Using Bluetooth LE with the imp004m

Advertising, scanning and data-exchange

This document covers functionality not yet part of a production impOS release and is therefore subject to change

Electric Imp has enabled Bluetooth LE (Low Energy) operation on the imp004m, which is the first imp module to support this wireless technology. Bluetooth LE is a subset of the Bluetooth standard, and was introduced in Bluetooth 4.0. The imp004m can therefore communicate with almost any device that supports Bluetooth 4.0 and up. For example, all of the current mobile device platforms support Bluetooth LE natively.

The imp004m's Bluetooth LE functionality is already part of its FCC/IC and TELEC modular approval, and CE test results are available. In addition, the imp004m and its Bluetooth stack are listed as a qualified design by the Bluetooth SIG, so no further Bluetooth testing is required before shipment by an SIG member.

To support Bluetooth, impOS™ 38 incorporates a rich API to access the imp004m's BLE functionality. impOS 38 is expected to be made available to customers in 2018, and beta builds are available now.

Bluetooth Hardware

The imp004m's WiFi chip is the Cypress Semiconductor CYW43438 communication chip, which also supports Bluetooth 4.1. The CYW43438 Bluetooth unit is not connected to the imp004m's MCU, but the CYW43438's Bluetooth-specific pins are brought out to the imp004m pin-out to make them accessible to hardware designers who wish to make use of the module's Bluetooth features in their products. To do so, you will need to:

- Select an imp UART bus and connect its RX and TX pins to CYW43438 pins BT_UART_TXD and BT_UART_RXD, respectively
- Pull down CYW43438 pin BT_UART_CTS_N
- Supply a 32.768kHz clock signal to CYW43438 pin LPO_IN if power usage is important, otherwise pull LPO_IN low
    - **Note** This configuration is not supported in impOS 38 but is expected to be added in the following public impOS release

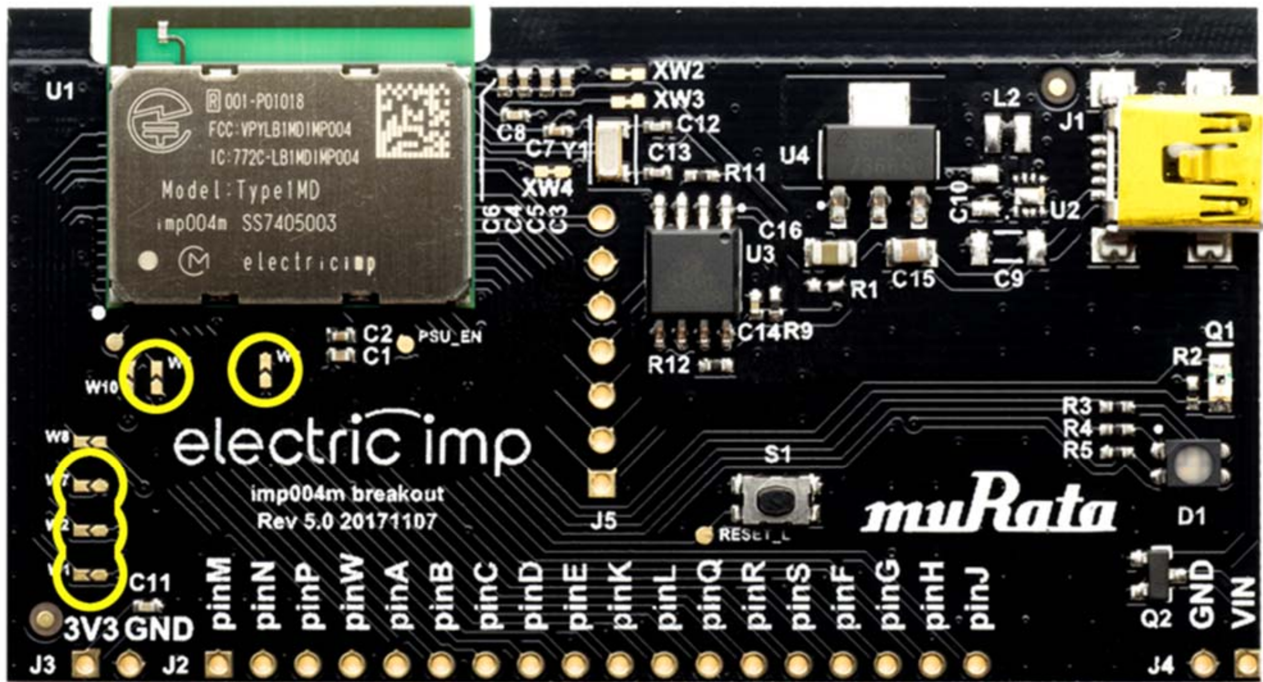
- Set CYW43438 pin BT_REG_ON high to power the Bluetooth sub-system

After making use of Bluetooth, you should set CYW43438 pins BT_REG_ON and LPO_IN low and if you wish to power down and fully disable the imp004m's Bluetooth sub-system.

This is made easier for you when you use certain development hardware. For example, Rev 4.0 of the imp004m Breakout Board features a series of solder bridges which, when soldered, connect these pins as follows:

- W5 — BT_REG_ON to pin J
- W6 — LPO_IN to pin E
- W1 — BT_UART_TXD to pin G
- W2 — BT_UART_RXD to pin F
- W7 — BT_UART_CTS to GND

These bridges are highlighted in the photograph of the imp004m Breakout Board, below:



Working With Bluetooth

The following guide focuses on the imp004m Breakout Board, but the principles it covers apply equally to all imp004m-based devices intended to be used for Bluetooth communications.

For full details of impOS 38's Bluetooth LE API, please see the imp API documentation.

Initializing Bluetooth on the imp004m Breakout Board

In order to make use of the imp004m’s Bluetooth capability in software, you will need to initialize the Bluetooth hardware using the imp API method **open()**. This returns an instance of the imp API’s **bluetooth** class, and you will use this instance to manage all Bluetooth communications.

The **open()** method has three parameters. The first, *uart* is an imp UART object through which the imp004m’s MCU will communicate with the CYW43438’s Bluetooth unit. For the imp004m Breakout Board, your Squirrel code should pass **uartFGJH** as *uart*’s argument. The imp004m Breakout Board’s **uartFGJH** TX and RX pins are connected to the module’s BT_UART_TXD and BT_UART_RXD pins by way of the W1 and W2 bridges mentioned above.

The UART’s CTS and RTS functionality is not needed, so pin J is used as a GPIO to control the Bluetooth sub-system’s power feed (BT_REG_ON, connected to pin J on the imp004m Breakout Board via the W5 bridge). It’s then sufficient to configure pin J as a digital output set high.

For power-sensitive applications, the Bluetooth unit’s LPO_IN pin should be fed with the output from a 32.768kHz clock. This article does not cover such applications, so the examples that follow pull LPO_IN low. LPO_IN is connected to pin E on the imp004m Breakout Board via bridge W6, so we configure pin E as a digital output set low:

```
// Set up Bluetooth on the imp004m Breakout Board
// Alias the relevant UART bus, LPO and power-on pins
bt_uart <- hardware.uartFGJH;
bt_lpo_in <- hardware.pinE;
bt_reg_on <- hardware.pinJ;

// Boot up Bluetooth: ground LPO_IN and set BT_REG_ON to High
bt_lpo_in.configure(DIGITAL_OUT, 0);
bt_reg_on.configure(DIGITAL_OUT, 1);
```

This done, it’s necessary to insert a short pause before attempting to call **open()**, to give the imp004m’s Bluetooth unit time to boot:

```
// Pause while BT boots...
imp.sleep(0.1);
```

It’s a good idea while debugging to wrap your **open()** call in a `try... catch` structure to trap any errors you may encounter during development (such as omitting the pre-instantiation **imp.sleep()** call). For information, we also measure the initialization time:

```
// Add a global variable to hold the bluetooth instance
bt <- null;

local start = hardware.millis();
```

```
try {
  // Instantiate BT
  bt = hardware.bluetooth.open(bt_uart, BT_FIRMWARE);;
  server.log("BLE initialized after " + (hardware.millis() -
start) + " ms");
} catch (err) {
  server.error(err);
  server.log("BLE failed after " + (hardware.millis() - start) +
" ms");
}
```

The global variable *bt* now points to a **bluetooth** instance that is ready for use. All of the methods described in this article target this instance.

Bluetooth Firmware

The method **open()**'s second parameter is called *firmware* and takes as its argument a string or blob containing the imp004m Bluetooth unit's firmware. This is not pre-loaded into the imp004m. In the example above, the firmware is passed in as the constant *BT_FIRMWARE*, which is defined in this Squirrel file stub and can be pasted in at the top of your development code.

While Electric Imp plans to include this code in forthcoming Bluetooth library, it is currently necessary to include the firmware in your application code directly. As such, the firmware will use of 16KB of application storage space (out of 256KB). An alternative approach, which eliminates this limitation, is to load the firmware into the imp004m's external SPI flash in the space not required by impOS and thus made available for customer use. It can then be read into RAM from flash, passed into **open()** and finally flushed from RAM.

Squirrel Suspension and Bluetooth

When a device's Squirrel Virtual Machine is suspended, no new callbacks (for Bluetooth or for any other reason) can be executed until the VM is resumed. When this happens, the whole Bluetooth stack is paused so that it does not cause new events requiring Squirrel execution. This can mean that any in-progress operations will be paused, and may cause large latency spikes in Bluetooth operations — even ones which don't require Squirrel execution.

In addition, GATT read and write callbacks are called in a context where suspending is not possible. Therefore, if any operation that suspends the VM occurs within these callbacks, a Squirrel error will be raised.

With these two factors in mind, it is strongly recommended that the *RETURN_ON_ERROR* send timeout policy is selected as the first line in Squirrel that uses Bluetooth features.

Shutting Down Bluetooth

Should you need to shut down the Bluetooth sub-system, call the imp API method **close()**:

```
bt.close();
```

This call releases control of the UART and frees the memory allocated to the Bluetooth stack. Note that it is invoked implicitly if *bt* goes out of scope, which is why we establish *bt* as a global variable in the code above.

Bluetooth Operation

The Generic Access Profile (GAP)

The Bluetooth standard's Generic Access Profile (GAP) defines a number of core functions which a Bluetooth LE device can perform, primarily to advertise its presence, to look for other Bluetooth devices with which it might communicate, and then to engage in that communication.

Let's look at the first of these.

GAP Advertising

The imp API provides the method **startadvertise()** to manage the process of informing other Bluetooth devices that the imp004m can communicate with them. The GAP specification mandates a structure for the information a device will transmit to the network and this includes up to 31 bytes of advertising data. The organization of this data is beyond the scope of this article — for more information, you should consult the Bluetooth 5.0 specification and the Bluetooth Core Specification Supplement.

However you format your device's advertising data, you will need to pass it as a string or blob into **startadvertise()**'s first parameter, *advert*. You will also need to provide arguments for two further parameters: *minAdInterval* and *maxAdInterval*. These are the minimum and maximum intervals, in milliseconds, between which the imp004m will send out advertising signals. Both must be between 20ms and 10,240ms, and the maximum must be greater than the minimum, though they can be the same. A common value for both is 100ms.

There is a fourth, optional parameter called *scanResponse*, but we'll examine this later, in the section on GAP scanning.

Advertising Example: imp004m as an Apple iBeacon

The following code shows just one of many advertising applications: the use of the imp004m Breakout Board as an Apple iBeacon.

```
// Define the iBeacon advertising data
local iBeacon =
"\x02\x01\x06\x1A\xFF\x4C\x00\x02\x15\x92\x77\x83\x0A\xB2\xEB\x4
9\x0F
```

```
    \xA1\xDD\x7F\xE3\x8C\x49\x2E\xDE\x00\x01\x00\x02\xC5";

// Convert the iBeacon data into a hex string for logging
local beaconString = "";
for (local i = 0 ; i < iBeacon.len() ; i++) {
  // Write each byte as hex
  beaconString = beaconString + format("%02x", iBeacon[i]);

  // Add dashes to separate the Proximity UUID sub-sections
  if (i == 12 || i == 14 || i == 16 || i == 18) beaconString =
beaconString + "-";
}

// Log the details of the iBeacon
server.log("Advertising the following iBeacon:");
server.log("UUID: "  + beaconString.slice(18,54));
server.log("Major: " + beaconString.slice(54,58));
server.log("Minor: " + beaconString.slice(58,62));

bt.startadvertise(iBeacon, 100, 100);
```

The advertising data is structured as follows:

- Data Field
  - Size: 0x02 — Two bytes.
  - Type: 0x01 — Indicates that the following data contains Bluetooth flags.
  - Data: 0x06 — The flags define the advertising packet as BLE General Discoverable and BR/EDR high-speed incompatible, ie. only broadcasting, not connecting.
- Data Field
  - Size: 0x1A — 26 bytes.
  - Type: 0xFF — Indicates that the following data is manufacturer specific.
  - Data: 0x4C00 — Apple's Bluetooth manufacturer ID in little-endian form (ie. the value is 0x004C).
  - Data: 0x15 — The following iBeacon data size: 21 bytes.
  - Data: 0x9277830AB2EB490FA1DD7FE38C492EDE — The 16-byte iBeacon 'Proximity UUID'.
  - Data: 0x0001 — The two-byte iBeacon 'Major' value (big-endian).
  - Data: 0x0002 — The two-byte iBeacon 'Minor' value (big-endian).
  - Data: 0xC5 — A reference RSSI value: the power in dBM at one meter.

Typically, all the iBeacons at a given location, or used by the same organization, will use the same Proximity UUID, with the major and minor values allowing the beacons to be subdivided by use-case or zone. For example:

- Proximity UUID — Indicates iBeacons at a certain location.
  - Major — Indicates iBeacons on a given floor of that location.
    - Minor — Indicates an iBeacon at a particular place on that floor.

You can use a mobile app like Radius Neworks' Locate (iTunes/Google Play) to detect an imp004m operating as an iBeacon using the above code. You will need to enter the proximity UUID, and major and minor values into the app, which only reports beacons it has been set up to scan for.

To cease broadcasting the advert, just call **stopadvertise()**.

GAP Scanning

While an imp004m is advertising is presence, it may also scan for other Bluetooth devices in the vicinity. Such scans are initiated by calling **startscan()**, but you will typically want to refine the scan parameters before making that call. The imp API provides two further methods to help you: **setscanparams()** and **setscanfilter()**.

The first of these establishes the basic parameters of the scan: will it be active or passive, how long will it pause between scans, and will it scan continuously or at some other duty cycle. Each of these scan parameters form parameters for the method: *active*, *interval* and *window*. All are optional, as is **setscanparams()** itself.

The latter two parameters take integers between 3ms and 10,240ms; *window* specifically takes a time value derived from the duty cycle percentage. For example, if *interval* is 200ms and you require a 50 per cent duty cycle, then *window* needs to be passed 100ms.

The *active* parameter takes `true` or `false`, the latter indicating the scan should be passive. Active scans involve issuing a scan request when an appropriate advertisement is detected. This request is not made during passive scanning, ie. the imp004m does not reveal that it has detected the advertiser.

In response to the scan request, the advertiser will transmit a scan response. The imp0004m will do this if you have made use of **startadvertise()**'s fourth parameter, *scanResponse*. This takes up to 31 bytes of string or blob data. Again, like the **startadvertise()** *advert* parameter's argument, it is up to you to format your *scanResponse* correctly — the imp API cannot do this for you because the data is so highly application specific.

Filtering Scans

The second scan configuration method, **setscanfilter()**, allows you to define rules against which all scan results are matched. Only those which are allowed by your rules will be issued to your application by way of the callback function you register using **startscan()**.

You provide **setscanfilter()** with an array of tables. Each table is a rule configured by its keys and their values. The **setscanfilter()** documentation provides a full list of these, and they allow you to filter by criteria such as Bluetooth address, the received signal strength, the type of advertisement being detected, and/or byte sequences within the advertisement data.

For example, the following code uses **setscanfilter()** to ignore all advertisements but the iBeacon mentioned earlier:

```
server.log("Scanning...");

// Set scan parameters for passive scanning
bt.setscanparams(false, 100, 100);

// Filter out iBeacons,
// ie. advertisements of 'type' 3 (Non-connectable undirected advertisements),
// and whose 'data' contains the iBeacon preamble bytes
bt.setscanfilter([{ "type" : 3,
                    "data" :
"\x02\x01\x1A\x1A\xFF\x4C\x00\x02\x15" }]);
```

With the filter prepared, the code can call **startscan()**. This method has a single parameter, *callback*, into which you pass a function that will be called whenever the imp004m detects an advertisement that matches any rules you have provided. The callback has a parameter of its own, called *adverts*, into which an array of detected advertisements are passed. Each is a table containing a number of keys as listed in the **startscan()** documentation.

Example: Scanning for iBeacons

With the scan set up as above, the **startscan()** might look something like this:

```
beacons <- [];

bt.startscan(function(adverts) {
  foreach (advert in adverts) {
    // Convert the advert's data payload to hex string
    local payload = "";
    for (local i = 0 ; i < advert.data.len() ; i++) {
      payload = payload + format("%02x", advert.data[i]);
    }

    // This is a beacon so record it
    local beacon = {};
    beacon.uuid <- payload.slice(18, 50);
    beacon.majorString <- payload.slice(50, 54);
    beacon.minorString <- payload.slice(54, 58);

    if (beacons.len() == 0) {
      beacons.append(beacon);
    } else {
      local got = false;
```

```
      foreach (aBeacon in beacons) {
        if (aBeacon.uuid == beacon.uuid
            && aBeacon.majorString == beacon.majorString
            && aBeacon.minorString == beacon.minorString) {
          got = true;
          break;
        }
      }

      if (!got) beacons.append(beacon);
    }
  }
});
```

This code can be easily expanded to log beacon UUID, major and minor values, and (using the advert table's *rssi* key) and approximate distance. This replicates the functionality of the Radius Networks Locate app mentioned in the previous example, which you can do if you have at least two imp004m Breakout Boards.

Scans can be suspended by calling **stopscan()**.

Device-to-Device Communications

Making Connections

The Bluetooth GAP specification also covers the establishment of one-to-one connections between nearby Bluetooth devices. Typically, one device will issue a connection request to another. If the imp004m is to respond to such a request — it currently can only respond to requests, not initiate them — you must register a function that will be called when a connection request is received. You do this by calling the imp API method **onconnect()**.

The supplied function needs a parameter of its own into which a new **btconnection** instance is passed. This **btconnection** instance provides a means for you to manage the connection now established between the two devices. **btconnection** has the following methods:

- **address()** — Returns the Bluetooth IP address of the remote device as a 12-character hexadecimal string
- **onclose()** — Registers a function that will be called when the connection is closed by any means outside of the application's (or impOS') control, eg. the remote device shuts down or goes out of range.
- **close()** — The application's means to close the connection

Closed **btconnection** instances, or those that go out of scope, cannot be used or re-used. If this happens while the devices are connected and the remote device subsequently breaks the connection, your application will not be notified (via **onclose()**).

If you want the imp004m to stop handling connection attempts, call **onconnect()** again, but pass in `null` in place of the handler function.

The Bluetooth Generic Attribute Profile (GATT)

How do two Bluetooth-enabled devices exchange information? To do so, both devices make use of the standard's Generic Attribute Profile, or GATT. This profile defines a client-server relationship that can be established between two devices.

The client typically sends a request to the GATT server and is able to read data (an 'attribute') from the server and write values back to that attribute. The server makes its attributes available to the client when the client connects to it. Clients can also operate as servers, and vice versa.

The imp004m can become a GATT server by calling the imp API method **servergatt()** and passing in an array of tables. Each table defines a 'service' provided by the server: a UUID to identify the service and a subsidiary array of tables, each of which defines a 'characteristic' of that service, ie. a data point. A full list of the keys that may be included in the definition of a characteristic can be found in the **servergatt()** documentation.

Many characteristics are predefined by the Bluetooth standards, but you can make use of your own by supplying a non-reserved UUID.

When configured using the imp API, each characteristic can include references to getter (*read*) and setter (*write*) functions. These functions must have at least one parameter: a **btconnection** object which provides information about the connected device. The read function will return the value of the characteristic; the write function has a second parameter, a string or blob value from which it updates the value of the characteristic. The functions can return zero, or no value at all to indicate success; returning a non-zero value indicates failure — the value is typically an application-specific error code.

Note If you plan to advertise the services offered by the GATT server by including service UUIDs in the BLE advertisement payload *(see below)*, you should ensure your code calls **servegatt()** before it calls **startadvertise()**.

GATT Security

impOS 37.15 adds support for GATT connection security. This is enabled by default, but can be disabled by calling `bt.setsecurity(1);`, where *bt* is your Bluetooth instance. This insecure mode is used in the following example — we will be adding guidance on using secure connections shortly.

Example: Refreshing an imp004m's WiFi Credentials

Using what we have learned so far, we can build an application that is able to receive new WiFi credentials via Bluetooth and, if requested, apply them.

The first thing we do is establish a data structure to hold the imp004m's WiFi credentials and manage updates, *data*, and a record of the connection *connection*:

```
connection <- null;

data <- {};
data.ssid <- "";
data.pwd <- "";
data.updated <- false;
data.update <- function() {
  local bs = "############################".slice(0,
this.pwd.len());
  server.log("Switching to SSID: " + this.ssid + ", PSK: " +
bs);
  imp.setwificonfiguration(this.ssid, this.pwd);
  if (incoming != null) incoming.close();
  imp.onidle(function() {
    server.disconnect();
    server.restart();
  });
};
```

The next step is to set up the service that the imp004m will provide. This has three characteristics: one each for writing the network name (SSID) and the password to *data*, and a third to trigger the application of those credentials:

```
server.log("Setting up services...");

local service = {};
// WiFi Credential Refresh Service
service.uuid <- "FADA47BEC45548C9A5F2AF7CF368D719";
service.chars <- [];

local chara = {};
// Set SSID characteristic
chara.uuid <- "5EBA195632D347C681A6A7E59F18DAC0";
chara.write <- function(conn, v) {
  data.ssid = v.tostring();
  server.log("Written SSID");
  data.updated = true;
};
service.chars.append(chara);

// Set PWD characteristic
chara = {};
chara.uuid <- "ED694AB947564528AA3A799A4FD11117";
chara.write <- function(conn, v) {
```

```
  data.pwd = v.tostring();
  server.log("Written PWD");
  data.updated = true;
};
service.chars.append(chara);

// Set dummy characteristic
chara = {};
chara.uuid <- "F299C3428A8A4544AC4208C841737B1B";
chara.write <- function(conn, v) {
  if (data.updated) data.update();
};
service.chars.append(chara);

bt.setsecurity(1);
bt.servegatt([service]);
server.log("Serving GATT...");
```

The code sets UUIDs for the service and each of the characteristics, for which it also provides write functions. The first two update *data*'s *ssid* and *pwd* properties, and record the fact by setting its *updated* flag. This is checked in the the third characteristic's write function. This doesn't actually write any data — any value passed in from the connected Bluetooth device is ignored — but uses this as a proxy to trigger the application of the new credentials — if there are any; this is why *data*'s *updated* flag is checked. Finally, the service is placed in an array and passed into **servergatt()**.

The characteristics defined above can't be read or written until a remote device connects to the imp004m. We make this possible by now adding the following code:

```
bt.onconnect(function(conn) {
  connection = conn;
  server.log(conn.address() + " connected");

  conn.onclose(function() {
    server.log(connection.address() + " disconnected");
  });
});

server.log("Awaiting connections...");
```

Here we define a function that will be called on an attempt to connect. The connection is referenced by the global variable (set earlier) *connection* to keep it in scope and available for future use: in the function registered with the **onclose()** method. Finally, we need to signal to other devices that the im004m can be contacted, and for this we use the GAP advertising process described earlier in this guide:

```
bt.startadvertise("\x11\x07\x19\xD7\x68\xF3\x7C\xAF\xF2\xA5\xC9\x48\x55\xC4\xBE\x47\xDA\xFA", 100, 100);
```

```
server.log("Advertising services...");
```

What does the first of these lines do? As we saw earlier, it sets the advertising data and the minimum and maximum advertising intervals — the latter two are both set to 100ms.

The advertising data makes use of the Bluetooth specification as follows: the first byte is the data size (17 bytes); it's followed by a data-type indicator: 0x07 indicates that what follows is a complete list of 128-bit service UUIDs provided by the device. You can find a list of pre-defined advertising data types here. After this come the UUIDs themselves. In this case, there is only one, the service UUID we set in the code at the start of this example. You should note that though the imp API takes UUIDs in little-endian form, the specification requires them to be transmitted in big-endian form. This is why the UUID bytes are reversed in the advertising data.

With this code in place, it's possible to write a mobile app which can scan for Bluetooth devices offering a service with the UUID set above — this is made possible by the advertising signal. Detecting the device may then result in further discovery operations as the app interrogates the imp004m for the characteristics of that service. The app may then connect to the imp004m and this allows it to write data to the module's GATT server and, as we've seen, trigger the device to apply new WiFi credentials, disconnect from the network, restart and reconnect, this time to the new network.

Monitoring Your imp004m's GATT Server

You can make use of Punch Through's mobile app LightBlue Explorer (on iOS and Android) to scan for your imp004m and its GATT services. For example, you might set up the imp004m to serve standard Bluetooth LE device information:

```
// Device information service
local dis = { "uuid": 0x180A,
              "chars": [
                { "uuid": 0x2A29, "value": "Electric Imp" },
// Manufacturer name
                { "uuid": 0x2A25, "value":
hardware.getdeviceid() },    // Serial number
                { "uuid": 0x2A24, "value": imp.info().type },
// Model number
                { "uuid": 0x2A26, "value":
imp.getsoftwareversion() }] // Firmware version
};

// Offer the BlinkUp service
bt.servegatt([dis]);
```

This will be detected and presented by LightBlue Explorer:

Mobile OS Bluetooth Attribute Caching

By default, iOS and Android cache the attribute information they discover about devices. This is done to ensures that future discovery requests need not use the radio, conserving power. This makes sense because Bluetooth peripherals generally do not change the services they offer, or the characteristics of those services. However, it also means that if you change your Squirrel app's served attributes during development, the changes will not be detected by the app. For example, if you replace the service listed above with a different one, apps like LightBlue will continue to show

the Device Information details. This is because the host OS is providing that information from its cache.

The easiest approach to dealing with this is to disable then re-enable Bluetooth on your mobile device. You may also need to power-cycle the device. Both actions cause the device's Bluetooth peripheral cache to be cleared, forcing the OS to go to the device for service information.

Security

impOS allows you to set the minimum connection security level the imp will then require for all GAP connections made to its current **bluetooth** instance. You make your choice with the imp API method **setsecurity()**. Using the scheme used in the Bluetooth 4.2 standard, it provides three security levels:

securityLevel	**Standard Name**	**imp IO Capabilities**
1	LE Security Mode 1 Level 1: No security	*NoInputNoOutput*
3	LE Security Mode 1 Level 3: Authenticated pairing with encryption	*KeyboardOnly*
4	LE Security Mode 1 Level 4: Authenticated LE Secure Connections pairing with encryption	*KeyboardOnly*

Note Other values are illegal and will cause a Squirrel error.

Security levels 3 and 4 require a six-digit pairing code which is passed into **bluetooth.setsecurity()** as its second argument. Security level 4 causes any connection which only achieves level 3 or lower to be automatically closed after pairing. Squirrel will not be notified when this happens.

The default setting is level 4. A random pairing code is generated when **bluetooth.open()** is called. This code can be retrieved using **bluetooth.getsecuritycode()**. It will be changed by calling **bluetooth.setsecurity()**.

The imp initiates the pairing procedure when a new connection is made, and will not answer GATT queries until pairing completes successfully. And only then will the Squirrel GAP connection callback registered using **bluetooth.onconnect()** be called.

Choosing Pairing Code Values

Bluetooth GAP connections are authenticated by pairing code. Unfortunately, this method can leak one new bit of the code to an active adversary on each run of the protocol. Because the code is

fixed at six decimal digits, only 20 failed runs are needed to recover the whole code. For this reason, you should choose your code carefully.

There are four possible levels of pairing code security:

Mode	Lifetime	Scope	Security Strength	Notes
1	Forever	Global	None	Effectively opts out of security
2	Forever	Per-imp	Limited	Allows the pairing code to be etched on the product or printed on the packaging
3	Squirrel Bluetooth session	Squirrel Bluetooth session	Good	This is the default
4	Time-limited	Single connection	Strong	

For mode 2, you generate a random pairing code and both store it in the device under test (DUT) and either relay it to the label printing or case-etching station so the end-user can read it and enter it into the mobile app that is being used to activate the device, or store it in a database from which it can be retrieved by the mobile app.

For mode 3, impOS chooses a random pairing code when a **bluetooth** object is instantiated. This is the default behavior and provides good security, but requires your product to incorporate a means of relaying the code to the end-user, such as a display.

Mode 4 provides the best security. The imp chooses a new random pairing code after every successful connection *and* every time a fixed period of time has elapsed. This period must be long enough to allow pairing to take place, but short enough to limit exposure, ie. minutes rather than seconds (too short) or hours (too long). Again, your product must incorporate a means of relaying the latest code to the end-user.