My browser reports the site as “Not secure” because I’m using a self-signed certificate for testing purposes, by the way.

Let’s get started on the first set of exercises and meet the Web Bluetooth API!

State Tracking

Our UI will need to allow or restrict various functions depending on whether or not we’ve selected and connected to a LE device and whether or not GATT service discovery has completed. We’ll track these two states in boolean variables. We’ll also update the Status section of the UI and the text on the Discover Devices button, depending on the current state.

Let’s start by adding some variables for state tracking and a couple of functions we can call to update state variables and to reflect this in the UI. I’ve highlighted the changes in red:

<script>

var connected = false;

var services\_discovered = false;

function setConnectedStatus(status) { connected = status;

document.getElementById('status\_connected').innerHTML = status; if (status == true) {

document.getElementById('btn\_scan').innerHTML = "Disconnect";

} else {

document.getElementById('btn\_scan').innerHTML = "Discover Devices";

}

}

function setDiscoveryStatus(status) { services\_discovered = status;

document.getElementById('status\_discovered').innerHTML = status;

}

Device Discovery

Our next job is to complete the discoverDevices () function. The discoverDevicesOrDisconnect() will invoke discoverDevices() in response to the associated UI button being clicked, provided we’re not already connected to a device. If we are already connected, it will disconnect the device. We’ll come to that case later.

discoverDevices() will scan for advertising Bluetooth peripheral devices and offer them in a list for the user to select from and connect to. Web Bluetooth does not allow you to connect to devices without an explicit request via the user interface, from the user. This is another security feature of the API.

Device discovery is performed by the requestDevice API function. It causes a UI dialogue to pop up and to be populated with suitable LE devices, detected by scanning. It’s a “thenable” promise which becomes fulfilled when the user picks one of the listed devices or rejected if anything else happens, such as clicking the cancel button. If fulfilled, a JSON parameter object with properties that describe the selected device is passed to the function inside the “then block”.

Update your code so that it includes the following:

<script>

var connected = false;

var services\_discovered = false; var selected\_device;

var connected\_server;

function discoverDevicesOrDisconnect() { console.log("discoverDevicesOrDisconnect"); if (!connected) {

discoverDevices();

} else {

// TODO disconnect from the current device

}

}

function discoverDevices() { console.log("discoverDevices");

var options = {

acceptAllDevices: true

}

navigator.bluetooth.requestDevice(options)

.then(device => {

console.log('> Name: ' + device.name);

console.log('> Id: ' + device.id); console.log('> Connected: ' + device.gatt.connected); selected\_device = device;

console.log(selected\_device);

})

.catch(error => {

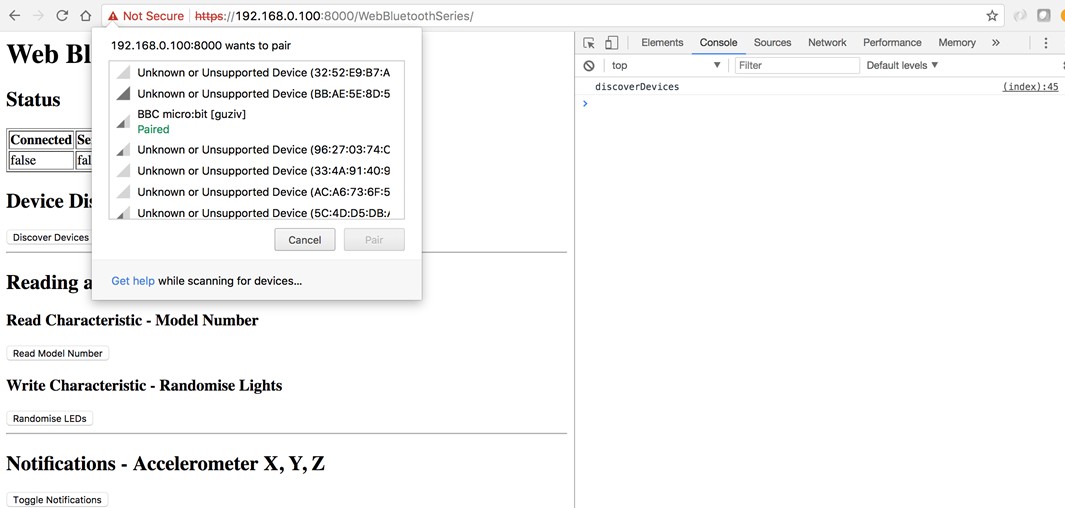
alert('ERROR: ' + error); console.log('ERROR: ' + error);

});

}

Reload the index.html page and open the Chrome developer console so you can watch the log. Now click the Discover Devices button. You should see something like this, with the list content differing according to the LE devices discovered in your environment.

Note: If you are running on Windows and get unexpected results from the Discover Devices button, check the console. If you see this message “Web Bluetooth is experimental on this platform. See <https://github.com/WebBluetoothCG/web-bluetooth/blob/gh-pages/implementation-status.md>” it means that you have not properly installed and enabled the Web Bluetooth polyfill for Windows, described in the Getting Started section, above.



Select your test device, assuming you see it on the list, and click Pair. Note that in Web Bluetooth “Pair” means “select” rather than the usual meaning in the context of Bluetooth. You’ll see details of the selected device logged to the developer console.

discoverDevices

(index):23 BluetoothDevice {id: "kQGGIaP9Qv572mPDwe1cBg==", name: "BBC micro:bit [guziv]", gatt: BluetoothRemoteGATTServer, ongattserverdisconnected: null}

BBC micro:bit [guziv] kQGGIaP9Qv572mPDwe1cBg== false

(index):17 > Name: (index):18 > Id: (index):19 > Connected:

Filtering

The way we’re using requestDevice currently, all advertising devices are being listed. It’s perhaps more usual to filter out devices so that only devices of a type which is relevant to the application, are listed. Filtering is accomplished using the options object which currently looks like this:

var options = {

acceptAllDevices: true

}

Let’s change the options object so that only BBC micro:bits are listed, according to the device name advertised. Furthermore, we’ll also indicate the UUIDs of the GATT services we want access to. This is another security requirement.

**Note: skip the next bit if you’re test device is using the node.js bleno code provided at the end of**

**this article.**

Update your code to define the UUIDs of the GATT services we’ll be using and to change the options object we use with the requestDevice API as shown here, so that only devices whose advertised name starts with “BBC” are selected.

// service UUIDs

ACCELEROMETER\_SERVICE = 'e95d0753-251d-470a-a062-fa1922dfa9a8'; LED\_SERVICE = 'e95dd91d-251d-470a-a062-fa1922dfa9a8'; DEVICE\_INFORMATION\_SERVICE = '0000180a-0000-1000-8000-00805f9b34fb';

function discoverDevices() { console.log("discoverDevices");

var options = {

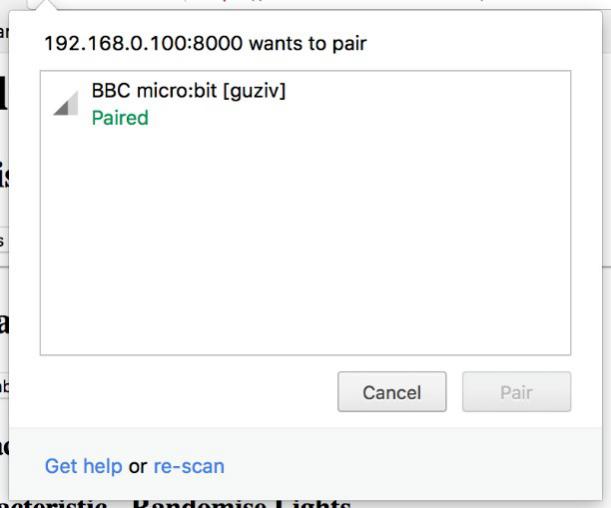
filters: [{ namePrefix: 'BBC' }], optionalServices: [DEVICE\_INFORMATION\_SERVICE,

ACCELEROMETER\_SERVICE, LED\_SERVICE]

}

Reload the index.html page and click Discover Devices. This time you should see only devices whose

name starts with “BBC” listed.



Connecting

Next, we’ll modify our code so that we connect to the device selected by the user. We’re already

storing the JSON object representing the selected device. This object includes a

BluetoothRemoteGattServer object and it’s this object which contains the API function we use to initiate a connection.

BluetoothDevice {id: "kQGGIaP9Qv572mPDwe1cBg==", name: "BBC micro:bit [guziv]",

**gatt: BluetoothRemoteGATTServer**, ongattserverdisconnected: null}

Update the requestDevice code so that a new function, connect() is called:

' + device.name); ' + device.id);

' + device.gatt.connected);

navigator.bluetooth.requestDevice(options)

.then(device => {

console.log('> Name: console.log('> Id: console.log('> Connected: selected\_device = device; console.log(selected\_device); connect();

and add the following new functions:

function connect() {

if (connected == false) { console.log("connecting"); selected\_device.gatt.connect().then(

function (server) {

console.log("Connected to " + server.device.id); console.log("connected=" + server.connected); setConnectedStatus(true);

connected\_server = server;

selected\_device.addEventListener ( 'gattserverdisconnected',

onDisconnected);

},

function (error) {

console.log("ERROR: could not connect - " + error); alert("ERROR: could not connect - " + error); setConnectedStatus(false);

});

}

}

function onDisconnected() { console.log("onDisconnected"); resetUI();

}

function resetUI() {

setConnectedStatus(false); setDiscoveryStatus(false);

document.getElementById('model\_number').innerHTML = ""; document.getElementById('accelerometer\_data').innerHTML = "";

}

The gatt.connect() call initiates the connection process and returns a BluetoothDevice object in the server parameter when a connection has been established. As you can see, we store the BluetoothDevice object in the connected\_server variable so we can use it later.

Reload index.html and test. You’ll see something like the following output in the console and if your device has been programmed to visually indicate it has accepted a connection, you’ll see it respond accordingly (I always display a “C” on the micro:bit LED display when it accepts a connection).

(index):27 BluetoothDevice {id: "kQGGIaP9Qv572mPDwe1cBg==", name: "BBC micro:bit [guziv]", gatt: BluetoothRemoteGATTServer, ongattserverdisconnected: null} (index):48 connecting

(index):51 Connected to kQGGIaP9Qv572mPDwe1cBg== (index):52 connected=true

BBC micro:bit [guziv] kQGGIaP9Qv572mPDwe1cBg== false

discoverDevices (index):23 > Name: (index):24 > Id: (index):25 > Connected:

Services and Characteristics

Our penultimate task is to discover the GATT services and characteristics which are on the connected device. We need to make a note of the services discovered so that we can validate that the device is suitable for our purposes. We also need to cache those characteristics we want to work with.

Add some variables which we’ll use shortly:

// presence of services and characteristics var has\_accelerometer\_service = false;

var has\_accelerometer\_data = false;

var has\_led\_service = false;

var has\_led\_matrix\_state = false;

var has\_device\_information\_service = false; var has\_model\_name\_string = false;

// characteristic UUIDs

ACCELEROMETER\_DATA = 'e95dca4b-251d-470a-a062-fa1922dfa9a8'; LED\_MATRIX\_STATE = 'e95d7b77-251d-470a-a062-fa1922dfa9a8'; MODEL\_NUMBER\_STRING = '00002a24-0000-1000-8000-00805f9b34fb';

// cached characteristics var accelerometer\_data; var led\_matrix\_state; var model\_number\_string;

Now update your code so that when a connection has been established, a call to a new function, discoverSvcsAndChars is made.

selected\_device.gatt.connect().then( function (server) {

console.log("Connected to " + server.device.id); console.log("connected=" + server.connected); setConnectedStatus(true);

connected\_server = server; selected\_device.addEventListener('gattserverdisconnected',

onDisconnected);

discoverSvcsAndChars();

},

And add the discoverSvcsAndChars function itself:

function discoverSvcsAndChars() {

console.log("discoverSvcsAndChars server=" + connected\_server); connected\_server.getPrimaryServices()

.then(services => {

has\_accelerometer\_service = false; has\_led\_service = false; has\_device\_information\_service = false;

services\_discovered = 0; service\_count = services.length;

console.log("Got " + service\_count + " services");

services.forEach(service => {

if (service.uuid == ACCELEROMETER\_SERVICE) { has\_accelerometer\_service = true;

}

if (service.uuid == LED\_SERVICE) { has\_led\_service = true;

}

if (service.uuid == DEVICE\_INFORMATION\_SERVICE) { has\_device\_information\_service = true;

service.uuid);

characteristic.uuid); ACCELEROMETER\_DATA) {

characteristic;

LED\_MATRIX\_STATE) {

}

console.log('Getting Characteristics for service ' +

service.getCharacteristics().then(characteristics => { console.log('> Service: ' + service.uuid); services\_discovered++; characteristics\_discovered = 0; characteristic\_count = characteristics.length; characteristics.forEach(characteristic => {

characteristics\_discovered++; console.log('>> Characteristic: ' +

if (characteristic.uuid ==

accelerometer\_data = has\_accelerometer\_data = true;

}

if (characteristic.uuid ==

led\_matrix\_state = characteristic; has\_led\_matrix\_state = true;

MODEL\_NUMBER\_STRING) {

characteristic;

}

if (characteristic.uuid == model\_number\_string =

has\_model\_name\_string = true;

}

if (services\_discovered == service\_count

&& characteristics\_discovered == characteristic\_count) {

console.log("FINISHED DISCOVERY"); setDiscoveryStatus(true);

});

}

});

});

}

});

### Our new code works like this; we make a call to the getPrimaryServices function of the BluetoothDevice object which we were given when the connection was established. This produces

an argument called services which contains an array of BluetoothRemoteGATTService objects, which we then iterate through. For each service, we iterate through its array of BluetoothRemoteGATTCharacteristic objects. Boolean variables get set to note which of the services and characteristics we’re interested in were discovered and we store the characteristic objects themselves.

Reload and test and your console should look something like this:

(index):43 BluetoothDevice {id: "kQGGIaP9Qv572mPDwe1cBg==", name: "BBC micro:bit [guziv]", gatt: BluetoothRemoteGATTServer, ongattserverdisconnected: null} (index):64 connecting

(index):67 Connected to kQGGIaP9Qv572mPDwe1cBg== (index):68 connected=true

(index):89 discoverSvcsAndChars server=[object BluetoothRemoteGATTServer] (index):92 Getting Characteristics...

(index):108 > Service: e95d0753-251d-470a-a062-fa1922dfa9a8 (index):110 >> Characteristic: e95dfb24-251d-470a-a062-fa1922dfa9a8 (index):110 >> Characteristic: e95dca4b-251d-470a-a062-fa1922dfa9a8 (index):108 > Service: e95dd91d-251d-470a-a062-fa1922dfa9a8 (index):110 >> Characteristic: e95d0d2d-251d-470a-a062-fa1922dfa9a8 (index):110 >> Characteristic: e95d93ee-251d-470a-a062-fa1922dfa9a8 (index):110 >> Characteristic: e95d7b77-251d-470a-a062-fa1922dfa9a8 (index):108 > Service: 0000180a-0000-1000-8000-00805f9b34fb

(index):110 >> Characteristic: 00002a26-0000-1000-8000-00805f9b34fb (index):110 >> Characteristic: 00002a24-0000-1000-8000-00805f9b34fb FINISHED DISCOVERY

BBC micro:bit [guziv] kQGGIaP9Qv572mPDwe1cBg== false

discoverDevices (index):39 > Name: (index):40 > Id: (index):41 > Connected:

The Service Discovery Completed panel in the Status section of the UI should now contain “true”.

If you don’t get this result, check your code and if necessary, replace with the full solution to this part of the tutorials below. Make sure your LE device does have the three GATT services and child characteristics we’re working with and that they have the correct UUIDs.

Disconnecting

Last, we still need to disconnect from the remote device when necessary. Update the discoverDevicesOrDisconnect function to complete the else clause.

function discoverDevicesOrDisconnect() { console.log("discoverDevicesOrDisconnect"); if (!connected) {

discoverDevices();

} else {

selected\_device.gatt.disconnect(); resetUI();

}

Close

We’ve accomplished a lot in this tutorial. You should now have a good feel for how device discovery works in Web Bluetooth, including how to filter unwanted devices and how to ensure the services we will want to access are accessible. You can also create a connection and then discover and cache services and characteristics.

In the next part we’ll learn how to read and write characteristic values.

Call to Action

Web Bluetooth is not yet a W3C standard. I think it needs to be. Web Bluetooth is not yet implemented in all browsers either1. And it really needs to be in my humble opinion. Right now, you’ll find Web Bluetooth in Chrome on most platforms. The “caniuse” URL in the footnote will give you full information.

If, you feel the same as I do, then I invite and encourage you to petition browser implementers to get behind Web Bluetooth and progress it through the W3C standards process. IoT needs this. You need this.

|  |  |
| --- | --- |
| **Mozilla Firefox** | <https://bugzilla.mozilla.org/show_bug.cgi?id=674737> |
| **Microsoft Edge** | [https://developer.microsoft.com/en-us/microsoft-](https://developer.microsoft.com/en-us/microsoft-edge/platform/status/webbluetooth/) [edge/platform/status/webbluetooth/](https://developer.microsoft.com/en-us/microsoft-edge/platform/status/webbluetooth/)  [https://wpdev.uservoice.com/forums/257854-](https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/9775308-implement-the-web-bluetooth-gatt-client-api) [microsoft-edge-developer/suggestions/9775308-](https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/9775308-implement-the-web-bluetooth-gatt-client-api) [implement-the-web-bluetooth-gatt-client-api](https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/9775308-implement-the-web-bluetooth-gatt-client-api) |
| **WebKit**  **(Safari)** | <https://bugs.webkit.org/show_bug.cgi?id=101034> |

The W3C also host a public email list for Web Bluetooth: [https://lists.w3.org/Archives/Public/public-](https://lists.w3.org/Archives/Public/public-web-bluetooth/) [web-bluetooth/](https://lists.w3.org/Archives/Public/public-web-bluetooth/)

Solution

The solution to the tutorial exercises that we’ve covered so far is as follows:

<html>

<head>

<script>

var connected = false;

var services\_discovered = false; var selected\_device;

var connected\_server;

// service UUIDs

ACCELEROMETER\_SERVICE = 'e95d0753-251d-470a-a062-fa1922dfa9a8'; LED\_SERVICE = 'e95dd91d-251d-470a-a062-fa1922dfa9a8'; DEVICE\_INFORMATION\_SERVICE = '0000180a-0000-1000-8000-00805f9b34fb';

// presence of services and characteristics var has\_accelerometer\_service = false;

var has\_accelerometer\_data = false;

var has\_led\_service = false;

var has\_led\_matrix\_state = false;

var has\_device\_information\_service = false; var has\_model\_name\_string = false;

// characteristic UUIDs

ACCELEROMETER\_DATA = 'e95dca4b-251d-470a-a062-fa1922dfa9a8'; LED\_MATRIX\_STATE = 'e95d7b77-251d-470a-a062-fa1922dfa9a8'; MODEL\_NUMBER\_STRING = '00002a24-0000-1000-8000-00805f9b34fb';

1 See [https://caniuse.com/#search=web%20bluetooth](https://caniuse.com/#search%3Dweb%20bluetooth)

// cached characteristics var accelerometer\_data; var led\_matrix\_state; var model\_number\_string;

function discoverDevicesOrDisconnect() { console.log("discoverDevicesOrDisconnect"); if (!connected) {

discoverDevices();

} else {

selected\_device.gatt.disconnect(); resetUI();

}

}

function discoverDevices() { console.log("discoverDevices");

setConnectedStatus(false); setDiscoveryStatus(false);

var options = {

filters: [{ namePrefix: 'BBC' }], optionalServices: [DEVICE\_INFORMATION\_SERVICE,

ACCELEROMETER\_SERVICE, LED\_SERVICE]

}

device.name);

navigator.bluetooth.requestDevice(options)

.then(device => {

console.log('> Name: ' +

console.log('> Id: ' + device.id);

console.log('> Connected: ' +

device.gatt.connected);

})

selected\_device = device; console.log(selected\_device); connect();

.catch(error => {

alert('ERROR: ' + error); console.log('ERROR: ' + error);

});

}

function readModelNumber() { console.log("readModelNumber");

}

function randomLEDs() { console.log("randomLEDs");

}

function toggleAccelerometerNotifications() { console.log("toggleAccelerometerNotifications");

}

function connect() {

if (connected == false) { console.log("connecting"); selected\_device.gatt.connect().then(

function (server) {

console.log("Connected to " +

server.device.id); server.connected);

console.log("connected=" +

setConnectedStatus(true); connected\_server = server;

selected\_device.addEventListener('gattserverdisconnected', onDisconnected);

discoverSvcsAndChars();

},

+ error); error);

function (error) {

console.log("ERROR: could not connect - " alert("ERROR: could not connect - " +

setConnectedStatus(false);

});

}

}

function onDisconnected() { console.log("onDisconnected"); resetUI();

}

function discoverSvcsAndChars() { console.log("discoverSvcsAndChars server=" +

connected\_server);

services");

connected\_server.getPrimaryServices()

.then(services => {

has\_accelerometer\_service = false; has\_led\_service = false; has\_device\_information\_service = false;

services\_discovered = 0; service\_count = services.length;

console.log("Got " + service\_count + "

ACCELEROMETER\_SERVICE) {

services.forEach(service => { if (service.uuid ==

has\_accelerometer\_service = true;

}

if (service.uuid == LED\_SERVICE) { has\_led\_service = true;

DEVICE\_INFORMATION\_SERVICE) {

true;

service ' + service.uuid);

}

if (service.uuid ==

has\_device\_information\_service =

}

console.log('Getting Characteristics for

service.getCharacteristics().then(characteristics => {

console.log('> Service: ' +

service.uuid);

characteristics.length;

services\_discovered++; characteristics\_discovered = 0; characteristic\_count =

characteristics.forEach(characteristic => {

characteristics\_discovered++; Characteristic: ' + characteristic.uuid); ACCELEROMETER\_DATA) {

characteristic;

console.log('>>

if (characteristic.uuid == accelerometer\_data =

has\_accelerometer\_data = true;

LED\_MATRIX\_STATE) {

characteristic;

= true;

MODEL\_NUMBER\_STRING) {

characteristic;

= true;

}

if (characteristic.uuid == led\_matrix\_state = has\_led\_matrix\_state

}

if (characteristic.uuid == model\_number\_string = has\_model\_name\_string

}

if (services\_discovered ==

service\_count && characteristics\_discovered == characteristic\_count) {

console.log("FINISHED

DISCOVERY");

setDiscoveryStatus(true);

});

}

});

});

}

});

status; "Disconnect";

function setConnectedStatus(status) { connected = status;

document.getElementById('status\_connected').innerHTML =

if (status == true) {

document.getElementById('btn\_scan').innerHTML =

} else {

document.getElementById('btn\_scan').innerHTML =

"Discover Devices";

}

}

status;

function setDiscoveryStatus(status) { services\_discovered = status;

document.getElementById('status\_discovered').innerHTML =

}

function resetUI() {

setConnectedStatus(false); setDiscoveryStatus(false);

document.getElementById('model\_number').innerHTML = ""; document.getElementById('accelerometer\_data').innerHTML = "";

}

</script>

</head>

<body>

<h1>Web Bluetooth</h1>

<h2>Status</h2>

<table border="1">

<tr>

<td>

</td>

<b>Connected</b>

<td>

<b>Service Discovery Completed</b>

</td>

<td>

<b>Notifications</b>

</td>

</tr>

<tr>

<td id="status\_connected">false</td>

<td id="status\_discovered">false</td>

<td id="status\_notifications">false</td>

</tr>

</table>

<h2>Device Discovery and Connection</h2>

<button id="btn\_scan" onclick="discoverDevicesOrDisconnect()">Discover Devices</button>

<hr>

<h2>Reading and Writing</h2>

<h3>Read Characteristic - Model Number</h3>

<button id="btn\_read" onclick="readModelNumber()">Read Model Number</button>

<div id="model\_number"></div>

<h3>Write Characteristic - Randomise Lights</h3>

<button id="btn\_write" onclick="randomLEDs()">Randomise LEDs</button>

<hr>

<h2>Notifications - Accelerometer X, Y, Z</h2>

<button id="btn\_notify" onclick="toggleAccelerometerNotifications()">Toggle Notifications</button>

<div id="accelerometer\_data"></div>

</body>

</html>

# Peripheral Device Code

## BBC micro:bit - C/C++ using Yotta

#include "MicroBit.h" MicroBit uBit;

void onConnected(MicroBitEvent)

{

uBit.display.print("C");

}

void onDisconnected(MicroBitEvent)

{

uBit.display.print("D");

}

int main()

{

// Initialise the micro:bit runtime. uBit.init();

uBit.display.print("X");

uBit.messageBus.listen(MICROBIT\_ID\_BLE, MICROBIT\_BLE\_EVT\_CONNECTED, onConnected);

uBit.messageBus.listen(MICROBIT\_ID\_BLE, MICROBIT\_BLE\_EVT\_DISCONNECTED, onDisconnected);

// services: note that the device information service is included by default

// see config.json property microbit-dal.bluetooth. device\_info\_service

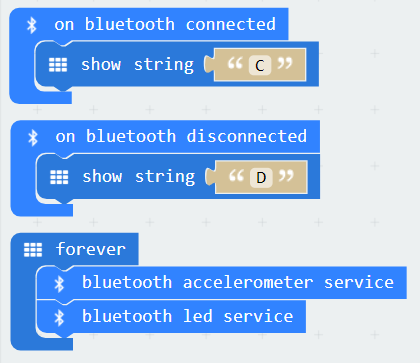
new MicroBitAccelerometerService(\*uBit.ble, uBit.accelerometer); new MicroBitLEDService(\*uBit.ble, uBit.display);

release\_fiber();

}

BBC micro:bit - <https://makecode.microbit.org/>

### Edit and run the code live at: <https://makecode.microbit.org/_fFH7fodFJ7U3>



Arduino 101

#include <SoftwareSerial.h> int connected = 0;

// bluetooth

#include <CurieBLE.h>

// BLE objects

BLEPeripheral blePeripheral;

// GAP properties

char device\_name[] = "BBC micro:bit [xxxxx]";

// Characteristic Properties

unsigned char model\_number\_string\_props = BLERead | 0; unsigned char led\_matrix\_state\_props = BLEWrite | BLERead | 0;

unsigned char accelerometer\_data\_props = BLERead | BLENotify | 0;

// Services and Characteristics

BLEService device\_information\_service("180A");

BLECharacteristic model\_number\_string("2A24", model\_number\_string\_props, "BBC micro:bit");

BLEService led\_service("E95DD91D-251D-470A-A062-FA1922DFA9A8");

BLECharacteristic led\_matrix\_state("E95D7B77-251D-470A-A062-FA1922DFA9A8", led\_matrix\_state\_props, 5);

BLEService accelerometer\_service("E95D0753-251D-470A-A062-FA1922DFA9A8"); BLECharacteristic accelerometer\_data("E95DCA4B-251D-470A-A062-FA1922DFA9A8", accelerometer\_data\_props, 6);

unsigned char initial\_led\_state[] = { 0x00, 0x00, 0x00, 0x00, 0x00 };

unsigned char initial\_accelerometer\_data[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

};

long timestamp = 0; void setup(void)

{

Serial.begin(115200); Serial.println(F("Arduino setup"));

randomSeed(analogRead(0));

// set advertising packet content blePeripheral.setLocalName(device\_name); blePeripheral.setDeviceName(device\_name);

// add services and characteristics blePeripheral.addAttribute(device\_information\_service); blePeripheral.addAttribute(model\_number\_string); blePeripheral.addAttribute(led\_service); blePeripheral.addAttribute(led\_matrix\_state); blePeripheral.addAttribute(accelerometer\_service); blePeripheral.addAttribute(accelerometer\_data); led\_matrix\_state.setValue(initial\_led\_state,5); accelerometer\_data.setValue(initial\_accelerometer\_data,6); Serial.println("attribute table constructed");

// begin advertising blePeripheral.begin(); Serial.println("advertising");

}

void loop()

{

// listen for BLE peripherals to connect: BLECentral central = blePeripheral.central();

// if a central is connected to peripheral: if (central) {

connected = 1;

// while the central is still connected to peripheral: while (central.connected()) {

long current\_ms = millis();

// if 500ms have passed notify with faked accelerometer data if (current\_ms - timestamp >= 500) {

timestamp = current\_ms; notifyAccelerometerData();

}

if (led\_matrix\_state.written()) { Serial.println("LED matrix state changed:"); printDec2Bin(led\_matrix\_state.value()[0]); printDec2Bin(led\_matrix\_state.value()[1]); printDec2Bin(led\_matrix\_state.value()[2]); printDec2Bin(led\_matrix\_state.value()[3]); printDec2Bin(led\_matrix\_state.value()[4]);

}

}

}

// when the central disconnects, print it out: if (connected == 1) {

connected = 0;

}

}

void notifyAccelerometerData() {

unsigned char accel\_data[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

// random wobble

unsigned char X\_wobble = random(0, 250); accel\_data[0] = X\_wobble;

unsigned char Y\_wobble = random(0, 250); accel\_data[2] = Y\_wobble;

unsigned char Z\_wobble = random(0, 250); accel\_data[4] = Z\_wobble;

accelerometer\_data.setValue(accel\_data,6);

}

void printDec2Bin(char value)

{

int i = 0;

for(i = 5; i >= 0; i--){

if((value & (1 << i)) != 0){

Serial.print("1");

}else{

Serial.print("0");

}

}

Serial.println(" ");

}

### Arduino usage note:

*Show the Arduino IDE Serial Monitor to watch values written to the LED Matrix State characteristic.*

RaspBerry Pi and other devices that run node.js

Here’s the code for a Raspberry Pi or other device which has an LE stack and runs node.js. You need

to install the bleno library using

npm install bleno

and then place this code in a file called main.js

var bleno = require('bleno');

bleno.on('stateChange', function (state) { console.log('on -> stateChange: ' + state);

if (state === 'poweredOn') { bleno.startAdvertising('BBC micro:bit [xxxxx]');

} else { bleno.stopAdvertising();

}

});

bleno.on('accept', function (clientAddress) { console.log('on -> accept, client: ' + clientAddress); bleno.updateRssi();

});

bleno.on('disconnect', function (clientAddress) { console.log("Disconnected from address: " + clientAddress);

});

bleno.on('rssiUpdate', function (rssi) { console.log('on -> rssiUpdate: ' + rssi);

});

bleno.on('advertisingStart', function (error) {

console.log('on -> advertisingStart: ' + (error ? 'error ' + error : 'success'));

if (!error) { bleno.setServices([

// device information service - do not include if executing on a Mac new bleno.PrimaryService({

uuid: '180a', characteristics: [

// Model Number String

new bleno.Characteristic({ value: null,

uuid: '2a24', properties: ['read'],

onReadRequest: function (offset, callback) { console.log("Read request received"); callback(this.RESULT\_SUCCESS, new Buffer("BBC micro:bit"));

}

})

]

}),

// micro:bit LED service new bleno.PrimaryService({

uuid: 'e95dd91d-251d-470a-a062-fa1922dfa9a8', characteristics: [

// LED matrix state

new bleno.Characteristic({ value: null,

uuid: 'e95d7b77-251d-470a-a062-fa1922dfa9a8', properties: ['write'],

onWriteRequest: function (data, offset, withoutResponse, callback) { this.value = data;

console.log('Write request'); if (data.length == 5) {

printDec2Bin(data[0]); printDec2Bin(data[1]); printDec2Bin(data[2]); printDec2Bin(data[3]); printDec2Bin(data[4]);

} else {

console.log("ERROR: invalid characteristic value length");

]

}),

}

callback(this.RESULT\_SUCCESS);

}

})

// micro:bit accelerometer service new bleno.PrimaryService({

uuid: 'e95d0753-251d-470a-a062-fa1922dfa9a8', characteristics: [

// Accelerometer Data

new bleno.Characteristic({ value: null,

uuid: 'e95dca4b-251d-470a-a062-fa1922dfa9a8', properties: ['read', 'notify'], onReadRequest: function (offset, callback) {

console.log("Read request received"); callback(this.RESULT\_SUCCESS, new Buffer("BBC micro:bit"));

},

onSubscribe: function (maxValueSize, updateValueCallback) { console.log("subscribed to accelerometer notifications"); this.intervalId = setInterval(function () {

accel\_data = fakeAccelerometerData(); console.log(accel\_data); updateValueCallback(accel\_data);

}, 1000);

},

// If the client unsubscribes, we stop broadcasting the message onUnsubscribe: function () {

console.log("unsubscribed from accelerometer notifications"); clearInterval(this.intervalId);

}

})

]

})

]);

}

});

bleno.on('servicesSet', function (error) {

console.log('on -> servicesSet: ' + (error ? 'error ' + error : 'success'));

});

function printDec2Bin(value) { binary = '';

i = 0;

for (i = 5; i >= 0; i--) {

if ((value & (1 << i)) != 0) { binary = binary + "1";

} else {

binary = binary + "0";

}

}

console.log(binary);

}

function fakeAccelerometerData() { accel\_data\_buffer = new ArrayBuffer(6); accel\_data = new Uint8Array(accel\_data\_buffer);

// random wobble

X\_wobble = Math.floor(Math.random() \* 250); accel\_data[0] = X\_wobble;

Y\_wobble = Math.floor(Math.random() \* 250); accel\_data[2] = Y\_wobble;

Z\_wobble = Math.floor(Math.random() \* 250); accel\_data[4] = Z\_wobble;

return accel\_data;

}

### Run with the command

sudo BLENO\_DEVICE\_NAME="BBC micro:bit [xxxxx]" node main.js

***Note****: Due to limitations in the bleno API, the device name will appear in the Shortened Local Name field rather than the Complete Local Name field and consequently the Web Bluetooth filtering will not select the device. When you get to the Filtering part of the tutorial, therefore retain the acceptAllDevices filter options and do not implement the device name filtering.*

***Note****: If you’re running the above node.js script on an Apple Mac, remove the code shown in red. Apple do not allow you to instantiate the device information service and various other standard services with UUIDs issued by the Bluetooth SIG. The service will automatically be there with device model string name set by Apple to reflect the model of computer you’re running on. On mine it says “MacBookPro12,1” for example.*

General Note:

The bare minimum services and characteristics have been implemented and, strictly speaking the services do not comply with their specifications, which contain other mandatory characteristics. But for learning and testing purposes, this is fine.