Chapter 6A: Classic Bluetooth – The Wireless Serial Port Profile (SPP)

Time: 2 ½ Hours

At the end of this chapter you will understand the basics of Classic Bluetooth and how to create a simple Classic Bluetooth application on WICED devices. This section is focused on the simplest Bluetooth connection, one Master (Android, Mac or PC) and one Slave (your WICED Bluetooth Device). By the end you should understand Inquiry, Page, Pair, Bond, SDP, L2CAP, RFCOMM and the Serial Port Profile (SPP).

6A.1 WICED Bluetooth Classic System Lifecycle Overview

The Bluetooth Classic Spec has a bewildering amount of complexity. Clearly this must have been one of the motivations for creating the much simpler BLE standard. Like Chapter 4 we will take the approach of creating the simplest example application possible to get things going.

The simplest Bluetooth Classic scenario has two devices, a Master and a Slave. Slaves are passive – not transmitting – until they hear an Inquiry broadcast from a Master, at which point the Slave broadcasts basic information about itself (Name, BDADDR, Services). The Master then Pages (connects) to the Slave and discovers the Services - i.e. the capabilities of the Slave. If the Master is interested, they will establish a secure link which includes Pairing on the first connection. Finally, a service level connection is established which in the simplest case is the Serial Port Profile.

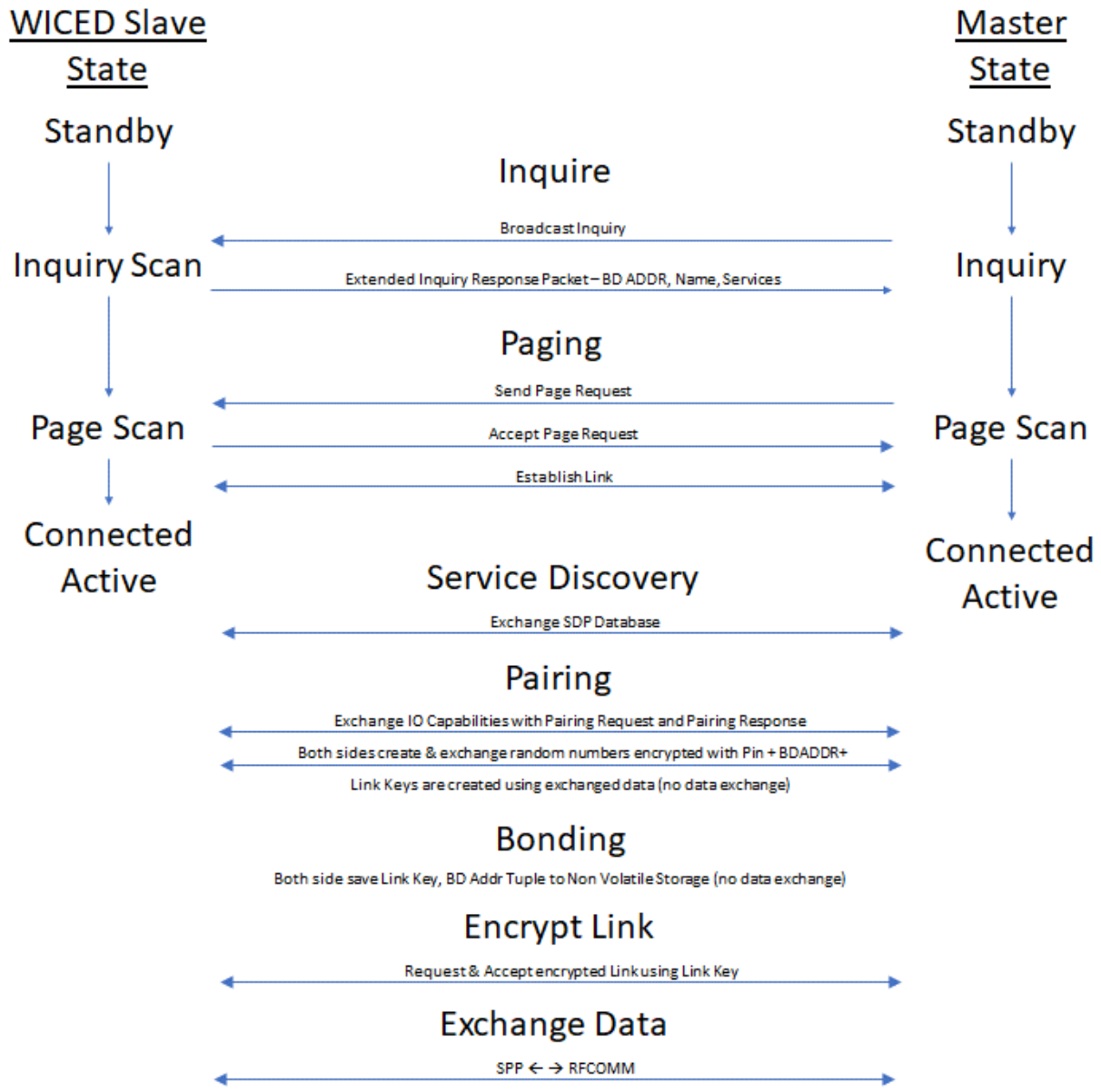
The five steps are:

1. Inquiry – Master finds a Slave to Connect
2. Paging – Master connects to Slave
3. Service Discovery (SDP) – The Master figures out what the Slave can do
4. Pair & Bond – A secure, authenticated connection is created
5. Establish SPP connection and Exchange Data using the Serial Port Profile

The architecture of a Bluetooth Classic device is essentially the same as that of a BLE device. It is composed of five layers.

	Application	The code that you write to implement your system functionality.
Bluetooth Classic Stack	Profile Library	Source libraries including implementation of standard Bluetooth Profiles such as SPP.
	Host Stack	Provides multiple connection paths to the application each with its own features (reliable, ordered, time critical, etc.). It also provides Services to the local and remote application.
	Controller	Establishes and maintain links between devices.
Hardware	Radio	RF magic & the best reason to use Cypress chips.

Here is the overall picture of the simplest Bluetooth Classic system:



6A.1.1 Inquiry

The purpose of the Inquiry process is for a Bluetooth Master to find all the Bluetooth Slaves that are within its radio range that might provide some interesting Service. This is exactly the opposite of BLE where a Peripheral advertises it availability and the BLE Central Scans for those packets.

A Bluetooth Classic Slave sits in state called Inquiry Scan - i.e. a listening only state - until it hears a Bluetooth Master broadcast an Inquiry Request message. The Slave Application is responsible for putting the Stack into the Inquiry Scan state using the correct Stack API.

Upon hearing an Inquiry Request the Slave will broadcast an Extended Inquiry Response (EIR) packet that contains its Name, Bluetooth Address (BDADR) and list of Services. These responses are handled completely by the Controller part of the Stack - i.e. your Application is not aware of these Inquiry Requests happening.

You should be aware that because of the vagaries of the Bluetooth Radio frequency hopping scheme, these Inquires may take several seconds.

6A.1.2 Page / Connect

The Paging process is used for a Bluetooth Master to connect to a Bluetooth Slave. The Master is "Paging" the Slave device (remember the old school pagers?).

A Bluetooth Classic Slave sits in state called Page Scan - i.e. a listening only state - until a Bluetooth Master initiates the connection process by sending a Page Request. The Slave is responsible for putting the Stack into the Page Scan state using the correct Stack API.

A Slave can - and often will be - in both the Page Scan and Inquiry Scan modes at the same time, meaning a Master can initiate a connection to a Slave without Inquiring if it already knows of the existence of the Slave from a previous connection.

6A.1.3 Discover the Services using Service Discovery Protocol (SDP)

A simple conceptual model of a Bluetooth Classic device is a Server that is running one or more Services that are attached to Ports. This is the same model that we use in IP Networking.

One question that arises from this idea is: "How do I figure out what Services are available and what Port each one is listening on?". The answer to both questions is the Service Discovery Protocol.

The SDP has a database embedded in it that contains a list of Services and what Port each one is running on. The SDP Protocol allows both sides of a connection to query the SDP database.

More details on this in section 6A.2

6A.1.4 Pair & Bond

The whole Bluetooth communication system depends on having a shared symmetric encryption key called the Link Key. Bluetooth Classic uses a process called Secure Simple Pairing that exchanges enough

information for the Link Key to be created. (There are other legacy Pairing methods, but they are largely obsolete at this point).

The Secure Simple Pairing process was designed to minimize the chances that the communication link could be compromised by an eavesdropper or by a man-in-the-middle. The process is the same as BLE.

As with BLE, Bonding is just saving the BDADDR/Link Key into non-volatile memory so that it can be reused to speed up re-initiating a connection.

I'll talk about this process in more detail in a minute in section 6A.3

6A.1.5 Exchange Data with the Serial Port Profile

Once Service Discovery is complete, the Bluetooth Master knows the Port number that it should use to connect to the Serial Port Profile (SPP). The SPP is just one of these Servers (from the last section) that acts like a serial port. You put bytes in one side and they come out the other.

The Bluetooth Master then opens a connection to the SPP Server running on the Bluetooth Slave. At this point you can commence the final step in your first basic application: actually exchanging data.

Again, we'll talk about this in much more detail in section 6A.3

6A.2 Service Discovery Protocol (SDP)

From the Bluetooth Core Spec – "The service discovery protocol (SDP) provides a means for Applications to discover which Services are available and to determine the characteristics of those available services." The SDP sits on top of the L2CAP layer – and when communicating generates a bunch of L2CAP traffic.

The Bluetooth SIG specifies the SDP database format in Volume 3 Part B of the Bluetooth Core Spec. The database is composed of one or more Service Records each containing one or more Service Attributes. Each Service Attribute is a Key/Value pair. There are several Bluetooth SIG Specified Service Attributes, but you can also create custom Attributes.

Some of the legal Attributes include:

ServiceRecordHandle – A 32-bit number uniquely identifying that Service in the SDP.

ServiceClassIDList – Identifies what type of Service this record represents, specifically a list of classes of Service.

ProtocolDescriptorList – A list of the protocol stacks that may be used to access this Service.

ServiceName – A plain text description of the Service.

The SDP provides the means for the Client to Search for Services and Attributes and request the values of the same.

6A.3 Secure Simple Pairing

Classic Bluetooth has the same four Pairing methods as BLE:

Method 1 is called "Just works". In this mode you have no protection against MITM.

Method 2 is called "Out of Band". Both sides of the connection need to be able to share the PIN via some other connection that is not Bluetooth such as NFC.

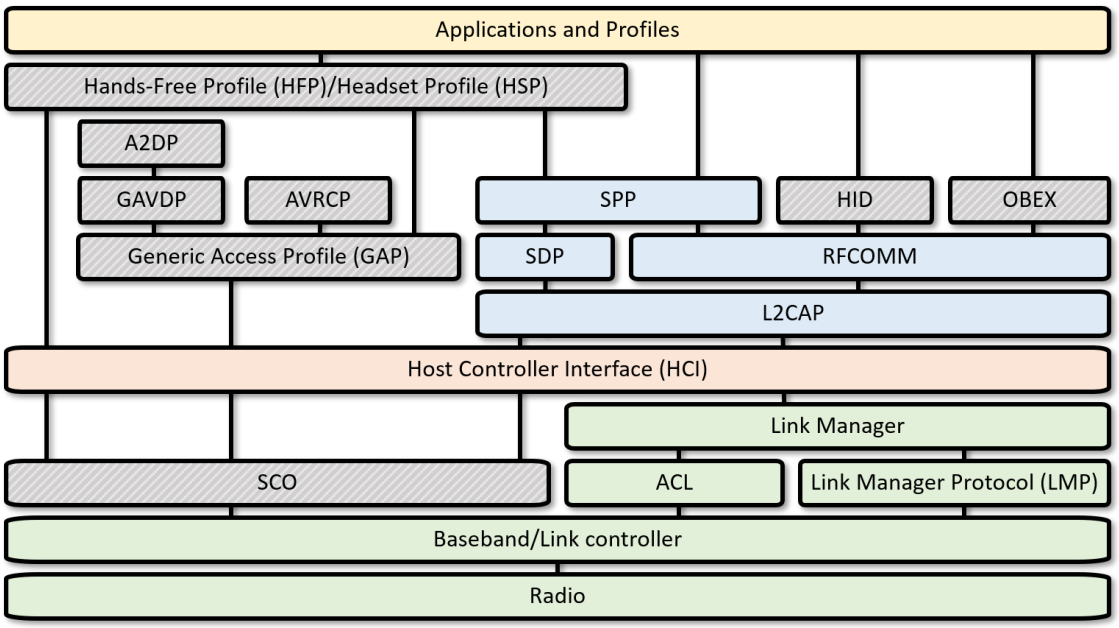
Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on the random numbers used to generate the shared key. The user observes both devices. If the number is the same on both, then the user confirms on both sides. If there is a MITM, then the random numbers on both sides would be different so the 6-digit codes would not match.

Method 4 is called "Passkey Entry" (V2.PH.7.2.3). For this method to work, at least one side needs to be able to enter a 6-digit numeric code. The other side must either be able to display a code that is randomly generated or else have the ability to enter the same code. In the latter case, the user chooses their own random code that is entered on both sides. Then, an exchange and comparison process starts with the Passkeys being divided up, encrypted, exchanged and compared with the other side.

6A.4 L2CAP, RFCOMM & the Serial Port Profile

The Bluetooth Classic system has a stack of software and hardware built into it. For the purposes of this simple Bluetooth Classic example, three blocks in the Host are relevant: L2CAP, RFCOMM and the Serial Port Profile.

You can see the three blocks in this simplified diagram of the Stack.



6A.4.1 L2CAP

L2CAP is an acronym that stands for Logical Link Control and Adaptation-layer Protocol. L2CAP has one main function in the system: it serves as a data packet multiplexor that lets you have multiple streamed connections from the higher level going into one interlaced set of packets going out the Radio. It obviously implements the de-multiplexor function as well, taking a single stream of interlaced packets and turning it back into complete streams on the other side of the link.

The L2CAP divides up the streams of data into L2CAP Channels that:

1. Divides up streams of data into smaller packets that will fit through the Radio.
2. Provides quality of service to each of the L2CAP channels.
3. Provides flow control.

6A.4.2 RFCOMM

RFCOMM was built as a wired RS232 replacement protocol. It supports all the normal wires for a serial port including Rx, Tx, CTS, RTS, DSR, DTR, CD and Ri. Depending on the implementation, RFCOMM gives you up to 60 Server Channels of streams of serial data. The protocol is built on top of L2CAP (a packet-based system). It appears to the Application developer with an API that makes it look like a UART.

6A.4.3 Serial Port Profile

The Serial Port Profile specifies all the protocols and procedures required to setup, discover and connect two virtual serial ports over an RFCOMM connection. If you are replacing a serial port interface like RS-232 or a UART with Bluetooth, then SPP is the profile you are looking for.

FYI, for iOS devices, the SPP is locked so it is only usable for MFi license holders. Their implementation is called iAP2.

6A.5 WICED Bluetooth Stack Events

The Stack generates Events based on what is happening in the Bluetooth world. After an event is created, the Stack will call the callback function which you registered when you turned on the Stack. Your callback firmware must look at the event code and the event parameter and take the appropriate action.

For your Basic Application these are the relevant BTM Events:

Event	Description
BTM_ENABLED_EVT	When the Stack has everything going. This event data will tell if you it happened with WICED_SUCCESS or !WICED_SUCCESS. This is typically where you will launch most of your application code.
BTM_SECURITY_REQUEST_EVT	For BLE, this is used to retrieve the local identity key for RPA. For Classic BT you don't need to do anything for this event.
BTM_PAIRING_IO_CAPABILITIES_BR_EDR_REQUEST_EVT	The Stack is asking what IO capabilities this device has (Display, Keyboard etc.). You need to update the structure sent to you in the event data.
BTM_PAIRING_COMPLETE_EVT	The Stack is informing you that you are now paired.
BTM_ENCRYPTION_STATUS_EVT	The Stack is informing you that the link is now encrypted…or not depending on the event data.
BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT	The Stack is asking you find and return the link key for the BDADDR that was sent in the event data.
BTM_USER_CONFIRMATION_REQUEST_EVT (for numeric comparison bonding method)	The Stack is asking you to ask the user if the PIN you are displaying matches the PIN from the other side. This state should print the passkey (e.g. to UART or some other display). You can allow the user to verify the key only on the other side, or you can verify the user's input here before sending back the confirmation.
BTM_PASSKEY_NOTIFICATION_EVT (for passkey entry method)	The Stack is notifying you that the other side of the connection wants a passkey. You should print the passkey (e.g. to UART or some other display) so that the user can enter it on the other device.
BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	The Stack is asking you to read the local identify keys from the NVRAM and return them to the Stack.
BTM_PAIRING_IO_CAPABILITIES_BR_EDR_RESPONSE_EVT	The Stack is informing you of the I/O capabilities of the other side of the connection.
BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT	The Stack is asking your firmware to store the BDADDR/Link Keys (which are passed in the event data).

6A.6 WICED Classic Bluetooth Firmware Architecture

In the exercises, you will use and modify a code example that demonstrates the Bluetooth SPP profile. This section briefly explains the architecture of that example. The example is contained in the starter application called "RFCOMM-20819EVB02".

6A.6.1 Overview

There are two main files that configure/implement the Bluetooth SPP functionality. They are:

1. spp.c which contains the following functions:
    a. *APPLICATION_START* which is the entry point for the firmware.
    b. *application_init* which provides a place for you to get your application stuff going.
    c. *app_management_callback* which is a template BTM event handler function.
    d. Various other helper functions.
2. A file called wiced_bt_cfg.c containing:
    a. All the basic Bluetooth configuration settings to get the stack going.
    b. Configuration of the buffer pools.

6A.6.2 Application Code (spp.c)

As mentioned above, the *APPLICATION_START* function is the entry point of the firmware. By default, that function will:

- Initialize the memory pools (just like BLE)
- Configure the debugging UART to allow you to see WICED_BT_TRACE messages
- Call wiced_bt_stack_init with the event handler to start the stack

The *application_init* function is created for you as a place to initialize your application. It is called in the BTM event handler after the stack starts. By default, this function:

- Starts the SPP service
- Initializes the SDP database
- Makes your device pairable
- Makes your device connectable (turns on Page Scan)
- Makes your device discoverable (turns on Inquiry Scan)

You will typically add your own application startup functionality to this function. For example, you may initialize hardware, start periodic timers or create threads to handle your application's needs.

6A.6.3 The Serial Port Profile

To make the SPP work a few things need to take place. The server needs to be initialized and callbacks need to be provided for starting/stopping the connection as well as receiving data.

makefile

Notice that spp_lib is included. This is the middleware library that contains all the lower level functions for the SPP server.

```
#
# Components (middleware libraries)
#
COMPONENTS += spp_lib
.
.
.
# paths to shared_libs targets
SEARCH_LIBS_AND_INCLUDES=$(CY_BSP_PATH) $(CY_BASELIB_PATH)
$(CY_SHARED_PATH)/dev-kit/libraries/btsdk-rfcomm
```

spp.c

There are four key global variables to support SPP. These are:

1. A uint16_t called spp_handle which holds the current handle of the spp connection.

```
uint16_t    spp_handle;
```

2. A structure of type wiced_bt_spp_reg_t called spp_reg which holds all the configuration information for the SPP Server.

```
wiced_bt_spp_reg_t spp_reg =
{
    SPP_RFCOMM_SCN,                 /* RFCOMM service channel number for SPP connection */
    MAX_TX_BUFFER,                  /* RFCOMM MTU for SPP connection */
    spp_connection_up_callback,     /* SPP connection established */
    NULL,                           /* SPP connection establishment failed, not used because
                                       this app never initiates connection */
    NULL,                           /* SPP service not found, not used because this app never
                                       initiates connection */
    spp_connection_down_callback,   /* SPP connection disconnected */
    spp_rx_data_callback,           /* Data packet received */
};
```

3. An array of type uint8_t called app_sdp_db that holds the actual SDP database.

```
const uint8_t app_sdp_db[] = // Define SDP database
{
    SDP_ATTR_SEQUENCE_2(142),
    SDP_ATTR_SEQUENCE_1(69),                                                    // 2 bytes
        SDP_ATTR_RECORD_HANDLE(0x10003),                                        // 8 bytes
        SDP_ATTR_CLASS_ID(UUID_SERVCLASS_SERIAL_PORT),                          // 8
        SDP_ATTR_RFCOMM_PROTOCOL_DESC_LIST( SPP_RFCOMM_SCN ),                   // 17 bytes
        SDP_ATTR_BROWSE_LIST,                                                   // 8
        SDP_ATTR_PROFILE_DESC_LIST(UUID_SERVCLASS_SERIAL_PORT, 0x0102),         // 13 byte
        SDP_ATTR_SERVICE_NAME(10),                                              // 15
        'S', 'P', 'P', ' ', 'S', 'E', 'R', 'V', 'E', 'R',

    // Device ID service
    SDP_ATTR_SEQUENCE_1(69),                                                    // 2 bytes, length of the record
        SDP_ATTR_RECORD_HANDLE(0x10002),                                        // 8 byte
        SDP_ATTR_CLASS_ID(UUID_SERVCLASS_PNP_INFORMATION),                      // 8
        SDP_ATTR_PROTOCOL_DESC_LIST(1),                                         // 18
        SDP_ATTR_UINT2(ATTR_ID_SPECIFICATION_ID, 0x103),                        // 6
        SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID, 0x0f),                                // 6
        SDP_ATTR_UINT2(ATTR_ID_PRODUCT_ID, 0x0401),                             // 6
        SDP_ATTR_UINT2(ATTR_ID_PRODUCT_VERSION, 0x0001),                        // 6
        SDP_ATTR_BOOLEAN(ATTR_ID_PRIMARY_RECORD, 0x01),                         // 5
        SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID_SOURCE, DI_VENDOR_ID_SOURCE_BTSIG) // 6
};
```

A uint16_t called app_sdp_db_len that holds the size of the SDP database.There is a function to start up the SPP Server. It is called with the configuration structure defined above as its only argument.

```
// Initialize SPP library
wiced_bt_spp_startup(&spp_reg);
```

There are connection up and down callbacks provided in the configuration structure. These callbacks send information via the BT Trace and set/unset a global variable that keeps track of the SPP handle.

```
/*
 * SPP connection up callback
 */
static void spp_connection_up_callback(uint16_t handle, uint8_t* bda)
{
    WICED_BT_TRACE("%s handle:%d address:%B\n", __FUNCTION__, handle, bda);
    spp_handle = handle;
}

/*
 * SPP connection down callback
 */
static void spp_connection_down_callback(uint16_t handle)
{
    WICED_BT_TRACE("%s handle:%d\n", __FUNCTION__, handle);
    spp_handle = 0;
}
```

When data is received it is just dumped out onto the screen in the RX data callback that was specified in the configuration structure. Note that there is a macro called LOOPBACK_DATA that you can define if you want the RX function to re-transmit the data it receives. By default it is disabled.

```
/*
 * Process data received over EA session.  Return TRUE if we were able to allocate buffer to
 * deliver to the host.
 */
static wiced_bool_t spp_rx_data_callback(uint16_t handle, uint8_t* p_data, uint32_t data_len)
{
    int i;
```

```
//    wiced_bt_buffer_statistics_t buffer_stats[4];

//    wiced_bt_get_buffer_usage (buffer_stats, sizeof(buffer_stats));

//    WICED_BT_TRACE("0:%d/%d 1:%d/%d 2:%d/%d 3:%d/%d\n", buffer_stats[0].current_allocated_count,
buffer_stats[0].max_allocated_count,
//                   buffer_stats[1].current_allocated_count, buffer_stats[1].max_allocated_count,
//                   buffer_stats[2].current_allocated_count, buffer_stats[2].max_allocated_count,
//                   buffer_stats[3].current_allocated_count, buffer_stats[3].max_allocated_count);

//    wiced_result_t wiced_bt_get_buffer_usage (&buffer_stats, sizeof(buffer_stats));

    WICED_BT_TRACE("%s handle:%d len:%d %02x-%02x\n", __FUNCTION__, handle, data_len, p_data[0],
p_data[data_len - 1]);

#if LOOPBACK_DATA
    wiced_bt_spp_send_session_data(handle, p_data, data_len);
#endif
    return WICED_TRUE;
}
```

The code example has two functions that send data: app_send_data which sends data on an interrupt; and app_timeout which sends data whenever a timer expires. We will disable these in our exercises so that we can control sending of data from a UART terminal, but those two functions are instructive on how to send data. Namely, you need to call the function wiced_bt_spp_send_session_data with the handle to the SPP service, a pointer to the data to send, and the length of the data in bytes.

wiced_bt_cfg.c

The last thing to notice is the buffer pool settings. SPP uses more memory than most BLE designs, so some of the values are larger than usual. Specifically, the MTU is set in the config file to 515 so the large buffer pool must be at least 527 (MTU + 12).

```
const wiced_bt_cfg_buf_pool_t wiced_bt_cfg_buf_pools[WICED_BT_CFG_NUM_BUF_POOLS] =
{
/*  { buf_size, buf_count, }, */
    { 64,       16,        }, /* Small Buffer Pool */
    { 360,      10,         }, /* Medium Buffer Pool (used for HCI & RFCOMM control messages, min
recommended size is 360) */
    { 1056,     3,         }, /* Large Buffer Pool  (used for HCI ACL messages) */
    { 1056,     1,         }, /* Extra Large Buffer Pool (used for SDP Discovery) */
};
```

6A.7 Exercises

Exercise - 6A.1 Create a Serial Port Profile Application

Application Creation

For this example, you will need to:

1. Use the SPP code example from the starter application RFCOMM-20819EVB02 to create an application called **ch06a_ex01_spp**.
2. Comment out lines 90 and 91 in spp.c to disable sending data on interrupts and on timer timeouts. For this exercise, we will only investigate using RX.
3. Open the makefile and set BT_DEVICE_ADDRESS to random.
4. Change the name of your device by editing the variable BT_LOCAL_NAME in the wiced_bt_cfg.c file.
    a. Hint: The default name is “spp test”, remember to use your initials in the device name so that you can find it in the list of devices that will be advertising (e.g. spp <inits>).
    b. Hint: The value must be null terminated so keep '\0' at the end of the array.
5. Review the file spp.c to familiarize yourself with the way that everything is configured.

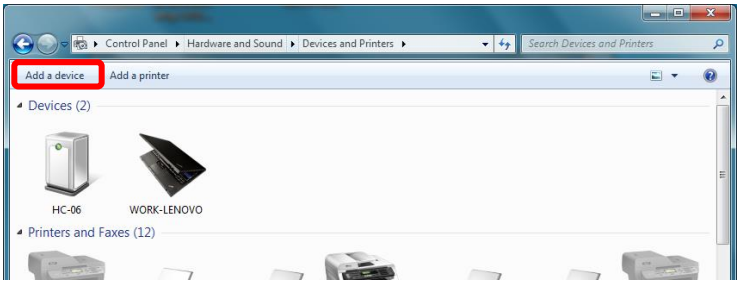
Testing

Once your application has been built and programmed to the kit, you can attach to it using Windows 7, Windows 10, MacOS or Android. Instructions for each are provided below.

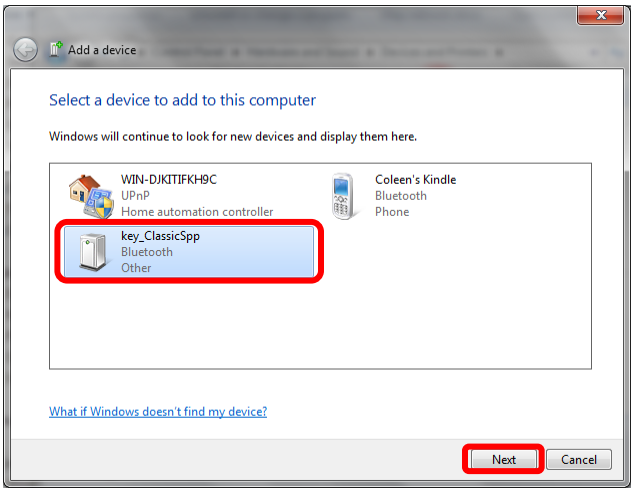
Note that iOS does not support SPP directly, so you can't use an iPhone to test this application. Apple supports Classic Bluetooth with iAP2 (iPod Accessory Protocol) which works a bit differently than SPP and requires an MFi license.

PC Instructions (Windows 7)

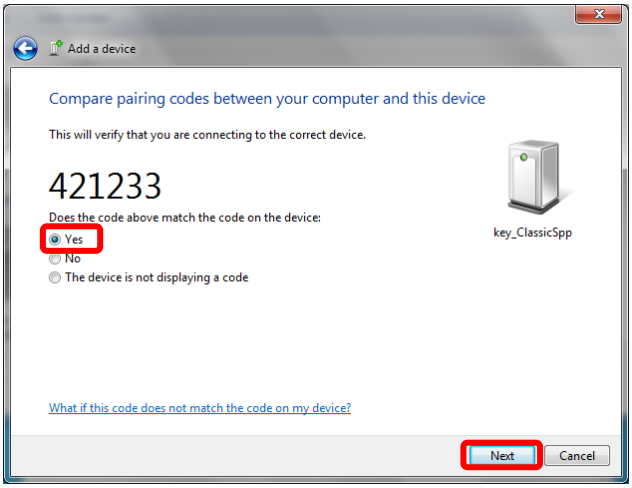
The first step is to pair your PC with the WICED Bluetooth device. Go to Control Panel -> Hardware and Sound -> Devices and Printers -> Add a Device.



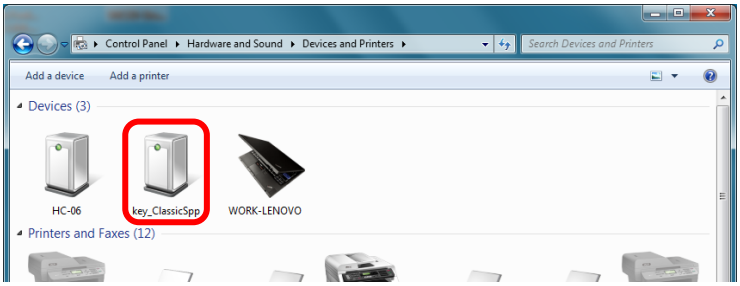
Wait until your device shows up in the list. Select it and click "Next".



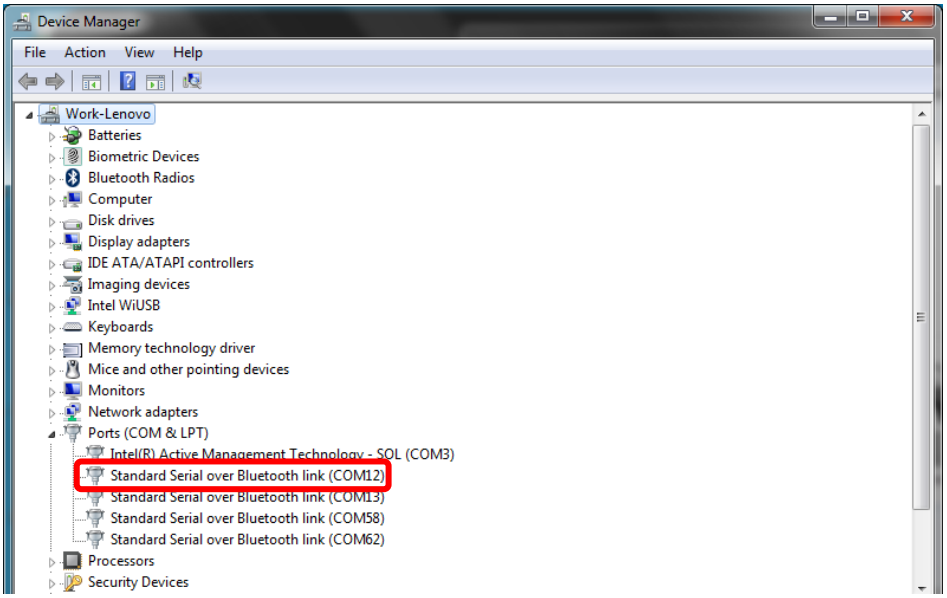
Compare the 6-digit code with the one displayed on your UART terminal. If the two numbers match, make sure "Yes" is selected and click "Next".



Click "Close" once the device has been added. Your device will now show up in the list of devices and drivers will automatically install.



Go to the Device Manager and look under Ports (COM & LPT) to find the COM port for the SPP interface of your Bluetooth device. It will be listed as "Standard Serial over Bluetooth link". If you see multiple ports listed for Standard Serial over Bluetooth link, the lowest numbered port is the one you want to use.



Open a serial terminal program of your choice (such as Putty) and connect to the SPP COM port. Now you can type in characters in the terminal window for the Bluetooth device and you will see them being received in the WICED kit by watching in terminal window connected to the kit's PUART. The WICED kit will display a line like the following with the number of characters received specified by "len=":

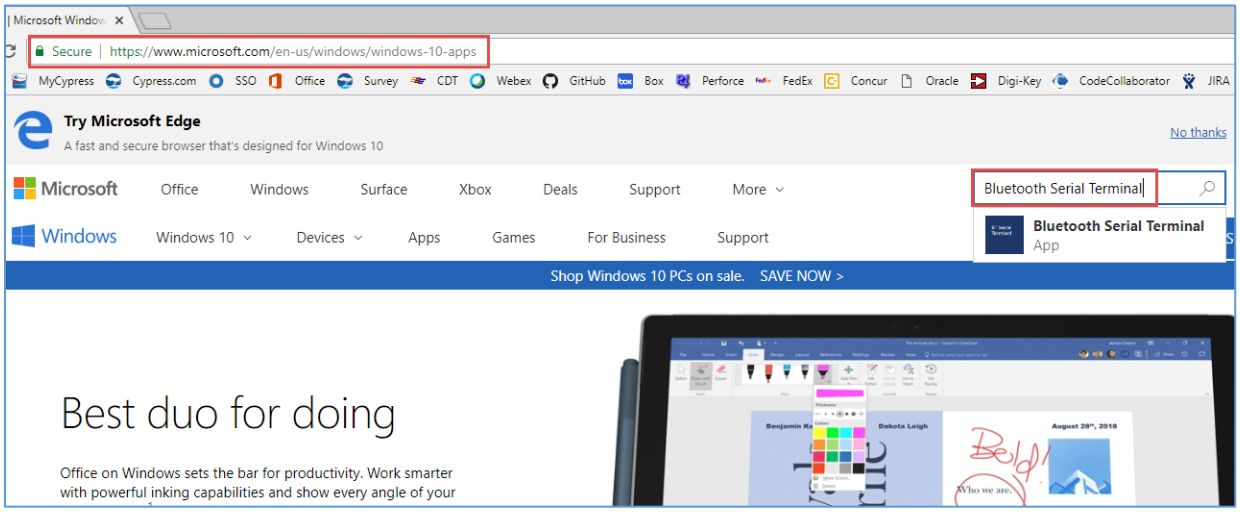
```
spp_rx_data_callback handle:2 len:2 78-78
```

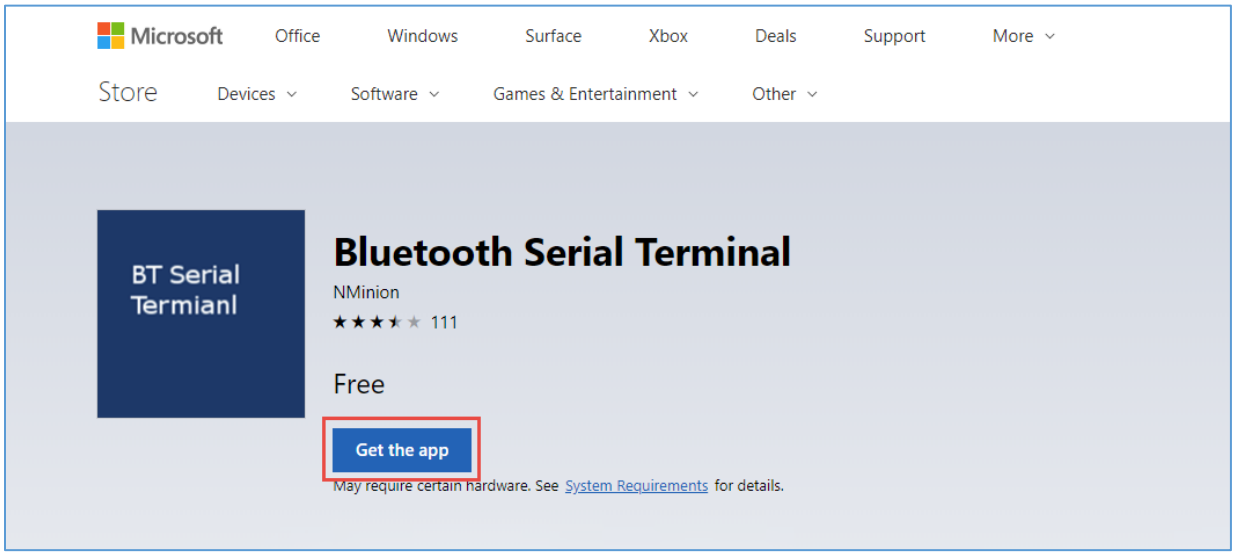
When you are done testing, close the Bluetooth SPP terminal window and then go to Control Panel -> Hardware and Sound -> Devices and Printers. Right click on the WICED Bluetooth device, select "Remove Device" and click "Yes" to remove the device's paring information.

PC Instructions (Widows 10)

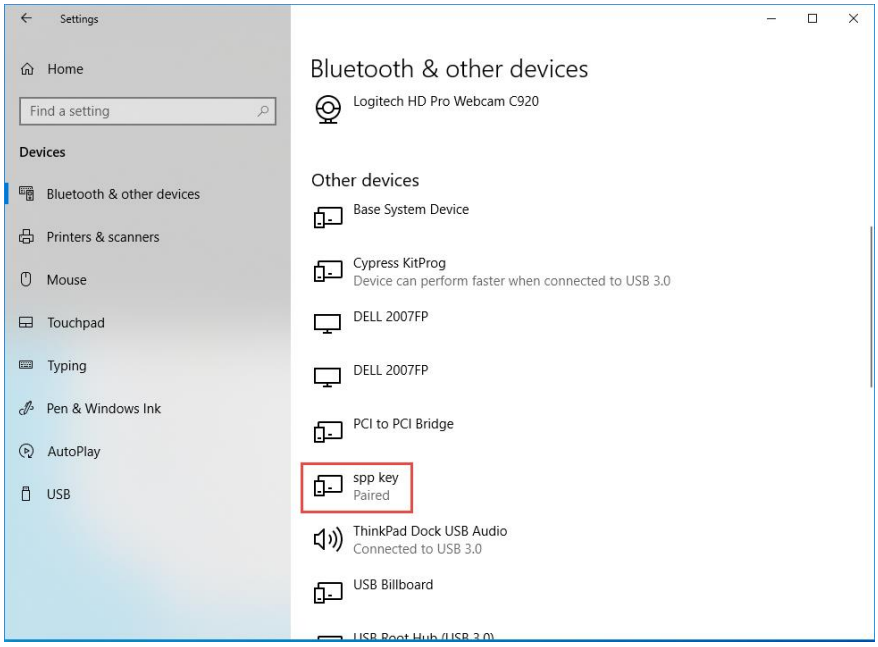
For Windows 10 you can use the same procedure as for Windows 7. Alternately, in Windows 10 you have the option to install the "Bluetooth Serial Terminal" from the Microsoft App Store which provides a "slick" interface. That option is discussed here.

First, go to the Windows 10 Apps store (https://www.microsoft.com/en-us/windows/windows-10-apps), search for "Bluetooth Serial Terminal", and install it.

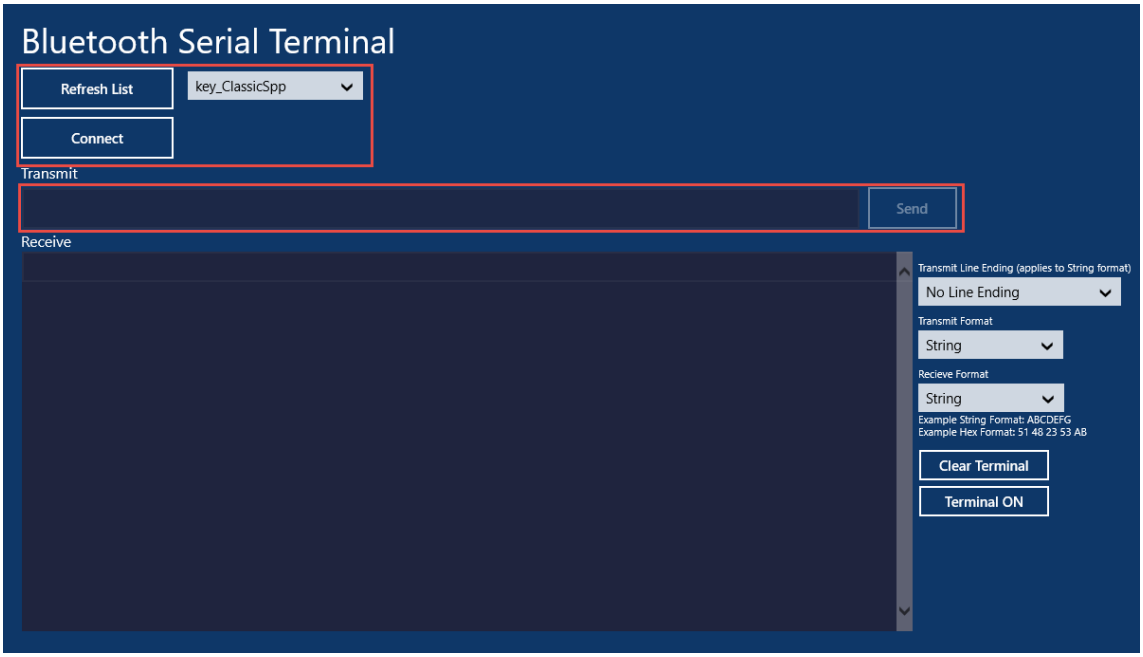




As with Windows 7, you need to pair with your device before it will show up as a serial port. To do this, go to *Settings -> Devices -> Add Bluetooth or other device -> Bluetooth*. When you see your device in the list, click on it. Click on "Connect" and "Done". Your device should now show up in the list of Other devices as "Paired":



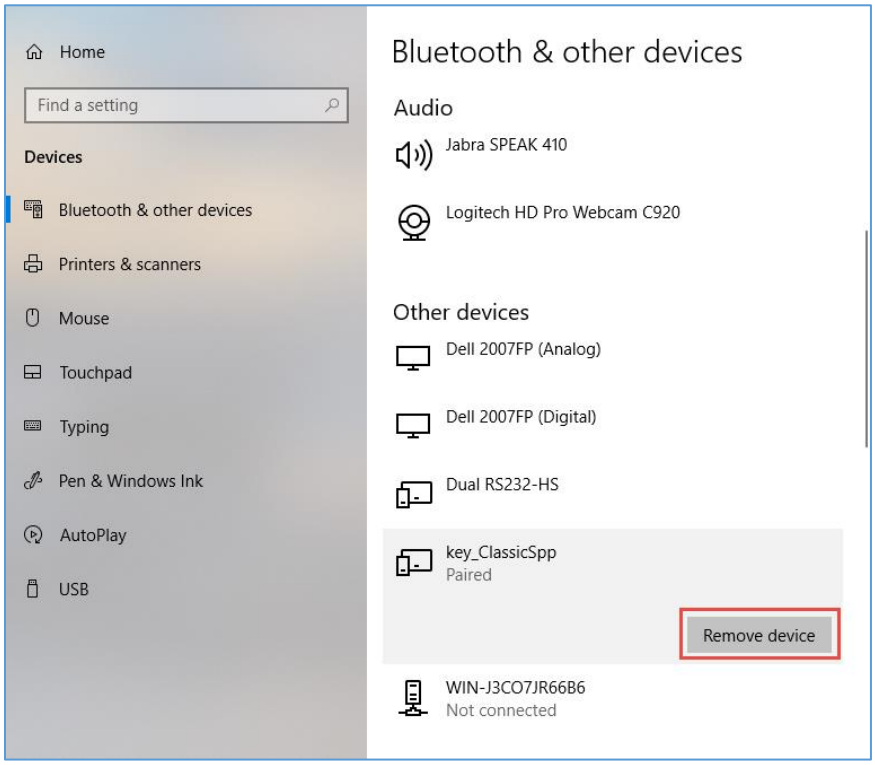
Now open the Bluetooth Serial Terminal app that you installed earlier. Your device should show up in the list. If not click "Refresh List".



Once you see it in the list, click "Connect". Now you can type strings in the Transmit window and click "Send" to send them to the WICED SPP Application. Observe the WICED kit by watching in the UART terminal. The WICED kit will display a line like the following with the number of characters received specified by "len=":

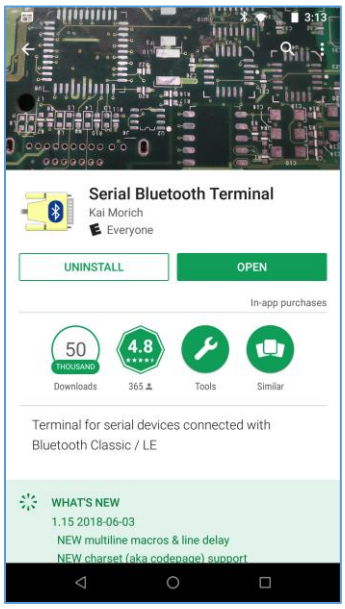
```
spp_rx_data_callback handle:2 len:2 78-78
```

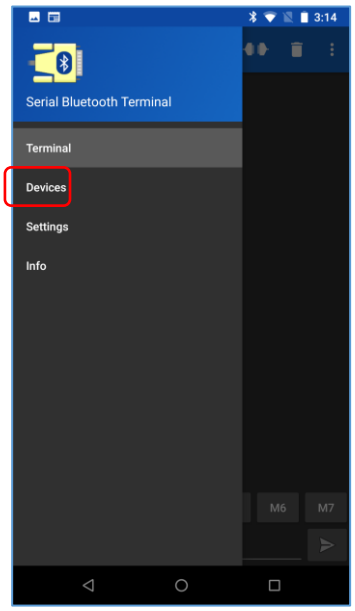
When you are done testing, click "Disconnect", close the Bluetooth Serial Terminal app and then go into the computer's Bluetooth settings to remove the device's paring information (Settings -> Devices -> <appname>_ClassicSpp -> Remove device -> Yes).

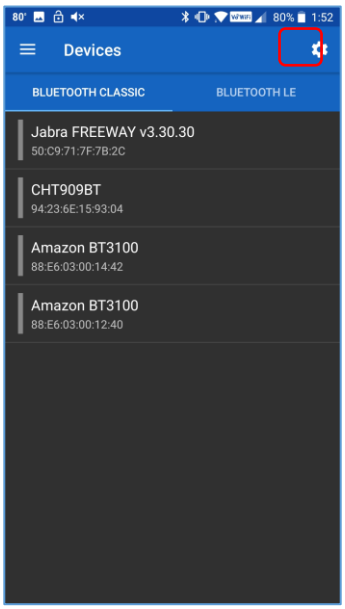


Android Instructions

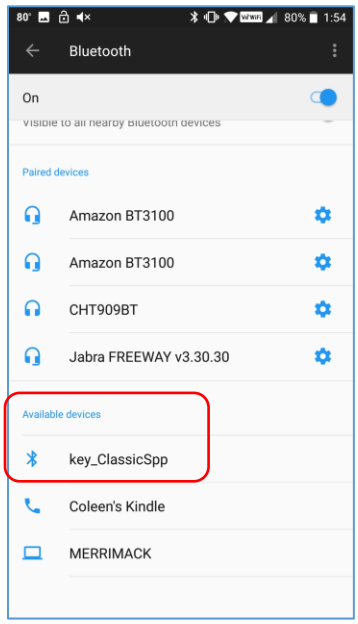
On an Android phone, you can install "Serial Bluetooth Terminal" from the Google Play Store. When you run the App, you will need to pair with your development kit. To do that open the menu (3 lines near the upper left corner) and tap on "Devices". From the Devices page, click on the "Gear" icon. This will take you to your phone's Bluetooth settings.

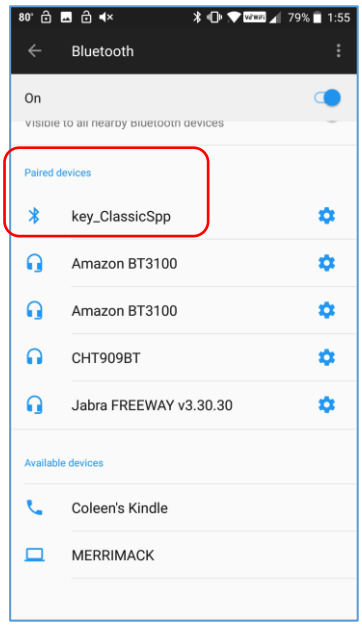




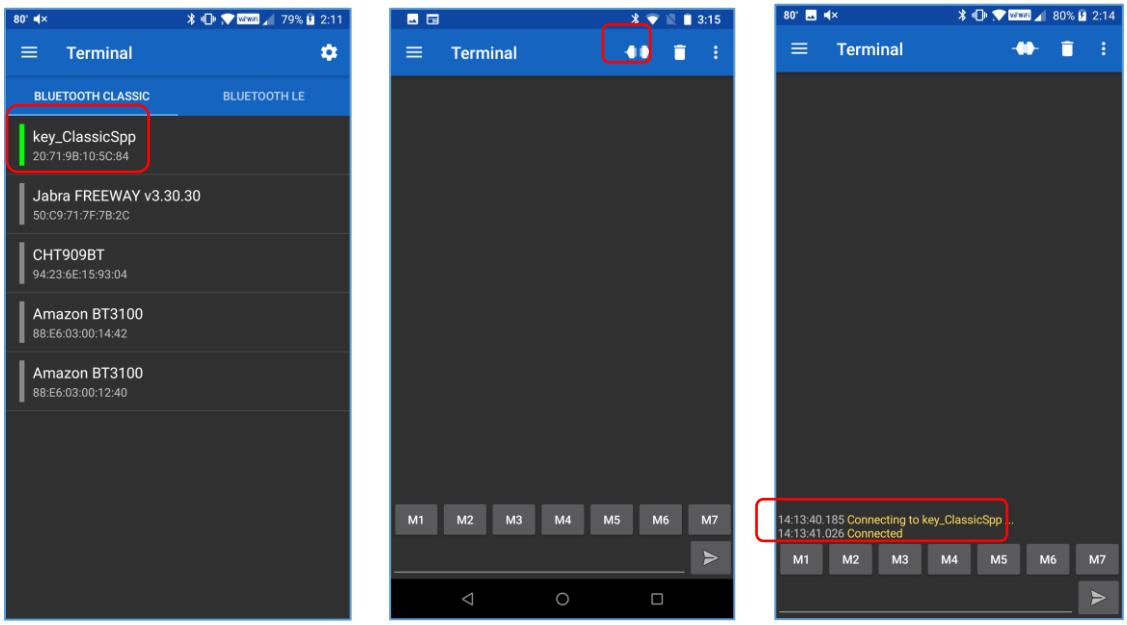


Find your device in the list and Pair with it (the exact procedure may be slightly different depending on the version of Android you are running).



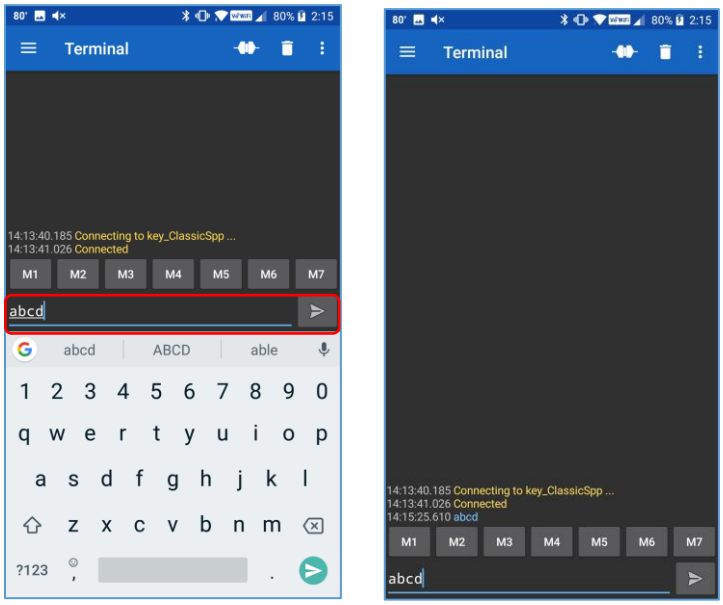


Once the device has been added, press the back arrow and you will see that your device appears in the Devices list. Tap on it to make it the active device (it will have a green bar to the left of the name when it is active). Then open the menu and select "Terminal" to see the blank terminal window. Next, tap the plug icon near the upper right corner to open the SPP server connection to your development kit. It will say "Connected" in the terminal window.

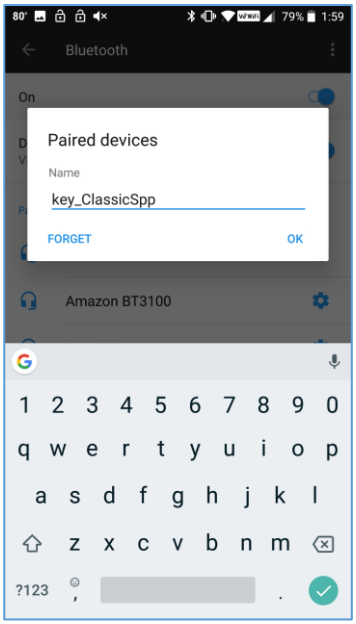


Now you can send data to the SPP server by entering it at the bottom of the window and clicking the Send arrow. You will see the data transmitted on the PUART terminal window for the kit. The WICED kit will display a line like the following with the number of characters received specified by "len=":

spp_rx_data_callback handle:2 len:2 78-78



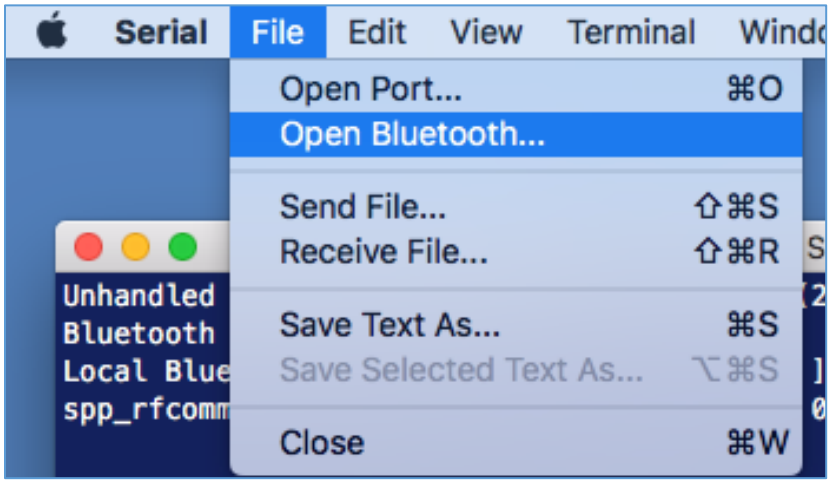
When you press the plug again, it will disconnect. You can then go back to the menu, select Devices, click on the Gear icon, and delete the Bonding information for your device (aka Forget) from the Bluetooth settings. Again, the exact procedure to forget the device will vary based on the version of Android you are running.



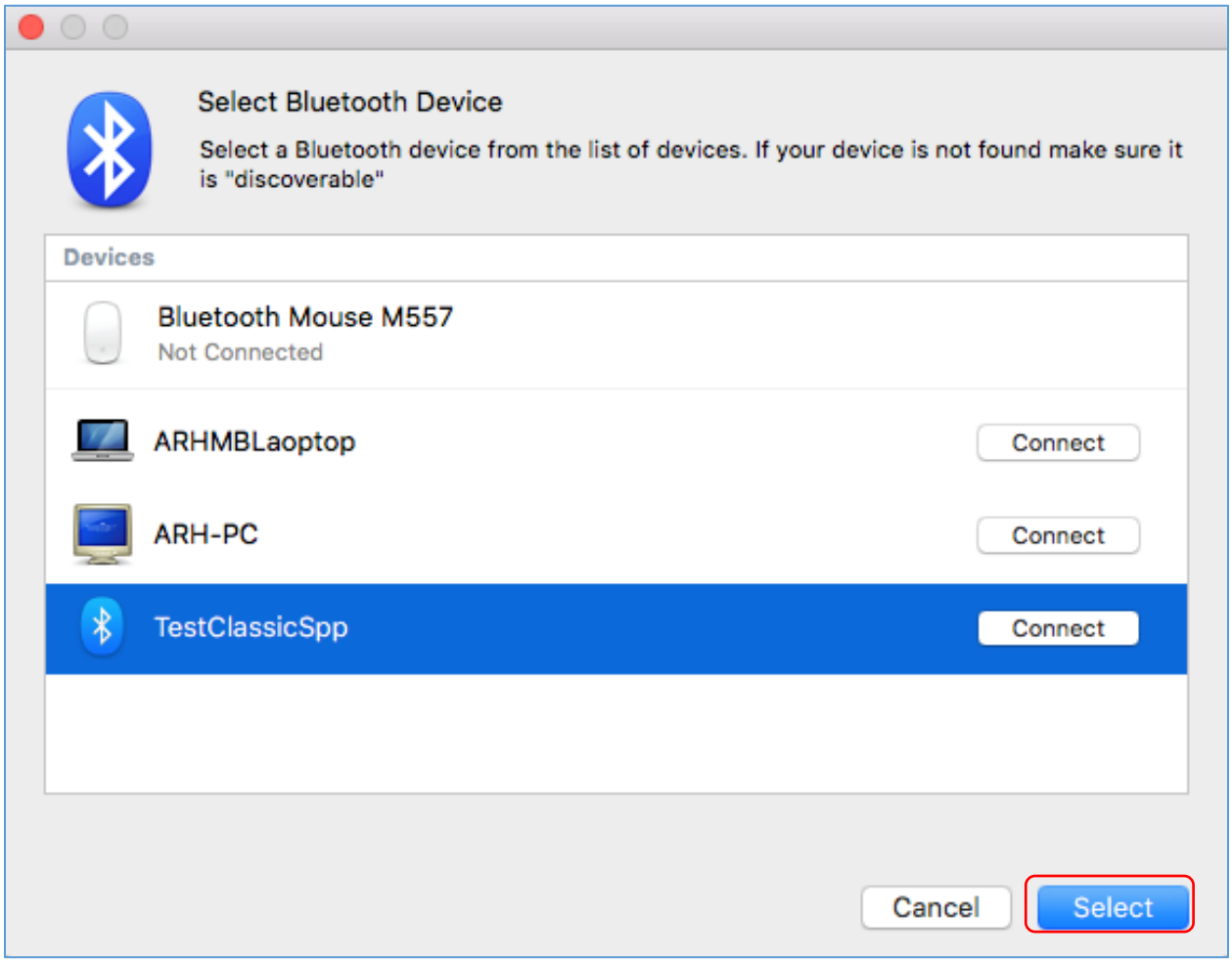
Mac Instructions

Install "Serial" from Decisive Tactics onto your Mac. You can get it in the App Store.

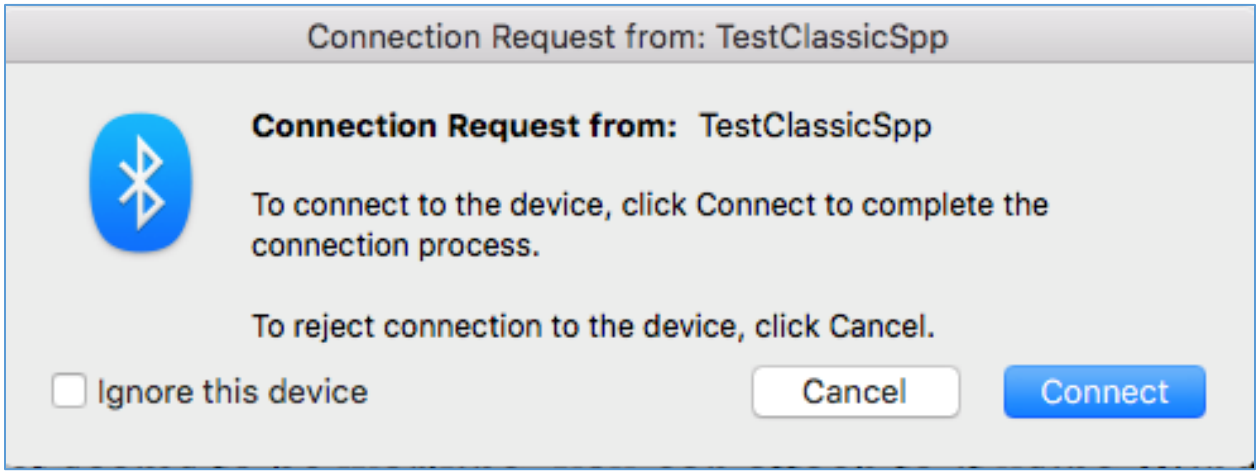
Once you have programmed the development kit you need to connect to it from the Mac. In the Serial program choose File → Open Bluetooth.



Then click on your application and press "Select". This will pair to the development kit and open a window.



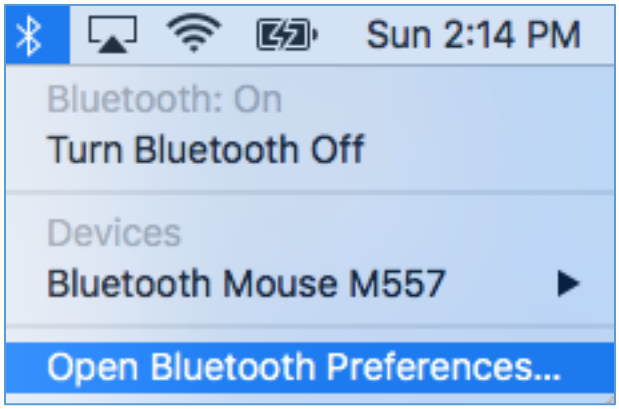
You will be asked to confirm the connection.



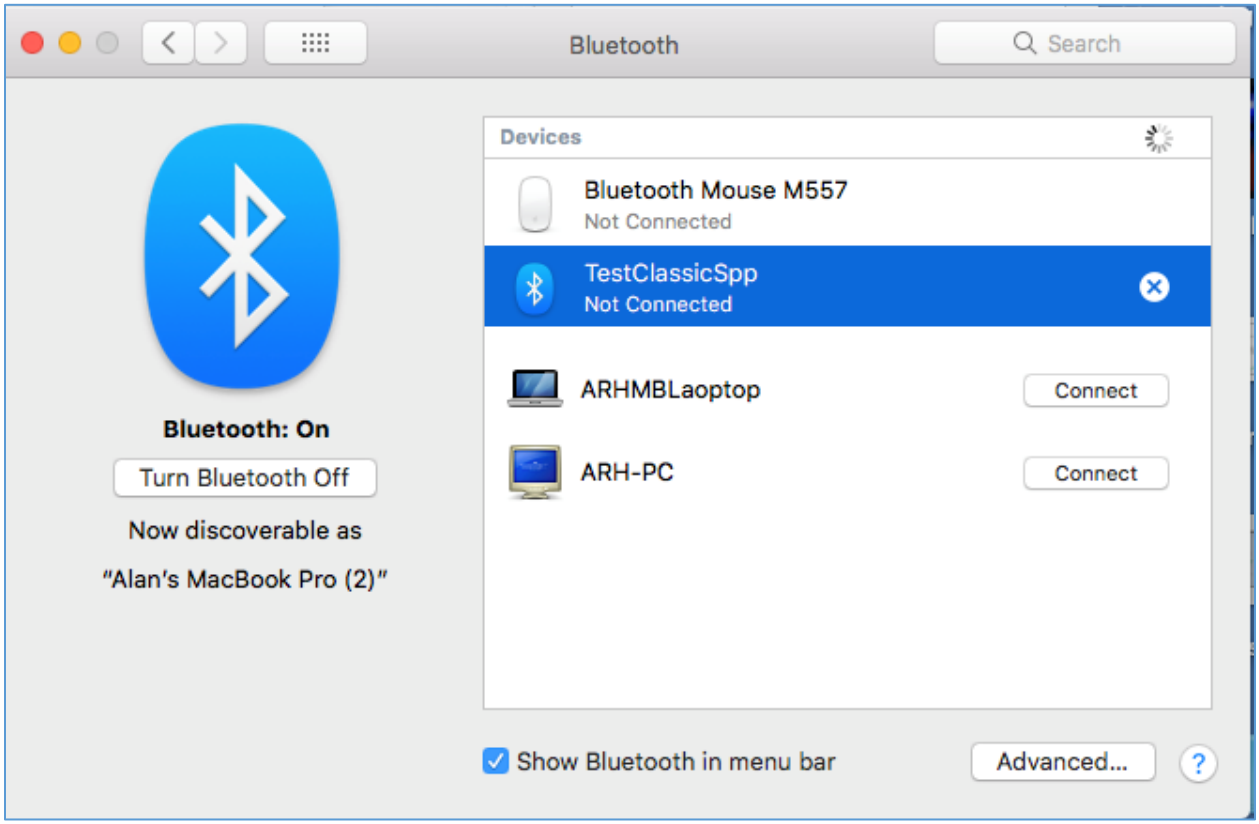
Once it is connected, everything you type will appear in the console window of the WICED Development kit. Below you can see that I typed "asdf".

```
Local Bluetooth Address: [20 71 9b 17 19 a2 ]
spp_rfcomm_start_server: rfcomm_create Res: 0x0  Port: 0x0001
IO_CAPABILITIES_BR_EDR_RESPONSE peer_bd_addr: 78 4f 43 a2 64 f6 , peer_io_cap: 1, peer_oob_data: 0, peer_auth_req:
 2
BR/EDR Pairing IO cap Request
numeric_value: 664177
Pairing Complete 0.
BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT
NVRAM ID:512 written :136 bytes result:0
Encryption Status event: bd ( 78 4f 43 a2 64 f6  ) res 0
spp_rfcomm_control_callback : Status = 0, port: 0x0001  SCB state: 0  Srv: 0x0001  Conn: 0x0000
RFCOMM Connected  isInit: 0  Serv: 0x0001   Conn: 0x0001 78 4f 43 a2 64 f6
spp_connection_up_callback handle:1 address:78 4f 43 a2 64 f6
rfcomm_data: len:1 handle 1 data 61-61)
spp_session_data: len:1, total: 1 (session 1 data 61-61)
spp_rx_data_callback handle:1 len:1 61-61
a
rfcomm_data: len:1 handle 1 data 73-73)
spp_session_data: len:1, total: 2 (session 1 data 73-73)
spp_rx_data_callback handle:1 len:1 73-73
s
rfcomm_data: len:1 handle 1 data 64-64)
spp_session_data: len:1, total: 3 (session 1 data 64-64)
spp_rx_data_callback handle:1 len:1 64-64
d
rfcomm_data: len:1 handle 1 data 66-66)
spp_session_data: len:1, total: 4 (session 1 data 66-66)
spp_rx_data_callback handle:1 len:1 66-66
f
```

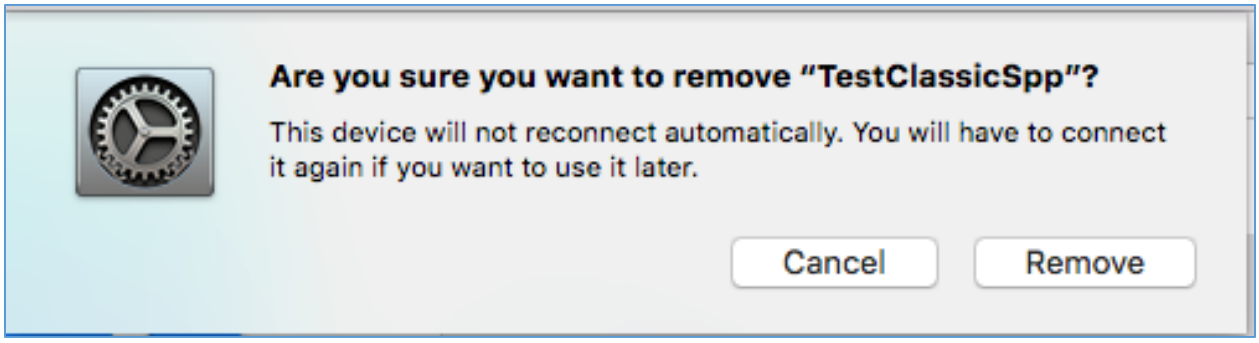
To unpair your development kit, select the Bluetooth symbol and pick "Open Bluetooth Preferences"



Select your device and click the "X".



You will need to confirm that you want to remove the Bonding information from the Mac BT Stack.



Exercise - 6A.2 Add UART Transmit

1. Use the SPP code example from the starter application RFCOMM-20819EVB02 to create an application called **ch06a_ex02_spp_uart**.
2. Comment out lines 90 and 91 in spp.c to disable sending data on interrupts and on timer timeouts. We will use the PUART interface to send data to the kit that will then be transmitted over Bluetooth using the SPP send data function.
3. Open the makefilt and set BT_DEVICE_ADDRESS to random.
4. Change the name of your device by editing the variable BT_LOCAL_NAME in the wiced_bt_cfg.c file.
    a. Hint: The default name is “spp test”, remember to use your initials in the device name so that you can find it in the list of devices that will be advertising (e.g. spp uart <inits>).
    b. Hint: The value must be null terminated so keep '\0' at the end of the array.
5. Modify spp.c to include a transmit function so that you can send data in both directions. You will read characters from the PUART terminal window so that when you type keys on your PC keyboard those values will be transmitted over Bluetooth to the Bluetooth Serial window.
    a. Hint: There is an example in the Peripherals chapter that receives characters from a PUART terminal window. Refer to that if you need help determining how to read characters from the PUART.
    b. Hint: Add a new function called *spp_tx_data* to *spp.c* to send the data. It will take a pointer to the data to send and the length as parameters and will call the *wiced_bt_spp_send_session_data* function to send the data. You will call the *spp_tx_data* function whenever a keystroke is received from the PUART.
6. Once you have built the application and programmed the kit, connect the same way you did for the previous exercise. In this case, try typing characters in the PUART terminal to see them show up on the PC or Android Bluetooth terminal window.

Exercise - 6A.3 (Advanced) Add Multiple Device Bonding Capability

In this exercise, you will add the capability to store bonding information from multiple devices. You will need to make several changes to your current SPP implementation:

1. Use the template in folder “templates/ch06a_ex04_spp_mult” to create an application called **ch06a_ex04_spp_mult**.
2. Change the device name in wiced_bt_cfg.c to include your initials.
3. Handle saving multiple link keys into the NVRAM. Let's use 8 for the maximum number of saved link keys. Use one VSID to save a one-byte count of how many are being used. Then use VSID = VSID_Start+count to save each additional Address/Key Bonding pair.
4. Handle reading multiple link keys. When you get the event BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT, the event data will be a pointer to a wiced_bt_device_link_keys_t structure. That structure contains the BDADDR of the device that is trying to pair. You need to search through the VSIDs to find the BDADDR of the saved link keys. If you find one that matches return it. Otherwise return a WICED_ERROR so that a new device can be added.
5. Update BTM_ENABLED_EVENT to load the number of bonded devices and to overwrite the next slot when the max number of devices has been reached.
6. Update BTM_PAIRING_COMPLETE_EVT to save the BDADDR of the host to NVRAM
    a. Remember to increment the number of bonded devices and the next free slot as well as save this information to NVRAM
7. Update BTM_ENCRYPTION_STATUS_EVT to search for the BDADDR trying to connect in NVRAM. If found restore values to a current host structure.
8. Add BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT and write the code to save the new link keys to NVRAM.
9. Add BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT and write the code to search for the BD_ADDR in NVRAM. If not return WICED_BT_ERROR and have the stack generate new keys and call BTM_LINK_KEYS_UPDATE_EVT so that they are stored.
10. Add BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT to save local keys to NVRAM and BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT to read local keys from NVRAM.

Once you have the application completed, try bonding with two different devices (e.g. phone and PC). Connect and disconnect back and forth to verify that bonding information for both is retained and is used when reconnecting.

Hint: Look at chapter 4b exercise 6 if you get stuck.