Hands-On Lab

Bluetooth Smart Ready Application for BlackBerry 10

Lab version: 2.0

Last updated: 11/24/2014

Contents

Revision History

Version	Date	Author	Changes
1.0	3rd September 2014	Martin Woolley	First version, released in Smart Starter Kit V2.0.

Overview

This document is a step-by-step guide to creating a Bluetooth Smart Ready app for BlackBerry 10, using a Bluetooth Smart device implementing a GATT profile. For details on the specific profile and how to configure your Bluetooth Smart Device, please refer to the *Getting Started With Arduino Hands-on Lab* document downloaded with this package. In this lab we will not cover any basics of BlackBerry 10 programming, nor is the resulting code suitable for production – the lab, tasks and code are designed to provide instruction in the principles and practice of Bluetooth Smart.

NOTE: This lab is not a complete commercial solution. All instructions relate directly to the kit list as published.

Objectives

This lab provides instructions to achieve the following:

- Discover a nearby Bluetooth Smart Device.
- Communicate with a Bluetooth Smart device.
- Implement a "key finder" app on BlackBerry 10, using a Bluetooth Smart device.

FYI: to complete this lab you will first need to configure the Arduino Uno and Bluetooth Smart Shield. Please refer to the Hands-on Lab - Getting Started With Arduino document downloaded with this package.

System Requirements

- Momentics IDE for BlackBerry 10 (minimum 2.1)
- Other requirements per https://developer.blackberry.com/native/downloads/requirements/

Equipment Requirements

- Arduino Uno
- A-to-B type USB cable
- Red Bear Labs Bluetooth Low Energy Shield
- BlackBerry 10 device running 10.2.0 or later

Exercises

This hands-on lab contains the following exercises:

1. Scanning for Bluetooth Smart devices
2. Communicating with a Bluetooth Smart device

Estimated time to complete this lab: 45 to 60 minutes

Exercise 1: Getting Started

Our BlackBerry 10 lab application has much code in it which is not concerned with Bluetooth; a user interface created with QML and signals and slots to allow signals to be used to communicate events between components. To make life easier for you, we provide a version of the application with all the key Bluetooth functionality removed to act as our start point. Your first task therefore, is to import this code into your Momentics workspace, build it and install it on your BlackBerry 10 device to ensure you have a solid basis from which to proceed.

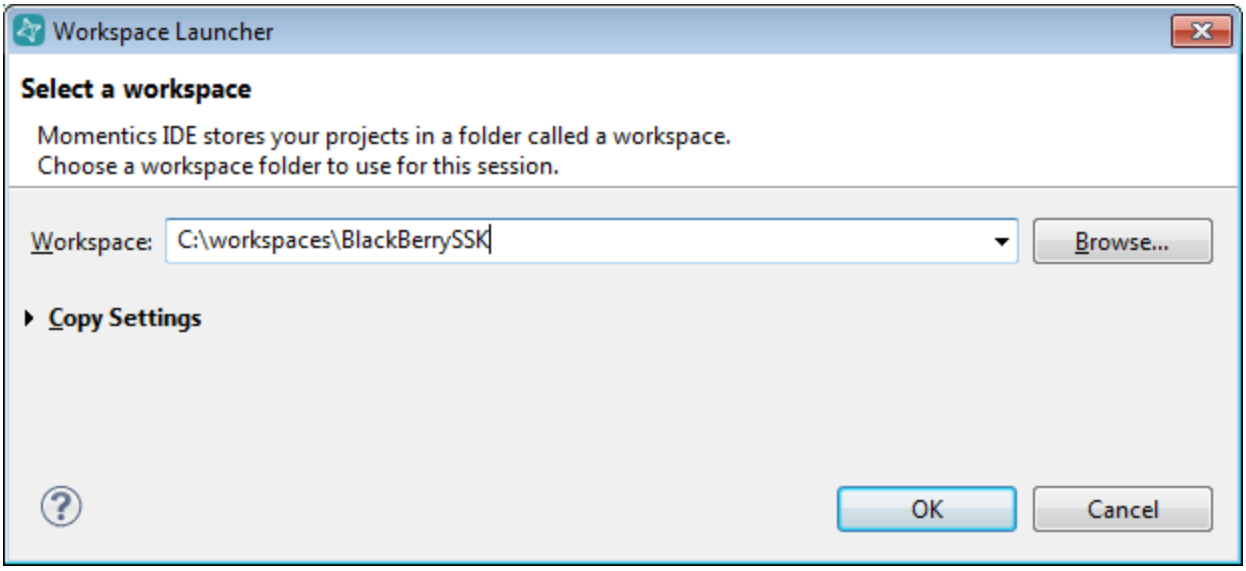
Task 1 – Set up your environment and the starter BlackBerry 10 Project

For this task, you will need a BlackBerry 10 device running 10.2.0 or above and a development machine to install and use the BlackBerry Momentics IDE on.

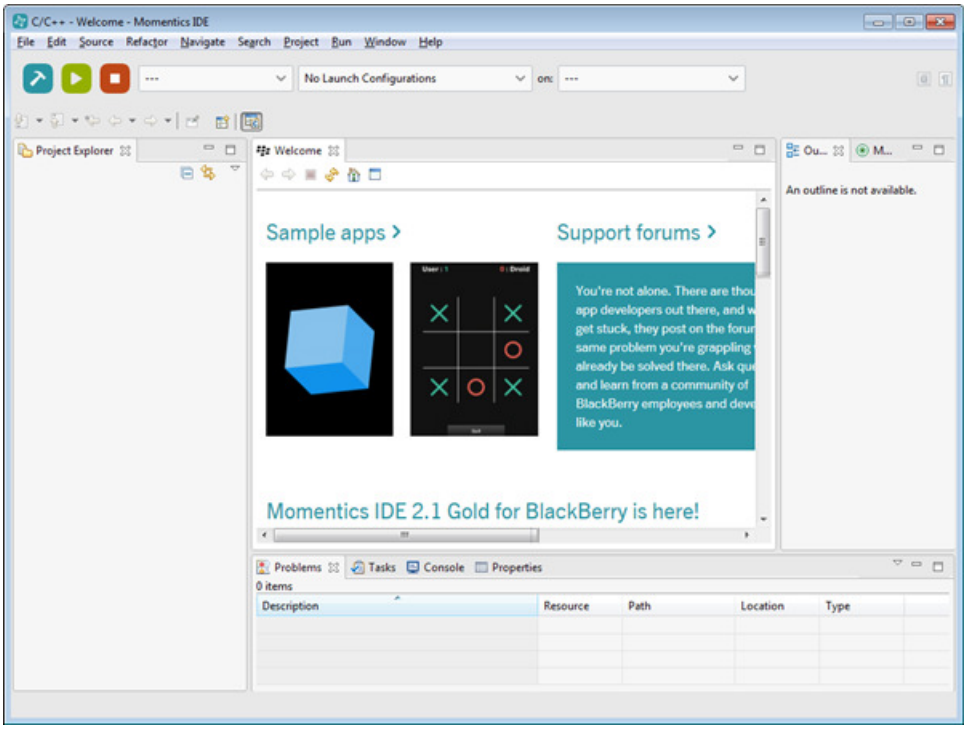
In this lab, we're developing a BlackBerry 10 *native* application. As such you need the Momentics IDE installed on your machine. Go to https://developer.blackberry.com/native/ and follow the instructions given for setting up Momentics if you don't already have it installed.

Once you have Momentics installed:

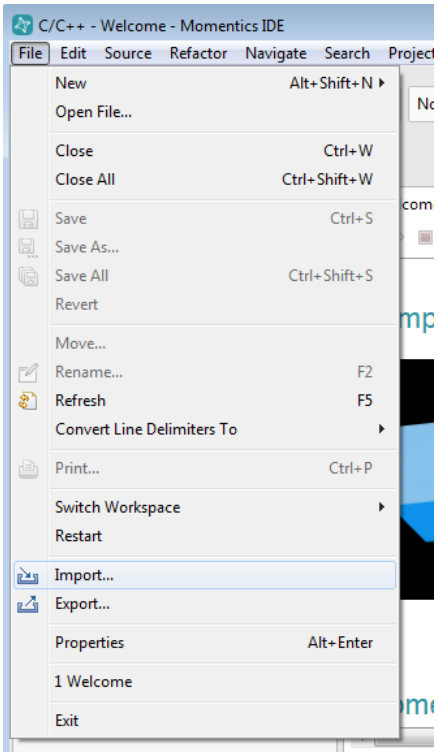
1. launch it and select a directory to act as the root of your project workspace.



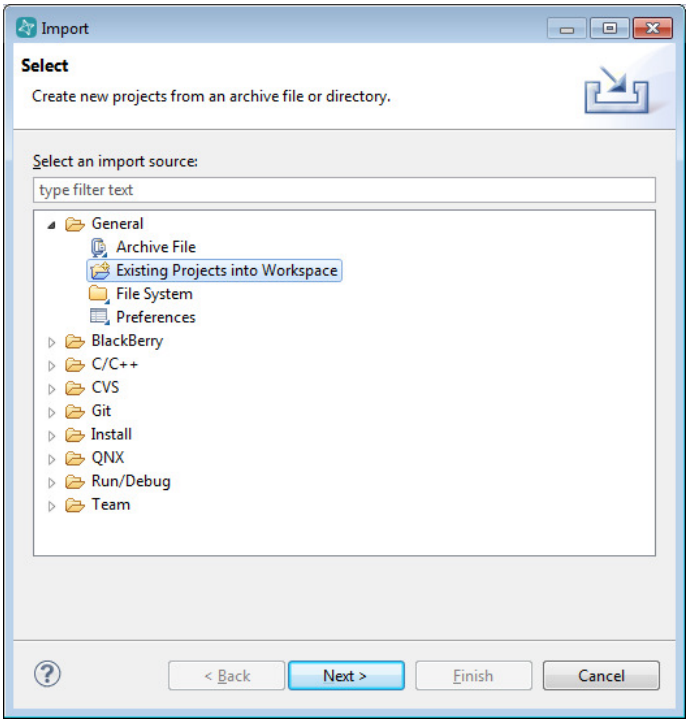
2. If you chose to create a new workspace rather than use an existing one, you will be presented with a view such as this:



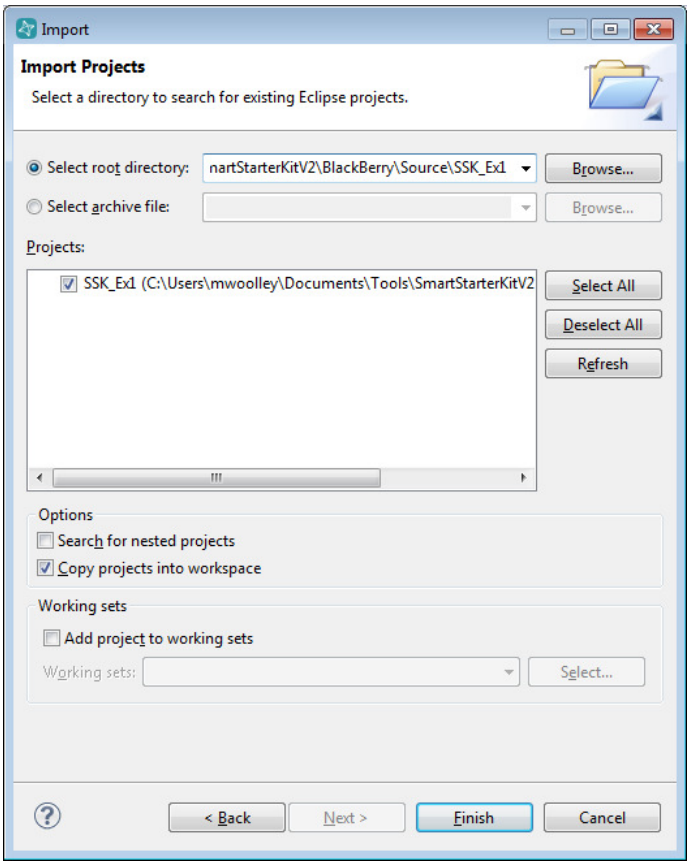
3. Next, import the SSK_Ex1 project provided in this download package in BlackBerry\source\SSK_Ex1 using File->Import:



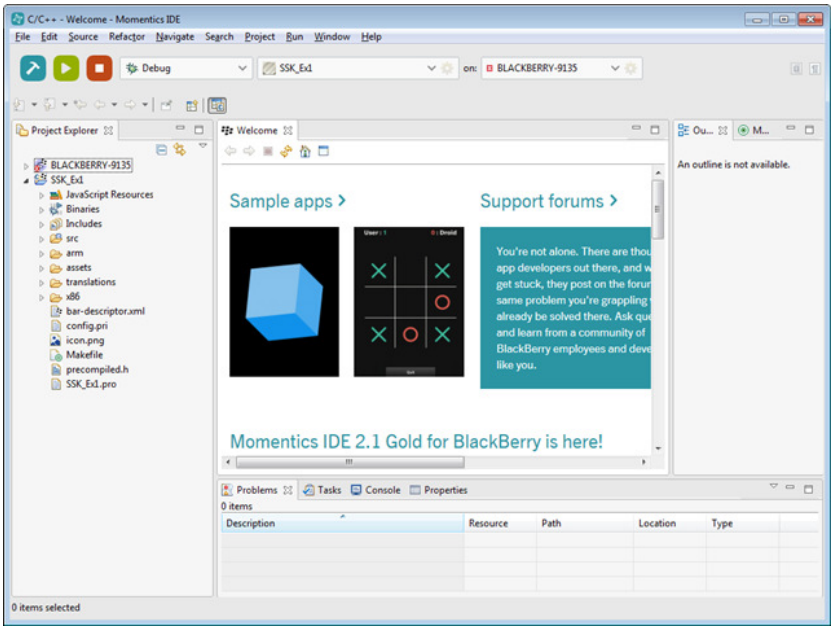
4. Choose to import “Existing Projects into the Workspace”



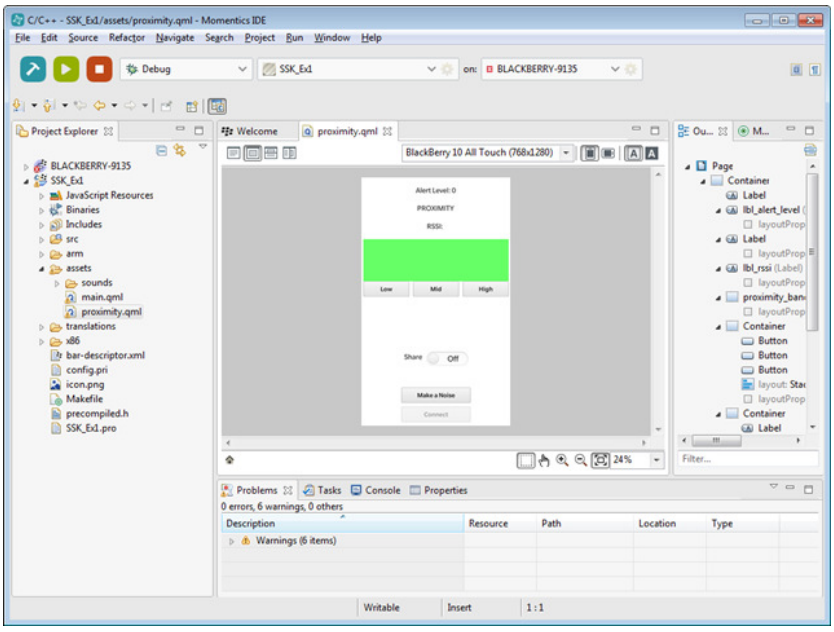
5. Click Next and then at the next dialogue, select “Copy projects into workspace”.



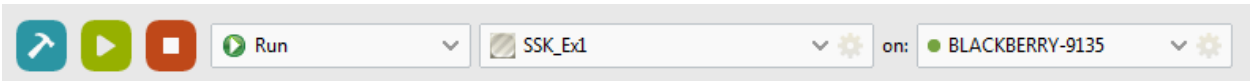
6. The project should now be visible in the Project Explorer pane on the left hand side of the Momentics IDE:



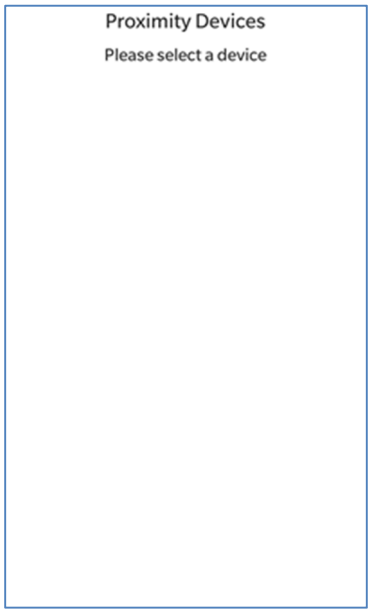
7. Open assets\proximity.qml and select design mode using the second icon in the resultant proximity.qml tabbed pane and you'll see the UI we already prepared for use in this lab.



8. Build the project with CTRL+B, connect your BlackBerry 10 device using USB or wifi and then run the app using the controls at the top of the IDE:



9. The starter application should install onto your BlackBerry 10 device and automatically start. You should see a fairly blank screen looking like this:



If this is the result you got then you're ready to move on to exercise 2.

Exercise 2: Communicating with the Bluetooth Smart Device

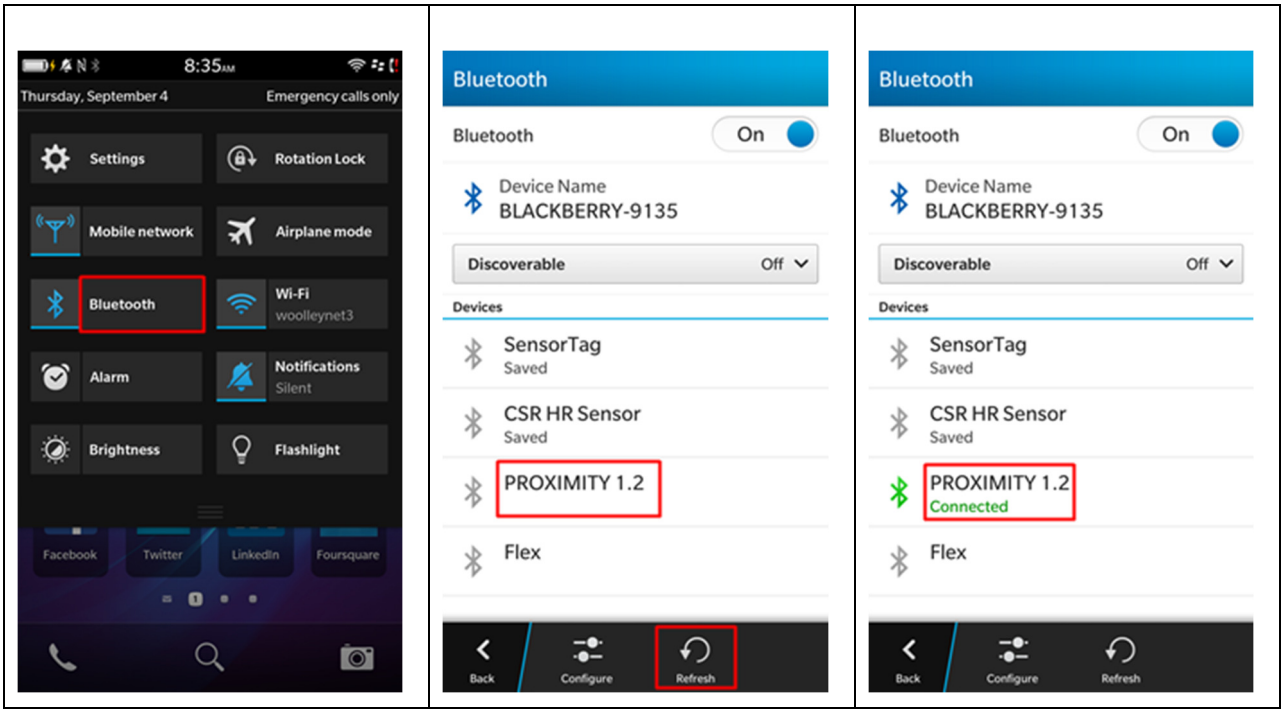
Exercise 2 is all about producing a BlackBerry 10 application which can interact with the Arduino based Bluetooth Smart device which is created in the Arduino lab from this Smart Starter Kit.

Task 2 – Scanning for peripherals

For this task, you will need a BlackBerry 10 device running 10.2.0 or above, your Momentics development machine, and a Bluetooth Smart proximity device, ideally the Arduino based device that is built as part of the Smart Starter Kit, powered up and in range.

The aim of this task is to make changes to our project code so that the first screen lists all Bluetooth Smart proximity devices which you've paired the BlackBerry smart phone with.

Your first job, therefore, is to power on the proximity device (I shall assume you're using the Arduino based device which is described in the Arduino part of this kit), find it in the BlackBerry 10 Bluetooth Settings screen and pair with it.



If you have any problems pairing your device, then clear the bonding keys from the Arduino and delete the proximity device from your BlackBerry 10 Bluetooth Settings screen if it's already there from a previous attempt. Use the BlackBerry 10 Refresh button to scan and find your proximity device again

and attempt to pair once more. Note that to clear Bluetooth bonding keys from the Arduino you need to connect the 3.3V pin to pin 6 and press the reset button. Wait a few seconds, disconnect 3.3v from pin 6 and press reset again. You can monitor progress via the serial monitoring facility of the Arduino Sketch IDE.

Let's review and make some changes to the application next.

In order to use Bluetooth features in an application, we need to ensure the build process can link against the right libraries. We already took care of this for you in the starter project but just so you are aware of how this was done, open the file SSK_Ex1.pro now. The LIBS entry includes –lbtapi which is a flag indicating the library containing the BlackBerry 10 Bluetooth API should be included in the linking process.

SSK_Ex1.pro

```
APP_NAME = SSK_Ex1

CONFIG += qt warn_on cascades10
LIBS += -lbtapi -lbbsystem -lbbdata -lbbplatform
include(config.pri)
```

Open file assets\main.qml. This is the QML page that will list devices we have paired with. Scroll down and you'll see a ListView component within the Page component. We'll shortly make some changes to our C++ code so that this ListView is populated with a list of device names. As you can see, the ListView gets its data from a data model which we refer to as _bt.deviceListing.model.

```
ListView {
    dataModel: _bt.deviceListing.model
    listItemComponents: [
        ListItemComponent {
            type: "listItem"
            StandardListItem {
                title: ListItemData.deviceName
            }
        }
    ]
    onTriggered: {
        var selectedItem = dataModel.data(indexPath);
        proximityPage = proximityDefn.createObject();
        selected_device_address.text = selectedItem.deviceAddress;
        selected_device_name.text = selectedItem.deviceName;
        _bt.setRemoteDevice(selected_device_address.text);
        _app.monitorProximity(selected_device_address.text, selected_device_name.text);
        console.log("QQQQ PUSHING PROXIMITY PAGE");
        navigationPane.push(proximityPage);
    }
```

Open src\applicationui.cpp. You can see in the constructor that we create an object of type BluetoothHandler and register it in our QML context with the name "_bt". So in QML, we can reference this object and any of its member variables which have been exposed to QML using the Q_PROPERTY macro using a prefix of "_bt".

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app)
: QObject(app)
, _handler(new BluetoothHandler(this))
{

    qmlRegisterType<DeviceListing>();
    qmlRegisterType<Timer>("CustomTimer", 1, 0, "Timer");

    QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);

    qml->setContextProperty("_app", this);
    qml->setContextProperty("_bt", _handler);
```

Take a look at BluetoothHandler.hpp. You'll see it includes this Q_PROPERTY.

```
Q_PROPERTY(DeviceListing* deviceListing READ deviceListing CONSTANT)
```

And in DeviceListing.hpp we have:

```
Q_PROPERTY(bb::cascades::DataModel* model READ model CONSTANT)
```

So this is how we're able to reference our DataModel object from QML. If you're an experienced BlackBerry developer you probably already knew this!

Our first coding task is to initialize the Bluetooth library and establish some call back functions and then populate the DataModel object so that our UI gains a list of devices of the appropriate type to offer the user to select from.

Open BluetoothHandler.cpp and locate its constructor.

Underneath the comment that starts "// 1." insert the following code:

```
    // 1. Initialise and scan for proximity devices
        if (!bt_initialised) {
            // Initialize the Bluetooth device and allocate the required resources
            // for the library
            bt_device_init(btEvent);
            // make sure the Bluetooth radio is switched on
            if (!bt_ldev_get_power()) {
                bt_ldev_set_power(true);
            }
            // populate the DataModel with paired proximity devices
            _deviceListing->listPairedDevices();
            _localDeviceInfo->update();
            // register some call back functions for GATT events
```

```
            bt_gatt_init(&gatt_callbacks);
            bt_initialised = true;
        }
```

Open DeviceListing.cpp. Locate the listPairedDevices method and add the following line of code after comment 1.1. This is the call to the BlackBerry Bluetooth API which establishes a list of previously paired devices.

```
    // 1.1 discover devices
       remoteDeviceArray = bt_disc_retrieve_devices(BT_DISCOVERY_PREKNOWN, 0);
```

As you can see from the code we provided, this returns an array of bt_remote_device_t structs which we then iterate through, selecting those which we consider to be proximity devices and inserting details into the Data Model from which our UI list is built.

Let's complete the code which determines whether or not a device is a "proximity device" next.

Still within DeviceListing.cpp, locate comment 1.2 and add the following line of code immediately after it.

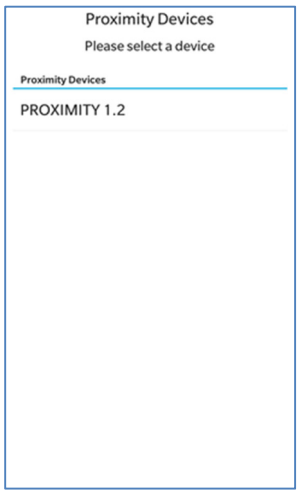
```
    // 1.2 Get list of services this device implements
       char **servicesArray = bt_rdev_get_services_gatt(remoteDevice);
```

And delete the existing line which simply initializes the variably currently:

```
    // delete this line
       char **servicesArray = 0;
```

Bt_rdev_services_gatt obtains an array of the GATT services that the specified remote device supports. We then iterate through this list and check for the existence of the services that matter to us, namely Immediate Alert, Link Loss and TX Power.

Build your application, install and run it. Provided you have already paired with the Arduino proximity device, you should see it (and maybe other devices you have) listed on the first QML page. You can also select the device by touching it and this will cause the second screen to be displayed. If this is the result you get, you're ready for the next task!



Task 3 – Selecting a Device and Starting Monitoring

By touching a device in the Proximity Devices list, you cause the onTriggered event handler in the ListView QML component to be executed. This causes the proximity.qml page to be displayed and a call to a C++ function called monitorProximity to be made. This function is in the applicationui.cpp class but aside from storing the details of the selected device, all it does is call the startProximityMonitoring() function in the BluetoothHandler class. We need to make some changes in this code so that services of interest are connected to.

Find comments 2A – 2D in the startProximityMonitoring function and insert code below each as shown next:

```
// 2A. Connect to the Immediate Alert Service

    /* BEGIN WORKAROUND - Temporary fix to address race condition */
    do {
        function_result = bt_gatt_connect_service(device_addr.toAscii().constData(),
                IMMEDIATE_ALERT_SERVICE_UUID, NULL, &conParm, this);
        retry_count++;
        delay(50);
    } while ((retry_count < 50) && (function_result == -1) && (errno == EBUSY));
```

```
// 2B. Connect to the Link Loss Service

    /* BEGIN WORKAROUND - Temporary fix to address race condition */
    do {
```

```
            function_result = bt_gatt_connect_service(device_addr.toAscii().constData(),
                    LINK_LOSS_SERVICE_UUID, NULL, &conParm, this);
            retry_count++;
            delay(50);
        } while ((retry_count < 50) && (function_result == -1) && (errno == EBUSY));
```

```
    // 2C. Connect to the TX Power Service

        /* BEGIN WORKAROUND - Temporary fix to address race condition */
        do {
            function_result = bt_gatt_connect_service(device_addr.toAscii().constData(),
                    TX_POWER_SERVICE_UUID, NULL, &conParm, this);
            retry_count++;
            delay(50);
        } while ((retry_count < 50) && (function_result == -1) && (errno == EBUSY));
```

```
    // 2D. Connect to the custom Proximity Monitoring Service
        /* BEGIN WORKAROUND - Temporary fix to address race condition */
        do {
            function_result = bt_gatt_connect_service(device_addr.toAscii().constData(),
                    PROXIMITY_MONITORING_SERVICE_UUID, NULL, &conParm, this);
            retry_count++;
            delay(50);
        } while ((retry_count < 50) && (function_result == -1) && (errno == EBUSY));
```

Per the comments in the preceding code fragments, some versions of BlackBerry 10 have a known issue regarding a race condition and so this code implements a retry algorithm to work around it.

As you can see, we make use of various constants defined for each service UUID.

When a service has been successfully connected to, our code will receive a call back to the gatt_service_connected function. We have various things to do in this code, depending on the service that was connected to, but the primary one is to establish handle values for the characteristics we will want to read and write later on. As you can see from the code, we have a function called establishHandles which does just that. We need to finish this function off by inserting some missing API calls however.

```
        if (strcmp(service, IMMEDIATE_ALERT_SERVICE_UUID) == 0) {
            // we connected to the immediate alert service or the link lost service so...
            immediate_alert_connected = true;
            // create a remote BT device structure since we need this for RSSI monitoring
    later
            bt_remote_device_t *bt_device = bt_rdev_get_device(bdaddr);
```

```
            DataContainer *dc = DataContainer::getInstance();
            dc->setCurrentDevice(bt_device);
            establishHandles(INDEX_IMMEDIATE_ALERT_SERVICE, instance);
            return;
        }
```

Locate the establishHandles function and insert code as shown below:

```
    // 3A. Count the number of characteristics the service has
       num_characteristics = bt_gatt_characteristics_count(instance);
```

```
    // 3B. Retrieve a list of all characteristics owned by this service
       number = bt_gatt_characteristics(instance, characteristicList, num_characteristics);
```

Review the remainder of this function and note how we iterate through the list of characteristics and save handle values in a singleton object of type DataContainer for future use.

Also in the gatt_service_connected function, you'll notice that we call startRssiPolling() when we connect to the TX Power service. As the name suggests, this function reads the RSSI value at regular intervals. It does so in a background thread which executes the function pollRssi. We need to make a change there to use the Bluetooth API to read the RSSI value. Locate comment 4 and insert the code shown.

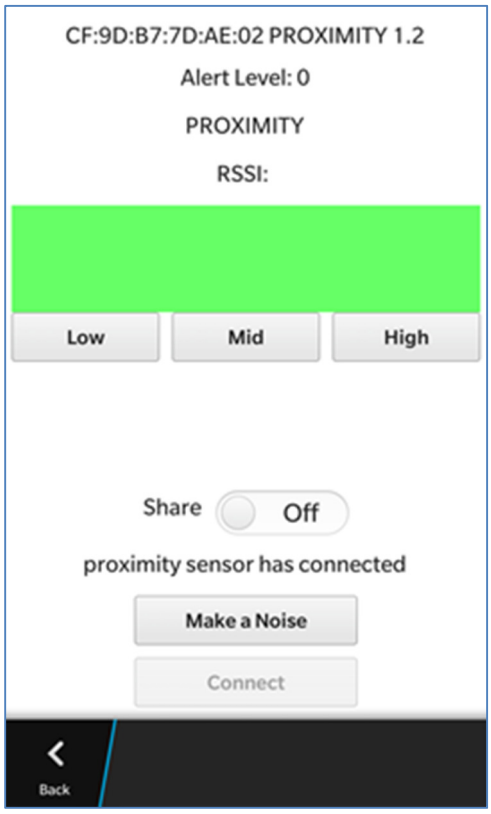
```
    // 4. Read and signal RSSI value
       result = bt_rdev_get_current_rssi(dc->getCurrentDevice(), rssi);
```

Build and run the application on your BlackBerry 10 device.

At this point, our application can find paired Proximity Devices, allow one to be selected for use, connect to GATT services and start receiving RSSI values at intervals.

Task 4 – Set Link Loss Alert Level

The monitoring UI page looks like this:



After a brief delay, you should see a message indicating that the smart phone and Arduino have connected and then RSSI values should start to appear and the rectangular “proximity band” indicator will change colour according to the RSSI values received.
The Low, Mid and High buttons are there to allow the user to set the Alert Level characteristic belonging to the Link Loss service to either 0, 1 or 2 according to the button pressed. Your next task therefore is to complete the code required for this functionality.

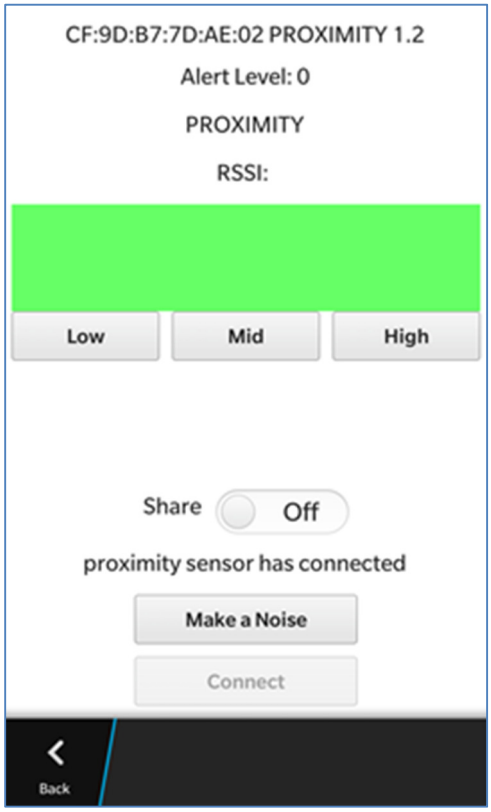
Take a look at the Button components in the proximity.qml file for the Low, Mid and High buttons. You’ll see that they call sendAlertOff(), sendAlertMid() and sendAlertHigh() respectively. These are just functions in the main.qml file whose context includes the C++ objects we’ve registered. Look at main.qml and you’ll see that those functions make calls to the BluetoothHandler object’s functions sendAlertOff(), sendAlertMid() and sendAlertHigh(). Each of these functions call setAlertLevel with an argument indicating the value which the Alert Level characteristic must be set to. Review that function and find comment 5. Add the code below.

```
// 5. write to Link Loss service's alert level characteristic
if (bt_gatt_write_value(ll_alert_service_instance, ll_alert_level_value_handle, 0, level,
    sizeof(level)) == EOK) {
        qDebug() << "XXXX BluetoothHandler::setAlertLevel() - value written successfully";
} else {
        qDebug() << "XXXX BluetoothHandler::setAlertLevel() - errno=(" << errno << ") :"
                    << strerror(errno);
}
```

Here, the bt_gatt_write_value function is the Bluetooth API function for writing to a characteristic.

Build and test your application. You should now find that the Low, Mid and High buttons work. If you've used our Arduino proximity device and bread board circuit, changing the Link Loss service's Alert Level with one of these buttons will cause an LED of the appropriate colour to flash as an ackknlwedgement.

Task 5 – Make a Noise with the Immediate Alert Service



At the foot of our UI we have a button labelled “Make a Noise”. Clicking this button should trigger a response from the Arduino which would help a person find it. The response depends on the value of the Alert Level characteristic that is owned by the Immediate Alert service and this will be the same as was selected with the Low, Mid and High buttons. A low value (0) will cause all three LEDs to flash, whilst a value of 1 or 2 will cause the LEDs to flash and the buzzer to sound in sync with the LED flashing.

Let’s complete the code for this feature now. If you look at proximity.qml and then main.qml you’ll find that clicking the Make a Noise button causes a call to BluetoothHandler.cpp function sendAlertRequest() and you’ll find comment 6 in the sendAlertRequest function of BluetoothHandler.cpp.

Add the code shown.

```
// 6. write to Immediate Alert service's alert level characteristic
      if (bt_gatt_write_value_noresp(ia_alert_service_instance, ia_alert_level_value_handle, 0,
level, sizeof(level)) == EOK) {
          qDebug() << "XXXX BluetoothHandler::sendAlertRequest() - value written successfully";
      } else {
```

```
        qDebug() << "XXXX BluetoothHandler::sendAlertRequest() - errno=(" << errno << ") :"
                    << strerror(errno);
    }
```

Build the application and test the Make a Noise button.

Task 6 – Proximity Sharing

The UI includes a sliding switch labelled “Share”. When this switch is ON, the intention is that it should cause the RSSI value measured by the BlackBerry 10 application and a value indicating the proximity band this corresponds to (1=near, 2=middle distance, 3=far) to be written to a characteristic belonging to a custom service implemented in the Arduino device. The custom service is called the Proximity Monitoring service and its purpose is to allow a connected client to share it’s proximity data back with the GATT server device which in our case, will use coloured LEDs to indicate the value of proximity band the client has shared. The RSSI value could also be shown on an LCD display should one be included in the bread board circuit and the Arduino sketch adjusted.

In this final task we will add the code to share RSSI and “proximity band” values with the Arduino’s custom Proximity Monitoring service when the sharing switch is enabled in the UI.

Take a look at the onRssi method in class applicationui.cpp. As you can see we test a bool *sharing* and if this is true, we make a call to the sendClientProximityData method of our BluetoothHandler class. The sharing bool gets set when the Sharing switch is set to ON of course.

```
void ApplicationUI::onRssi(QVariant proximity_indicator, QVariant rssi) {
    qDebug() << "XXXX onRssi";
    if (sharing) {
        qDebug() << "XXXX sharing proximity data";
        _handler->sendClientProximityData(proximity_indicator.toInt(), rssi.toInt());
    } else {
        qDebug() << "XXXX not sharing proximity data";
    }
    QMetaObject::invokeMethod(_root, "onRssi", Q_ARG(QVariant, proximity_indicator),
Q_ARG(QVariant, rssi));
}
```

Locate the sendClientProximityData method in BluetoothHandler.cpp and within it, comment 7. Add the following code after the comment:

```
// 7. Write to the custom Proximity Monitoring service's client proximity characteristic
if (bt_gatt_write_value_noresp(pm_service_instance,
        pm_client_proximity_value_handle, 0,
            proximity_data, sizeof(proximity_data)) == EOK) {
        qDebug() << "XXXX BluetoothHandler::sendClientProximityData() - value written
successfully";
    } else {
        qDebug() << "XXXX BluetoothHandler::sendClientProximityData() - errno=(" << errno
<< ") :"
                << strerror(errno);
    }
```

All we're doing here is writing to the client proximity characteristic of the custom Proximity Monitoring Service using the API function bt_gatt_write_value_noresp.

Task 7 – Disconnecting and Disconnections

The Back button in the proximity.qml page makes a call to the BluetoothHandler::disconnect() method.

```
            paneProperties: NavigationPaneProperties {
                backButton: ActionItem {
                    title: "Back"
                    onTriggered: {
                        console.log("QQQQ disconnecting then going back");
                        _bt.disconnect();
                        navigationPane.pop();
                    }
                }
            }
```

The intention is that this method disconnects from the various GATT services prior to letting the user choose a different device from the device list on the main page. Find comment 8 in BluetoothHandler.cpp and add this code after it:

```
        bt_gatt_disconnect_service(device_addr.toAscii().constData(),
        IMMEDIATE_ALERT_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(), LINK_LOSS_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(), TX_POWER_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(),
        PROXIMITY_MONITORING_SERVICE_UUID);
```

GATT service disconnections can happen for other reasons however. For example the link may be lost due to the Bluetooth Smart Arduino device going out of range. We'll need to make a few changes for the "link lost" use case.

Find comment 9 and insert the higlighted code within the block shown:

```
            if (event == BT_EVT_LE_DEVICE_DISCONNECTED) {
                immediate_alert_connected = false;
                link_loss_connected = false;
                tx_power_connected = false;
                rssi_polling_required = false;
                DataContainer *dc = DataContainer::getInstance();
                dc->setDeviceConnected(false);
                dc->setAlertServiceInstance(0);
                dc->setLinkLossServiceInstance(0);
                dc->setTxPowerServiceInstance(0);
```

```
            _handler->emitSignalSetMessage("proximity sensor has disconnected");
            if (alarm_on_lost_link) {
                _handler->emitSignalLostLink();
            }
            return;
```

As you can see, we're emiting a signal if a bool variable *alarm_on_lost_link* is true. We need to manage that variable.

Set it to true in this block of code in the btEvent function:

```
        if (event == BT_EVT_LE_DEVICE_CONNECTED) {
            _handler->emitSignalSetMessage("proximity sensor has connected");
            DataContainer *dc = DataContainer::getInstance();
            dc->setDeviceConnected(true);
            alarm_on_lost_link = true;
            _handler->emitSignalLinkEstablished();
            return;
        }
```

And set it to false in this block of code in the btEvent function:

```
    void BluetoothHandler::disconnect() {
        qDebug() << "XXXX BluetoothHandler::disconnect()";
        DataContainer* dc = DataContainer::getInstance();
        QString device_addr = dc->getCurrentDeviceAddr();
        alarm_on_lost_link = false;
        bt_gatt_disconnect_service(device_addr.toAscii().constData(),
    IMMEDIATE_ALERT_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(),
    LINK_LOSS_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(), TX_POWER_SERVICE_UUID);
        bt_gatt_disconnect_service(device_addr.toAscii().constData(),
    PROXIMITY_MONITORING_SERVICE_UUID);
    }
```

Task 9 – Deinitialise

When your application shuts down, you should perform a few clean up operations. Find comment 10 and add this code in the BluetoothHandler destructor:

```
        bt_device_deinit();
        bt_gatt_deinit();
```

Task 10 – Test your Application

You're ready to take your BlackBerry 10 app for a test drive! Here's a summary of the expected behaviours for you to check when testing:

Feature	Expected Behaviour
Scan Button	finds your Arduino key fob device which should have the name "Proximity 1.2".
Connect	connects to the Arduino using Bluetooth Smart
Select any of the Low, Mid or High buttons	Write 0, 1 or 2 to the Link Loss service Alert Level characteristic in the Arduino. The corresponding LED (0=green, 1=yellow, 2=red) on your board should flash 4 times as an acknowledgement.
Make a Noise	The Alert Level characteristic of the Immediate Alert service should be written to using the value last selected by pressing one of Low, Mid or High. If the alert level is 1 or 2, the board should respond by beeping 5 times and flashing all three LEDs in unison. If 0 then the LEDs should flash but the buzzer should be silent.
Sharing ON	The board's LED of the same colour as the proximity rectangle on the UI should be illuminated. The buzzer will be silent. As you move the phone away or towards the Arduino, as the rectangular proximity indicator changes value, the illuminated LED should change value after a short delay.
Sharing OFF	All LEDs should be switched off unless enabled by some other action.
Link Lost / Disconnected	The phone and board should make a noise for an extended period of time and all LEDs should flash.

Summary of Exercise 2

You now have all the pieces to run the full demonstration scenario.

Well done! *By following this hands-on lab you have created an application that scans for Bluetooth Smart devices, scans their services, connects to them, reads and writes characteristic values, and monitors the RSSI value. It can also share RSSI and proximity band data back with the remote Bluetooth Smart Arduino device using a custom service.*