



# 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 project 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 ..... 2**
  - 6A.1.1 INQUIRY .....4
  - 6A.1.2 PAGE / CONNECT.....4
  - 6A.1.3 DISCOVER THE SERVICES USING SERVICE DISCOVERY PROTOCOL (SDP).....4
  - 6A.1.4 PAIR & BOND.....4
  - 6A.1.5 EXCHANGE DATA WITH THE SERIAL PORT PROFILE.....5
- 6A.2 SERVICE DISCOVERY PROTOCOL (SDP)..... 5**
- 6A.3 SECURE SIMPLE PAIRING..... 6**
- 6A.4 L2CAP, RFCOMM & THE SERIAL PORT PROFILE ..... 6**
  - 6A.4.1 L2CAP .....8
  - 6A.4.2 RFCOMM .....8
  - 6A.4.3 SERIAL PORT PROFILE .....8
- 6A.5 WICED BLUETOOTH DESIGNER ..... 9**
- 6A.6 WICED BLUETOOTH STACK EVENTS ..... 16**
- 6A.7 WICED CLASSIC BLUETOOTH FIRMWARE ARCHITECTURE ..... 17**
  - 6A.7.1 OVERVIEW .....17
  - 6A.7.2 APPLICATION CODE (<APPNAME>.C) .....17
  - 6A.7.3 SDP DATABASE (<APPNAME>\_SDB\_DB.C/.H).....18
  - 6A.7.4 HANDLE PAIRING.....19
  - 6A.7.5 HANDLE BONDING .....20
  - 6A.7.6 SUPPORT THE SERIAL PORT PROFILE .....22
- 6A.8 EXERCISES..... 27**
  - EXERCISE - 6A.1 CREATE A SERIAL PORT PROFILE PROJECT .....27
  - EXERCISE - 6A.2 ADD UART TRANSMIT .....40
  - EXERCISE - 6A.3 (ADVANCED) IMPROVE SECURITY BY ADDING IO CAPABILITIES (YES/NO).....41
  - EXERCISE - 6A.4 (ADVANCED) ADD MULTIPLE DEVICE BONDING CAPABILITY .....42



## 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 project 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.

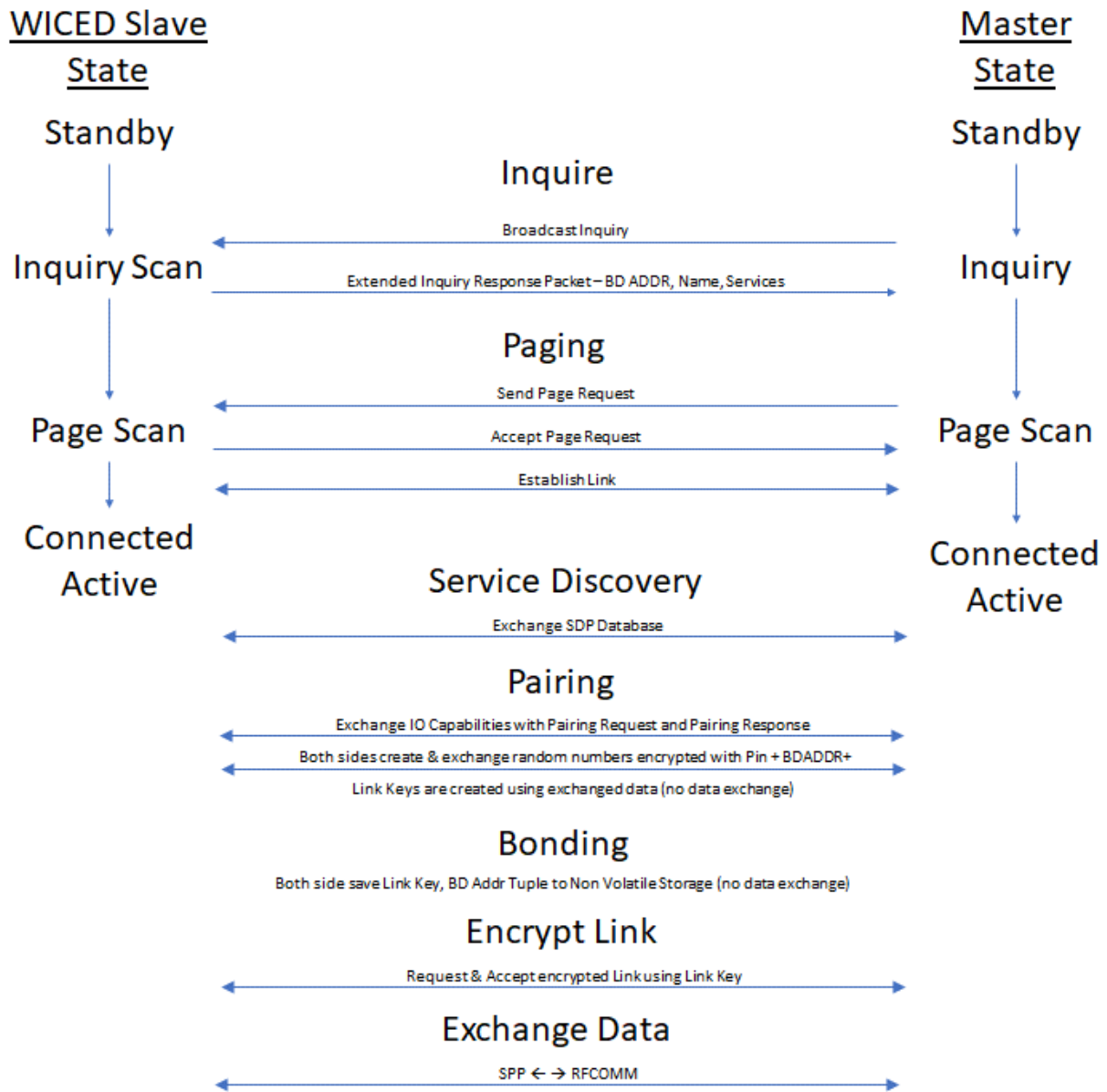
The five steps are:

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 its 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 [paggers?](#)).

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 project: 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.

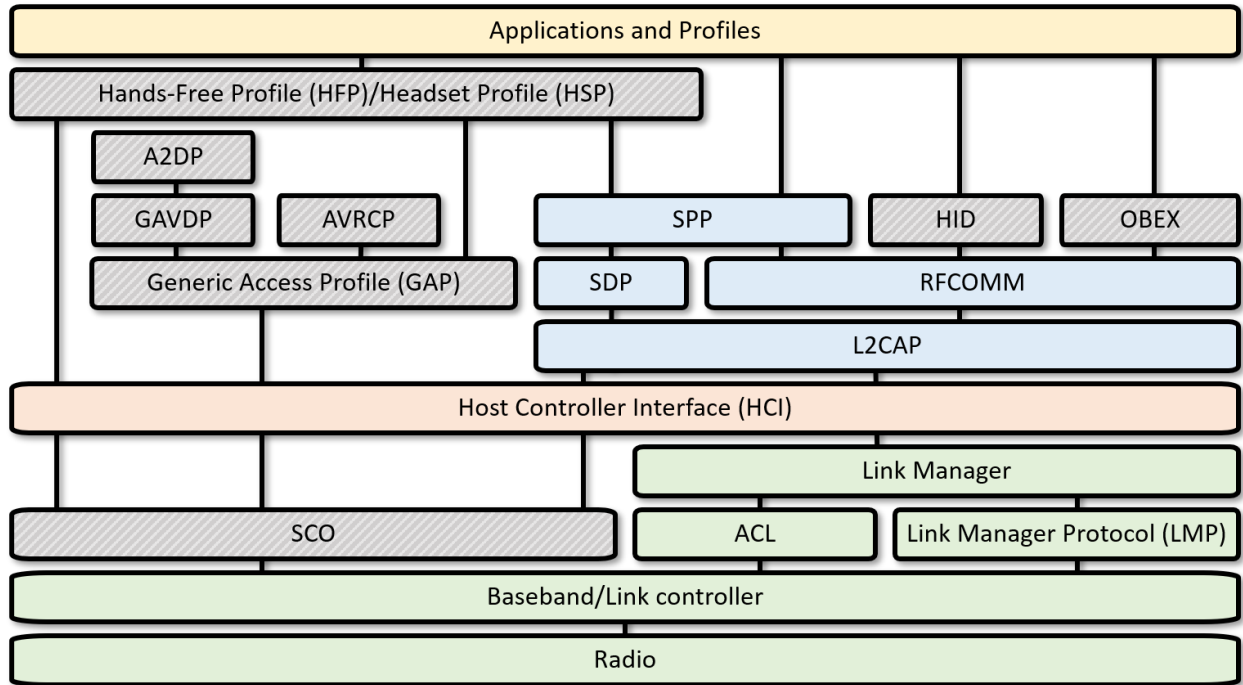
Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on the random numbers used to generate the shared key. The user observes both devices. If the number is the same on both, then the user 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.

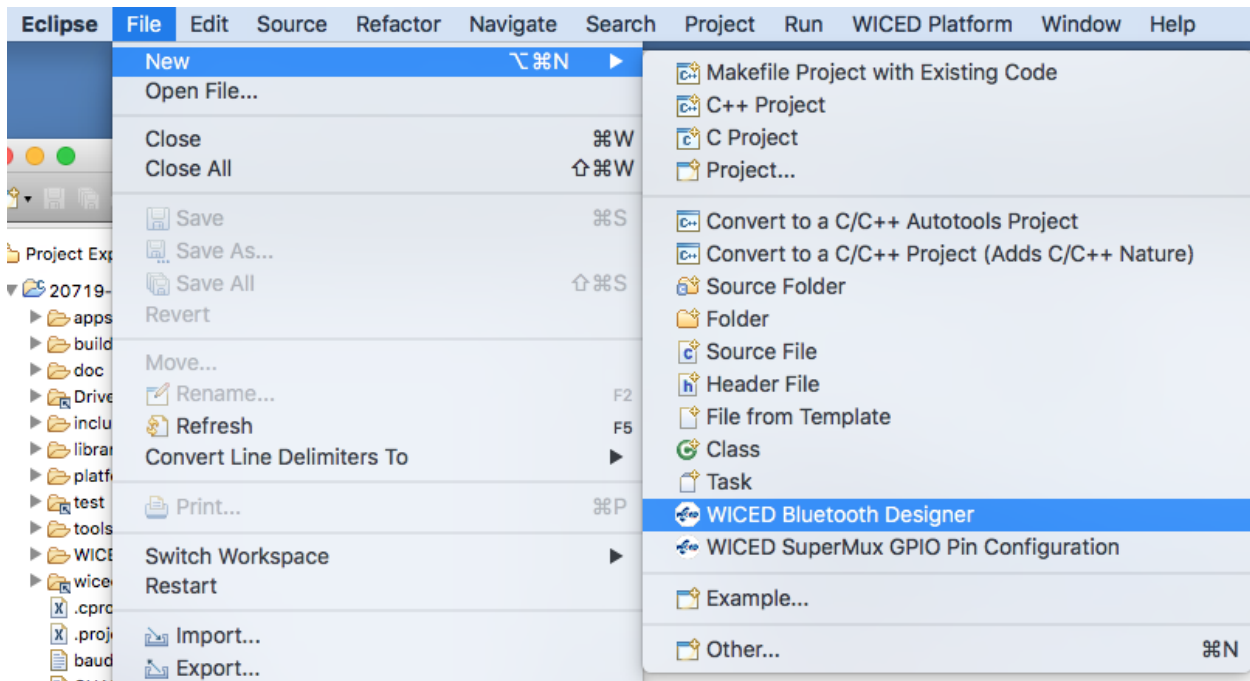


## 6A.5 WICED Bluetooth Designer

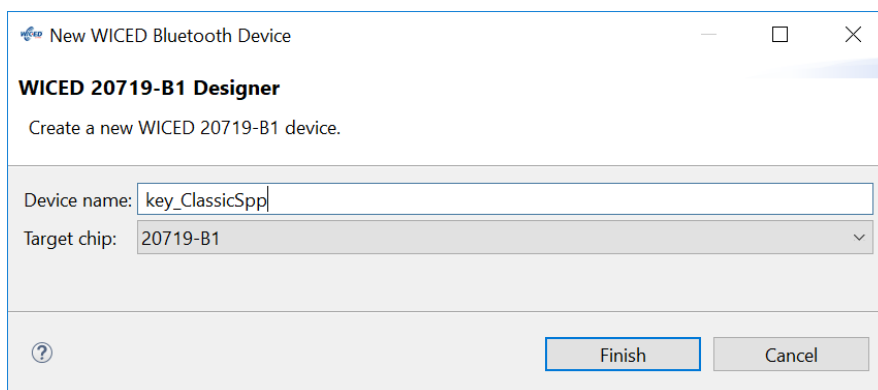
As with BLE, WICED Bluetooth Designer can be used to help you create a WICED Bluetooth Classic Project. Specifically, it will help you:

- Make a template project with all the required files
- Create the SDP Database
- Create a Make Target

To run the tool, go to File → New → WICED Bluetooth Designer

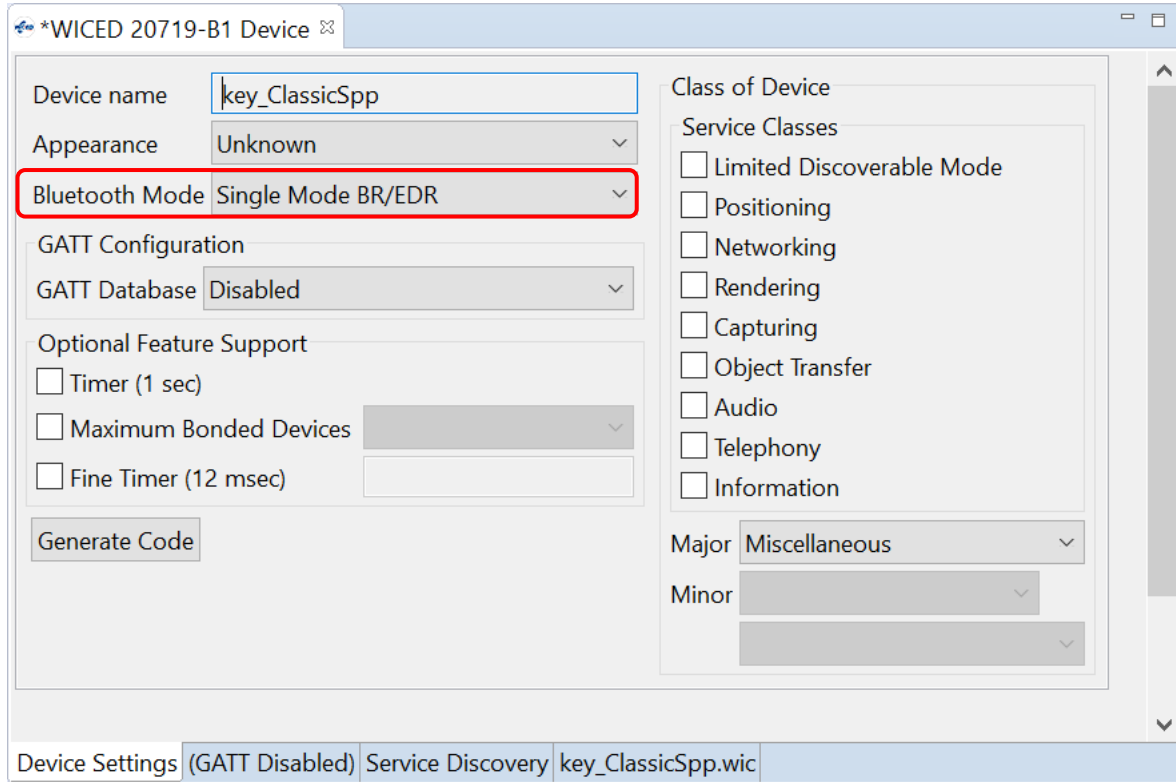


Give your project a name, in this case I call it "key\_ClassicSp". **When you do this yourself, be sure to make the name unique so that you will be able to identify which device is yours. For example, <inits>\_ClassicSp where <inits> is your initials.**

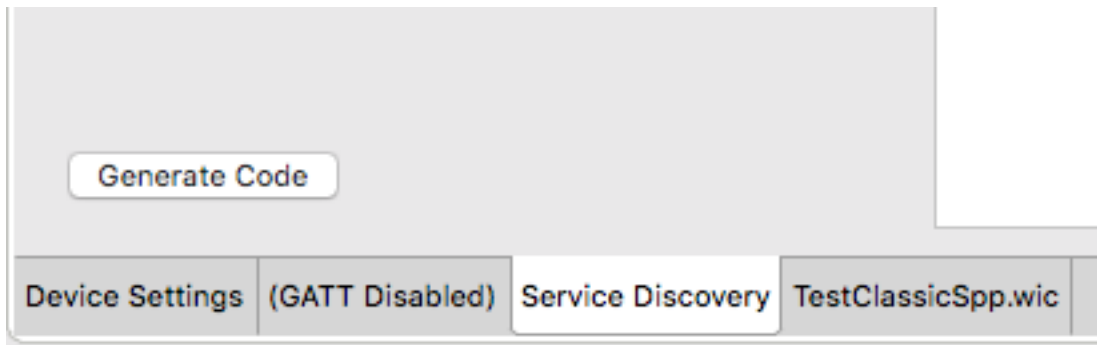


Click "Finish" to open the configuration window.

The default setting is Single Mode LE, but we want a Bluetooth Classic BR/EDR project so change the Bluetooth Mode to "Single Mode BR/EDR".



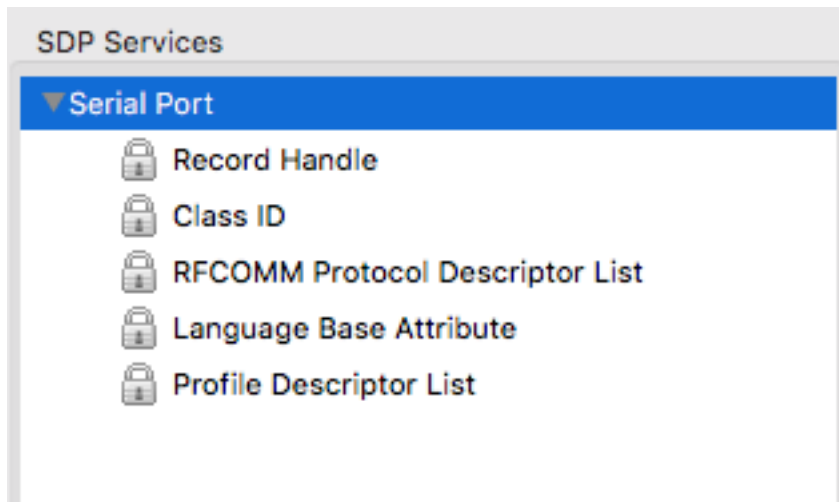
The biggest benefit of Bluetooth Designer is helping you build the "Service Discovery" database. Click on the Service Discovery Tab at the bottom of the window.



The SDP database starts with nothing in it. To add Services to the database, click on the drop-down menu under "Add SDP Service", pick out the Service you want (in this case, Serial Port) then click "+".

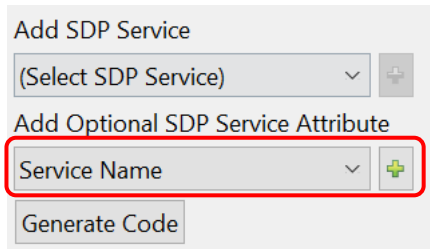


Now the SDP Database (which shows up in the window titled SDP Services) has the Serial Port. Click on the Arrow to the left of "Serial Port" to expand the Attributes that are created for the SPP:

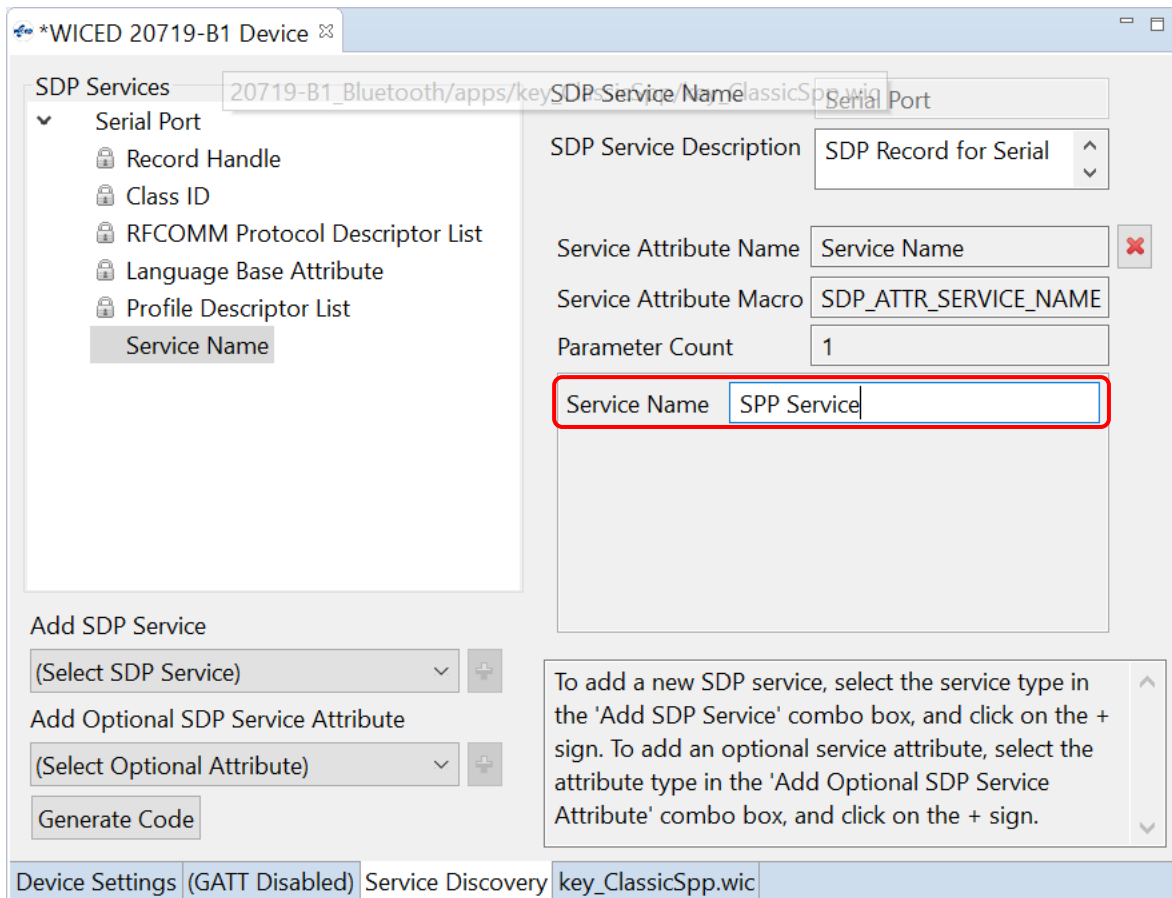


The process of looking for any offered services is termed browsing. In SDP, the mechanism for browsing for services is based on an attribute shared by all service classes. This attribute is called the “Browse List” attribute. The value of this attribute contains a list of UUIDs. Each UUID represents a browse group with which a service may be associated.

By default, the "Service Name" and "Browse List" are not included in the Attributes for the SPP. To add them, select them from the "Add Optional SDP Service Attribute" menu and press the "+". Here I add the "Service Name":



Next, I'll change the name to "SPP Service" by typing in the "Service Name" text box.



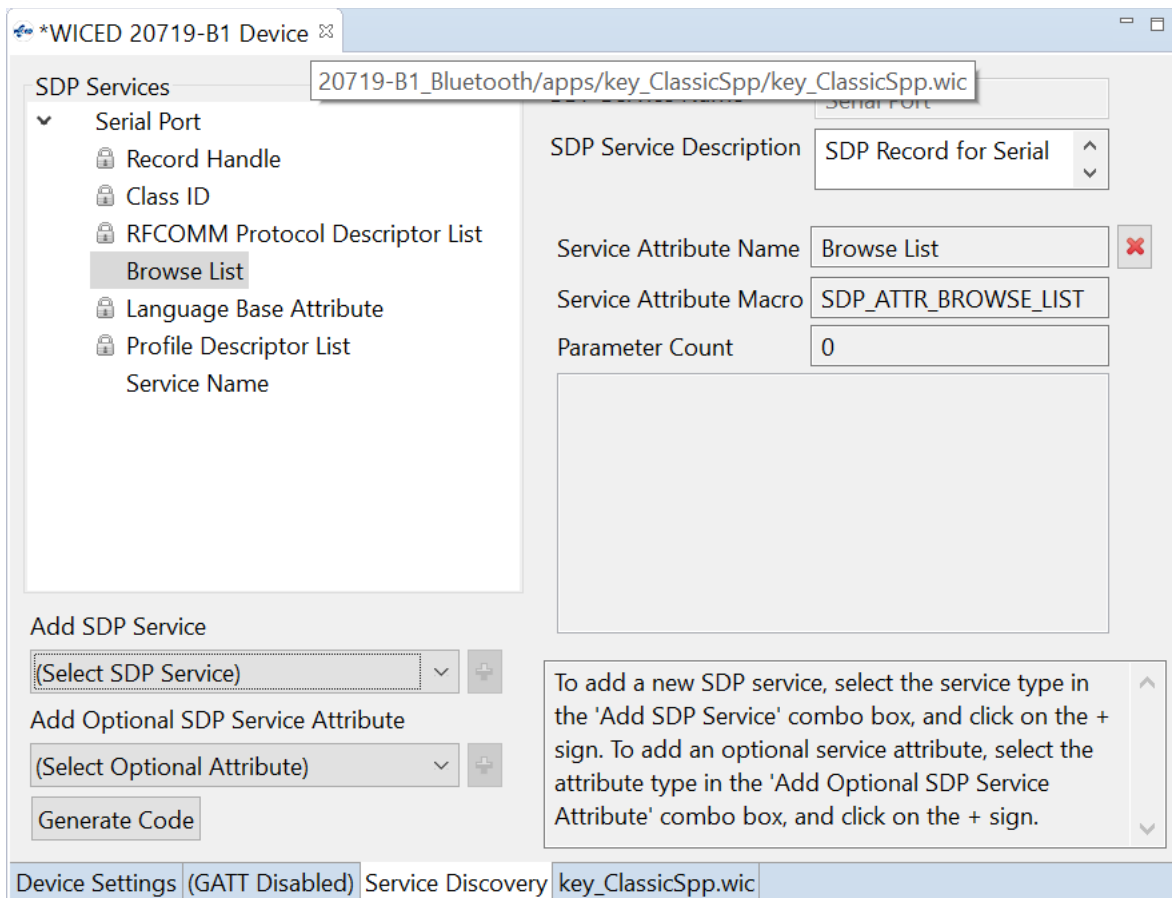
To add the Browse List to the SPP Service, select "Browse List" from the Optional SDP Service Attribute List and press "+".

Add SDP Service  
 (Select SDP Service) ▼ +

Add Optional SDP Service Attribute  
 Browse List ▼ +

Generate Code

I'll leave the settings for the Browse List at the default values.

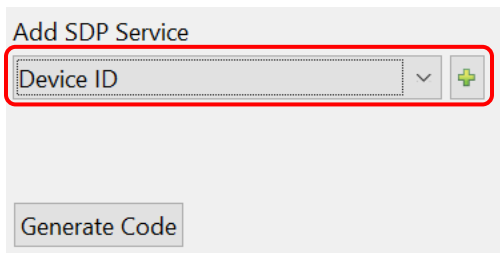


The screenshot shows the configuration window for the Serial Port service. The 'SDP Services' list on the left includes Record Handle, Class ID, RFCOMM Protocol Descriptor List, Browse List (selected), Language Base Attribute, Profile Descriptor List, and Service Name. The right pane shows the configuration for the selected service: SDP Service Description is 'SDP Record for Serial', Service Attribute Name is 'Browse List', Service Attribute Macro is 'SDP\_ATTR\_BROWSE\_LIST', and Parameter Count is '0'. A tooltip at the bottom right explains how to add services and attributes.

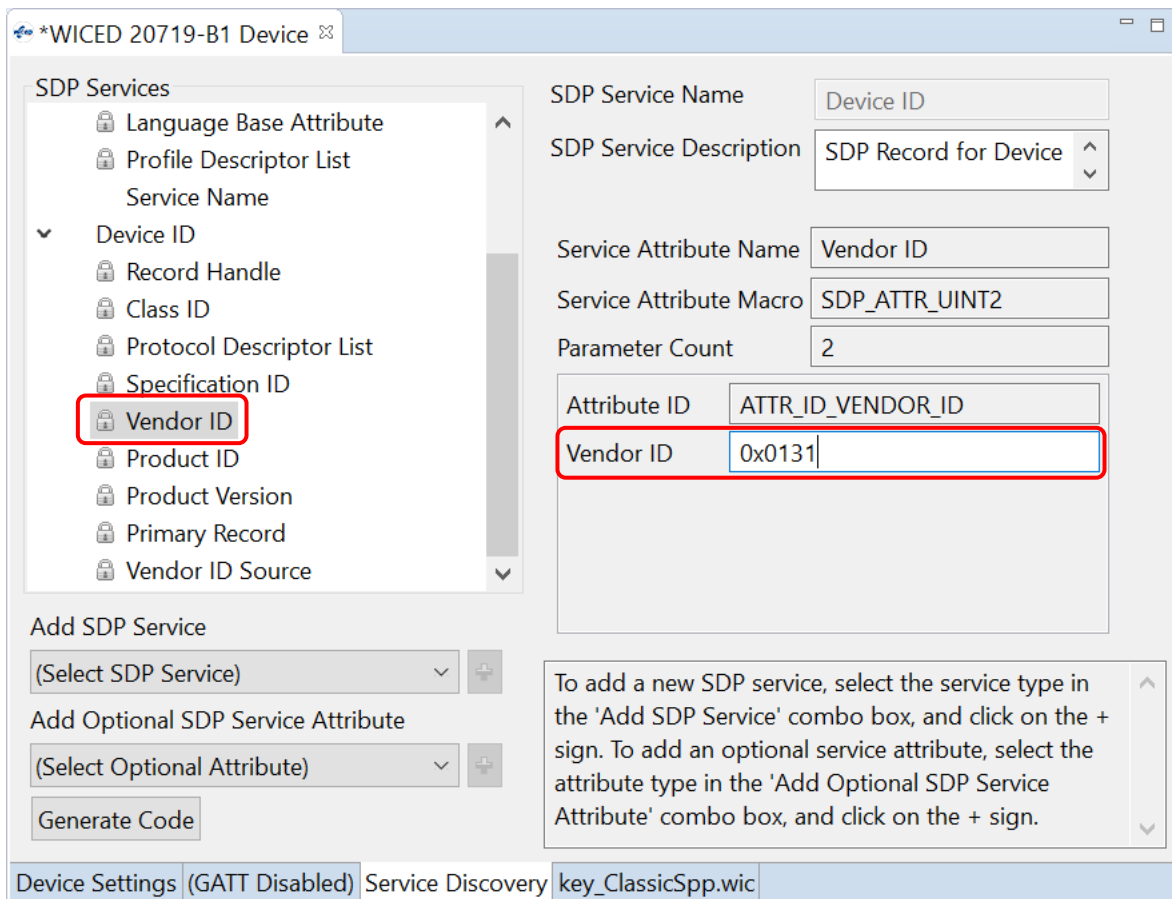
Device Settings | (GATT Disabled) | Service Discovery | key\_ClassicSpp.wic

This completes the setup for the SPP service.

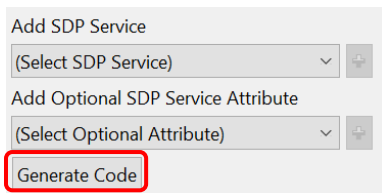
Next, I'll add another Service to the Database called the Device ID Service. This contains things like vendor ID, product ID, etc. which you can use to convey information about your device.



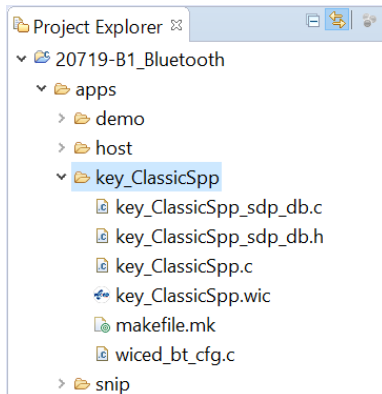
After it is added, you can see all the Attributes that go with it. You can change some of the values by clicking on the Attribute you want to modify and then entering the new values. Only certain values are editable. **The default value for Vendor ID is 0xFFFF. This is not a valid vendor ID, so it will not work properly in all situations. Therefore, you MUST change this to a valid Vendor ID. In this case, I'll use the Cypress Vendor ID which is 0x0131.**



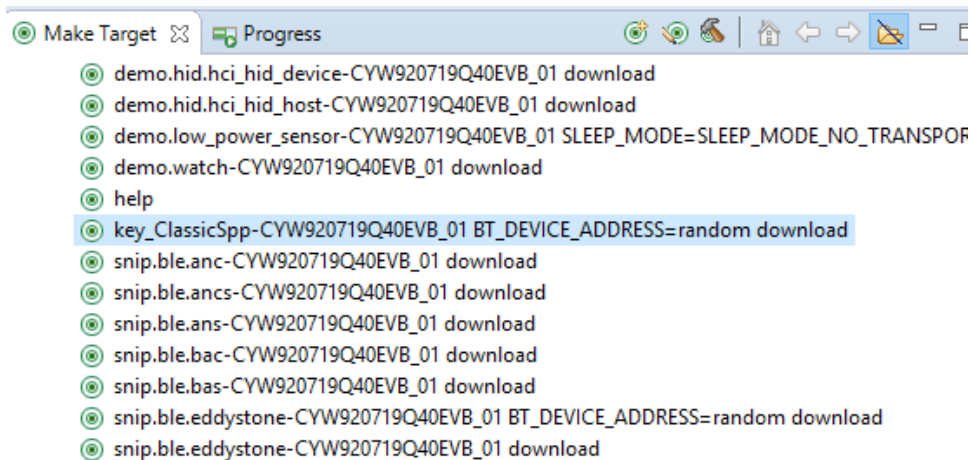
The last step in the process of generating the project is to press the Generate Code button which will create the project with C-code that you will customize later, and it will create a make target.



Here is the resulting workspace:



WICED Bluetooth Designer also creates a Make Target for your new project but it does not include the options `BT_DEVICE_ADDRESS=random`. I'll add that now so that I don't end up with a conflicting address.



## 6A.6 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   |
|--|---|
| <a href="#">BTM_ENABLED_EVT</a>  | 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.  |
| <a href="#">BTM_SECURITY_REQUEST_EVT</a>   | 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.  |
| <a href="#">BTM_PAIRING_IO_CAPABILITIES_BR_EDR_REQUEST_EVT</a>                               | 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.  |
| <a href="#">BTM_PAIRING_COMPLETE_EVT</a>   | The Stack is informing you that you are now paired.   |
| <a href="#">BTM_ENCRYPTION_STATUS_EVT</a>  | The Stack is informing you that the link is now encrypted...or not depending on the event data.   |
| <a href="#">BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT</a>                                     | The Stack is asking you find and return the link key for the BDADDR that was sent in the event data.  |
| <a href="#">BTM_USER_CONFIRMATION_REQUEST_EVT</a><br>(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. |
| <a href="#">BTM_PASSKEY_NOTIFICATION_EVT</a><br>(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.  |
| <a href="#">BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT</a>  | The Stack is asking you to read the local identify keys from the NVRAM and return them to the Stack.  |
| <a href="#">BTM_PAIRING_IO_CAPABILITIES_BR_EDR_RESPONSE_EVT</a>                              | The Stack is informing you of the I/O capabilities of the other side of the connection.   |
| <a href="#">BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT</a>                                      | The Stack is asking your firmware to store the BDADDR/Link Keys (which are passed in the event data).   |



## 6A.7 WICED Classic Bluetooth Firmware Architecture

### 6A.7.1 Overview

WICED Bluetooth Designer will create a skeleton of the firmware that you need start building your device. The skeleton includes:

1. A file named `<appname>.c` which contains the application functions:
  - a. `application_start` which is the entry point for the firmware.
  - b. `<appname>_app_init` which provides a place for you to get your application stuff going.
  - c. `<appname>_management_callback` which is a template BTM event handler function.
2. A `.c/.h` pair of files called `<appname>_sdp_db.c/.h` containing:
  - a. The `#defines` for the SDP database.
  - b. A `uint8_t` structure holding the actual database.
3. A file called `wiced_bt_config.h` containing:
  - a. All the basic Bluetooth configuration settings to get the stack going.

### 6A.7.2 Application Code (`<appname>.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 if you want `WICED_BT_TRACE` messages
- Call `wiced_bt_stack_init` with the event handler to start the stack

The `<appname>_app_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:

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

To make your project work you need to add the capability to `<appname>.c` to:

- Handle Pairing
- Handle Bonding
- Support the Serial Port Profile

These will each be covered in detail in a minute.

You also should change the code to route the debug messages to the PUART (i.e. uncomment the line to use `WICED_ROUTE_DEBUG_TO_PUART` and comment the line to use `WICED_ROUTE_DEBUG_TO_WICED_UART`).

### 6A.7.3 SDP Database (<appname>\_sdb\_db.c/.h)

The Service Discovery Database is created for you based on the configuration settings in BT Designer. It creates the files <appname>\_sdp\_db.c/.h

The file <appname>\_sdp\_db.h for the simple project with the SPP and Device ID Services that we created using BT Designer is shown here:

```
#ifndef __SDP_DATABASE_H__
#define __SDP_DATABASE_H__

// SDP Record for Serial Port
#define HDLR_SERIAL_PORT          0x10001
#define SERIAL_PORT_SCN          0x01
// SDP Record for Device ID
#define HDLR_DEVICE_ID           0x10002

// External definitions
extern const uint8_t sdp_database[];
extern const uint16_t sdp_database_len;

#endif /* __SDP_DATABASE_H__ */
```

The file <appname>\_sdp\_db.c simply contains the two Service Records that we defined in BT Designer: the SPP and the Device Info.

```
const uint8_t sdp_database[] = // Define SDP database
{
    SDP_ATTR_SEQUENCE_1(157),

    // SDP Record for Serial Port
    SDP_ATTR_SEQUENCE_1(84),
    SDP_ATTR_RECORD_HANDLE(HDLR_SERIAL_PORT),
    SDP_ATTR_CLASS_ID(UUID_SERVCLASS_SERIAL_PORT),
    SDP_ATTR_RFCOMM_PROTOCOL_DESC_LIST(SERIAL_PORT_SCN),
    SDP_ATTR_BROWSE_LIST,
    SDP_ATTR_LANGUAGE_BASE_ATTR_ID_LIST,
    SDP_ATTR_PROFILE_DESC_LIST(UUID_SERVCLASS_SERIAL_PORT, 0x0102),
    SDP_ATTR_SERVICE_NAME(11),
    'S', 'P', 'P', ' ', 'S', 'e', 'r', 'v', 'i', 'c', 'e',

    // SDP Record for Device ID
    SDP_ATTR_SEQUENCE_1(69),
    SDP_ATTR_RECORD_HANDLE(HDLR_DEVICE_ID),
    SDP_ATTR_CLASS_ID(UUID_SERVCLASS_PNP_INFORMATION),
    SDP_ATTR_PROTOCOL_DESC_LIST(1),
    SDP_ATTR_UINT2(ATTR_ID_SPECIFICATION_ID, 0x103),
    SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID, 0x0131),
    SDP_ATTR_UINT2(ATTR_ID_PRODUCT_ID, 0xFFFF),
    SDP_ATTR_UINT2(ATTR_ID_PRODUCT_VERSION, 0xFFFF),
    SDP_ATTR_BOOLEAN(ATTR_ID_PRIMARY_RECORD, 0x01),
    SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID_SOURCE, DI_VENDOR_ID_SOURCE_BTSIG),
};

// Length of the SDP database
const uint16_t sdp_database_len = sizeof(sdp_database);
```

The sequence numbers that you see above (SDP\_ATTR\_SEQUENCE) indicate the length of the database items. For example, 157 is the total number of bytes in the database (excluding itself). Likewise, 84 is the total number of bytes in the SPP service (again excluding itself).

The only action required by your application firmware to interact with the database is to register it. This is inserted into the function <appname>\_app\_init for you automatically by BT Designer:

```
/* Initialize SDP Database */
wiced_bt_sdp_db_init( (uint8_t*)sdp_database, sdp_database_len );
```

#### 6A.7.4 Handle Pairing

In the file <appname>.c you need to add code to handle pairing since it is not done completely by WICED Bluetooth Designer.

The BTM events involved in Pairing are:

- BTM\_PAIRING\_IO\_CAPABILITIES\_BR\_EDR\_RESPONSE\_EVT
- BTM\_PAIRING\_IO\_CAPABILITIES\_BR\_EDR\_REQUEST\_EVT
- BTM\_USER\_CONFIRMATION\_REQUEST\_EVT (for numeric comparison)
- BTM\_PASSKEY\_NOTIFICATION\_EVT (for passkey entry)
- BTM\_PAIRING\_COMPLETE\_EVT

When the Master attempts to Pair with you, it sends its I/O capabilities. When you get that event, you can decide what to do including nothing. In this case, we just print out the I/O capabilities. By default, BT Designer does not include this event in the template.

```
case BTM_PAIRING_IO_CAPABILITIES_BR_EDR_RESPONSE_EVT:
    WICED_BT_TRACE(
        "IO_CAPABILITIES_BR_EDR_RESPONSE peer_bd_addr: %B, peer_io_cap: %d, peer_oob_data: %d, peer_auth_req: %d\n",
        p_event_data->pairing_io_capabilities_br_edr_response.bd_addr,
        p_event_data->pairing_io_capabilities_br_edr_response.io_cap,
        p_event_data->pairing_io_capabilities_br_edr_response.oob_data,
        p_event_data->pairing_io_capabilities_br_edr_response.auth_req);
    break;
```

When you get the event asking for your I/O Capabilities, you need to respond by changing the event data. By default, BT Designer gives you an incomplete filling of the structure. You need to add the local IO capabilities. For example, if your device has a display and you want to use numeric comparison, you could use BTM\_IO\_CAPABILITIES\_DISPLAY\_AND\_YES\_NO\_INPUT as shown here:

```
case BTM_PAIRING_IO_CAPABILITIES_BR_EDR_REQUEST_EVT:
    /* Request for Pairing IO Capabilities (BR/EDR) */
    WICED_BT_TRACE("BR/EDR Pairing IO Capabilities Request\n");
    p_event_data->pairing_io_capabilities_br_edr_request.oob_data = BTM_OOB_NONE;
    p_event_data->pairing_io_capabilities_br_edr_request.auth_req = BTM_AUTH_SINGLE_PROFILE_GENERAL_BONDING_YES;
    p_event_data->pairing_io_capabilities_br_edr_request.is_orig = WICED_FALSE;
    p_event_data->pairing_io_capabilities_br_edr_request.local_io_cap = BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT;
    break;
```



```

/* Paired Device Link Keys Request */
WICED_BT_TRACE("Paired Device Link Request Keys Event\n");
/* Device/app-specific TODO: HANDLE PAIRED DEVICE LINK REQUEST KEY - retrieve from NVRAM, etc */
#if 1
    if (key_classicspp_read_link_keys( &p_event_data->paired_device_link_keys_request ))
    {
        WICED_BT_TRACE("Key Retrieval Success\n");
    }
    else
#endif
/* Until key retrieval implemented above, just fail the request - will cause re-pairing */
{
    WICED_BT_TRACE("Key Retrieval Failure\n");
    status = WICED_BT_ERROR;
}
break;

```

BT Designer does not provide a template for the keys update event. So, you need to add the case for that event, which simply calls a write function that we will create next.

```

case BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT:
    WICED_BT_TRACE("BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT\n");
    key_classicspp_write_link_keys(&p_event_data->paired_device_link_keys_update);
    break;

```

Next, you need to write the <appname>\_read\_link\_keys and <appname>\_write\_link\_keys functions. In our case we will write them so that they each support only one set of saved link keys, and they each use an entire VSID row.

```

// This function reads the first VSID row into the link keys
int key_classicspp_read_link_keys( wiced_bt_device_link_keys_t *keys )
{
    wiced_result_t result;
    int bytes_read;

    bytes_read = wiced_hal_read_nvram(WICED_NVRAM_VSID_START, sizeof(wiced_bt_device_link_keys_t),
        (uint8_t *)keys, &result);
    WICED_BT_TRACE("NVRAM ID:%d read :%d bytes result:%d\n", WICED_NVRAM_VSID_START, bytes_read, result);

    return bytes_read;
}

// This function write the link keys into the first VSID row
int key_classicspp_write_link_keys( wiced_bt_device_link_keys_t *keys )
{
    wiced_result_t result;
    int bytes_written = wiced_hal_write_nvram(WICED_NVRAM_VSID_START, sizeof(wiced_bt_device_link_keys_t),
        (uint8_t*)keys, &result);

    WICED_BT_TRACE("NVRAM ID:%d written :%d bytes result:%d\n", WICED_NVRAM_VSID_START, bytes_written, result);
    return (bytes_written);
}

```

After you write these two functions you need to add a forward declaration for each of them at the top of the file.

```

int key_classicspp_read_link_keys( wiced_bt_device_link_keys_t *keys );
int key_classicspp_write_link_keys( wiced_bt_device_link_keys_t *keys );

```

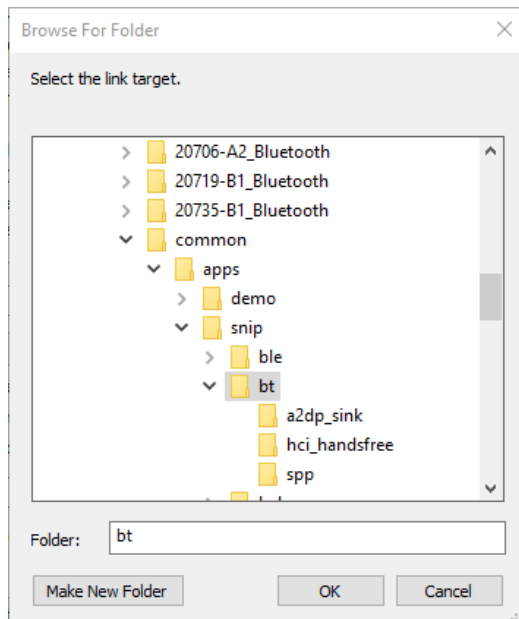
## 6A.7.6 Support the Serial Port Profile

To make the SPP work you need to initialize the server and provide callbacks for starting and stopping the connection and for receiving data. The WICED Bluetooth SDK contains all the code to implement an SPP server in two places: (1) the low-level functions are in a library called "spp\_lib" and; (2) an example application that demonstrates how to use the SPP library functions is in apps/snip/bt/spp.

After you have created your SPP project with the BT Designer tool, you can open snip/bt/spp/spp.c example to copy the additional blocks of code that you need. Typically, I create separate files spp.h and spp.c in my project to handle all the SPP server functionality.

Note: If you don't see snip/bt/spp in your workspace, follow these steps:

1. In the Project Explorer window, right click on "snip" and select New -> Folder.
2. Click the "Advanced >>" button.
3. Select "Link to alternate location (Linked Folder)".
4. Click the "Browse..." button.
5. Navigate to the WICED Studio Installation location (e.g. <User>/Documents/WICED-Studio-<version>) and select the folder ".../common/apps/snip/bt". The window should look like this:



6. Click on "OK".
7. Click on "Finish".

## makefile.mk

There are two changes that need to happen to the file `makefile.mk`. First, you need to add `spp_lib.a` so that you can link the SPP library functions. Second, when you add new source files to your project (like `spp.c`) you need to add them to `makefile.mk` as well:

```
APP_SRC = key_ClassicSpp.c
APP_SRC += key_ClassicSpp_sdp_db.c
APP_SRC += wiced_bt_cfg.c
APP_SRC += spp.c

C_FLAGS += -DWICED_BT_TRACE_ENABLE

# If defined, HCI traces are sent over transport/WICED HCI interface
C_FLAGS += -DHCI_TRACE_OVER_TRANSPORT

$(NAME)_COMPONENTS := spp_lib.a
```

## spp.h

Create a new file in the project folder called `spp.h`. In that file, I just add the `#pragma once` and then provide a function prototype for the public interface to the server:

```
#pragma ONCE
void spp_start();
```

## spp.c

Create another new file in the project folder called spp.c. First, I'll add the *#defines* that are needed to get access to the required functions:

In spp.c you first need add *#includes* to get all the required functions and *#defines* for some SPP parameters:

```
#include "spp.h"
#include "wiced.h"
#include "stddef.h"
#include "bt_types.h"
#include "wiced_bt_spp.h"
#include "wiced_bt_trace.h"

#define SPP_RFCOMM_SCN (1)
#define MAX_TX_BUFFER (1017)
```

Next you need to include forward declarations for the SPP handler functions that we will write in a minute.

```
static void      spp_connection_up_callback(uint16_t handle, uint8_t* bda);
static void      spp_connection_down_callback(uint16_t handle);
static wiced_bool_t spp_rx_data_callback(uint16_t handle, uint8_t* p_data, uint32_t data_len);
```

Then you declare a variable called `spp_handle` to hold the current handle of the SPP connection and a structure of type `wiced_bt_spp_reg_t` called `spp_reg` which holds all the configuration information for the SPP Server.

```
static uint16_t      spp_handle;

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 */
};
```

Then write a function to startup the SPP server. It just needs to call the startup function from the library with the configuration structure you defined above.

```
void spp_start()
{
    // Initialize SPP library
    wiced_bt_spp_startup(&spp_reg);
}
```



The connection up and down 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 you receive data just dump it out onto the screen in the RX data callback.

```

/*
 * 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]);

    for(i=0;i<data_len;i++)
        WICED_BT_TRACE("%c", p_data[i]);
    WICED_BT_TRACE("\n");

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

```

Note that the implementation above does not send data – it is RX only – but if you look at the #if LOOPBACK\_DATA directive you can see how data can be sent using the wiced\_bt\_spp\_send\_session\_data function. That function can be called anywhere in your application to send data over the SPP interface. It needs the handle to the SPP service, a pointer to the data to send, and the length of the data in bytes. Since it requires the handle, it is easiest to include an additional



function (e.g. `spp_send_tx_data`) in `spp.c/.h` as part of the public interface so that it has access to the `spp_handle` variable that we created earlier.

### <appname>.c

In the main application, you need to add an include for `spp.h`:

```
#include "spp.h"
```

Then you need to initialize and start the SPP server at the end of the `<appname>_app_init` function:

```
/* Start the SPP server */  
spp_start();
```

### wiced\_bt\_cfg.c

The last thing we need to do is to increase the buffer pool sizes. This is necessary because the SPP uses more memory than is allocated by BT Designer, so it will not work with the default values. 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 */  
    { 272, 8, }, /* 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, 2, }, /* Extra Large Buffer Pool (used for SDP Discovery) */  
};
```



## 6A.8 Exercises

### Exercise - 6A.1 Create a Serial Port Profile Project

#### Project Creation

For this example, you will need to:

1. Follow the instructions in section 6A.5 to create a new project using BT Designer.
2. Update the project to handle pairing as documented in section 6A.7.4
3. Update the project to handle bonding as documented in section 6A.7.5
4. Update the project with the SPP functionality as documented in section 6A.7.6

Hint: Remember to use your initials in the project name (i.e. device name) so that you can find it in the list of devices that will be advertising.

Hint: Remember to add the option `BT_DEVICE_ADDRESS=random` to the make target so that your device's address will not conflict with another kit in the class.

Once the project has been created, you can move it into the `wbt101/ch06a` folder if you want to keep things organized (e.g. `apps/wbt101/ch06a/ex01_<inits>_ClassicSpp`). If you do that, remember to update the Make Target path too.

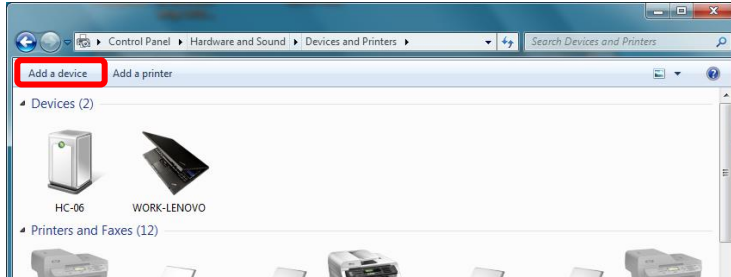
#### Testing

Once your project seems to be working, you can attach to it using Windows 7, Windows 10, MacOS or Android. Instructions for each are provided below.

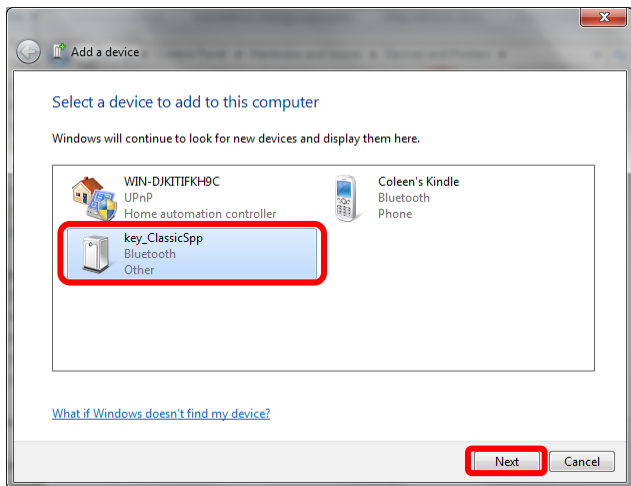
Note that iOS does not support SPP directly, so you can't use an iPhone to test this project. Apple supports Classic Bluetooth with iAP2 (iPod Accessory Protocol) which works a bit differently than SPP and requires an MFi license.

## PC Instructions (Windows 7)

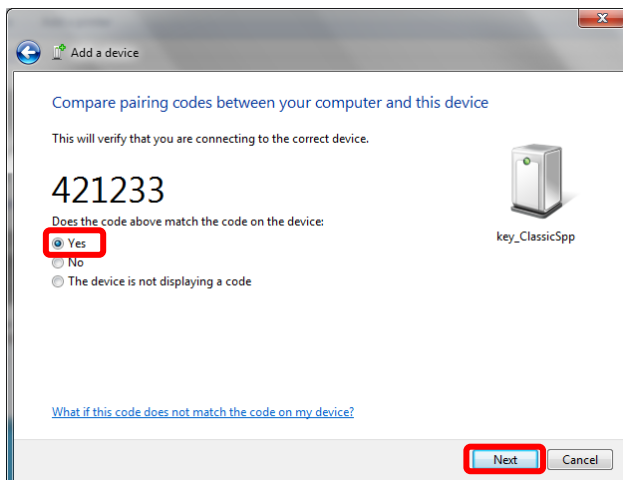
The first step is to pair your PC with the WICED Bluetooth device. Go to Control Panel -> Hardware and Sound -> Devices and Printers -> Add a Device.



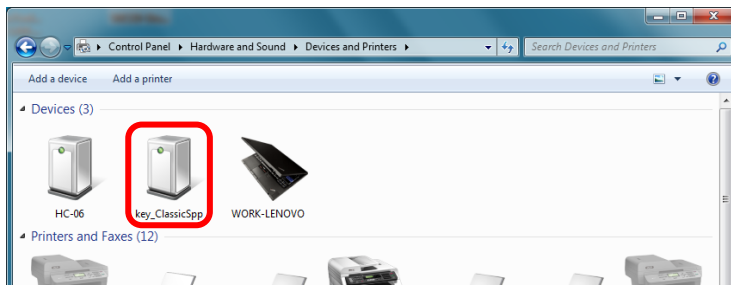
Wait until your device shows up in the list. Select it and click "Next".



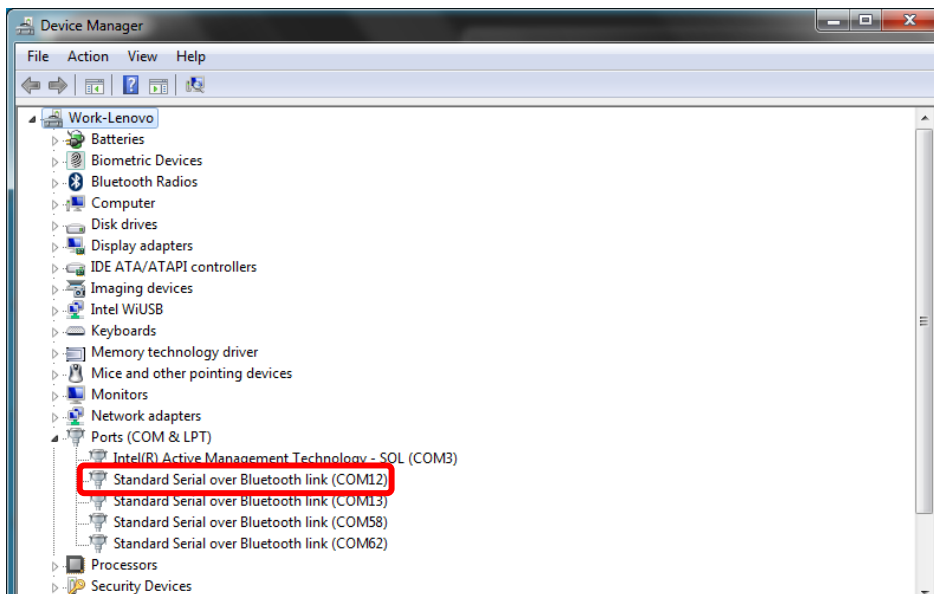
Compare the 6-digit code with the one displayed on your UART terminal. If the two numbers match, make sure "Yes" is selected and click "Next".



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.



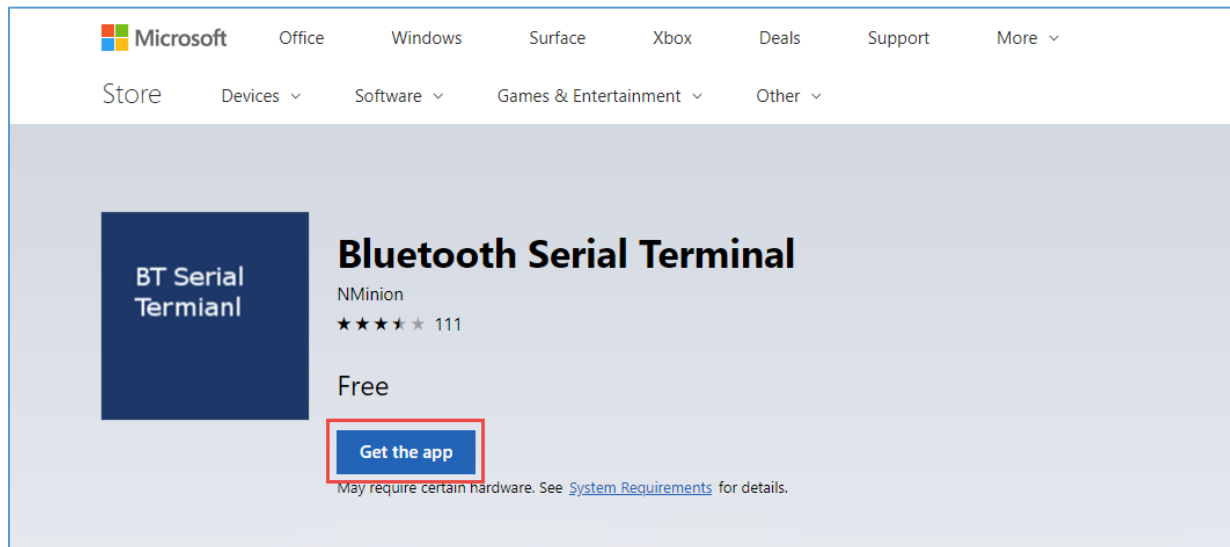
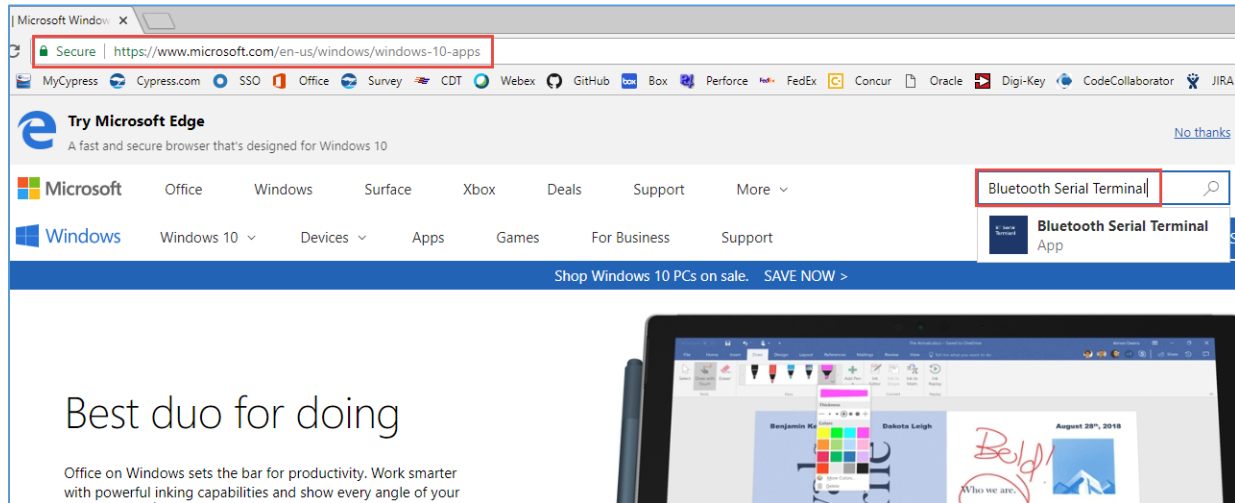
Open a serial terminal program of your choice (such as Putty) and connect to the SPP COM port. Now you can type in characters in the terminal window for the Bluetooth device and you will see them being received in the WICED kit by watching in terminal window connected to the kit's PUART.

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 pairing information.

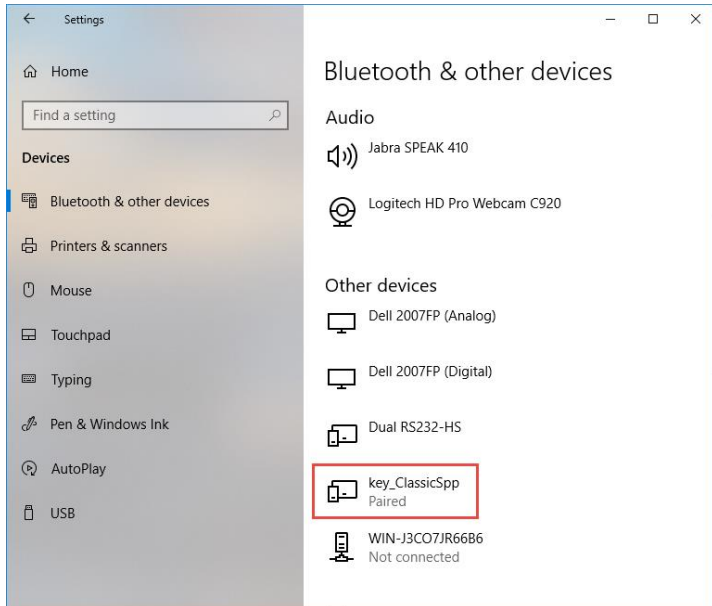
## PC Instructions (Widows 10)

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.

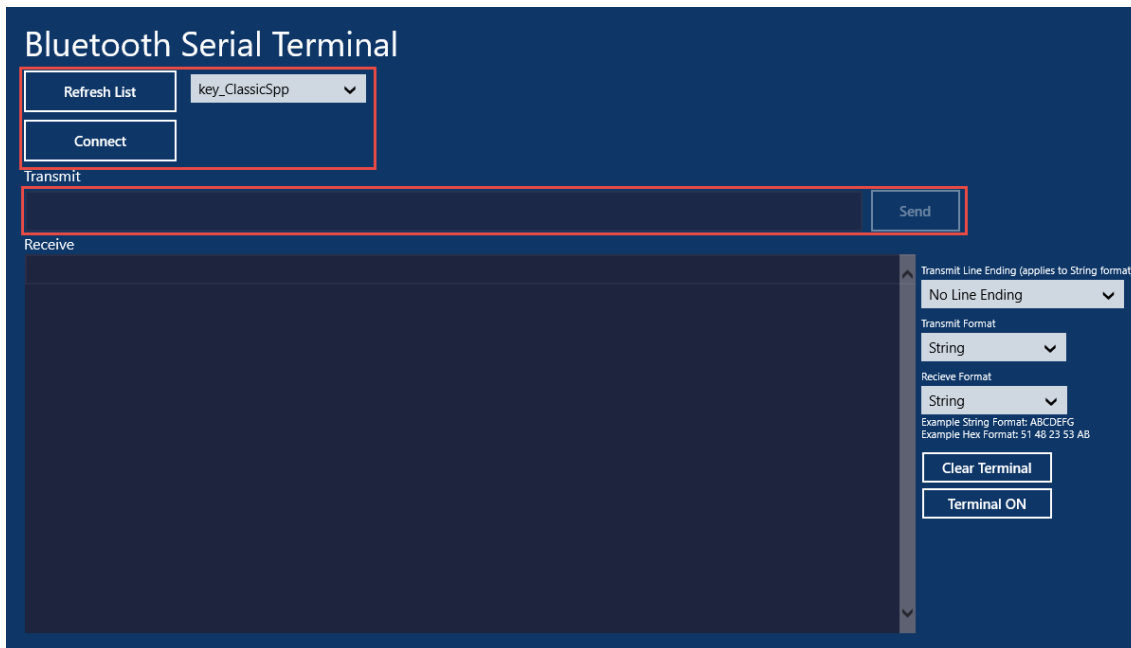
First, go to the Windows 10 Apps store (<https://www.microsoft.com/en-us/windows/windows-10-apps>), search for "Bluetooth Serial Terminal", and install it.



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. Compare the 6-digit code with the one displayed on your UART terminal. If the two numbers match, click on "Connect". Click "Done". Your device should now show up in the list of devices as "Paired":

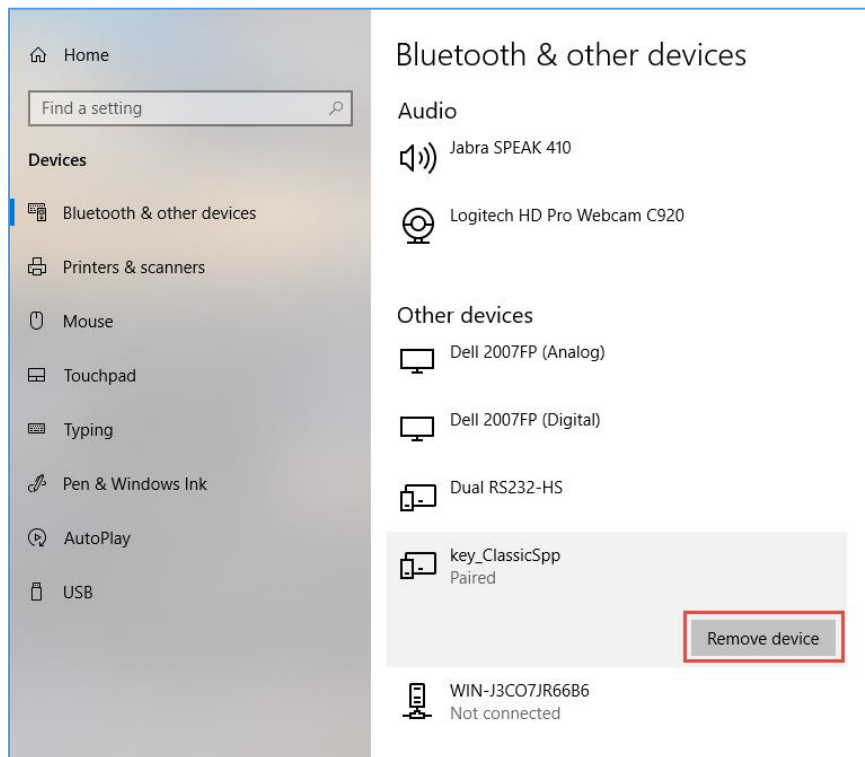


Now open the Bluetooth Serial Terminal app that you installed earlier. Your device should show up in the list. If not click "Refresh List".



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 Project. Observe the strings being received in the WICED kit by watching in the UART terminal.

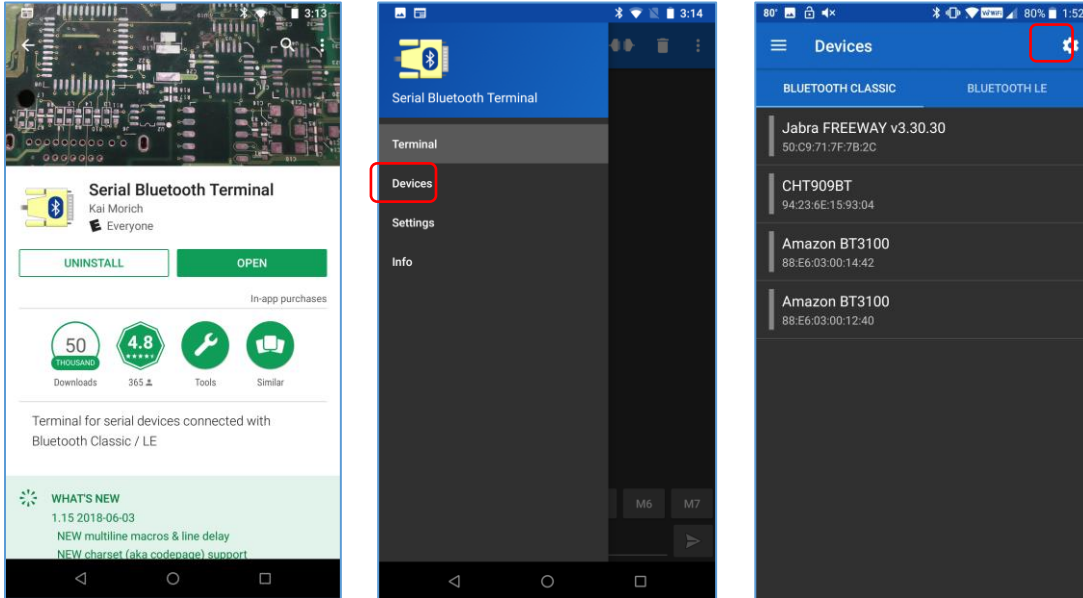
When you are done testing, click "Disconnect", close the Bluetooth Serial Terminal app and then go into the computer's Bluetooth settings to remove the device's pairing information (Settings -> Devices -> <appname>\_ClassicSpp -> Remove device -> Yes).



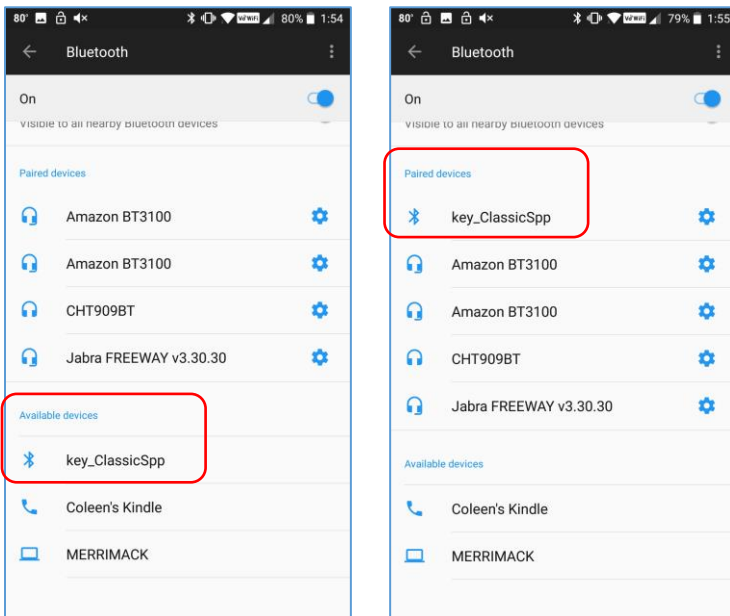


## Android Instructions

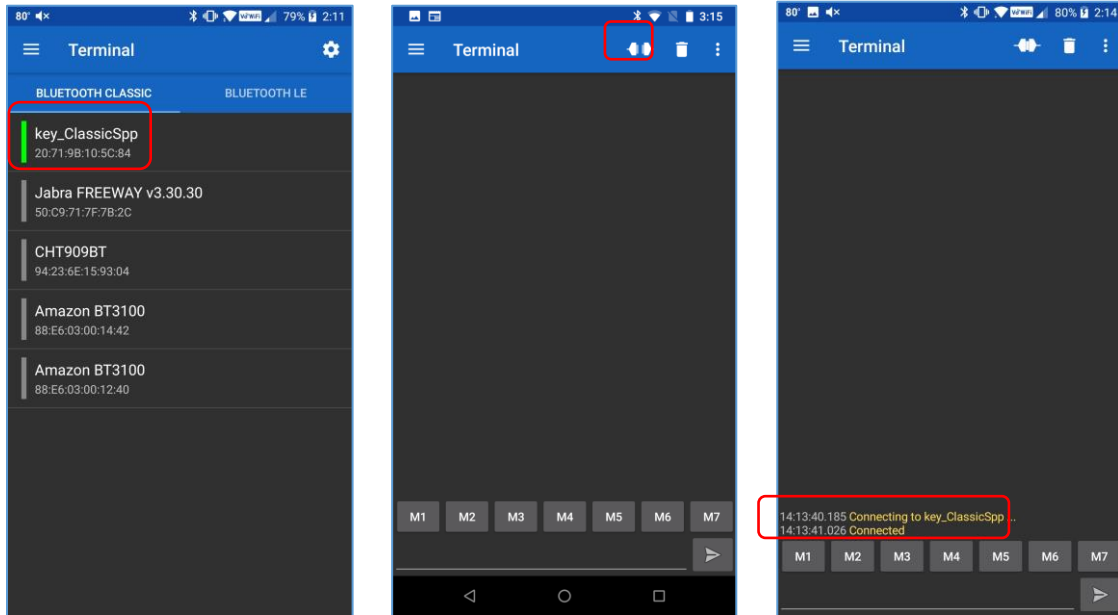
On an Android phone, you can install "Serial Bluetooth Terminal" from the Google Play Store. When you run the App, you will need to pair with your development kit. To do that open the menu (3 lines near the upper left corner) and tap on "Devices". From the Devices page, click on the "Gear" icon. This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



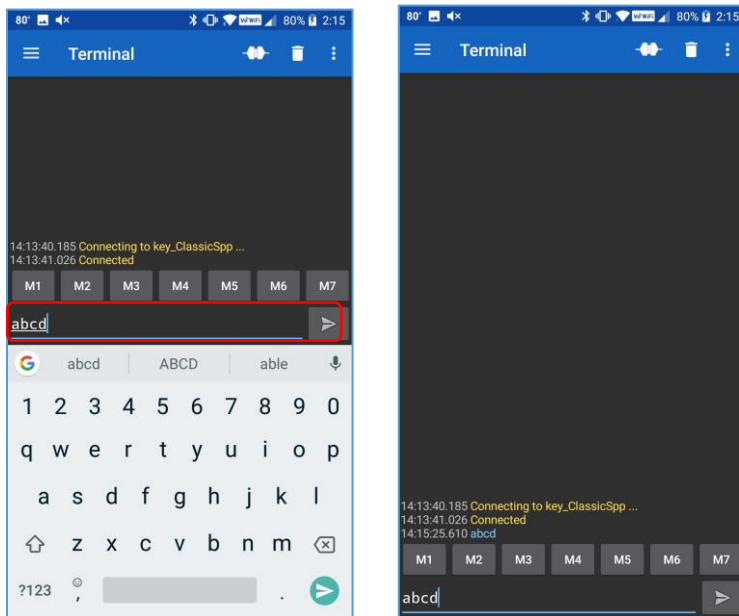
Find your device in the list and Pair with it (the exact procedure may be slightly different depending on the version of Android you are running).



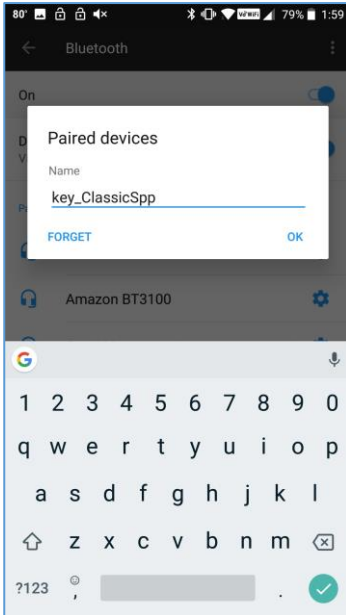
Once the device has been added, press the back arrow and you will see that your device appears in the Devices list. Tap on it to make it the active device (it will have a green bar to the left of the name when it is active). Then open the menu and select "Terminal" to see the blank terminal window. Next, 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.



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.

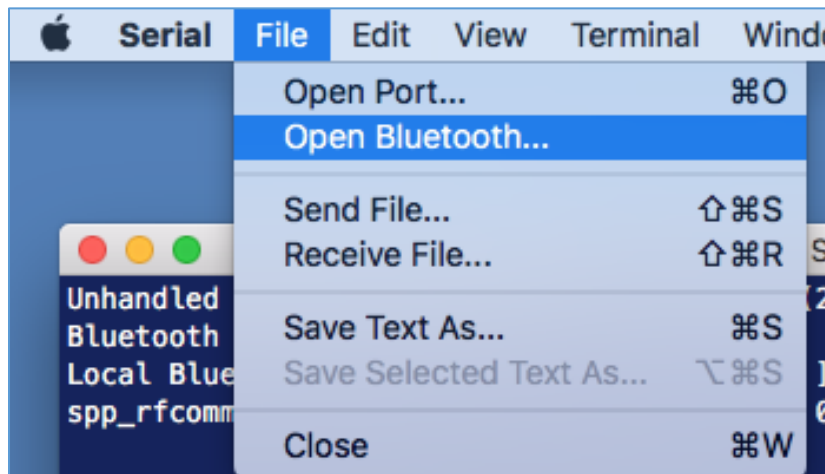


## Mac Instructions

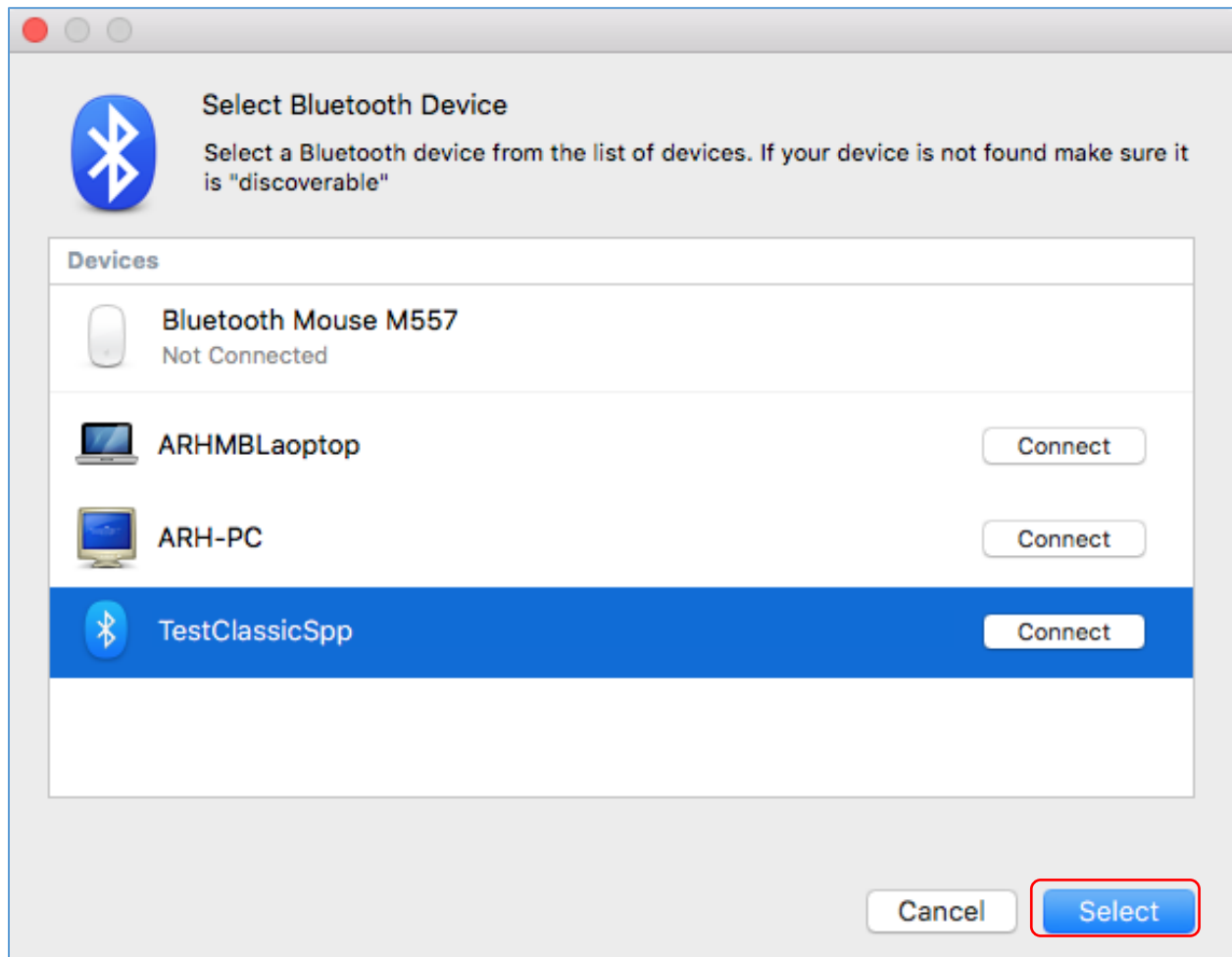
Install "Serial" from Decisive Tactics onto your Mac. You can get it in the App Store.



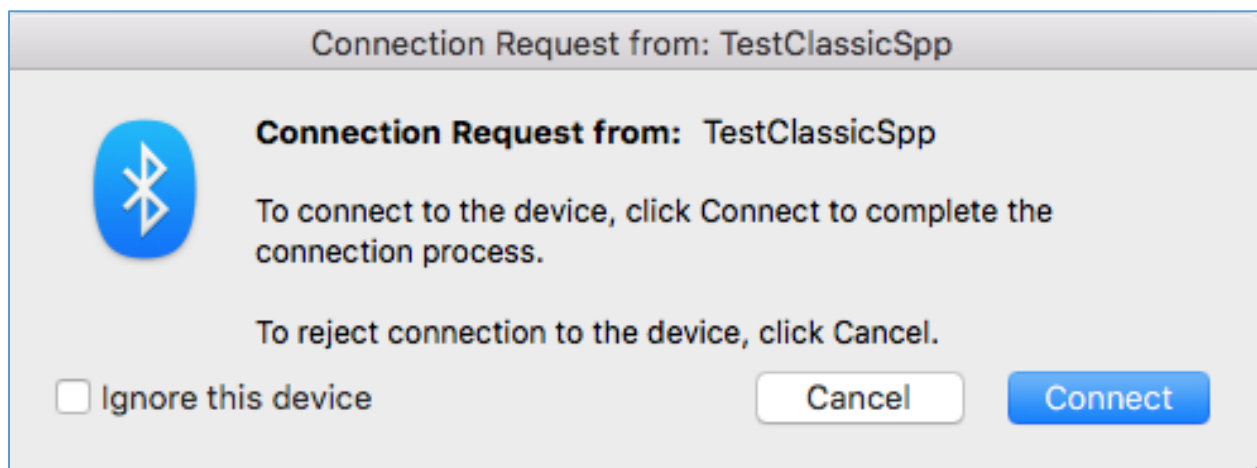
Once you have programmed the development kit you need to connect to it from the Mac. In the Serial program choose File → Open Bluetooth.



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



You will be asked to confirm the connection.



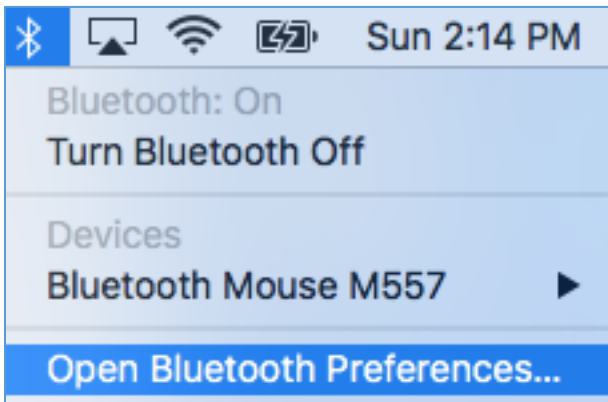
Once it is connected, everything you type will appear in the console window of the WICED Development kit. Below you can see that I typed "asdf".

```

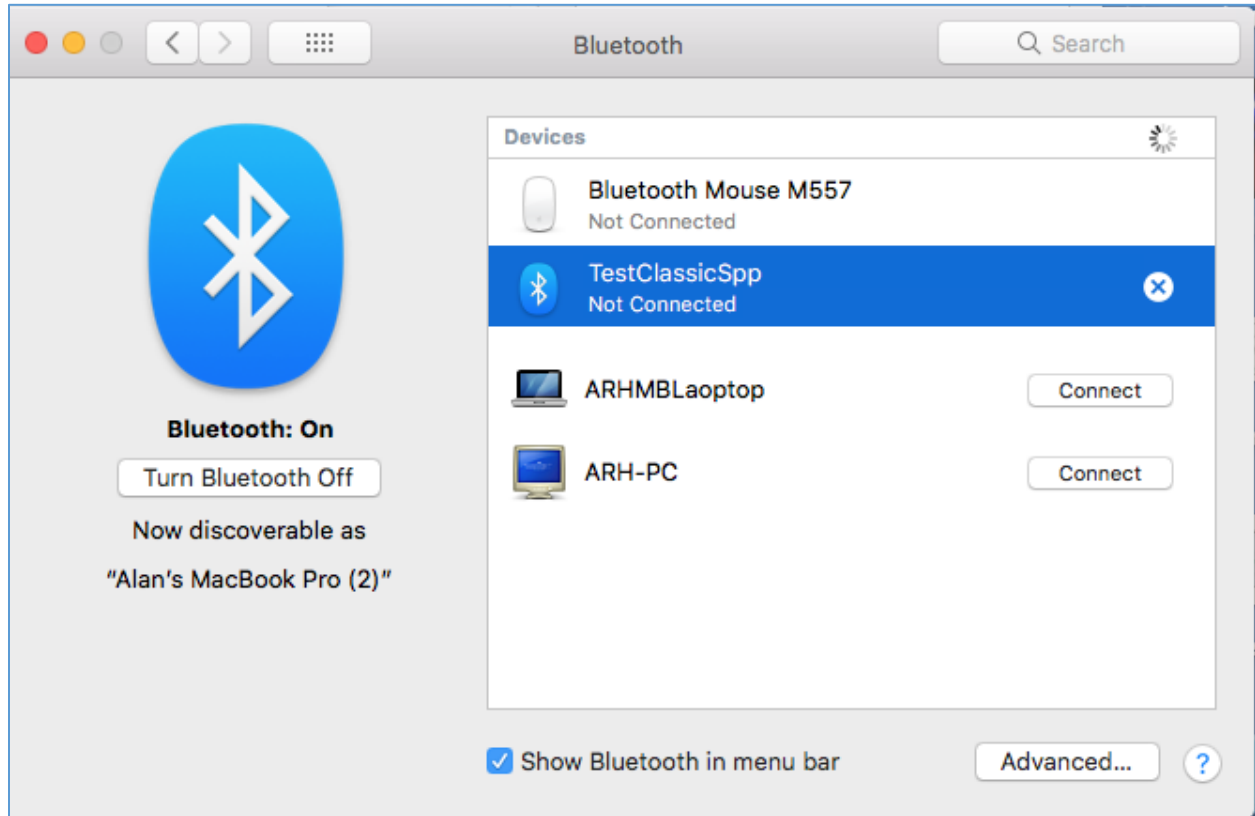
Local Bluetooth Address: [20 71 9b 17 19 a2 ]
spp_rfcomm_start_server: rfcomm_create Res: 0x0 Port: 0x0001
IO_CAPABILITIES_BR_EDR_RESPONSE peer_bd_addr: 78 4f 43 a2 64 f6 , peer_io_cap: 1, peer_oob_data: 0, peer_auth_req:
2
BR/EDR Pairing IO cap Request
numeric_value: 664177
Pairing Complete 0.
BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT
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

```

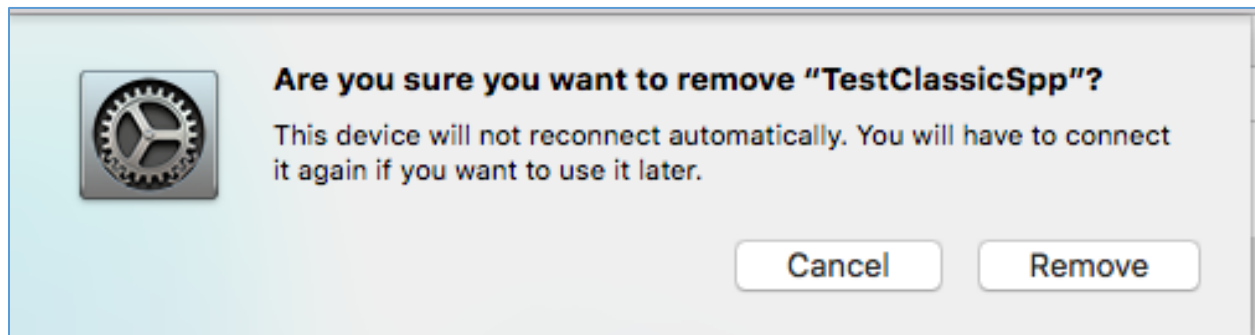
To unpair your development kit, select the Bluetooth symbol and pick "Open Bluetooth Preferences"



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

Make a copy of Exercise - 6A.1 and modify it 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.

Hint: There is an example in the Peripherals chapter that receives characters from a terminal window. Refer to that if you need help determining how to read characters from the PUART.

Hint: Add a new function called *spp\_tx\_data* to *spp.c* and *spp.h* 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 from the main application whenever a keystroke is received from the PUART.

Hint: You can rename the project/files in the new project if desired. If you want to change the name of the device that shows up on the Bluetooth scan, it is inside the file *wiced\_bt\_cfg.c*.





### Exercise - 6A.3 (Advanced) Improve Security by Adding IO Capabilities (Yes/No)

In this exercise, we are going to change the previous exercise to add confirmation on the WICED device. As before, a value will be displayed on both ends of the connection (WICED device and the phone or PC). The user will need to compare the two values and then perform the confirmation step on both devices before the connection is established.

Note: the phone or PC may or may not display the value and it may or may not require user input – this is up to the phone/PC application. In the case where the phone/PC automatically confirms the value you will still have to accept the connection on the WICED device end.

You will add the capability for the user to confirm that the numeric comparison value is correct on the WICED device using the "y" and "n" keys in the UART terminal.

To make this work you need to:

1. Make a copy of Exercise - 6A.2 and modify it as follows:
  - a. Add code to the `BTM_USER_CONFIRMATION_REQUEST_EVT` that will turn on the interrupts for the two mechanical buttons and inform the user that they should press the correct button.
    - i. Hint: This event will provide the `BDADDR` for the master that is trying to pair. You should save this value (hint: `memcpy`) to a global since you will need it when you reply in the interrupt.
  - b. Add a global variable called "doCompare" that is set when the user confirmation request is made and is reset after the user enters their Yes or No response.
  - c. In the PUART RX interrupt, if doCompare is set, look for "y" or "n" and then call `wiced_bt_dev_confirm_req_reply()` with the appropriate response. If doCompare is not set, then just send the value to the SPP interface as before.
    - i. Hint: Look at the function declaration to determine what information you need to send with the reply based on which button was pressed.
    - ii. Hint: Make sure you remove the existing `wiced_bt_dev_confirm_req_reply()` call from the `BTM_USER_CONFIRMATION_REQUEST_EVT`.

Hint: You can rename the project/files in the new project if desired. If you want to change the name of the device that shows up on the Bluetooth scan, it is inside the file `wiced_bt_cfg.c`.



## Exercise - 6A.4 (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 two changes to your current SPP implementation:

1. 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.
2. Handle reading multiple link keys. When you get the event `BTM_PAIRING_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.

Hint: You can rename the project/files in the new project if desired. If you want to change the name of the device that shows up on the Bluetooth scan, it is inside the file `wiced_bt_cfg.c`.

Once you have the project 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.