INTEGRATING WITH IOS

HANDS-ON LAB

Tuesday, 11 June 2013

Version 1.0

TABLE OF CONTENTS

1 Introduction

The purpose of this lab is to give you a high-level overview of how a BLE peripheral devices works with iOS. You will implement a basic Heart Rate sensor using the DKBLE112 development kit and test it (if possible) with the **BLE Demo** Xcode project provided by Bluegiga, exploring the relevant portions of that code that allow it to work with a heart rate sensor.

For testing purposes, you should ideally have access to an Apple device with Bluetooth Low Energy support. This means an iPhone 4S or newer, an iPad 3rd generation or newer, or an iPad Mini. Obtaining the BLE Demo project source code for reference is also necessary.

2 Creating the Heart Rate Sensor

In this section, you will create a very basic heart rate sensor BLE projectt. This project will run as a standalone on-module BGScript application, requiring no external host for module control.

2.1 Main project definition (**project.xml**)

The Heart Rate project implementation is started by first creating a project file (**project.xml**), which defines the resources use by the project and the firmware output file. This project file includes a description of each of the main elements of the project. Using Notepad++ or another suitable text editor, create a file called “project.xml” and give it the following contents:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
    <gatt in="gatt.xml" />
    <hardware in="hardware.xml" />
    <script in="heartrate.bgs" />
    <image out="out.hex" />
</project>
```

Figure 1: Project file for the Heart Rate sensor (project.xml)

NOTE: You can also give this file a ”.bgproj” extension, which will associate it with the BLE Update application if you have it installed. This will allow you to simply double-click on the project file from within Windows Explorer after you have finished, and it will automatically be opened in BLE Update, ready for flashing.

The main explanation of each of these tags is as follows:

- **<gatt>** Defines the XML file containing the GATT database
- **<hardware>** Defines the XML file containing the hardware configuration
- **<script>** Defines the BGScript file which contains the BGScript code
- **<image>** Defines the output HEX file containing the firmware image

Once you have added this content to **project.xml** (or **project.bgproj**) and saved the file, continue.

2.2 Hardware configuration (**hardware.xml**)

The **hardware.xml** file contains the hardware configuration for the BLE112 device. It describes which interfaces and functions are used and what their specific properties are. For our Heart Rate device, we need only very basic functionality on a hardware level.

```
<?xml version="1.0" encoding="UTF-8" ?>
<hardware>
    <sleeposc enable="true" ppm="30" />
    <usb enable="false" />
    <txpower power="15" bias="5" />
    <script enable="true" />
    <slow clock enable="true" />
    <pmux regulator_pin="7" />
</hardware>
```

Figure 2: Hardware configuration for Heart Rate sensor (hardware.xml)

- **<sleeposc>** The 32.768KHz sleep oscillator is enabled. Sleep oscillator allows the device to enter power mode 1 or 2 between *Bluetooth* operations, for example between connection intervals. This should **always** be used for BLE112 projects.
- **<usb>** USB interface is disabled to save power and allow the device to go to low-power modes. If USB is enabled, no low-power modes will be used. The USB port is enabled by default if this tag is omitted.
- **<txpower>** TX power is set to +3dBm value. Every step represents roughly a 2dBm change and the range of the parameter is 15 to 0, corresponding TX power values from +3dBm to -24dBm.
- **<script>** Scripting is enabled since the Heart Rate application is implemented with the BGScript scripting language.
- **<slow_clock>** The “slow clock” option is enabled to reduce the CPU core speed as much as possible when full speed is not required for BLE radio operations. This can save additional power, and is recommended for project which are trying to use the absolute minimum possible power. However, some timing-sensitive operations such as PWM and UART transmissions may stop working reliably if this option is enabled.
- **<pmux>** Enables automatic management of the DC/DC converter on the DKBLE112. This prevents momentarily large current draws from the CR2032 battery during transmissions, if battery power is used, and will extend the life of the battery.

Once you have added this content to **hardware.xml** and saved the file, continue.

2.3 GATT database definition (**gatt.xml**)

This section describes how to define the bare minimum Heart Rate profile's services. This is where we define the data structure and which kinds of operations are allowed on each attribute in the database. It is important to understand the structure of a GATT database and what is meant by "profile," "service," and "characteristic" (also referred to as an "attribute"). Here is a quick summary:

- A **profile** is a collection of one or more **services**
- A **service** is a collection of one or more specifically defined **characteristics**
- A **characteristic** is a single value with specific access rules (read/write/notify/indicate/authentication)

Note that while a **profile** is a logical collection of services and does not itself have a unique identifier, each service and each characteristic requires a universally unique identifier (UUID). The Bluetooth SIG has designated different UUID classes for officially adopted services and characteristics (16 bits in length) vs. custom/proprietary services and characteristics (128 bits in length).

The Heart Rate profile is an officially adopted profile made up of official services and characteristics, so all UUIDs we will use here are 16-bit values. The minimal GATT service/characteristic hierarchy looks like this:

a. **Generic Access Profile** service (UUID 0x1800)
    a. **Device Name** characteristic (UUID 0x2A00) – friendly name used of the device (read only)
    b. **Appearance** characteristic (UUID 0x2A01) – type of device (read only)
b. **Heart Rate** service (UUID 0x180D)
    a. **Measurement** characteristic – heart rate measurement value (notify only)

Represented visually, the minimal profile looks like this:

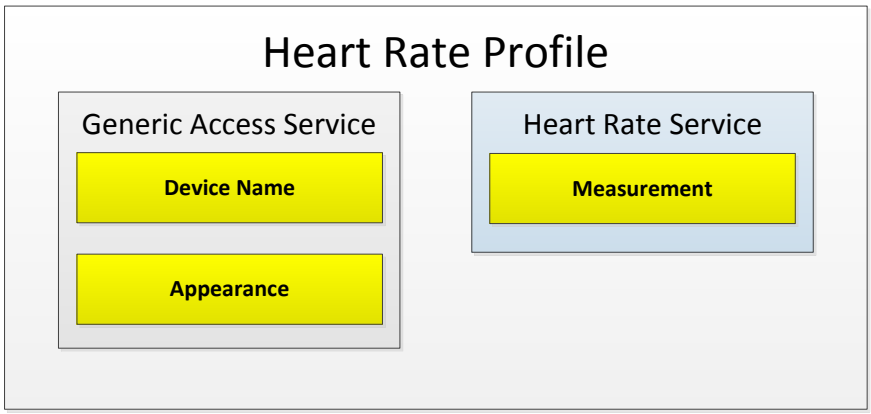

Figure 3: Visual GATT database structure

Every *Bluetooth* low energy device needs to implement a GAP service. The GAP service here is very simple and consists of only two characteristics. The Heart Rate service is even simpler, with only a single two-byte characteristic. This characteristic is set up to allow notifications (for pushing the heart rate measurement data to the remote client without acknowledgement). The measurement itself is only two bytes of data: one "flags" byte, and one "rate" byte which is the actual beats-per-minute (BPM) value.

Add the following content to your **gatt.xml** file before continuing.

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <service uuid="1800">
        <description>Generic Access Profile</description>

        <characteristic uuid="2a00">
            <properties read="true" const="true" />
            <value>Heart Rate Demo</value>
        </characteristic>

        <characteristic uuid="2a01">
            <properties read="true" const="true" />
            <value type="hex">4003</value>
        </characteristic>
    </service>

    <service uuid="180d" advertise="true">
        <description>Heart Rate Demo</description>
        <characteristic uuid="2a37" id="c_heartrate_measurement">
            <properties notify="true" />
            <value length="2" />
        </characteristic>
    </service>
</configuration>
```

Figure 4: Complete Heart Rate GATT database structure

IMPORTANT: If possible, assign a unique name to your device instead of simply using “Heart Rate Demo” here. This will make it easier to identify if there are many other heart rate sensor BLE devices nearby.

Note also in the XML definition above the use of the **id** attribute on <characteristic> tags. These named identifiers allow for programmatic access to characteristics from within your BGScript source file, rather than needing to use a numeric handle. The numeric handle is generated by the BGBuild compiler during the build process, and a simple text file called **attributes.txt** is created which contains a set of name/number pairs on each line. You can refer to this file if you need to know the numeric handle of any characteristic for which you have assigned a named identifier.

The **id** attribute on <service> tags is used only for including service definitions within other services, which we will not do in this exercise. Services cannot be directly referenced or modified from within BGScript, since all data operations are technically done on characteristics only, rather than services.

In both cases, the **id** value is completely arbitrary, but it is a good idea to follow a helpful naming convention. I tend to split words with underscores, and to begin service IDs with “s_” and characteristic IDs with “c_”. The only requirement is that they are alphanumeric (underscores allowed) and that they do not overlap with BGScript keywords.

Note: it is tempting sometimes to hard-code the numeric handle of your attribute(s) into your iOS application, because it is easy and eliminates the need to perform a service/descriptor discovery within iOS. However, this is not a good idea because it means any changes you make to your device in the future may modify the GATT structure and invalidate the previous handle values, requiring an app update of some kind. It is far better to design your app so that it knows what to look for by UUID rather than by handle.

2.4 Application Configuration (**config.xml**)

The **config.xml** file contains the extended application configuration for BLE112 device. This file is not necessary for this project since all of the default values are acceptable. Options which can be controlled in this file include UART optimization, script timeout (useful if you use “while” loops), and multiple connection support (only useful on the master end of things, which this is not).

2.5 BGScript for Heart Rate Project (**heartrate.bgs**)

Our Heart Rate project implements a standalone BLE peripheral device where no external host processor is needed. The application logic is created as a BGScript source file and the code is explained in this section.

BGScript uses an event-based programming approach. The script is executed when an event takes place, and the programmer may register listeners for various events. The Heart Rate project uses a few different event listeners, but will require only one user-declared variable. Let's review how the whole project will be implemented at a high level:

2.5.1 Functional overview

The basic Heart Rate project will work in the following way:

- Advertising will start automatically on boot. Upon disconnection, advertising will resume again.
- The heart rate value will be emulated by reading the P0_6 ADC pin (connected to the potentiometer on the DKBLE112 board) and scaling it to a semi-believable heart rate range, 20-224.
- The emulated heart rate measurement value will be pushed out via notifications once per second using a soft timer on the BLE112.

Now that we have an idea of the functionality, we can move on to the code.

2.5.2 Declaring variables

The Heart Rate project uses only one user-defined variable for building the measurement value immediately before writing it to the appropriate attribute. Add the following content to the top of your BGScript source file:

```
dim measurement(2)  # buffer for building measurement attribute value
```

2.5.3 System: Boot event (**system_boot**)

When the system is started or reset, a **system_boot** event is generated. This event listener is typically the entry point for all the BGScript applications, and provides a perfect opportunity for initializing any required variables or other behavior

In this lab, the following tasks take place in the **system_boot** event handler:

- Begin advertising
- Start the 1-second repeating timer

Add this event handler code to your script:

```
event system_boot(major, minor, patch, build, ll_version, protocol, hw)
    # start advertising in connectable mode
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)

    # start repeating 1-second timer
    call hardware_set_soft_timer(32768, 0, 0)
end
```

2.5.4 *Bluetooth*: Disconnection event (**connection_disconnected**).

If the *Bluetooth* connection is a lost, a disconnection event occurs. Because a Bluetooth *Smart* device will not automatically resume advertising when a connection is lost and instead will be put into an **idle** state by default, we have to intentionally restart advertising upon disconnection..

Add this event handler code to your script:

```
event connection_disconnected(handle, result)
    # start advertising again after disconnection
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)
end
```

2.5.5 Hardware: Soft Timer "Tick" (**hardware_soft_timer**)

The timer "tick" event is used to trigger a new ADC reading. Remember that we originally set up the timer by use of the following line back in the **system_boot** event handler:

```
call hardware_set_soft_timer(32768, 0, 0)
```

The first parameter specifies the interval—it is "x" in the value "x/32768", since 32768 is the crystal frequency. Using "32768" here gives us exactly 1 second. The second parameter is the timer handle (currently only one simultaneous timer is supported at a time, so this should always be 0). Third is the setting which controls whether it is a one-shot timer or continuous. In this case, we use "0" to make it continuous.

In the event handler here, we are triggering a new ADC read (conversion) operation. Once complete, the ADC result will be sent back to us in a new event (below). This might seem like extra work—after all, why can't we just read the ADC and store the result as a return value to our original "call" command? It turns out that the ADC is not instantaneous. At the highest **decimation** setting—the most precise measurement—it takes over 100 uSec to finish a single reading. That's not much time for humans, but it is a great deal of time for a microcontroller. Therefore, it must be handled via an event instead of a return value.

Add this event handler code to your script:

```
event hardware_soft_timer(handle)
    # initiate potentiometer reading on P0_6
    call hardware_adc_read(6, 1, 2)
end
```

2.5.6 Hardware: ADC conversion complete (**hardware_adc_result**)

This event comes back with the read ADC value ready for processing. Here, we simply scale it to a believable range (20-224) and write it directly to the **c_heartrate_measurement** attribute. If the client has enabled notifications on that attribute, then this action will push the new measurement to the client. Otherwise, it will simply be written locally once per second.

Add this event handler code to your script:

```
event hardware_adc_result(input, value)
    # build simple characteristic value response (flags byte = 0x02)
    measurement(0:1) = 2

    # scale the reading from heart rate value 20-224
    measurement(1:1) = value / 160 + 20

    # write value to measurement attribute
    call attributes_write(c_heartrate_measurement, 0, 2, measurement(0:2))
end
```

3 Compiling and flashing the Heart Rate project

You should now have a total of four files in your project folder:

a. project.xml (or project.bgproj)

b. hardware.xml

c. gatt.xml

d. heartrate.bgs

It is now ready to compile and flash onto the development kit.

3.1.1 Connect the CC Debugger

In order to flash a new firmware image onto the device, the CC debugger must be connected. This is a pretty straightforward process:

a. Connect a mini USB cable between the CC debugger and your PC

b. Connect the small 0.1” to 0.05” adapter board to the CC debugger

c. Connect the small 10-pin 0.05” ribbon cable between the CC debugger and the DKBLE112, making sure that both rows of pins are aligned properly in the holes and that the red wire (pin 1) is on the far right edge of the header on the dev kit, not on the side closest to the USB connector.

d. With the DKBLE112 powered and the CC debugger connected, press the small black button on the CC debugger and verify that the debugger’s LED is **green**, not red. This indiates a proper connection. Note that you may need to make extra sure that the 0.1” to 0.05” adapter board is seated properly, as it has been known to make intermittent connections.

e. The CC debugger may not enumerate properly without the correct drivers. If necessary, you can download the drivers directly from TI, or download and install the TI SmartRF Flash Programmer utility which includes drivers for the CC debugger.

3.1.2 Compile and flash the project with BLE Update

The following steps will build and flash the modified firmware onto the module, assuming the CC debugger is properly connected as described above:

a. Run the BLE Update utility

b. Make sure the CC debugger device is selected from the Port dropdown

c. Click the ”Info” button in BLE Update to verify that the debugger can communicate with the module

d. Click the ”Browse” button and locate the ”project.xml” file (or ”project.bgproj”) file your project folder

e. Click the ”BGBuild” menu and verify that an SDK is selected

- If not, click the ”Select manually...” item and browse to the **/bin/bgbuild.exe** file from the extracted SDK archive

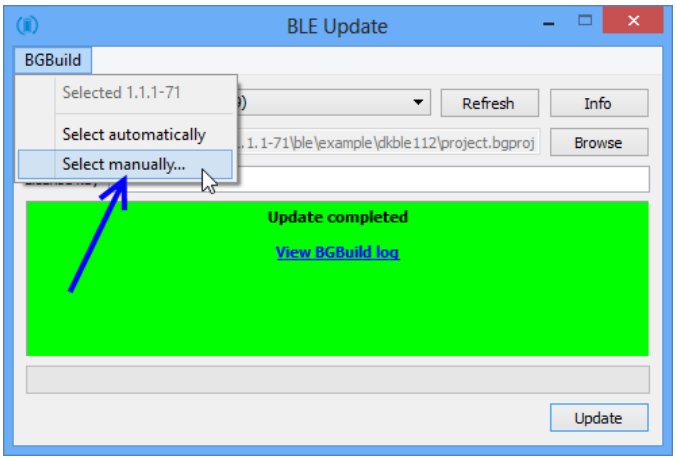

Figure 5: Manually selecting a BGBuild executable

f. Click the ”Update” button to compile and flash the project onto the module

You should now have a very basic heart rate project running on the development kit, and can proceed on to testing with the BLEGUI application and a BLED112 dongle.

4 Testing the Heart Rate sensor with iOS and BLE Demo

This section assumes you have already downloaded and extracted the BLE Demo Xcode project archive and are familiar with Xcode in general. If you do not have it the project archive yet, please download and extract it. If you are not familiar with Xcode, there are a number of good resources available from Apple to help get you started.

Note: this section requires an iOS device and a Mac running Xcode, or else an iOS device which already has the BLE Demo app installed on it for testing.

4.1 Install BLE Demo on your iOS device

To install the BLE Demo project onto your iOS device:

a. Extract the project archive to a known location on your Mac

b. Open the project in Xcode

c. Ensure that the proper developer signing configuration is in place to allow you to build and deploy 3rd party projects on your device

d. Build and run the BLE Demo project

4.2 Test BLE Demo on your iOS device

IMPORTANT: iOS caches the GATT structure of BLE devices which it has connected to in the recent past. If you have reflashed different firmware onto the DKBLE112 board for testing as of earlier today and have already connected to it during that time with your iOS device for testing, then you **will** need to clear the device cache before this process will work correctly. To clear the cache, you can do one of two things:

a. Go to the **Settings** app and into the **Bluetooth** area, and power-cycle the Bluetooth subsystem.

b. Power-cycle the entire device

Once you are reasonably sure the GATT cache will not be a problem, perform the following steps to test the device:

a. Tap the BLE Demo app icon to begin

b. Move the ”Scanner” slider control to the ”ON” position

c. Make sure your BLE peripheral device is powered on

d. Identify your device in the list (it should have a rapidly changing RSSI shown)

e. Tap on your device’s name to connect

f. Tap on the ”HeartRate” service name in the ”Services Found” screen that appears

g. Make sure the ”POTENTIOMETER” switch is set to ”ON” on the DKBLE112 board

h. Rotate the potentiometer as desired to change the reported heart rate measurement

If everything went smoothly, you connected to the BLE device from iOS and controlled the measured reading via a hardware component on the board. Congratulations!

5 Understanding iOS BLE Limitations

All Bluetooth Low Energy apps in iOS currently make use of the CoreBluetooth APIs. These are an entirely different set of API components than what is/was used for classic Bluetooth connections (ExternalAccessory or GameKit). CoreBluetooth is built for BLE usage and contains its own set of relevant objects and methods. One of the best overviews of the object structure of CoreBluetooth from both the Central side (typically the iOS device) and the Peripheral side (typically the BLE112 side) can be found here:

- http://weblog.invasivecode.com/post/39707371281/core-bluetooth-for-ios-6-core-bluetooth-was

The BLE Demo app code, like most other iOS apps which use BLE, follows a straightforward but particular set of operations and restrictions as documented in Apple's Bluetooth Design Guidelines:

- https://developer.apple.com/hardwaredrivers/BluetoothDesignGuidelines.pdf

The Bluetooth Low Energy portion starts currently on page 17 of the above document.

The BLE protocol as defined by the Bluetooth SIG allows for some very tight timing and potentially fast throughput (mathematically speaking, up to 270 kbit/sec under perfectly optimal conditions). Practically speaking due to hardware and RF environment limitations, even 60 kbit/sec under good conditions is about as fast as you can hope to get between two BLE devices. This is because the minimum connection interval (time between coordinating BLE packets) is 7.5ms according to the spec, and the BLE timeslots are 1.25ms meaning you can push through up to 6 unacknowledged 20-byte data packets each interval.

However, iOS imposes its own restrictions due to the following main reasons:

- The BLE radio must be shared among multiple apps, so one is not allowed to dominate
- The BLE radio can operate simultaneously as both a central and peripheral device

As a result, the following settings/restrictions are in effect:

- The connection interval may not be set below 20 ms
- The peripheral may not send more than 4 unacknowledged notification packets per interval
- All background scanning for peripherals must use a relatively long scan interval/window
- All background scanning for peripherals must include a filtering service UUID (no catch-all scans)

It is important to keep these things in mind when considering whether BLE is a good fit for a given application. BLE peripherals do not require the Apple authentication coprocessor, which makes it an enticing option for many developers; however, despite this it is not always the best fit, particularly if your application requires high throughput. The peak throughput possible with iOS is approximately 30 kbit/sec due to the above restrictions.

6 Contact Information

Sales: sales@bluegiga.com

Technical support: support@bluegiga.com
http://techforum.bluegiga.com

Orders: orders@bluegiga.com

WWW: www.bluegiga.com
www.bluegiga.hk

Head Office / Finland:

Phone: +358-9-4355 060
Fax: +358-9-4355 0660
Sinikalliontie 5A
02630 ESPOO
FINLAND

Postal address / Finland:

P.O. BOX 120
02631 ESPOO
FINLAND

Sales Office / USA:

Phone: +1 770 291 2181
Fax: +1 770 291 2183
Bluegiga Technologies, Inc.
3235 Satellite Boulevard, Building 400, Suite 300
Duluth, GA, 30096, USA

Sales Office / Hong-Kong:

Phone: +852 3182 7321
Fax: +852 3972 5777
Bluegiga Technologies, Inc.
19/F Silver Fortune Plaza, 1 Wellington Street,
Central Hong Kong