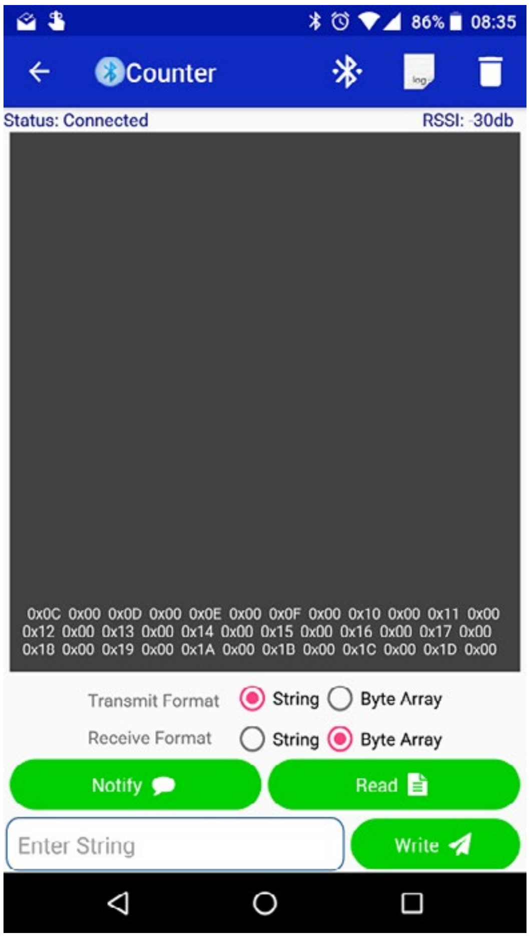
```
class CounterCharacteristic extends bleno.Characteristic {
    constructor() {
        super({
            uuid: COUNTER_CHAR_UUID,
            properties: [„notify”],
            value: null
        });

        this.counter = 0;
    }

    onSubscribe(maxValueSize, updateValueCallback) {
        console.log(`Counter subscribed, max value size is ${maxValueSize}`);
        this.updateValueCallback = updateValueCallback;
    }

    onUnsubscribe() {
        console.log(„Counter unsubscribed”);
        this.updateValueCallback = null;
    }

    sendNotification(value) {
        if(this.updateValueCallback) {
            console.log(`Sending notification with value ${value}`);

            const notificationBytes = new Buffer(2);
            notificationBytes.writeInt16LE(value);

            this.updateValueCallback(notificationBytes);
        }
    }

    start() {
        console.log(„Starting counter”);
        this.handle = setInterval(() => {
            this.counter = (this.counter + 1) % 0xFFFF;
            this.sendNotification(this.counter);
        }, 1000);
    }

    stop() {
        console.log(„Stopping counter”);
        clearInterval(this.handle);
        this.handle = null;
    }
}
```

There are two methods: onSubscribe and onUnsubscribe, which will be called when a client subscribes or unsubscribes from the characteristic. The former gives us the opportunity to store a callback, that will be later used to notify the client about changes in the data. The sendNotification method is called from the timer and makes use of the callback.

To start the peripheral:

```
    let counter = new CounterCharacteristic();
counter.start();


bleno.on(„stateChange”, state => {

    if (state === „poweredOn”) {

        bleno.startAdvertising(„Counter”, [COUNTER_SERVICE_UUID], err => {
            if (err) console.log(err);
        });

    } else {
        console.log(„Stopping...”);
        counter.stop();
        bleno.stopAdvertising();
    }
});

bleno.on(„advertisingStart”, err => {

    console.log(„Configuring services...”);

    if(err) {
        console.error(err);
        return;
    }

    let service = new bleno.PrimaryService({
        uuid: COUNTER_CHAR_UUID,
        characteristics: [counter]
    });

    bleno.setServices([service], err => {
        if(err)
            console.log(err);
        else
            console.log(„Services configured”);
    });
});
```

Again, we can use the generic client app to connect to the device and see the result.



```
0x0C 0x00 0x0D 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x11 0x00
0x12 0x00 0x13 0x00 0x14 0x00 0x15 0x00 0x16 0x00 0x17 0x00
0x18 0x00 0x19 0x00 0x1A 0x00 0x1B 0x00 0x1C 0x00 0x1D 0x00
```

Transmit Format [✓ String] Byte Array
Receive Format String [✓ Byte Array]

You can find full source code for the sample application here.

Summary

This short chapter explains how to deal with notifications initiated by the peripheral Bluetooth device. In the upcoming chapter, we'll show you how to connect and communicate with peripheral devices from mobile apps.

Chapter 1: Bluetooth with Raspberry Pi and bleno – **iBeacon**
Chapter 2: Bluetooth with Raspberry Pi and bleno – **GATT**
Chapter 4: Bluetooth with Raspberry Pi and bleno – **iOS client**

Chapter 4

Bluetooth with Raspberry Pi and bleno – iOS Client

Connecting, reading and writing values for a peripheral Bluetooth device using the BlueCapKit framework

by Konrad Roj

iOS Client

The example presented in this chapter explains how to connect, read, and write values for a peripheral Bluetooth device using the BlueCapKit framework. The sample application connects to the calculator service that was built in chapter 2 of the series.

The client source code is available here, while the GATT server code can be found here.

Discovering the peripheral device

Connecting to a Bluetooth device requires you to go through several steps.

The first one is to actually find the device you want to connect to.

We want to find a device that implements the calculator service, so we have to use the UUID service as a search parameter. If the phone/tablet has bluetooth turned on, we need to start searching for the peripheral. Otherwise, we need to inform the user that bluetooth is turned off (or that another problem has been detected).

```
let stateChangeFuture = manager.whenStateChanges()
let scanStream = stateChangeFuture.flatMap { [unowned self] state -> FutureStre-
am<Peripheral> in
    switch state {
    case .poweredOn:
        return self.manager.startScanning(forServiceUUIDs: [GattUUID.service])
    case .poweredOff, .unauthorized, .unsupported, .resetting, .unknown:
        self.disconnectFromGatt()
        throw CapError.serviceNotFound
    }
}
```

Establishing a connection

Once the device has been found, we can stop the search. The next step is to store the reference to the device and connect to it.

```
let peripherialStream = scanStream.flatMap { [unowned self] discoveredPeripheral
-> FutureStream<Void> in
    self.manager.stopScanning()

    self.peripheral = discoveredPeripheral

    return self.peripheral!.connect(connectionTimeout: 10, capacity: 5)
```

Find Characteristics

After connecting to the device and the selected service, we need to discover the characteristics that will be used to write arguments and read the result of the calculation.

```
let discoveryStream = peripherialStream.flatMap { [unowned self] () -> Futu-
re<Void> in
    return (self.peripheral?.discoverServices([GattUUID.service]))!
}.flatMap { [unowned self] () -> Future<Void> in
    let service = self.peripheral?.services(withUUID: GattUUID.service)?.first

    DispatchQueue.main.async {
        self.connectButton.setTitle(„Connected: \(service!.uuid)”, for: .nor-
mal)
    }

    return service!.discoverCharacteristics([GattUUID.arg1, GattUUID.arg2, Gat-
tUUID.result])
}
```

Save characteristics and prepare UI

Finally, we save the references of the discovered characteristics and prepare the UI.

```
 = discoveryStream.andThen { [unowned self] in
    let service = self.peripheral?.services(withUUID: GattUUID.service)?.first

    guard let resultCharacteristic = service?.characteristics(withUUID: GattU-
UID.result)?.first else {
        self.disconnectFromGatt()
        return
    }

    guard let arg1Characteristic = service?.characteristics(withUUID: GattUUID.
arg1)?.first else {
        self.disconnectFromGatt()
        return
    }

    guard let arg2Characteristic = service?.characteristics(withUUID: GattUUID.
arg2)?.first else {
        self.disconnectFromGatt()
        return
    }

    self.resultChar = resultCharacteristic
    self.arg1Char = arg1Characteristic
    self.arg2Char = arg2Characteristic

    self.prepareUI()
}
```

Data format conversion

We need a way to translate back and forth between the string and byte array (which is used to communicate with the Bluetooth device).

```
let uint = text.map { (val) -> UInt8 in
    return UInt8(String(val))!
}

let data = Data(bytes: uint, count: MemoryLayout<UInt8>.size)




let uint: [UInt8] = [UInt8](self.resultChar!.dataValue!)
let string = String(describing: uint.first ?? 0)
```

Writing for the characteristic

The following code writes an argument for the respective characteristic.

```
func writeArg1(text: String) {
    let uint = text.map { (val) -> UInt8 in
        return UInt8(String(val))!
    }

    let data = Data(bytes: uint, count: MemoryLayout<UInt8>.size)
    let writeFuture = self.arg1Char?.write(data: data)
    writeFuture?.onSuccess(completion: { [unowned self] (_) in
        print(„write succes”)
```

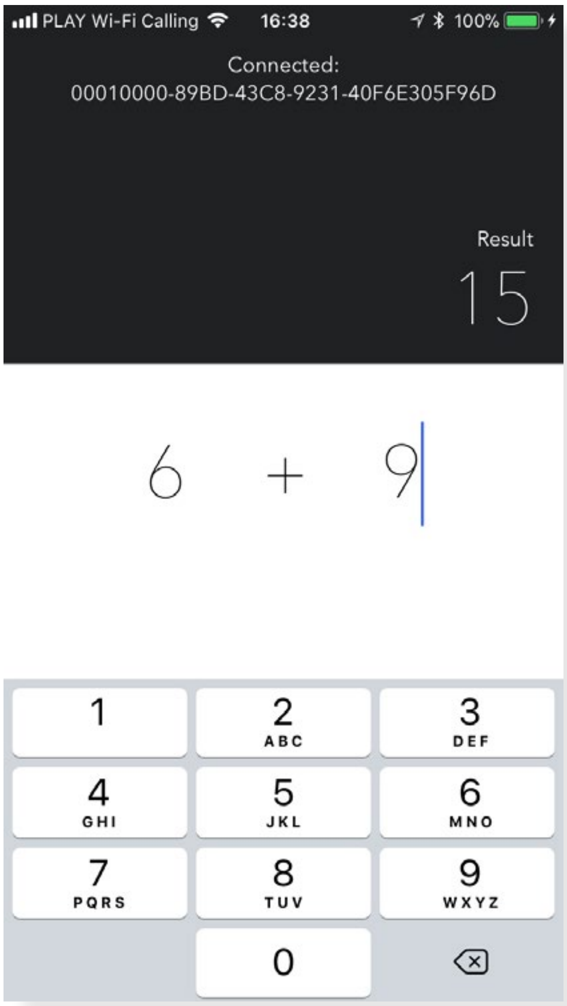
```
            self.readResult()
        })
        writeFuture?.onFailure(completion: { (e) in
            print(„write failed”)
        })
    }
```

Read from the characteristic

Here’s the method that reads the result of the calculation from the device.

```
    func readResult() {
        let readFuture = self.resultChar?.read(timeout: 5)
        readFuture?.onSuccess { [unowned self] (_) in
            let uint: [UInt8] = [UInt8](self.resultChar!.dataValue!)
            DispatchQueue.main.async {
                self.resultLabel.text = String(describing: uint.first ?? 0)
            }
        }
        readFuture?.onFailure { (_) in
            print(„read error”)
        }
    }
```

After putting all of it together, we can do the calculations and observe the results in the UI.



Summary

BlueCapKit is a wrapper over CoreBluetooth that provides an interface based on futures / streams. This allows you to build the logical application flow using closures, so it’s easier to read (you can get rid of many delegates from your code).

Chapter 1: Bluetooth with Raspberry Pi and bleno – **iBeacon**
Chapter 2: Bluetooth with Raspberry Pi and bleno – **GATT**
Chapter 3: Bluetooth with Raspberry Pi and bleno – **Notifications**

skyrise.tech

Contact with experts from Skyrise.tech now!

+48 513 128 281

office@skyrise.tech