

Hands-On Lab

Creating a Bluetooth Smart device using Arduino

Lab version: 2.0.4

Last updated: 02/02/2015

Contents

REVISION HISTORY	3
OVERVIEW	4
Goal	4
Equipment Requirements	5
Exercises Overview	5
EXERCISE 1: SETTING UP THE ARDUINO SOFTWARE AND HARDWARE	6
Software	6
Task 1 (OS X) — Installing the Arduino software	6
Task 1 (Windows) – Installing the Arduino IDE software	6
Task 2 (Windows) — Installing the Arduino drivers	6
Hardware	9
EXERCISE 2: YOUR FIRST ARDUINO SKETCH	11
Task 1 — Load the example sketch	11
EXERCISE 3: GENERATE A GATT PROFILE.....	14
Task 1 – Identifying your chip & installing the Bluetooth library	15
Task 2 – Generating a profile	15
Task 3 – Implementing the key fob behaviours	24
A) Link Loss Alert Level change	25
B) Immediate Alert	26
C) Link Loss	28
D) Proximity Monitoring	29
E) Time Monitoring	31
EXERCISE 4: OPTIONAL – ADD AN LCD DISPLAY	36

Revision History

Version	Date	Author	Changes
1.0.0	1 st May 2013	Matchbox	Initial version
2.0.0	1 st September 2014	Martin Woolley, Bluetooth SIG	Replaced Button Click custom service with the Proximity Monitoring Service. Introduced the Time Monitoring Service. Described implementation of link loss, link loss alert level change and immediate alert use cases and included Arduino code.
2.0.4	2 nd February 2015	Martin Woolley, Bluetooth SIG	Fixed some minor issues with the contents of this document; unpackaging instructions incorrectly assumed the Arduino BLE library would be in a zip file, which it is not. The circuit diagram for the suggested circuit had an error.

Overview

Bluetooth® Smart is the intelligent, power-friendly version of Bluetooth wireless technology. While the power-efficiency of Bluetooth Smart makes it perfect for devices needing to run off a tiny battery for long periods, the magic of Bluetooth Smart is its ability to work with an application on the smartphone or tablet you already own. Bluetooth Smart makes it easy for developers and OEMs to create solutions that will work with the billions of Bluetooth enabled products already in the market today.

Bluetooth Smart is an application-friendly technology supported by every major operating system. The technology is low-cost and offers a flexible mechanism for connecting everyday objects like heart-rate monitors, toothbrushes, and shoes with applications on Android, iOS and Windows.

This document is an introduction to the basic hardware and software setup required to begin programming a Bluetooth® Smart sensor on open-source hardware. We have used the Arduino Uno. The Smart Starter Kit includes this document and three accompanying platform-specific documents. Each document gives a series of short exercises that will build up a complete understanding of end-to-end Bluetooth Smart programming over the whole course. The full guide will take the reader from unboxing and assembling a Bluetooth peripheral to a completed app interfacing with Bluetooth Smart hardware.

The full solution source code for this lab accompanies these documents in the following folder structure:

[Lab Installation Folder]/Arduino/Solutions

Sub-folder	Contents
\ble_proximity_template	Solution to the primary exercises
\Optional Exercise	Solution to optional, extra exercise which adds an LCD display to the hardware

Note: This lab should not be considered a complete solution, nor the basis for any commercial product. All instructions relate directly to the kit list as published.

The source code you need to follow along is here in this document, ready for you to copy into the working project.

Goal

This document guides you through hardware assembly and configuration to produce an app which pairs with an Arduino board acting as a customized proximity tracker device (sometimes called a “key fob”).

This app will allow you to pair your phone and keys, have the key fob make a sound when prompted by the phone app, see a 'hot/cold' indicator on the app when moving near or away from your keys (except for with the Windows Phone version), and be alerted when you move your phone too far away from your keys. For Windows Phone (which does not allow access to the required signal strength data used in the hot/cold indication), you may instead exploit a custom service implemented in the Arduino sketch called the Time Monitoring Service and share the local time on the Windows Phone device with the Arduino sketch, updating it once per second. See the Windows Phone lab for further details.

You can create your app for iOS, Android, BlackBerry 10 or Windows Phone, depending on requirements and available devices.

You must have the following items to complete this lab:

- Microsoft Windows 7 or 8
- OR
- Apple OSX 10.8

Equipment Requirements

You should have the following items to complete this lab:

- [Arduino](#) Uno
- A-to-B type USB cable
- [RedBearLab](#) BLE Shield v2.0

Note that we used version ble-sdk-arduino-0.9.0.beta of the corresponding SDK.

If you use different hardware then the general goals still apply, but the specific steps will be different, depending on the manufacturer.

NOTE: We chose this specific hardware because it is easy to use and was readily available at the time we wrote the doc. You will get equally good results with any other standards-compliant combination of Bluetooth 4.0 hardware.

Exercises Overview

This hands-on lab offers three exercises:

1. Installing Software
2. Your First Sketch
3. Generating a Bluetooth Smart Profile

Estimated time to complete this lab: 15 to 30 minutes

Exercise 1: Setting up the Arduino Software and Hardware

Software

Using Mac OSX:

Task 1 (OS X) — Installing the Arduino software

1. Download the latest stable version of the Arduino Software for Mac OSX from [here](http://arduino.cc/en/Main/Software)¹.
2. Once downloaded, double click to open the archive. Drag the Arduino application into your Applications folder.

Using Windows 8 or 7:

Task 1 (Windows) – Installing the Arduino IDE software

1. Download the latest stable version of the Arduino Software from [here](http://arduino.cc/en/Main/Software). It is available either as an installer or a zip archive, though both contain the same files.
2. Run the installer or unzip the archive to a folder on your computer.

Task 2 (Windows) — Installing the Arduino drivers

1. Take your A-to-B USB cable and connect the Arduino board to your computer.
2. Once the Arduino board has been connected, Windows will attempt to detect the device and install the matching drivers. In most cases this will be successful and you can skip straight to the next exercise.

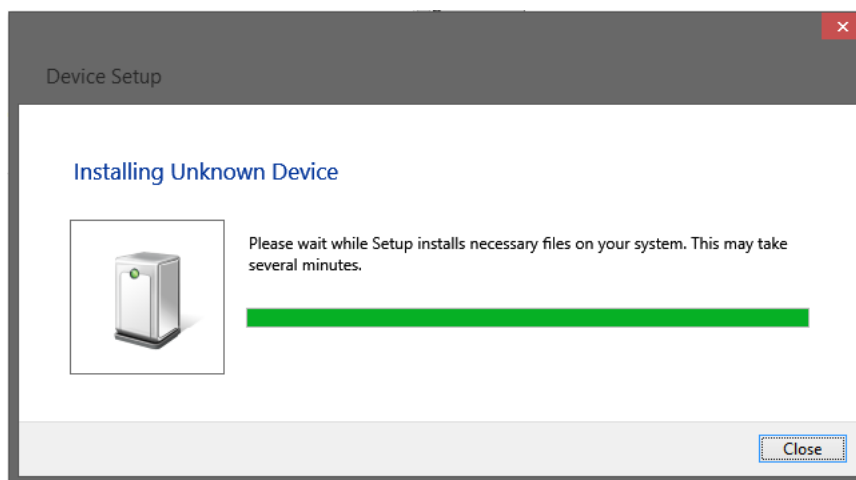


Figure 2: Windows 8 Automatic device installation dialog

¹ <http://arduino.cc/en/Main/Software>

3. If the automatic installation fails, you will see a screen saying, “Windows was unable to find driver software for this device”. In this case, open Device Manager. In the device tree, look under “Other devices” for an “Unknown device”.

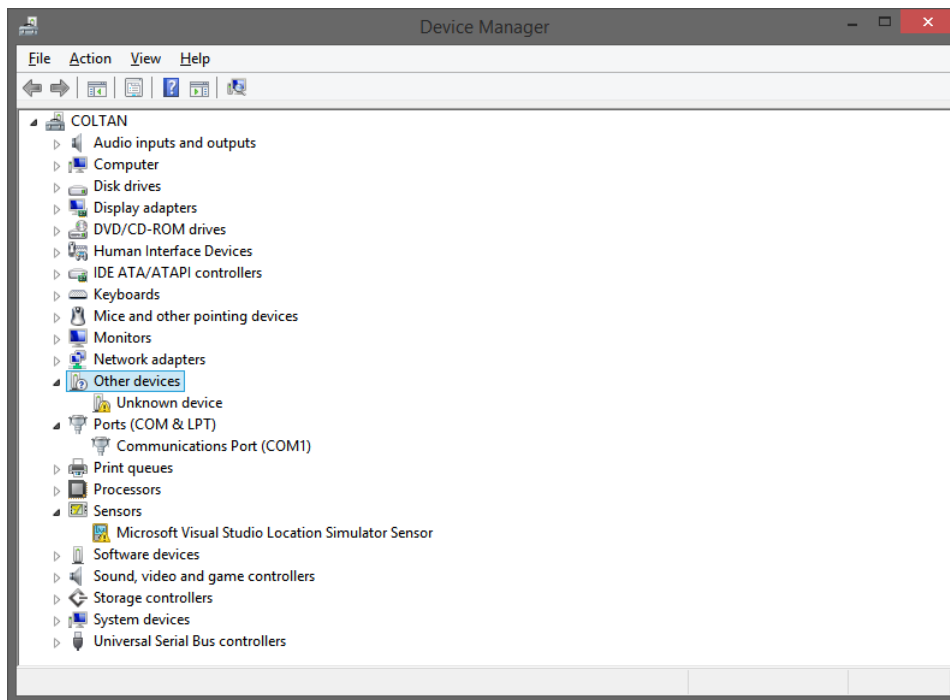


Figure 3: Windows 8 Device Manager showing “Other devices”

4. Right-click on the Unknown Device and select “Update Driver Software”.

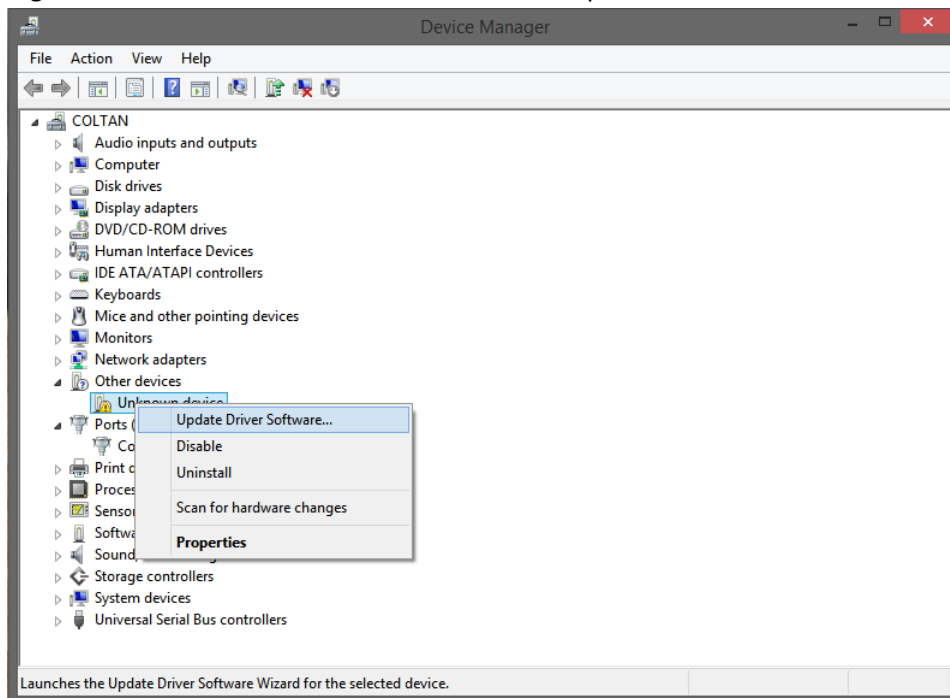


Figure 4: Windows 8 Automatic device installation dialog

- When prompted, select “Browse my computer for Driver software”.

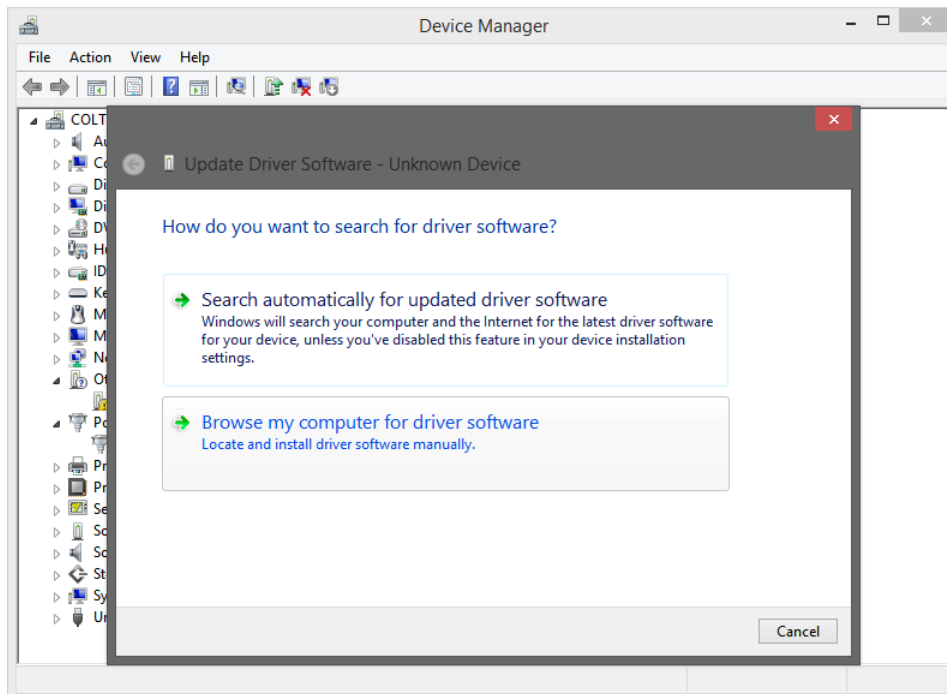


Figure 5: Windows 8 Update Driver Software dialog

- Navigate to the folder where you installed the Arduino software as part of the previous task.
- Look for “Arduino.inf” under \drivers. Select this and the installation should complete.

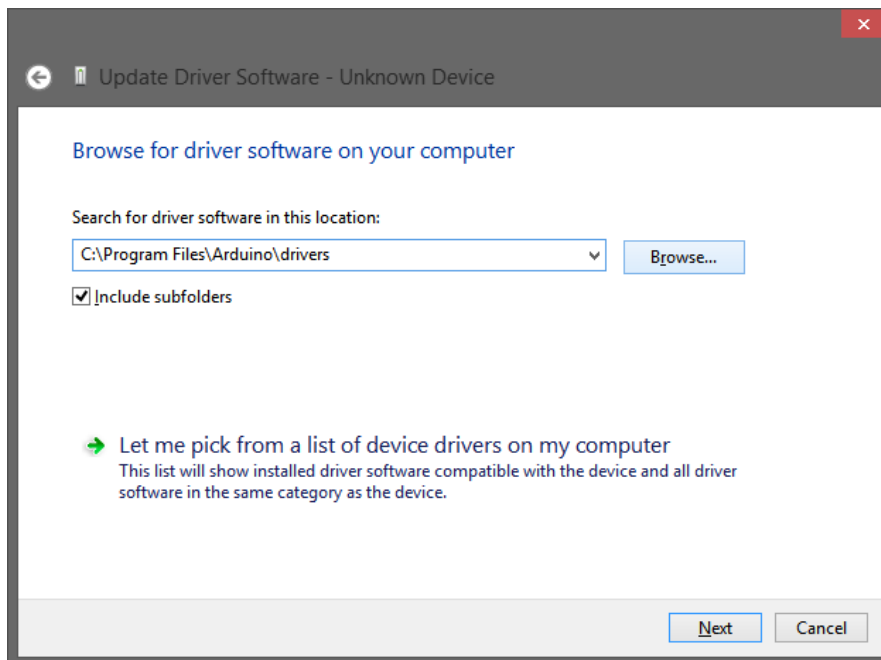
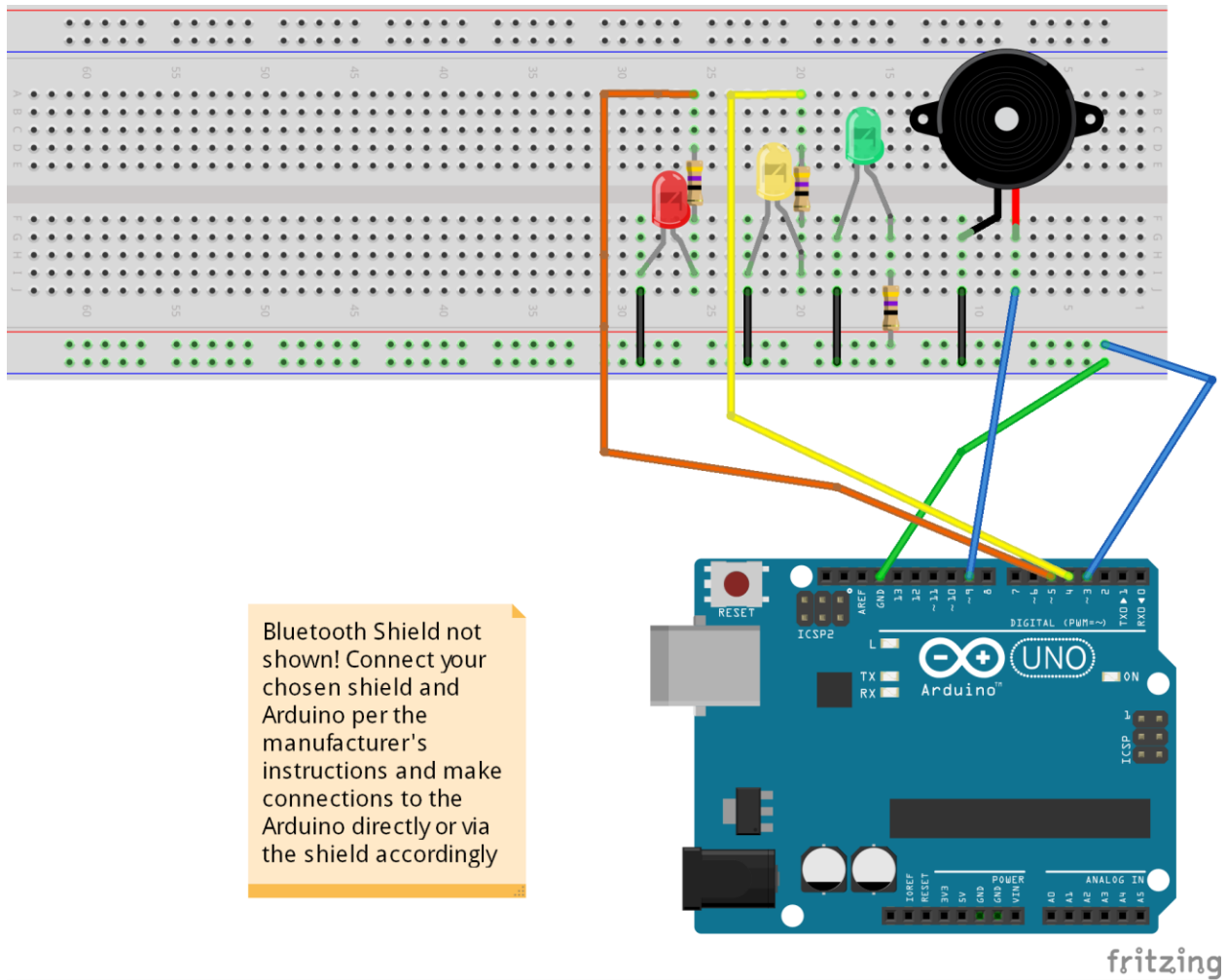


Figure 6: Windows 8 Update Driver Software dialog

Hardware

We designed a simple breadboard circuit for our lab exercises. It's depicted below. As you can see, it consists of the Arduino, a piezo buzzer and three colored LEDs, one green, one yellow and one red, with resistors of a suitable rating in place to protect the LEDs from overloading. One important component is not shown in the diagram however. We used the RedBearLab Bluetooth Low Energy shield (expansion board). This sits on top of the Arduino, plugging directly into all the main Arduino pins. It exposes similar pins which pass straight through to the Arduino and are therefore easily accessible for connecting to the bread board. See <http://redbearlab.com/getting-started-bleshield/> for details on setting up the RedBearLab shield with the Arduino.



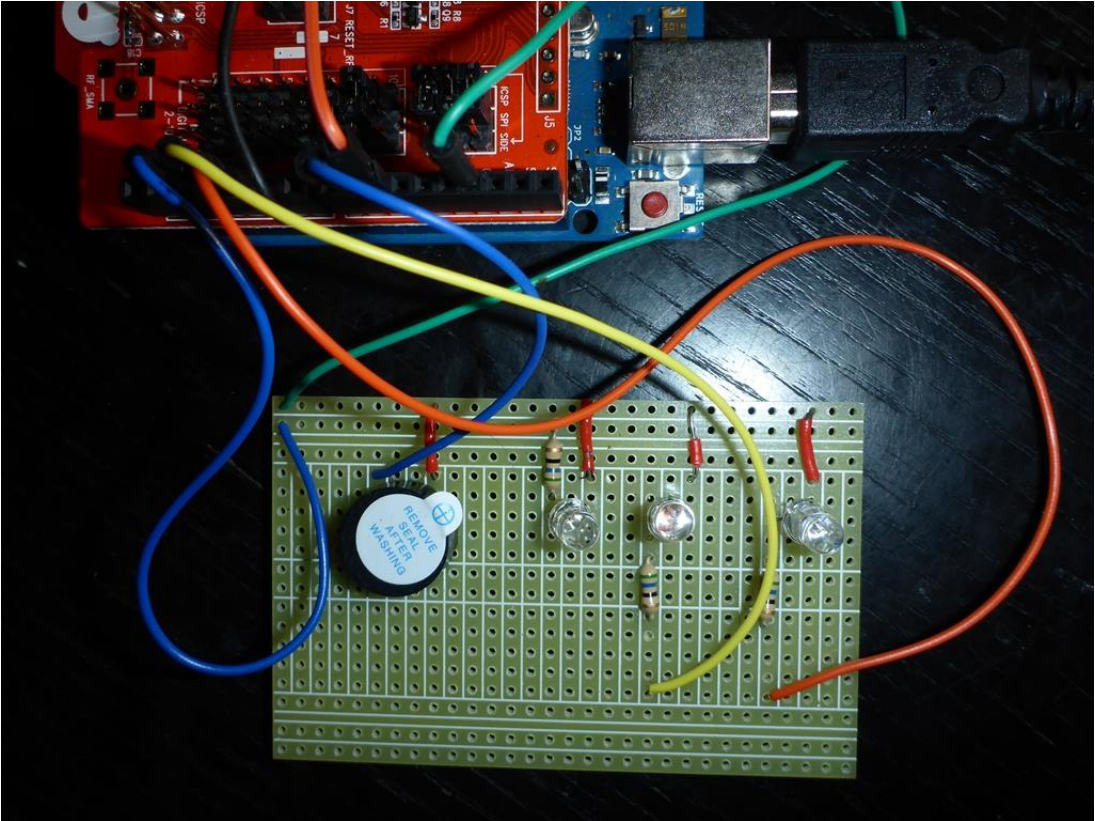
Arduino PIN Connections

PIN 3 -> Green LED +ve

PIN 4-> Yellow LED +ve

PIN 5 -> Red LED +ve

PIN 9 -> Buzzer



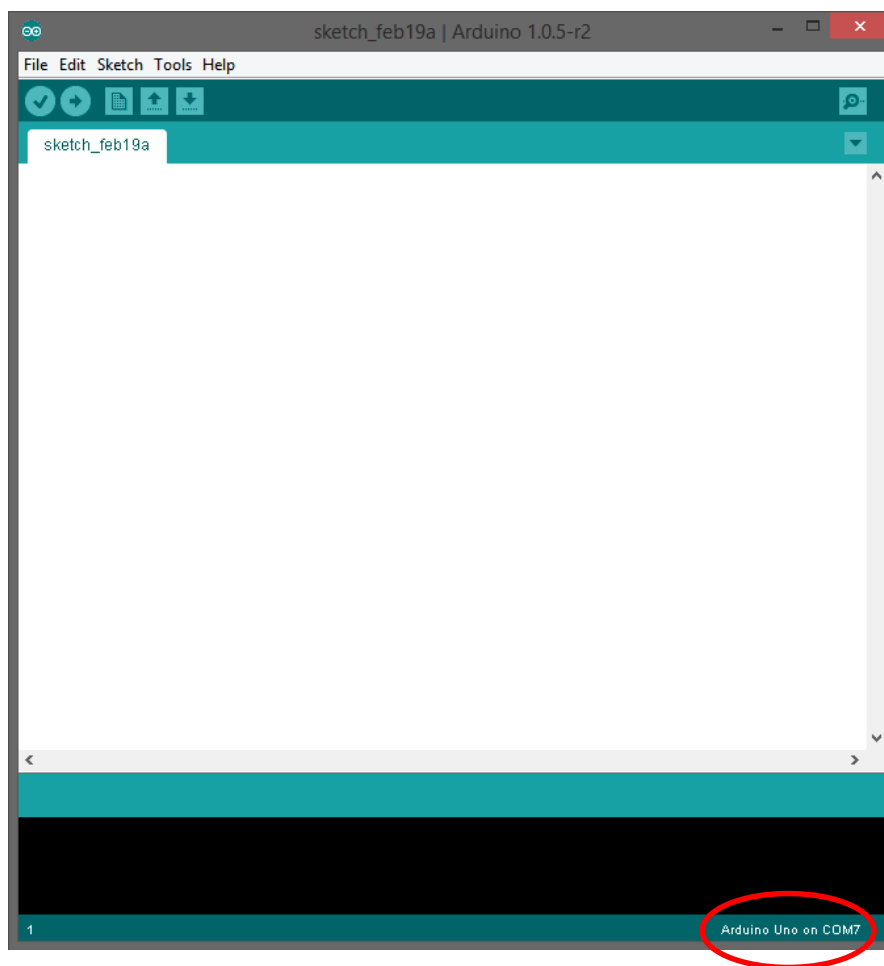
Exercise 2: Your First Arduino Sketch

In Arduino terminology, a program is called a *sketch*. In this exercise, we will go through uploading a basic example sketch to the Arduino device. Once we start creating a Bluetooth Smart application for any platform, all of our code for the Arduino will be implemented in these sketches.

Note: The majority of the information in this lab is also available in the “Getting Started with Arduino” article at the [Arduino website](#).

Task 1 — Load the example sketch

1. Launch the Arduino software; you will see a blank sketch.



Notice how the IDE reports a connected device on a specific COM port

Figure 7: Arduino IDE showing blank sketch

2. Ensure the correct board is selected from Tools > Board. In our case, we are using an Arduino Uno.

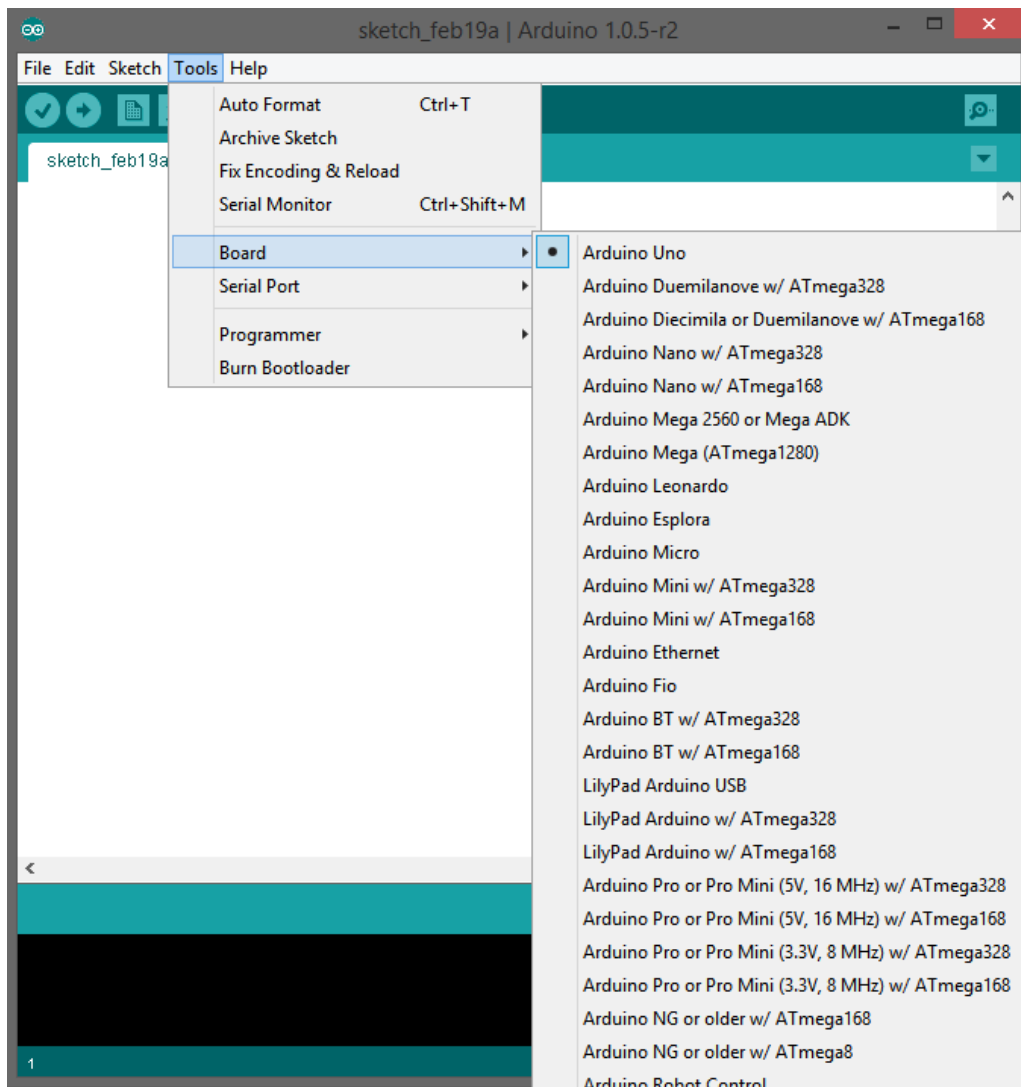
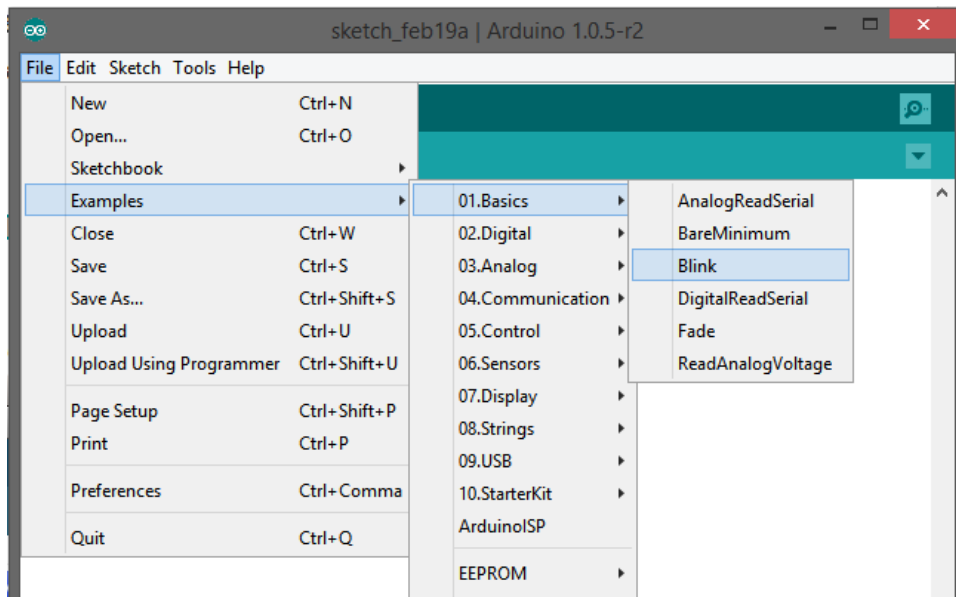


Figure 8
 Arduino IDE showing Tools > Board menu

Make sure the correct COM port is selected from Tools > Serial Port. If multiple ports are shown here, simply disconnect the Arduino from your machine, then reconnect. Select the COM port that disappears then reappears.

On Mac OSX the Arduino will be represented by an entry in the serial port menu beginning

3. Open the example sketch "Blink". Select File > Examples > 01.Basics > Blink



Upload the program to the Arduino board by selecting the “Upload” button from the toolbar. This looks like a horizontal arrow pointing to the right, as shown here.



HINT: mousing over the ‘arrow’ button shows the word “Upload” in the toolbar.

4. The LED on the board should now be blinking.

Congratulations, you’ve uploaded your first Arduino sketch!

Exercise 3: Generate a GATT Profile

Now that we understand the relationship between a sketch and Arduino, we will look at implementing a specific Bluetooth Smart behavior.

In order for Bluetooth Smart peripherals to work, their services must be described via a GATT Profile.

This is a construct that is used to tell a Bluetooth Smart Ready app what services to look for and where to find them. In other words, it will allow Bluetooth Smart Ready devices to discover and understand the capabilities of a given peripheral.

A GATT profile encapsulates the services and functions of a specific device. This means you need to either generate the profile or get it from the relevant hardware manufacturer.

For comprehensive details of the structure of a GATT profile, please refer to the Developer portal at the [Bluetooth website](#)

Throughout this lab we have used the RedBearLab BLE Shield v2.0, and so we have included in this download package the generated profile required for the use of this particular hardware. To generate your own version, or if you have other Bluetooth hardware, follow the instructions below.

If you have a Arduino board, and a RedBearLab v2.0 BLE Shield and want to just put our sketch on the board and skip to the next tutorials, follow these instructions.

Follow exercise 1, to install Arduino software

1. Copy [Lab Installation Folder]/SmartStarterKitV2\Arduino to your Arduino library folder. The Arduino Sketchbook application will tell you the location of this folder in File->Preferences. By default, on Windows it is *C:\Users\[user]\Documents\Arduino*
2. Connect the Arduino board
3. Run Arduino software
4. Goto Sketch>Import Library>Add Library
5. Navigate to the *[Lab Installation Folder]/Arduino/BLE folder, and select choose*
6. Follow exercise 2, except to load the sketch File>Sketchbook>BLE>ble_proximity_template

Task 1 – Identifying your chip & installing the Bluetooth library

In order to generate an appropriate GATT profile and successfully interface with a Bluetooth Smart peripheral, we must first identify the manufacturer of the Bluetooth 4.0 chip that is used on the hardware. Then we will install the correct 3rd-party library to use.

Each Bluetooth Smart shield (sometimes seen as: Bluetooth Low Energy shield) has a chip supplied by an OEM. To find out the OEM for your chip, look at the board and note down any serial numbers printed on the chips. The specific Bluetooth 4.0 chip could be any of the chips present, so it may be a process of elimination to get the correct one.

How we did it : We looked at the bip on the RedBearLabsBLE shield. The chip number is NRF8001, and entered that into <http://www.findchips.com/>. This showed the manufacturer was Nordic Semiconductor. On their web site we found nRFgo Studio for generating GATT profiles. For this lab we were using version 1.17.0.3211.

A web search for the chip number should help you identify the chip manufacturer. A visit to that manufacturer's website or portal should allow you to find software for generating GATT profiles and the manufacturer-specific Bluetooth Smart library to use.

NOTE: Skip this next section if you are not using the RedBearLabs BLE Shield.

We have used the RedBearLabs BLE shield, and are supplying the library files as part of the Smart Starter Kit. These are in a file called *BLE.zip*.

1. Unzip the *[Lab Installation Folder]/Arduino/BLE/BLE.zip* archive to the Arduino system libraries folder :- %ArduinoPath%/Libraries/

You should now see a new folder entitled %FOLDERNAME%. This folder will be automatically included when a sketch requires it.

Task 2 – Generating a profile

If you're generating a GATT Profile for the key-fob application using manufacturer-provided software² then you'll need to do the following:

1. Use the standard [Proximity profile](#), containing the [Link Loss](#), [Immediate Alert](#) and [Tx Power](#) services.
2. Add a custom service for allowing the smart phone application to "share" its proximity data back with the Bluetooth peripheral which in this case is our Arduino and shield.
3. Add a custom service for allowing the smart phone application to "share" its current time back with the Bluetooth peripheral which in this case is our Arduino and shield.

The specifics of the above steps will depend on which manufacturer's chip you use.

² The software varies between manufacturers but as the Bluetooth Smart services are standard across all chips then any adopted or standard profiles will always contain the same basic information.

How we did it. We started by downloading the required software for generating the profile from our chip manufacturers website. Our chip manufacturer provides a GUI tool called nRFGo that allows the creation or editing of profiles. They also provide a set of examples that can be modified and uploaded to the board as desired.

We took their example named '*ble_proximity_template*' that implements a basic Bluetooth Smart proximity detector as the starting point of this exercise. As previously mentioned the Proximity Profile already contains three services by default. To these we want to add two additional, custom services, the first of which we call the Proximity Monitoring Service and the second the Time Monitoring Service. We can then create a key fob finder that will allow you to press a button on your mobile phone and have the Arduino board emit a sound and flash lights, indicating its location (a use case known as Immediate Alert) or automatically make itself audible and visible if we move completely out of range from it (Link Loss). With our custom, Proximity Monitoring Service, we can use the signal strength measured by the smart phone to light an appropriate LED to indicate the approximate distance the smart phone is from the key fob. With the custom Time Monitoring Service, you can send the current time expressed as 3 distinct hour, minute and second values and use the second value to decide which if the three LEDs to light. Specifically, we divide a minute into 4 x 15 second quadrants and disable all LEDs if in the first quadrant, enable one (green) if in the second, two (green and yellow) for the third quadrant and all three LEDs (green, yellow and red) for the fourth.

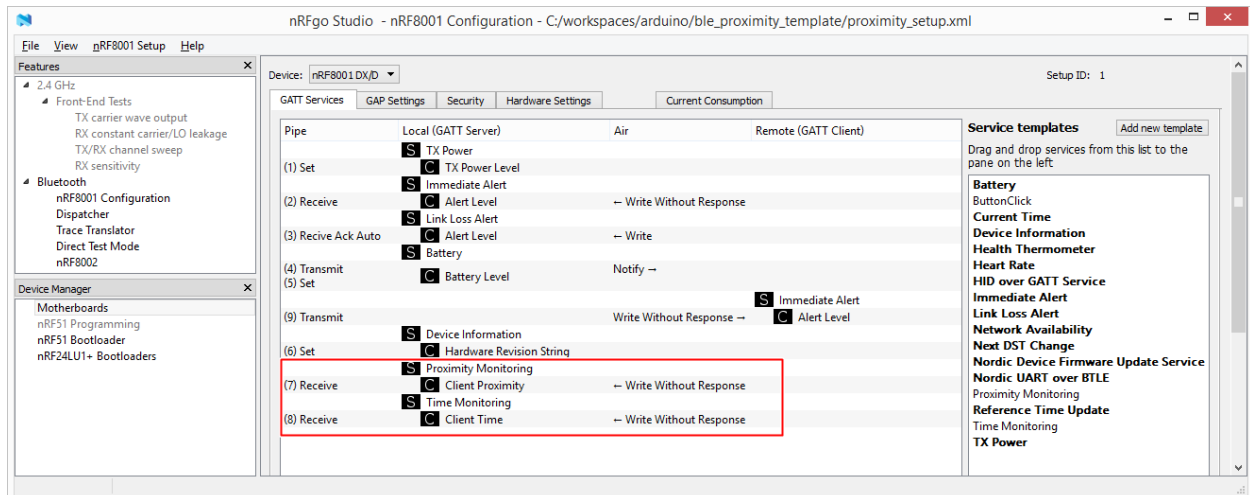
To add the Proximity Monitoring service, using the software, we created a new primary service named ***Proximity Monitoring*** with a Uuid of 3E099910-293F-11E4-93BD-AFD0FE6D1DFD.

The Proximity Monitoring service contains a single characteristic, "Client Proximity" which has the Write Without Response property. Clients may write a 2 octet value to this characteristic; the first octet should contain a proximity band value of 1, 2 or 3, meaning near, middle distance or far and corresponding to the green|yellow|red indication shown on the smart phone UI. The second should contain the client's measured RSSI (Received Signal Strength Indication) value.

To add the Time Monitoring service, using the software, we created a new primary service named ***Time Monitoring*** with a Uuid of 3E099912-293F-11E4-93BD-AFD0FE6D1DFD.

The Time Monitoring service contains a single characteristic, "Client Time" which has the Write Without Response property. Clients may write a 3 octet value to this characteristic; the first octet should contain the hour component of the current time, the second should contain the minute component and the third the second component.

Screen shots from nRFGo showing how these services and their characteristic were defined appear next.



The Proximity Monitoring and Time Monitoring services added to the standard Proximity Profile in nRFgo.

The 'Form' dialog box for 'Proximity Monitoring' service. The Name is 'Proximity Monitoring'. The UUID is '3E09 9910 -293F-11E4-93BD-AFD0FE6D1DFD'. The Base is 'Custom base 1'.

The Proximity Monitoring service UUID

The 'Form' dialog box for 'Time Monitoring' service. The Name is 'Time Monitoring'. The UUID is '3E09 9912 -293F-11E4-93BD-AFD0FE6D1DFD'. The Base is 'Custom base 1'.

The Time Monitoring service UUID

Client Proximity

Main

Characteristic Name Client Proximity

Characteristic UUID 3E09 9911 -293F-11E4-93BD-AFD0FE6D1DFD Base

Base: Custom base 1

Initial value 0000 Hex

Max data length 2 bytes

Use fixed length ☒

☐ Use Characteristic Presentation Format

Format boolean

Exponent 0

Unit 0000

Name Space 01

Descriptor 0000

Properties

Profile Specific Characteristic Descriptors

Advanced

OK Cancel

The Client Proximity characteristic of the Proximity Monitoring service.

Client Time

Main

Characteristic Name Client Time

Characteristic UUID 3E09 9913 -293F-11E4-93BD-AFD0FE6D1DFD Base

Base: Custom base 1

Initial value 00 Hex

Max data length 7 bytes

Use fixed length ☒

☐ Use Characteristic Presentation Format

Format boolean

Exponent 0

Unit 0000

Name Space 01

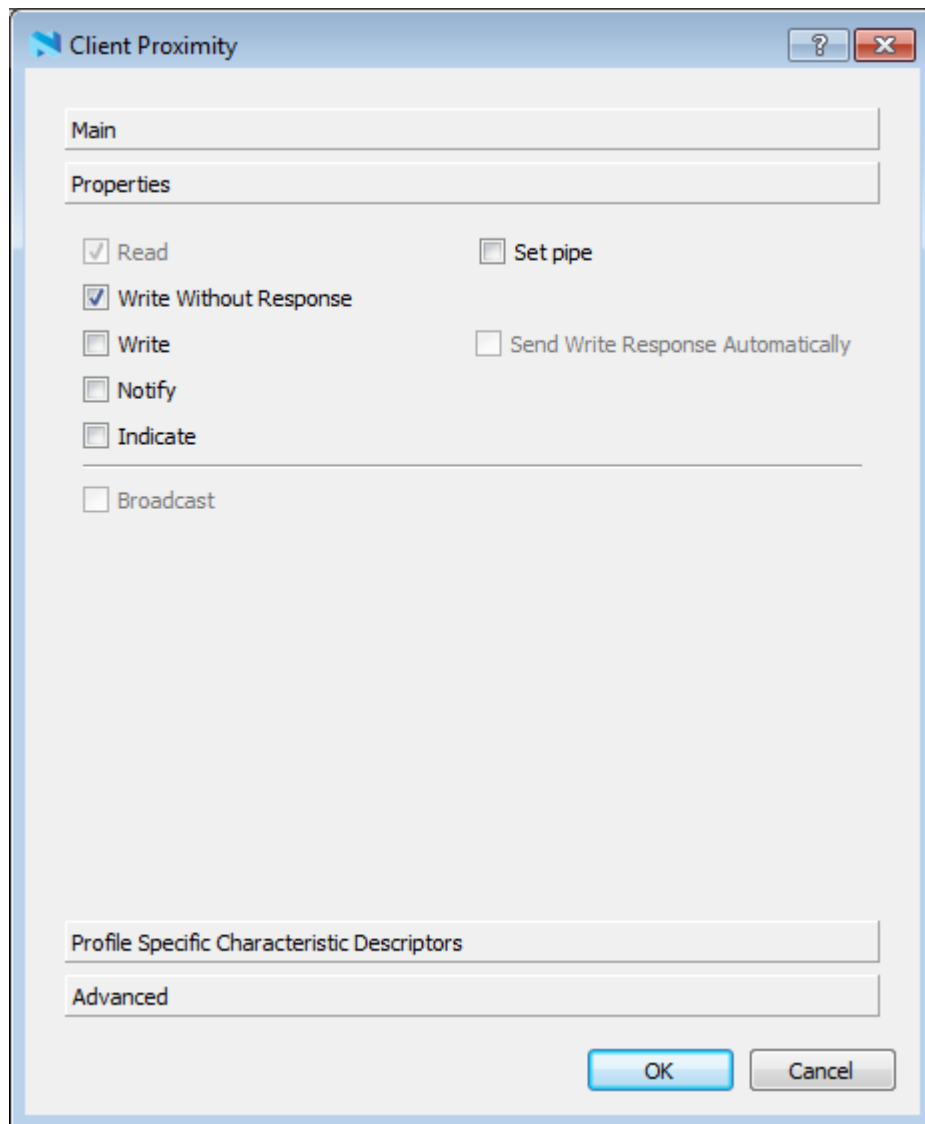
Properties

Profile Specific Characteristic Descriptors

Advanced

OK Cancel

The Client Time characteristic of the Time Monitoring service.



The image shows a Windows-style dialog box titled "Client Proximity". It has a standard title bar with a question mark icon and a close button (X). The dialog is divided into several sections. At the top is a "Main" section. Below it is a "Properties" section containing a list of checkboxes: "Read" (checked), "Write Without Response" (checked), "Write" (unchecked), "Notify" (unchecked), "Indicate" (unchecked), "Broadcast" (unchecked), "Set pipe" (unchecked), and "Send Write Response Automatically" (unchecked). Below the "Properties" section is a "Profile Specific Characteristic Descriptors" section. At the bottom is an "Advanced" section. The "OK" and "Cancel" buttons are located at the bottom right of the dialog.

Client Proximity

Main

Properties

☒ Read ☐ Set pipe

☒ Write Without Response

☐ Write ☐ Send Write Response Automatically

☐ Notify

☐ Indicate

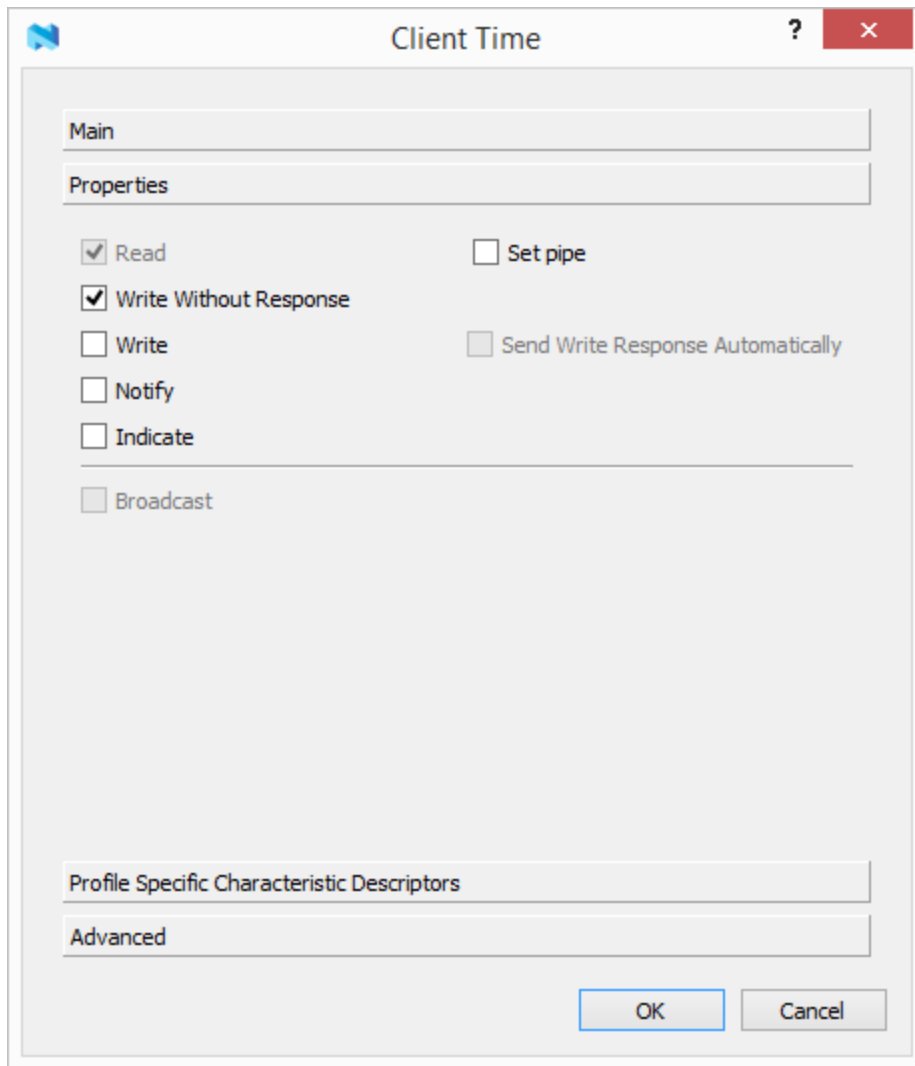
☐ Broadcast

Profile Specific Characteristic Descriptors

Advanced

OK Cancel

Properties of the Client Proximity characteristic



Properties of the Client Time characteristic

Once the two custom services have been added we are now ready to generate the source files for the profile. This is accomplished in the software we used by a '*generate source files/generate only services.h*' menu command but this may vary depending on your board manufacturer – you'll need to look for something similar. After the generation process has completed we have the auto generated files '*services.h*' and '*services_lock.h*'. These contain the profiles of the services we intend to use and are defined in hex format. They will be included into our main project in the next task.

The generated code uses the array defined in *services.h* by including *services.h* in the *ble_proximity_template.ino* file. If you use the array defined in *services_lock.h*, the service setup will be written to the OTP memory in the chip, so you can avoid having to load the setup at each startup. However, this memory can only be written once, so you should be certain that the setup is as you require it to be before you do this. To switch to *services_lock.h* you need to change the include in *ble_proximity_template.ino*. See comments in *ble_proximity_template.ino*.

Macro defined in services.h

```
#define SETUP_ID 1
#define SETUP_FORMAT 3 /** nRF8001 D */
#define ACI_DYNAMIC_DATA_SIZE 168

/* Service: TX Power - Characteristic: TX Power Level - Pipe: SET */
#define PIPE_TX_POWER_TX_POWER_LEVEL_SET 1
#define PIPE_TX_POWER_TX_POWER_LEVEL_SET_MAX_SIZE 1

/* Service: Immediate Alert - Characteristic: Alert Level - Pipe: RX */
#define PIPE_IMMEDIATE_ALERT_ALERT_LEVEL_RX 2
#define PIPE_IMMEDIATE_ALERT_ALERT_LEVEL_RX_MAX_SIZE 1

/* Service: Link Loss Alert - Characteristic: Alert Level - Pipe:
RX_ACK_AUTO */
#define PIPE_LINK_LOSS_ALERT_ALERT_LEVEL_RX_ACK_AUTO 3
#define PIPE_LINK_LOSS_ALERT_ALERT_LEVEL_RX_ACK_AUTO_MAX_SIZE 1

/* Service: Battery - Characteristic: Battery Level - Pipe: TX */
#define PIPE_BATTERY_BATTERY_LEVEL_TX 4
#define PIPE_BATTERY_BATTERY_LEVEL_TX_MAX_SIZE 1

/* Service: Battery - Characteristic: Battery Level - Pipe: SET */
#define PIPE_BATTERY_BATTERY_LEVEL_SET 5
#define PIPE_BATTERY_BATTERY_LEVEL_SET_MAX_SIZE 1

/* Service: Device Information - Characteristic: Hardware Revision String
- Pipe: SET */
#define PIPE_DEVICE_INFORMATION_HARDWARE_REVISION_STRING_SET 6
#define PIPE_DEVICE_INFORMATION_HARDWARE_REVISION_STRING_SET_MAX_SIZE 9

/* Service: Proximity Monitoring - Characteristic: Client Proximity -
Pipe: RX */
#define PIPE_PROXIMITY_MONITORING_CLIENT_PROXIMITY_RX 7
#define PIPE_PROXIMITY_MONITORING_CLIENT_PROXIMITY_RX_MAX_SIZE 2
```

```
/* Service: Time Monitoring - Characteristic: Client Time - Pipe: RX */
#define PIPE_TIME_MONITORING_CLIENT_TIME_RX          8
#define PIPE_TIME_MONITORING_CLIENT_TIME_RX_MAX_SIZE 7

/* Service: Immediate Alert - Characteristic: Alert Level - Pipe: TX */
#define PIPE_IMMEDIATE_ALERT_ALERT_LEVEL_TX_1        9
#define PIPE_IMMEDIATE_ALERT_ALERT_LEVEL_TX_1_MAX_SIZE 1
```

Note that “PIPE” is a term used in the Nordic Semiconductors SDK and conceptually means a given Bluetooth operation applied to a specified characteristic. So for example receiving a request to “write without response” a value to the Client Proximity characteristic is a “pipe”. Each pipe has a unique, numeric value which serves as an identifier and which is available within the Arduino sketch from the Nordic Semiconductors API. Switch statements are often built around pipe values.

Task 3 – Implementing the key fob behaviours

At this point, we have set up our environment and generated the header files that define the services we want to use to turn the Arduino board into a Bluetooth-enabled key fob. Now we need to add our own code to the proximity code example supplied by our board manufacturer so that our chosen behaviours are exhibited in response to the various events that may occur. The full solution source code is available as part of this download package, in the *[Lab Installation Folder]/[platform]/solutions/ble_proximity_template* folder.

Our aim is to make our Arduino key fob respond to the following events:

- A) The Link Loss service's alert level changing. This may be triggered by the user selecting one of three available values in the smart phone app UI. We want to store the selected alert level value for use when link loss occurs and we want to acknowledge the request by flashing one of the LEDs, selected according to the alert level value we received.
- B) The immediate alert service's alert level changing. This may be triggered by the user clicking a button labelled "Make a noise" in the smart phone app UI. For a value greater than zero we want to flash all three LEDs on and off a number of times and, sound the buzzer in unison. For a value of zero, we'll just flash the LEDs but make no noise.
- C) The link between Arduino / Shield peripheral and smart phone being lost ("Link Loss"). When this happens, we'll flash all three LEDs and sounder the buzzer, for an extended period of time.
- D) Proximity Monitoring data being received. To begin with, we'll just use the proximity band value to select and illuminate one of either the green (1), yellow (2) or red (3) LEDs. If a proximity band value of zero is received, we'll take that to mean that the user has disabled the sharing of proximity data and respond by switching off all LEDs. We won't use the RSSI value here but as an additional exercise, you are invited to add a Serial UART LCD display to the bread board and program the Arduino sketch to display the received RSSI value on it.
- E) Time Monitoring data being received. We'll use the second component of the client's time value to select and illuminate zero, one, two or three of the LEDs according to the 15 second quadrant of a minute that the second value falls within. As an additional exercise, you are invited to add a Serial UART LCD display to the bread board and program the Arduino sketch to display the received client time value on it.

Let's go through each of the events, one at a time.

A) Link Loss Alert Level change

So that our key fob device can respond to changes to the selected alert level for link loss events, we need to modify both the the standard link_loss.cpp file and the Arduino sketch file, ble_proximity_template.ino. We need to modify link_loss.cpp so that it includes a call to a new function called [link_loss_alert_level_change](#) which we'll add to ble_proximity_template.ino along with some other changes.

link_loss.cpp (extract)

```
void link_loss_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t *buffer)
{
    switch (pipe_num)
    {
        case PIPE_LINK_LOSS_ALERT_ALERT_LEVEL_RX_ACK_AUTO :
            alert_handle_on_link_loss = (alert_level_t)buffer[0];
            link\_loss\_alert\_level\_change((alert_level_t)buffer[0]);
            break;
    }
}
```

ble_proximity_template.ino (extract)

```
int ledPin1      = 3;
int ledPin2      = 4;
int ledPin3      = 5;

void link\_loss\_alert\_level\_change(alert_level_t level)
{
    alert_level_print(level);

    action = ACTION_SET_ALERT_LEVEL_REQUESTED;

    int ledpin = getPinNumber(level);

    flash(ledpin, 250, 4);
}
```

```

int getPinNumber(uint8_t level) {
    switch (level) {
        case 0:    return ledPin1;
                   break;
        case 1:    return ledPin2;
                   break;
        case 2:    return ledPin3;
                   break;
        default:   return ledPin1;
                   break;
    }
}

void flash(unsigned char led,unsigned char delaysms, unsigned char times){
    for (int i=0;i<times;i++) {
        digitalWrite(led, HIGH);
        delay(delaysms);
        digitalWrite(led, LOW);
        delay(delaysms);
    }
}

```

As you can see, the `link_loss_alert_level_change` function determines the LED that corresponds to the requested alert level value and causes it to flash very briefly, 4 times just to act as a visual acknowledgement of the requested alert level change.

B) Immediate Alert

An Immediate Alert event is triggered by a request to write a value to the Alert Level characteristic of the Immediate Alert service being received. The default `immediate_alert.cpp` file does not need to be changed as it already calls a function in our sketch called `immediate_alert_hook`. We'll make our changes in that function so that we can control the LEDs and buzzer in the way that we require.

Immediate_alert.cpp (extract – no change)

```
void immediate_alert_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t
*buffer)
{
    switch (pipe_num)
    {
        case PIPE_IMMEDIATE_ALERT_ALERT_LEVEL_RX :
            immediate_alert_hook((alert_level_t)buffer[0]);
            break;
    }
}
```

ble_proximity_template.ino (extract)

```
void immediate_alert_hook(alert_level_t level)
{
    alert_level_print(level);
    // flash more times for higher alert levels
    Counter = SOUND_BUZZER_FOR_3_SECONDS + level;
    alert_level = level;
}
```

Notice the Counter variable. We use this variable to control the buzzer. Once set, we count down in our main event loop and switch on /off the pin to which our buzzer is connected on the board. This will turn the buzzer on/off and allow us to audibly locate the ‘key fob’. We also flash all three LEDs when the buzzer is sounding so that there’s also a visual indication of the location of our Arduino key fob. Here are the key code fragments from ble_proximity_template.ino.

```
#define SOUND_BUZZER_FOR_3_SECONDS 3
#define SOUND_BUZZER_FOR_90_SECONDS 90
#define BUZZER_OFF 0
#define ONE_SECOND 1000

void loop()
{
```

```

    aci_loop();
    ControlBuzzer();
}
void ControlBuzzer(void)
{
    if (Counter > BUZZER_OFF && Counter-- > BUZZER_OFF)
    {
        beepAndFlashAll(ONE_SECOND);
    }
}

void beepAndFlashAll(unsigned char delaysms){
    digitalWrite(ledPin1, HIGH);
    digitalWrite(ledPin2, HIGH);
    digitalWrite(ledPin3, HIGH);
    analogWrite(speakerPin, volume[alert_level]);
    delay(delaysms);
    analogWrite(speakerPin, volume[0]);
    digitalWrite(ledPin1, LOW);
    digitalWrite(ledPin2, LOW);
    digitalWrite(ledPin3, LOW);
    delay(delaysms);
}

```

C) Link Loss

When the link between our Arduino key fob and the smart phone and its app is lost, we want to sound the buzzer and flash all three LEDs for an extended period of time (90 seconds in fact). This is a similar behaviour to the Immediate Alert case so we can reuse most of that code. The default proximity code for our shield already implements a function which is called when link loss occurs, so our job is simple. We just need to make a small change to that function so that the buzzer and LEDs are used in the way we require.

link_loss.cpp (extract – no change)

```

void proximity_disconnect_evt_rcvd(uint8_t disconnect_reason)
{
    if ((DISCONNECT_REASON_CX_CLOSED_BY_PEER_DEVICE !=
disconnect_reason)&&(DISCONNECT_REASON_CX_CLOSED_BY_LOCAL_DEVICE !=

```

```

disconnect_reason))
{
    link_loss_alert_hook(alert_handle_on_link_loss);
}
}

```

ble_proximity_template.ino (extract)

```

void link_loss_alert_hook(alert_level_t level)
{
    alert_level_print(level);
    if(level > 0)
    {
        Counter = SOUND_BUZZER_FOR_90_SECONDS;
    }
    else
    {
        Counter = BUZZER_OFF;
    }
}

```

D) Proximity Monitoring

If the user enables “proximity sharing” in their smart phone app, this will cause Write Without Response requests to be sent to our key fob, whenever the client’s RSSI value changes. 2 octets will be sent in the write request, the first containing a numeric indicator of the “proximity band” that the smart phone is deemed to fall within (near, middle distance, far) and the second, the RSSI value itself.

Here’s the code we Need for this use case:

client_proximity.h

```

#include <stdint.h>

#ifndef CLIENT_PROXIMITY__
#define CLIENT_PROXIMITY__

extern void client_proximity_hook(uint8_t proximity_band, uint8_t rssi);

```

```
void client_proximity_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t
*buffer);

#endif//CLIENT_PROXIMITY__
```

client_proximity.cpp

```
#include <stdint.h>
#include "lib_aci.h"
#include "services.h"
#include "client_proximity.h"

void client_proximity_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t
*buffer)
{
    switch (pipe_num)
    {
        case PIPE_PROXIMITY_MONITORING_CLIENT_PROXIMITY_RX :
            client_proximity_hook(buffer[0],buffer[1]);
            break;
    }
}
```

ble_proximity_template.ino (extract)

```
void client_proximity_hook(uint8_t proximity_band, uint8_t rssi)
{
    int rssi_value = rssi & 0xFF;
    allLedsOff();
    if (proximity_band == 0) {
        // means the user has turned off proximity sharing so we just want to
        switch off the LEDs
        return;
    }
    int ledpin = getPinNumber(proximity_band - 1);
    digitalWrite(ledpin, HIGH);
}
```

```
void allLedsOff() {  
    digitalWrite(ledPin1, LOW);  
    digitalWrite(ledPin2, LOW);  
    digitalWrite(ledPin3, LOW);  
}
```

E) Time Monitoring

If the user enables “time sharing” in their smart phone app, this will cause Write Without Response requests to be sent to our key fob at intervals, probably each second. 3 octets will be sent in the write request, the first containing the hour component of the client’s time, the second the minute component and the third the seconds i.e. we have HH, MM and SS in the three octets.

Here’s the code we Need for this use case:

client_time.h

```
#include <stdint.h>  
#ifndef CLIENT_TIME__  
#define CLIENT_TIME__  
  
extern void client_time_hook(uint8_t hh,uint8_t mm,uint8_t ss);  
  
void client_time_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t *buffer);  
  
#endif//CLIENT_TIME__
```

client_time.cpp

```
#include <stdint.h>
#include "lib_aci.h"
#include "services.h"
#include "client_time.h"

void client_time_pipes_updated_evt_rcvd(uint8_t pipe_num, uint8_t *buffer)
{
    switch (pipe_num)
    {
        case PIPE_TIME_MONITORING_CLIENT_TIME_RX :
            client_time_hook(buffer[0],buffer[1],buffer[2]);
            break;
    }
}
```

ble_proximity_template.ino (extract)

```
void client_time_hook(uint8_t hh, uint8_t mm, uint8_t ss)
{
    allLedsOff();
    String time_message = "Time: ";
    time_message = time_message + hh;
    time_message = time_message + ":";
    time_message = time_message + mm;
    time_message = time_message + ":";
    time_message = time_message + ss;

    // light 0 LEDs if less than 15, 1 if less than 30, 2 if less than 45 else
    all 3 LEDs
    int leds_to_light = ss / 15;
    if (leds_to_light == 0) {
        return;
    }
    for (int i=0;i<leds_to_light;i++) {
        int ledpin = getPinNumber(i);
```



```
        digitalWrite(ledpin, HIGH);  
    }  
}
```

What we have achieved

By now we have a complete implementation of a Bluetooth Smart profile on Arduino, including our very own custom service.

File	Comment
alert_level_characteristic.h	defines possibles alert levels
ble_proximity_template.ino	Modified proximty template
immediate_alert.cpp immediate_alert.h	Immediate alert service implementation
link_loss.cpp link_loss.h	Link Loss Service (for proximity profile) implementation
client_proximity.cpp client_proximity.h	Proximity Monitoring service, handles writes to the Client Proximity characteristic.
client_time.cpp client_time.h	Proximity Time service, handles writes to the Client Time characteristic.
ublu_setup.gen.out.txt	Output from run_me_compile_xml_to_nRF8001_setup.bat, that creates the binding between the ports and uuids
proximity_setup.xml	Created by the nordic software
run_me_compile_xml_to_nRF8001_setup.bat	Batach file creates the binding between the ports and uuid, generates services.h & service_lock.h
services.h	This file is autogenerated by nRFgo Studio
services_lock.h	This file is autogenerated by nRFgo Studio

Once you have successfully worked through this document, setting up your Arduino, you're ready to start on the platform-specific part of the Bluetooth Smart Starter Kit. Please refer to the relevant hands-on lab for the OS you'd like to work with.

Exercise 4: Optional – Add an LCD Display

At this stage, our `client_proximity_hook(uint8_t proximity_band, uint8_t rssi)` function does not use the RSSI value it receives from the smartphone client application. As an optional exercise, let's address that by adding a Serial UART LCD display to our circuit board and display the received RSSI value on it. We can also modify the `client_time_hook` function to display the hour, minute and second values it receives on the display.

Hook up your LCD display according to the instructions you received with your hardware. The component we used requires only three connections to be made; one to the 5V Arduino pin, one to its GND pin and the RX connection to one of the Arduino's digital pins (we used pin 10).

Once done, you just need to initialize access to your LCD's serial interface and then write a message to it when you want to. You'll find a full solution in the Optional Exercise folder within the Arduino folder of this package. Here are the key parts:

ble_proximity_template.ino.cpp – initializing the serial UART LCD

```
// Serial UART LCD control
#include <SoftwareSerial.h>
#define lcdTxPin 10
SoftwareSerial LCD = SoftwareSerial(0, lcdTxPin);
const int LCDdelay=10;
void setup(void)
{
....
    LCD.begin(9600);
    backlightOn() ;
    clearLCD();
    lcdPosition(0,0);
    LCD.print("Ready...");
}
```

ble_proximity_template.ino.cpp – initializing the serial UART LCD

```
void client_proximity_hook(uint8_t proximity_band, byte rssi)
{
    int rssi_value = (256 - (int) rssi) * -1;
    ...
    String rssi_message = "Client RSSI: ";
    rssi_message = rssi_message + rssi_value;
    clearLCD();
    LCD.print(rssi_message);
    ...
}

void client_time_hook(uint8_t hh, uint8_t mm, uint8_t ss)
{
    ...
    String time_message = "Time: ";
    time_message = time_message + hh;
    time_message = time_message + ":";
    time_message = time_message + mm;
    time_message = time_message + ":";
    time_message = time_message + ss;
    clearLCD();
    LCD.print(time_message);
    ...
}
```

ble_proximity_template.ino.cpp – serial UART LCD functions

```
void lcdPosition(int row, int col) {
    LCD.write(0xFE);    //command flag
    LCD.write((col + row*64 + 128));    //position
    delay(LCDdelay);
}

void clearLCD() {
    LCD.write(0xFE);    //command flag
```

```

    LCD.write(0x01);    //clear command.
    delay(LCDdelay);
}

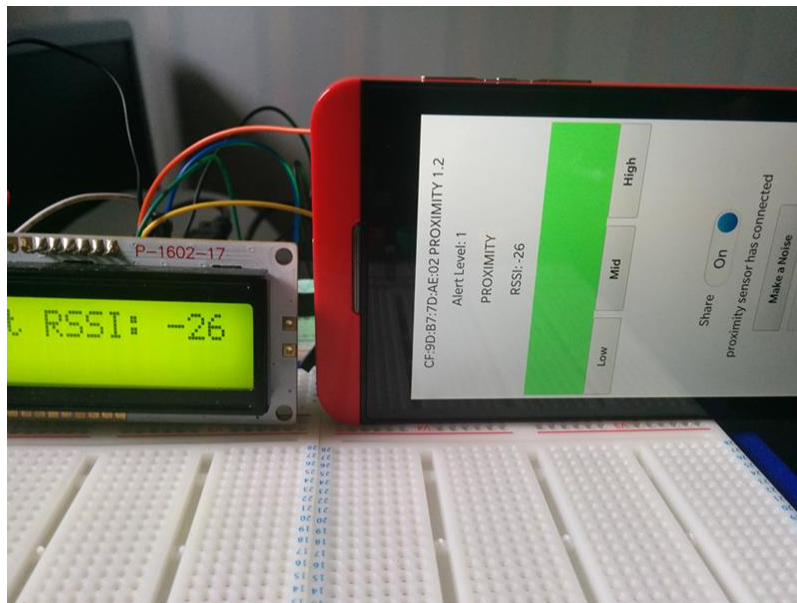
void backlightOn() {    //turns on the backlight
    LCD.write(0x7C);    //command flag for backlight stuff
    LCD.write(157);     //light level.
    delay(LCDdelay);
}

void backlightOff(){    //turns off the backlight
    LCD.write(0x7C);    //command flag for backlight stuff
    LCD.write(128);     //light level for off.
    delay(LCDdelay);
}

void serCommand(){      //a general function to call the command flag for
issuing all other commands
    LCD.write(0xFE);
}

```

Here's what you should end up with when you've completed this exercise!



RSSI data from the Proximity Monitoring Service shown on the LCD display



Client time from the Time Monitoring Service shown on the LCD display

Portions of this document were taken or adapted from the Arduino Getting Started Guide.

The text of the Arduino Getting Started Guide is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Code samples in the guide are released into the public domain.