

Chapter 5A: Classic Bluetooth – The Wireless Serial Port

Time: 3 ¾ 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).

5A.1 WICED BLUETOOTH CLASSIC SYSTEM LIFECYCLE	2
5A.1.1 INQUIRY	3
5A.1.2 PAGE / CONNECT.....	4
5A.1.3 PAIR & BOND.....	4
5A.1.4 DISCOVER THE SERVICES USING SERVICE DISCOVERY PROTOCOL (SDP).....	4
5A.1.5 EXCHANGE DATA WITH THE SERIAL PORT PROFILE.....	5
5A.2 SECURE SIMPLE PAIRING.....	5
5A.3 SERVICE DISCOVERY PROTOCOL (SDP).....	5
5A.4 L2CAP, RFCOMM & THE SERIAL PORT PROFILE	6
5A.4.1 L2CAP	6
5A.4.2 RFCOMM	7
5A.4.3 SERIAL PORT PROFILE	7
5A.5 WICED BLUETOOTH DESIGNER	7
5A.5.1 WICED STUDIO 6.2 – BT DESIGNER BUGS	15
5A.6 WICED BLUETOOTH STACK EVENTS	16
5A.7 WICED CLASSIC BLUETOOTH FIRMWARE ARCHITECTURE	17
5A.7.1 INITIALIZATION FUNCTIONS.....	17
5A.7.2 SDP DATABASE.....	17
5A.7.3 HANDLE PAIRING.....	20
5A.7.4 HANDLE BONDING.....	20
5A.7.5 SERIAL PORT PROFILE	22
5A.8 EXERCISES.....	26
EXERCISE - 5A.1 CREATE A SERIAL PORT PROFILE PROJECT	26
EXERCISE - 5A.2 ADD UART TRANSMIT	36
EXERCISE - 5A.3 IMPROVE SECURITY BY ADDING IO CAPABILITIES (DISPLAY)	36
EXERCISE - 5A.4 IMPROVE SECURITY BY ADDING IO CAPABILITIES (YES/NO).....	36
EXERCISE - 5A.5 ADD MULTIPLE DEVICE BONDING CAPABILITY	36

5A.1 WICED Bluetooth Classic System Lifecycle

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 they exchange and save Pairing information. The Master then discovers the Services - i.e. the capabilities of the Slave. Finally, a basic wireless Serial Port data exchange connection is created.

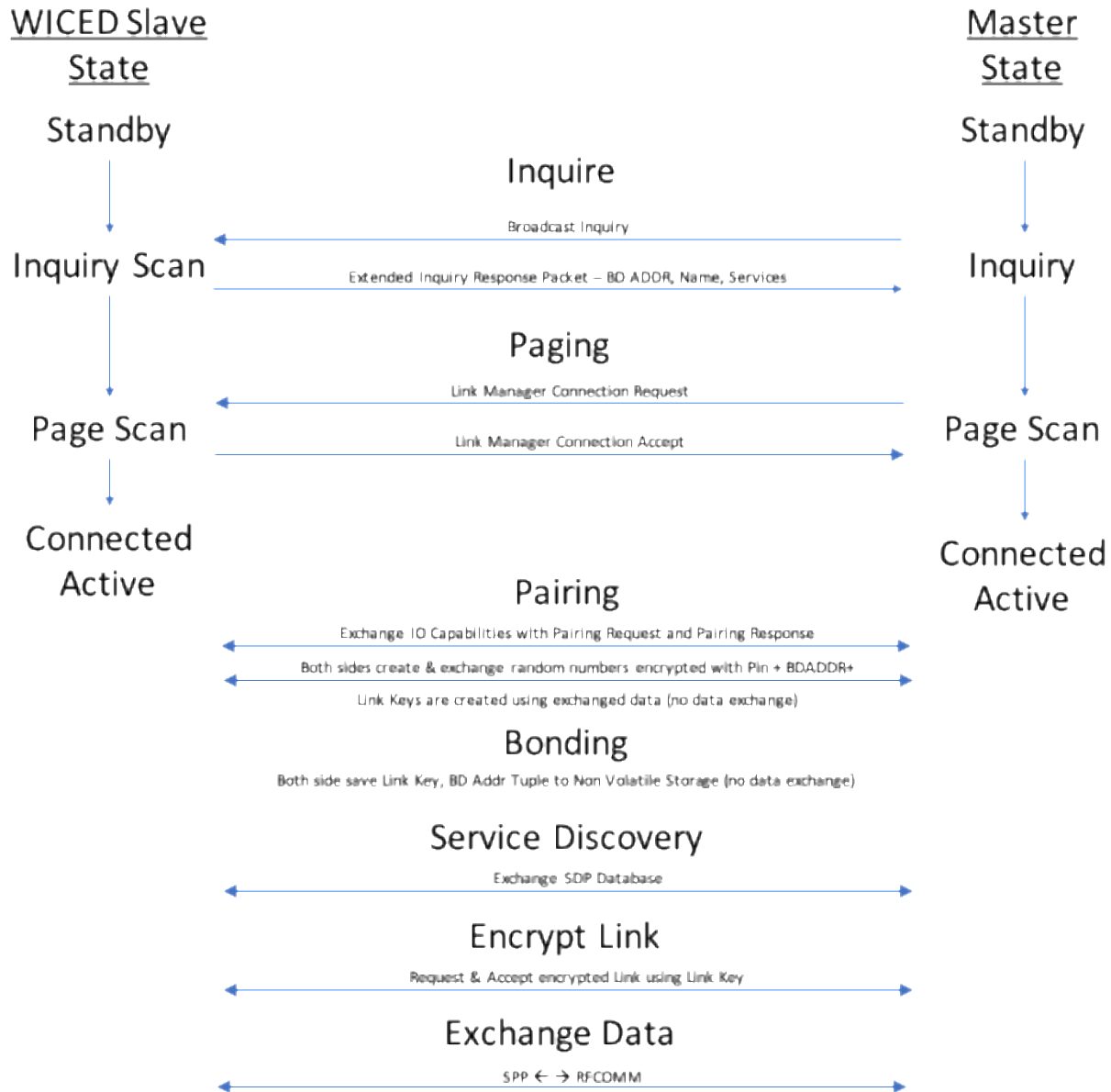
The five steps are:

1. Inquiry – Master finds a Slave to Connect
2. Paging – Master connects to Slave
3. Pair & Bond – A secure, authenticated connection is created
4. Service Discovery (SDP) – The Master figures out what the Slave can do
5. 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 the same four layers.

	Application	The code that you write to implement your system functionality.
Bluetooth Classic Stack	Host	Provides multiple connection paths to the application each with its own properties (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:



5A.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 a 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 (BDADDR) 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 make take up to 10ish seconds.

5A.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.

5A.1.3 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 the BLE process minus the Numeric Comparison method.

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 5A.2 .

5A.1.4 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 they are 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 the Bluetooth Master to query the SDP database.

More details on this in section 5A.3

5A.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 5A.4

5A.2 Secure Simple Pairing

Secure Simple Pairing is the same Pairing technique that we used in the BLE (except that the PIN is typically 4 digits instead of 6). You use a PIN code which is either:

1. Some trivial fixed value like '0000' or '1234' if you have no I/O capability
2. Displayed on one side, then entered on the other
3. Transmitted out of band (OOB), for example, using NFC

The PIN is then used to encrypt and transmit random numbers which are generated on both side of the connection.

Finally, the Pin + Random Numbers + some data about the device are combined into a Link Key which serves as a shared secret to identify and encrypt data between the devices.

5A.3 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.

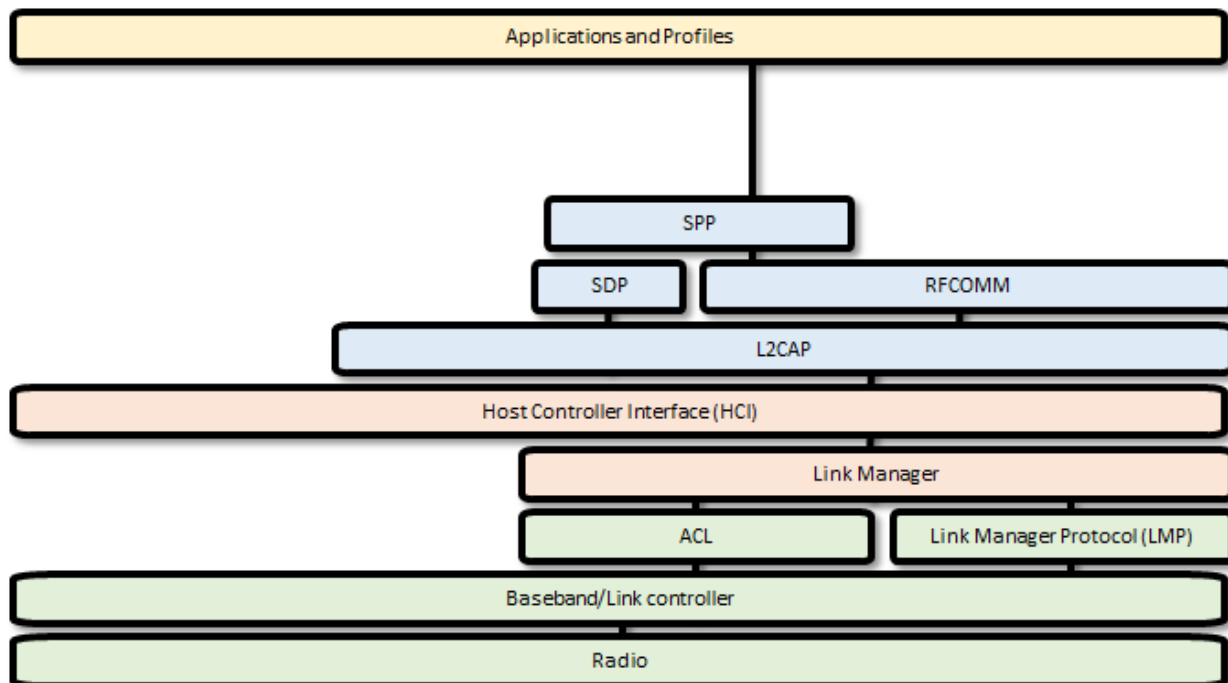
ServiceName – A plain text description of the Service.

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

5A.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.



5A.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. Provide flow control

5A.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.

5A.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.

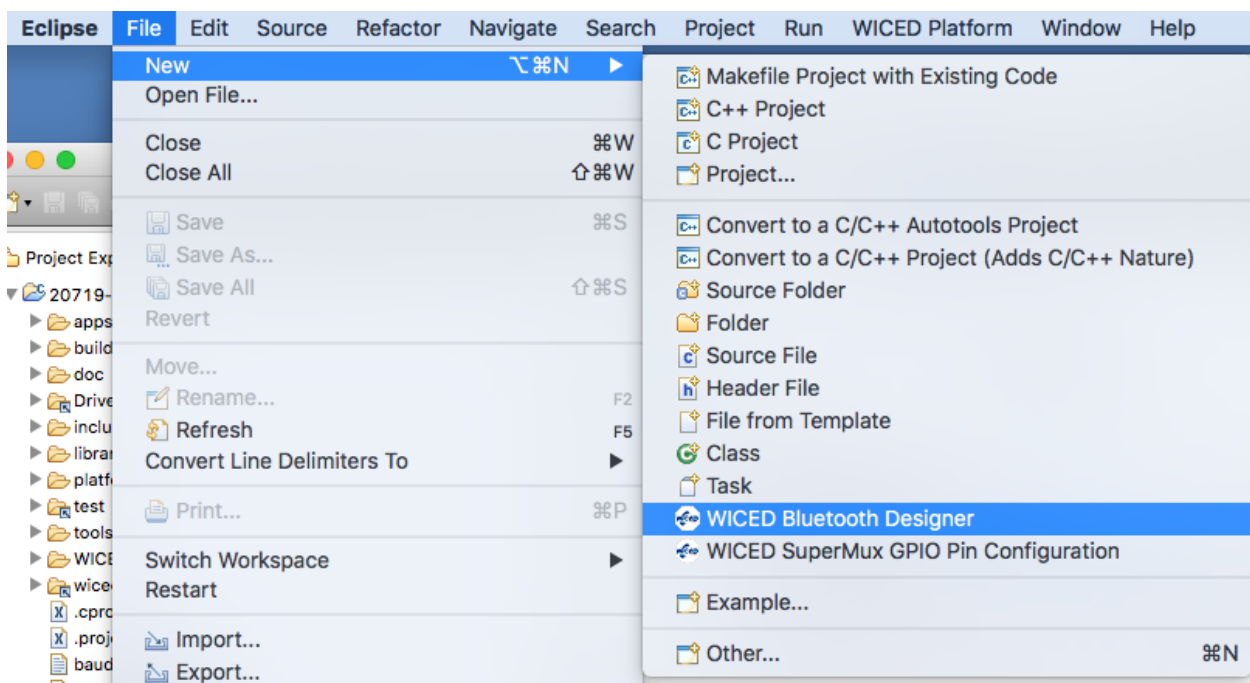
FYI, for iOS devices, the SPP is locked so it is only usable for MFi license holders. Their implementation is called iAP2.

5A.5 WICED Bluetooth Designer

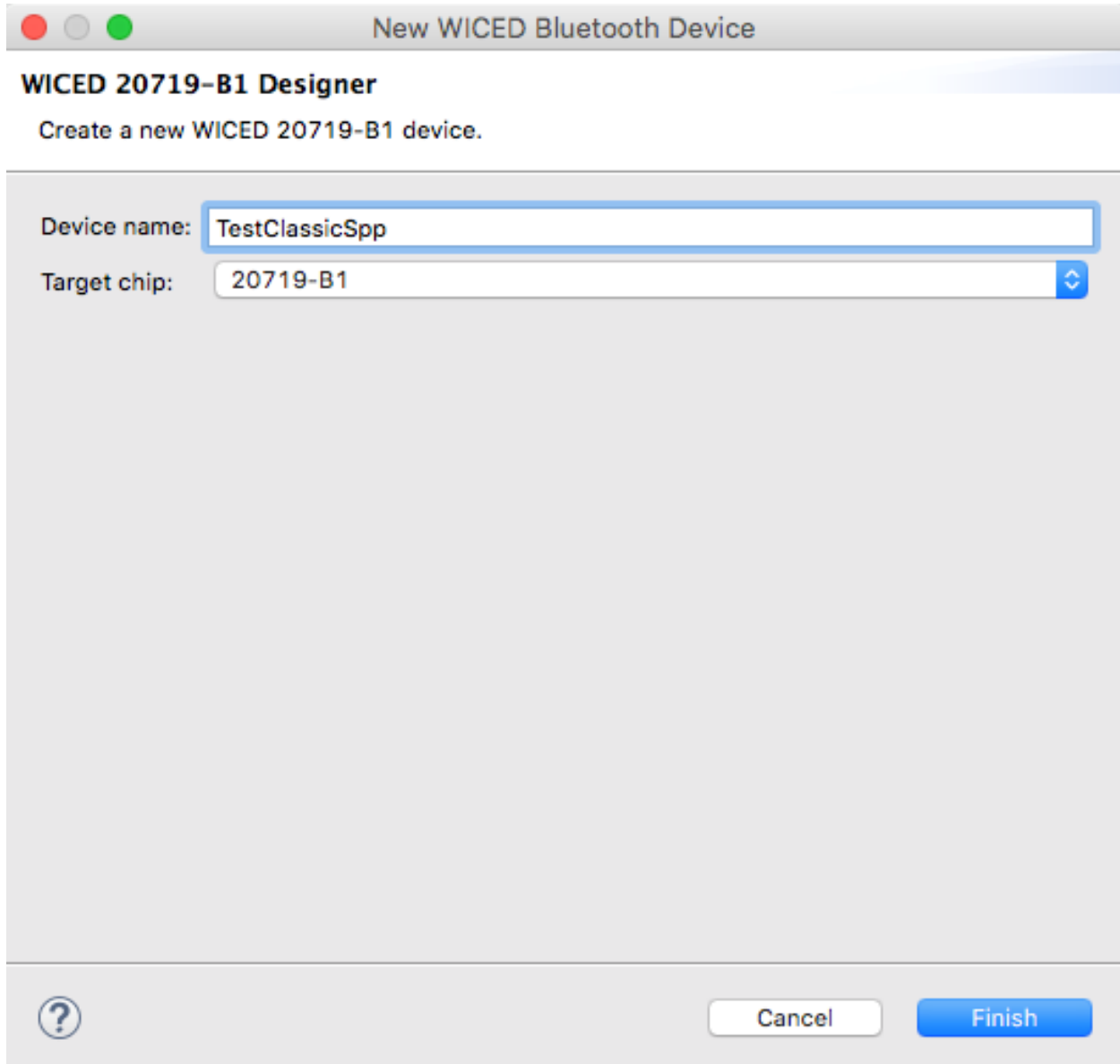
As with BLE, WICED Bluetooth Designer can be used to help you create a WICED Bluetooth Classic Project. Specifically, it will help you:

- 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 "TestClassicSpp".




New WICED Bluetooth Device

WICED 20719-B1 Designer

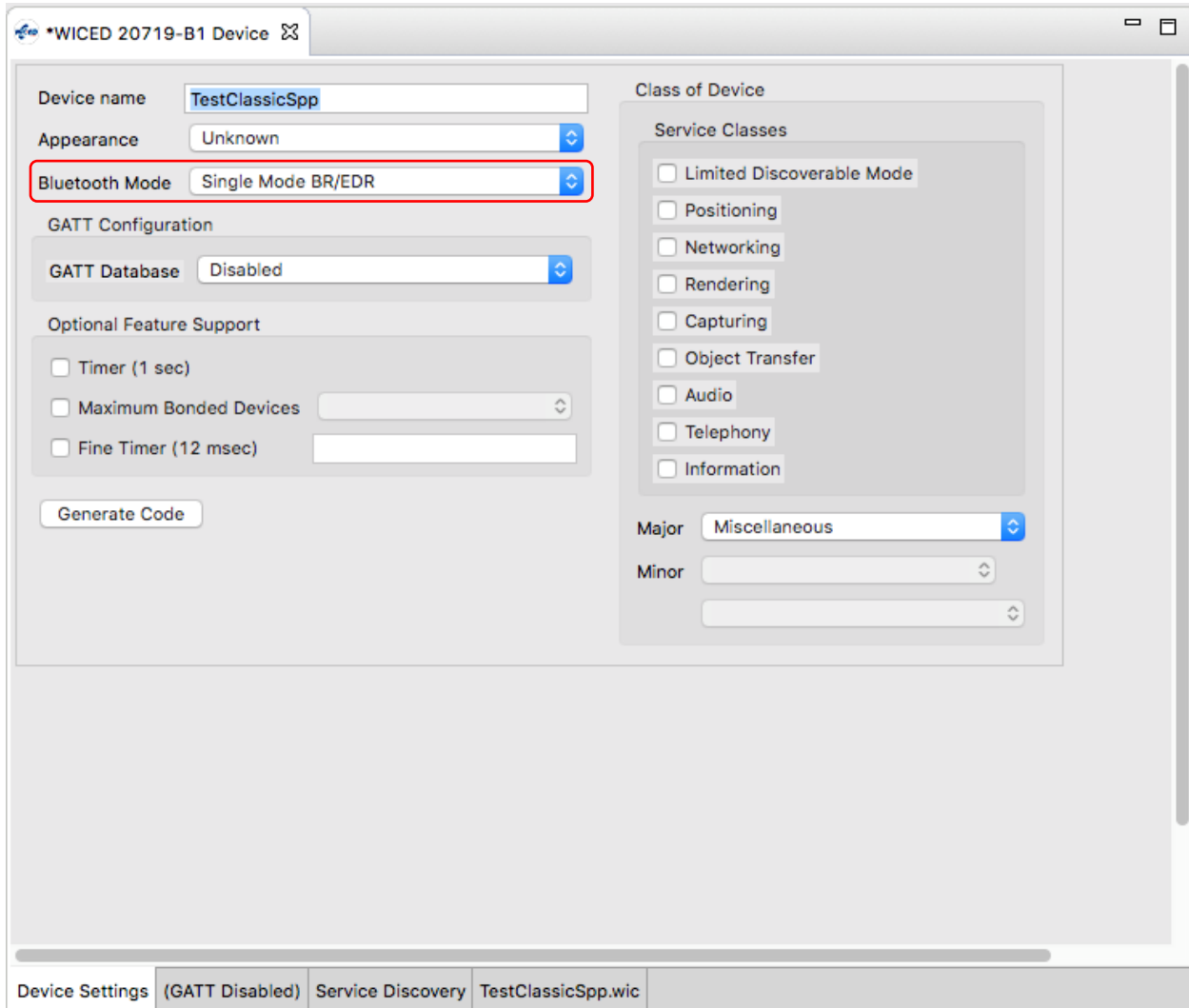
Create a new WICED 20719-B1 device.

Device name:

Target chip:



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”.



WICED 20719-B1 Device

Device name: TestClassicSpp

Appearance: Unknown

Bluetooth Mode: Single Mode BR/EDR

GATT Configuration

GATT Database: Disabled

Optional Feature Support

- ☐ Timer (1 sec)
- ☐ Maximum Bonded Devices
- ☐ Fine Timer (12 msec)

Generate Code

Class of Device

Service Classes

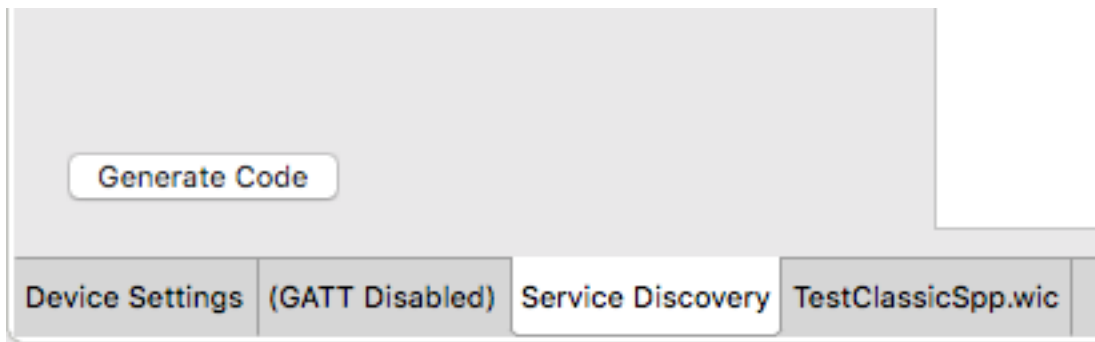
- ☐ Limited Discoverable Mode
- ☐ Positioning
- ☐ Networking
- ☐ Rendering
- ☐ Capturing
- ☐ Object Transfer
- ☐ Audio
- ☐ Telephony
- ☐ Information

Major: Miscellaneous

Minor:

Device Settings (GATT Disabled) Service Discovery TestClassicSpp.wic

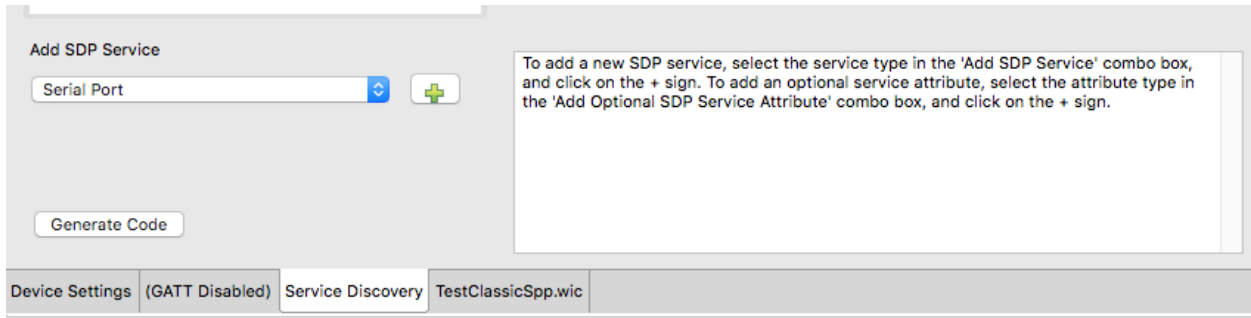
The biggest benefit of the Bluetooth Designer is helping you build the “Service Discovery” database. Click on the Service Discovery Tab.



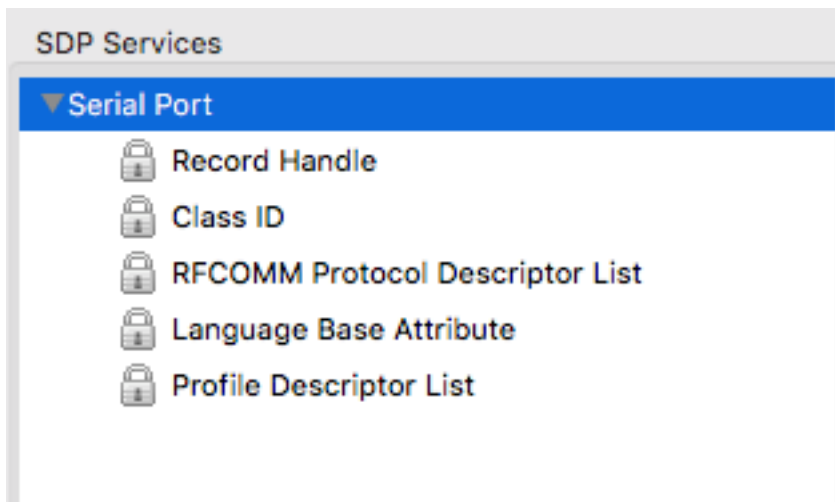
Generate Code

Device Settings (GATT Disabled) Service Discovery TestClassicSpp.wic

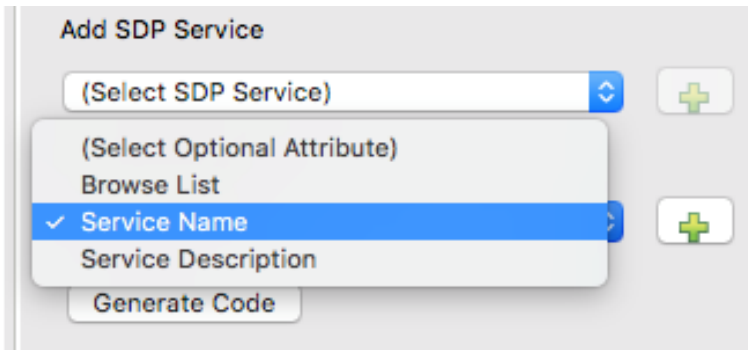
The SDP database starts with nothing in it. To add Services to the database, click on the drop-down menu title “Add SDP Service”, pick out the Service you want (in this case, Serial Port) then press the “+”.



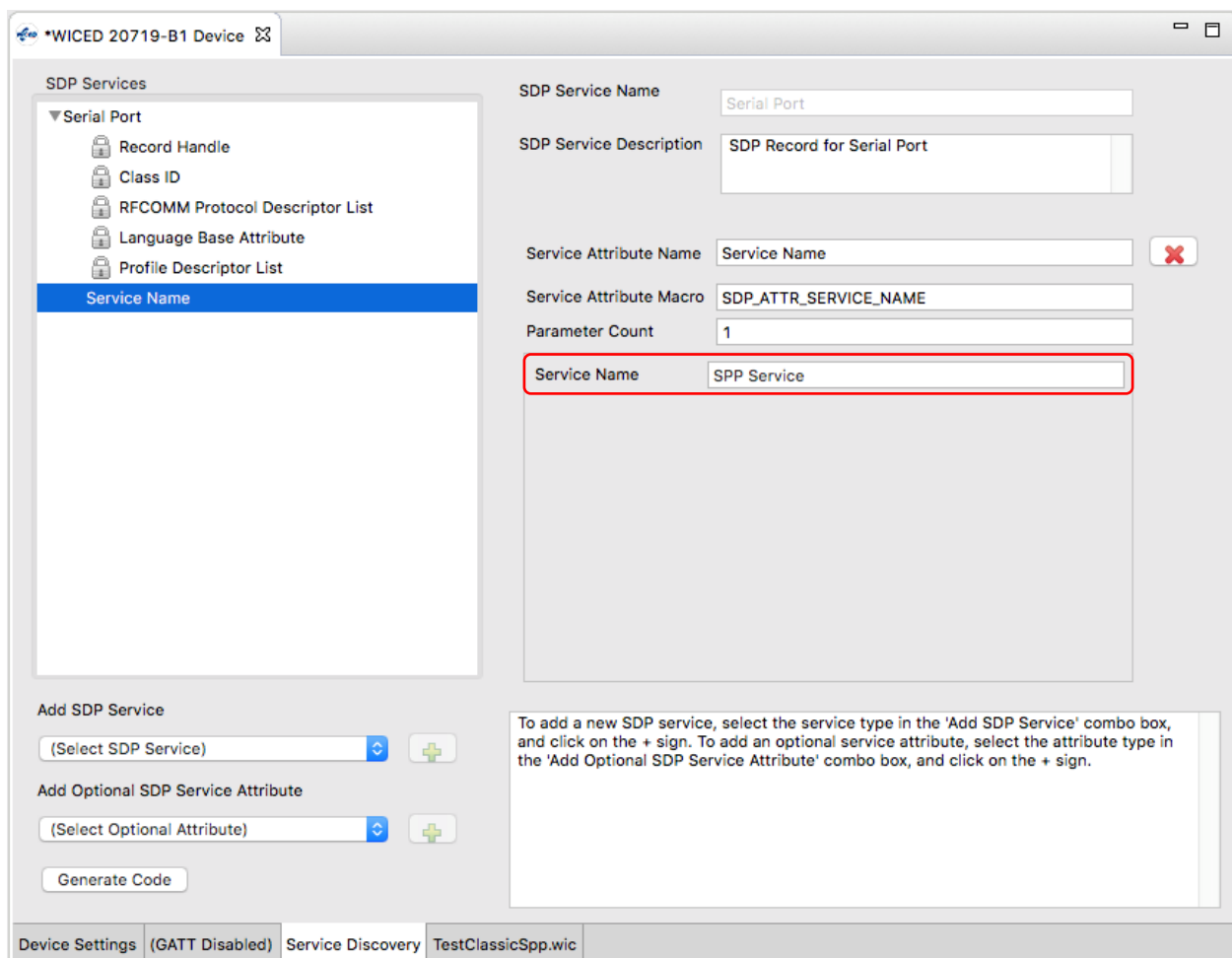
Now the SDP Database (which is in the window titled SDP Services) has the Serial Port and all the other stuff you need to go with it. Click on the Arrow to the left of “Serial Port” to expand the Attributes.



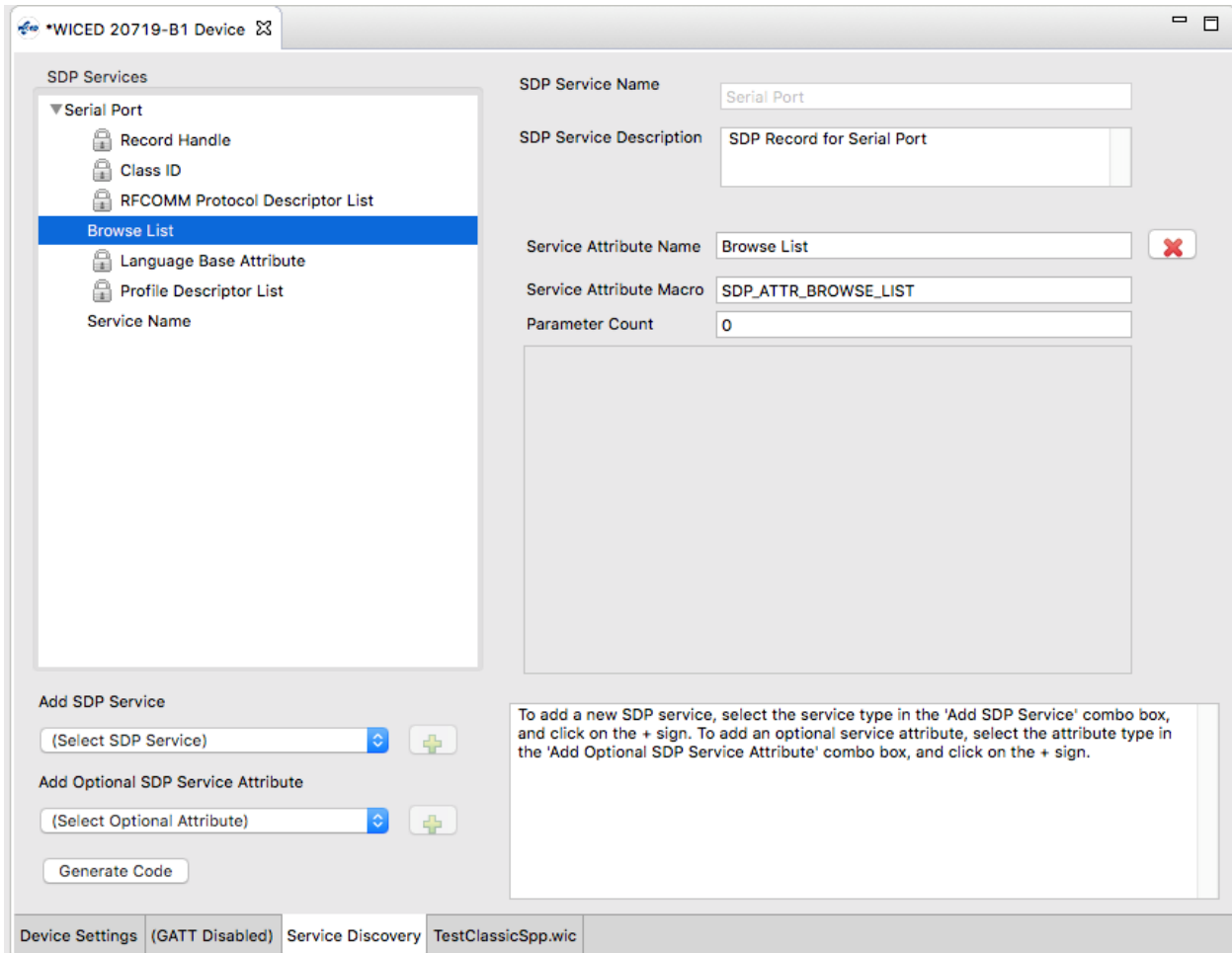
By default, the Service Name and Browse List are not included in the SDP Service Attributes. To add them, select them from the “Add Optional SDP Service Attribute” menu and press the “+”. Here I add the “Service Name”:



Next, I change the name to “SPP Service” by typing in the “Service Name” text box.



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



WICED 20719-B1 Device

SDP Services

- Serial Port
 - Record Handle
 - Class ID
 - RFCOMM Protocol Descriptor List
 - Browse List**
 - Language Base Attribute
 - Profile Descriptor List
 - Service Name

SDP Service Name: Serial Port

SDP Service Description: SDP Record for Serial Port

Service Attribute Name: Browse List

Service Attribute Macro: SDP_ATTR_BROWSE_LIST

Parameter Count: 0

Add SDP Service: (Select SDP Service) +

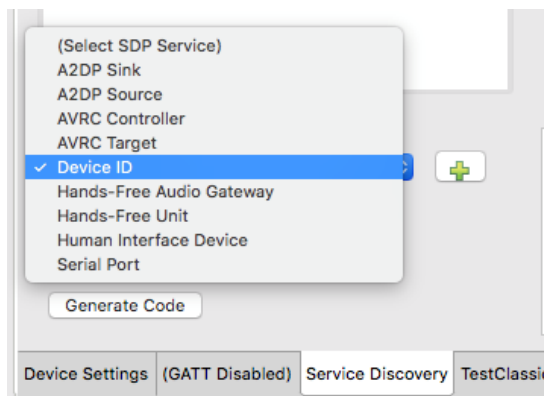
Add Optional SDP Service Attribute: (Select Optional Attribute) +

Generate Code

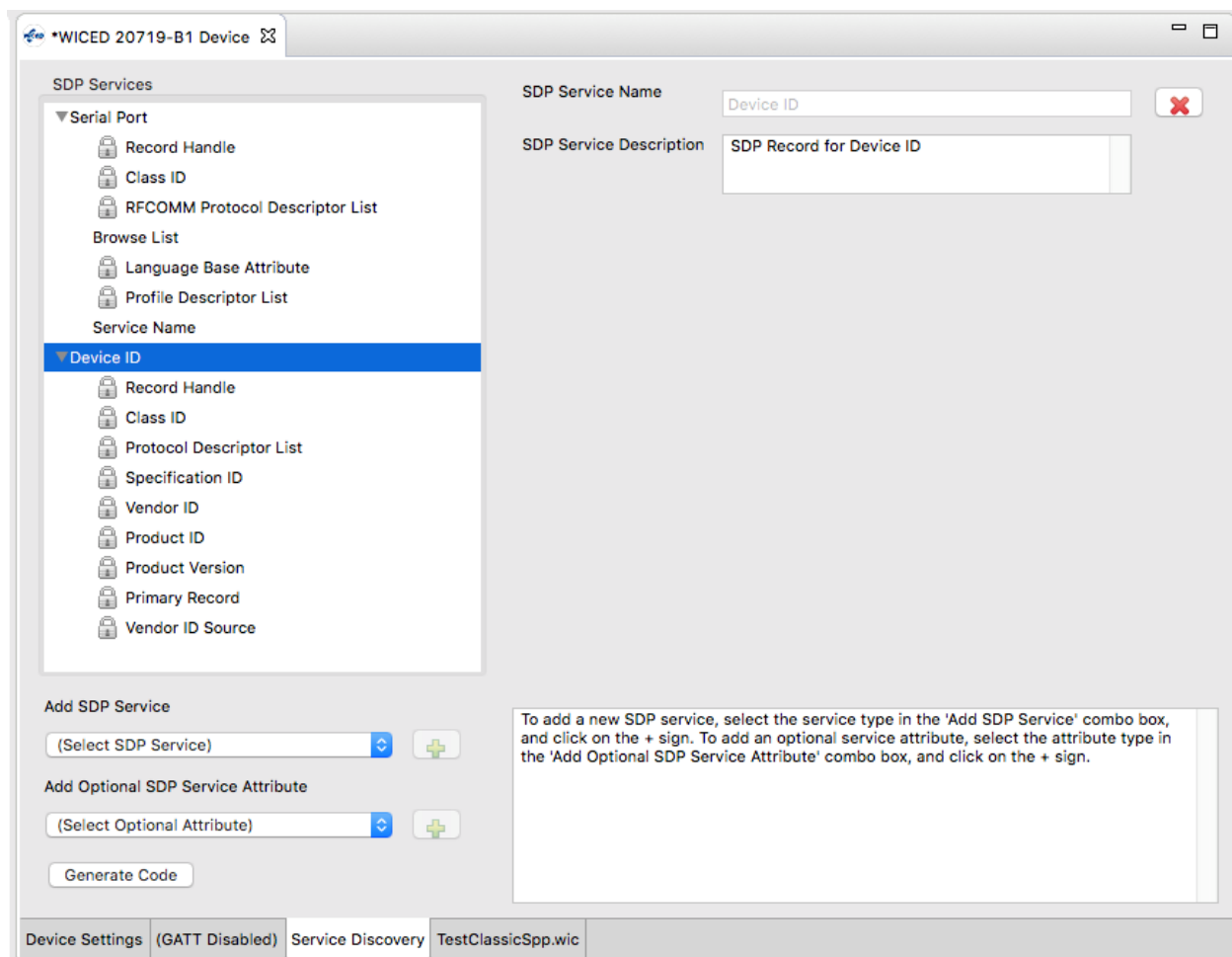
To add a new SDP service, select the service type in the 'Add SDP Service' combo box, and click on the + sign. To add an optional service attribute, select the attribute type in the 'Add Optional SDP Service Attribute' combo box, and click on the + sign.

Device Settings (GATT Disabled) Service Discovery TestClassicSpp.wic

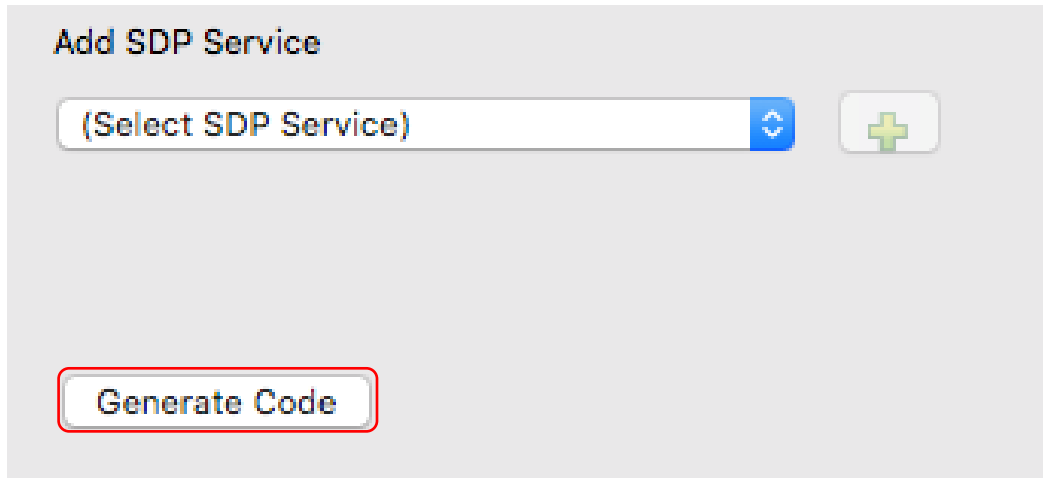
The next SDP Service that I add to the Database is the Device ID Service. This contains things like vendor ID, product ID, etc. which you can use to convey information about your device.



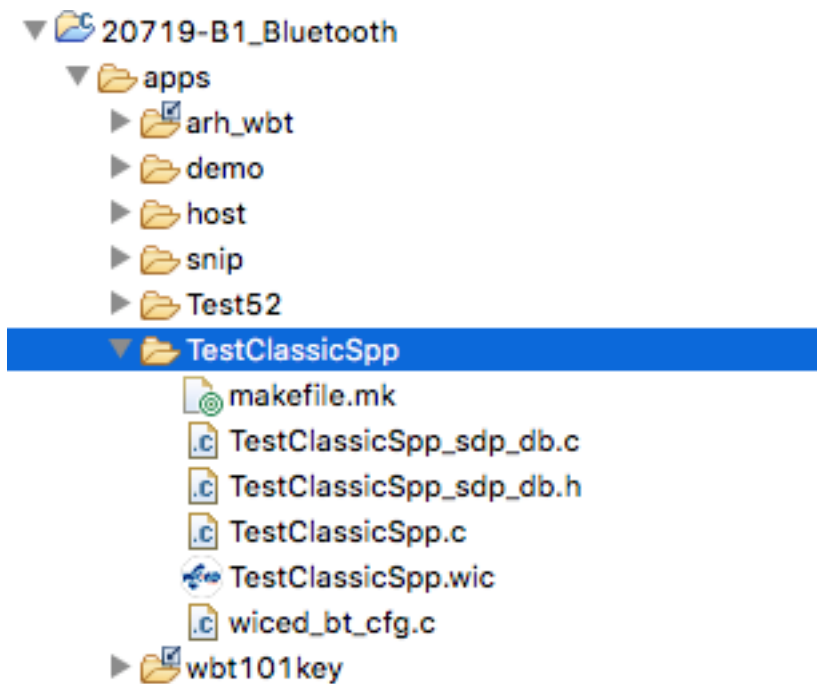
After it is added, you can see all the Attributes that go with it. You can change some of the data for the Attributes by clicking on the one you want to modify and then entering the new values. Only certain values are editable.



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.



If you look in TestClassicSpp_sdp_db.c you will find the array of uint8_t's that represent the database.

```
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, 0xFFFF),
    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_BT_SIG),
};

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

The sequence numbers that you see above indicate the length of the database. 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).

5A.5.1 WICED Studio 6.2 – BT Designer Bugs

In WICED Studio 6.2, BT Designer generates projects with missing includes. To get rid of these warnings you will need to add:

- #include "wiced_bt_stack.h"
- #include "wiced_bt_sdp.h"
- #include "wiced_hal_wdog.h"
- #include "wiced_bt_app_common.h"

5A.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
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_PAIRING_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	The Stack is asking you to ask the user if the PIN you are displaying matches the PIN from the other side.
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_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT	The Stack is asking your firmware to store the BDADDR/Link Keys (which are passed in the event data).

5A.7 WICED Classic Bluetooth Firmware Architecture

WICED Bluetooth Designer will create a skeleton of the firmware that you need start building your device. The skeleton includes a file named <appname>.c which contains:

- The initialization functions application_start which is the entry point for your firmware and <appname>_app_init which provides a place for you to get the Bluetooth stuff going.
- A template BTM event handler function.

A .c/.h pair of files called <appname>_sdp_db.c/.h that contain the #defines for the SDP database and the uint8_t structure holding the actual database.

A file wiced_bt_config.h which contains all the basic Bluetooth configuration settings to get the stack going.

To make your project work you need to add the capability to:

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

5A.7.1 Initialization Functions

The application_start function is the entry point of your 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 Paging)
- Makes your device discoverable (turns on Inquiry)

5A.7.2 SDP Database

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 SPP and Device ID Services that we created using BT Designer is shown here:

```
2⊕ * This file has been automatically generated by the WICED 20719-B1 Designer.[]
6
7 // TestClassicSpp_sdp_db.h
8
9 #ifndef __SDP_DATABASE_H__
10 #define __SDP_DATABASE_H__
11
12 // SDP Record for Serial Port
13 #define HDLR_SERIAL_PORT          0x10001
14 #define SERIAL_PORT_SCN          0x01
15 // SDP Record for Device ID
16 #define HDLR_DEVICE_ID           0x10002
17
18 // External definitions
19 extern const uint8_t sdp_database[];
20 extern const uint16_t sdp_database_len;
21
22 #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.

```

20 * This file has been automatically generated by the WICED 20719-B1 Designer.
6
7 // TestClassicSpp_sdp_db.c
8
9 #include "wiced_bt_uuid.h"
10 #include "wiced_bt_sdp.h"
11 #include "TestClassicSpp_sdp_db.h"
12
13 const uint8_t sdp_database[] = // Define SDP database
14 {
15     SDP_ATTR_SEQUENCE_1(157),
16
17     // SDP Record for Serial Port
18     SDP_ATTR_SEQUENCE_1(84),
19     SDP_ATTR_RECORD_HANDLE(HDLR_SERIAL_PORT),
20     SDP_ATTR_CLASS_ID(UUID_SERVCLASS_SERIAL_PORT),
21     SDP_ATTR_RFCOMM_PROTOCOL_DESC_LIST(SERIAL_PORT_SCN),
22     SDP_ATTR_BROWSE_LIST,
23     SDP_ATTR_LANGUAGE_BASE_ATTR_ID_LIST,
24     SDP_ATTR_PROFILE_DESC_LIST(UUID_SERVCLASS_SERIAL_PORT, 0x0102),
25     SDP_ATTR_SERVICE_NAME(11),
26     'S', 'P', 'P', ' ', 'S', 'e', 'r', 'v', 'i', 'c', 'e',
27
28     // SDP Record for Device ID
29     SDP_ATTR_SEQUENCE_1(69),
30     SDP_ATTR_RECORD_HANDLE(HDLR_DEVICE_ID),
31     SDP_ATTR_CLASS_ID(UUID_SERVCLASS_PNP_INFORMATION),
32     SDP_ATTR_PROTOCOL_DESC_LIST(1),
33     SDP_ATTR_UINT2(ATTR_ID_SPECIFICATION_ID, 0x103),
34     SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID, 0xFFFF),
35     SDP_ATTR_UINT2(ATTR_ID_PRODUCT_ID, 0xFFFF),
36     SDP_ATTR_UINT2(ATTR_ID_PRODUCT_VERSION, 0xFFFF),
37     SDP_ATTR_BOOLEAN(ATTR_ID_PRIMARY_RECORD, 0x01),
38     SDP_ATTR_UINT2(ATTR_ID_VENDOR_ID_SOURCE, DI_VENDOR_ID_SOURCE_BT_SIG),
39 };
40
41 // Length of the SDP database
42 const uint16_t sdp_database_len = sizeof(sdp_database);
  
```

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.

```

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

5A.7.3 Handle Pairing

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
- 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.

```

213 case BTM_PAIRING_IO_CAPABILITIES_BR_EDR_RESPONSE_EVT:
214     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",
215                     p_event_data->pairing_io_capabilities_br_edr_response.bd_addr,
216                     p_event_data->pairing_io_capabilities_br_edr_response.io_cap,
217                     p_event_data->pairing_io_capabilities_br_edr_response.oob_data,
218                     p_event_data->pairing_io_capabilities_br_edr_response.auth_req);
219     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 but no user entry, you would use BTM_IO_CAPABILITIES_DISPLAY_ONLY 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 cap 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_NO;
    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_ONLY;
    break;

```

When the pairing process is complete you can print a message, or more likely initialize some variable for that session.

```

case BTM_PAIRING_COMPLETE_EVT:
    /* Pairing is Complete */
    p_br_edr_info = &p_event_data->pairing_complete.pairing_complete_info.br_edr;
    WICED_BT_TRACE("Pairing Complete %d.\n", p_br_edr_info->status);
    break;

```

5A.7.4 Handle Bonding

To handle Bonding, you need to act on the BTM Events that request you to find the saved link key for a specific BD Address (BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT) and to save the Link Key when it is created or updated for a specific address (BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT).

This is best done by reserving a block of the NVRAM VSIDs to hold the key/BDADDR tuple. The VSID is just a row number in the NVRAM. Each Volatile Sector in the NVRAM is 255 bytes long. Each WICED

Device has #defines for the WICED_NVRAM_VSID_START and WICED_NVRAM_VSID_END in wiced_hal_nvr.h.

BT Designer provides a template for the keys request event. To use it change the #if to 1. You also need to provide the function <apname>_read_link_keys which we will do in a minute.

```

223     case BTM_PAIED_DEVICE_LINK_KEYS_REQUEST_EVT:
224         /* Paired Device Link Keys Request */
225         WICED_BT_TRACE("Paired Device Link Request Keys Event\n");
226         /* Device/app-specific TODO: HANDLE PAIRED DEVICE LINK REQUEST KEY - retrieve from NVRAM, etc */
227     #if 1
228         if (testclassicspp_read_link_keys( &p_event_data->paired_device_link_keys_request ))
229         {
230             WICED_BT_TRACE("Key Retrieval Success\n");
231         }
232         else
233     #endif
234         /* Until key retrieval implemented above, just fail the request - will cause re-pairing */
235         {
236             WICED_BT_TRACE("Key Retrieval Failure\n");
237             status = WICED_BT_ERROR;
238         }
239         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.

```

241     case BTM_PAIED_DEVICE_LINK_KEYS_UPDATE_EVT:
242         WICED_BT_TRACE("BTM_PAIED_DEVICE_LINK_KEYS_UPDATE_EVT\n");
243         testclassicspp_write_link_keys(&p_event_data->paired_device_link_keys_update);
244         break;
245

```

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.

```

262 // This function reads the first VSID row into the link keys
263 int testclassicspp_read_link_keys( wiced_bt_device_link_keys_t *keys )
264 {
265     wiced_result_t result;
266     int bytes_read;
267
268     bytes_read = wiced_hal_read_nvram(WICED_NVRAM_VSID_START, sizeof(wiced_bt_device_link_keys_t),
269                                     (uint8_t *)keys, &result);
270     WICED_BT_TRACE("NVRAM ID:%d read :%d bytes result:%d\n", WICED_NVRAM_VSID_START, bytes_read, result);
271
272     return bytes_read;
273 }
274
275 // This function write the link keys into the first VSID row
276 int testclassicspp_write_link_keys( wiced_bt_device_link_keys_t *keys )
277 {
278     wiced_result_t result;
279     int bytes_written = wiced_hal_write_nvram(WICED_NVRAM_VSID_START, sizeof(wiced_bt_device_link_keys_t),
280                                               (uint8_t *)keys, &result);
281
282     WICED_BT_TRACE("NVRAM ID:%d written :%d bytes result:%d\n", WICED_NVRAM_VSID_START, bytes_written, result);
283
284     return (bytes_written);
285 }
286
287
288

```

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

```

64
65 static int testclassicspp_read_link_keys( wiced_bt_device_link_keys_t *keys );
66 static int testclassicspp_write_link_keys( wiced_bt_device_link_keys_t *keys );
67
--

```

5A.7.5 Serial Port Profile

The WICED Bluetooth SDK contains all the code to implement an SPP server. To make it work you need to initialize the server and provide callbacks for starting and stopping the connection and receiving data. After you have created your SPP project with the BT Designer tool, you can open the snip.bt.spp example to copy the additional blocks of code that you need. Typically, I create separate files spp.h and spp.c to handle all the SPP server functionality.

This leads you to two changes that need to happen to the file makefile.mk. First you need to add the spp_lib.a so that you can link the SPP functions. Second, when you add new files to your project (like spp.c) you need to add them.

```

5 APP_SRC = TestClassicSpp.c
6 APP_SRC += TestClassicSpp_sdp_db.c
7 APP_SRC += wiced_bt_cfg.c
8 APP_SRC += spp.c
9
10 C_FLAGS += -DWICED_BT_TRACE_ENABLE
11
12 # If defined, HCI traces are sent over transport/WICED HCI interface
13 C_FLAGS += -DHCI_TRACE_OVER_TRANSPORT
14
15 #####
16 # Component(s) needed
17 #####
18 $(NAME)_COMPONENTS := spp_lib.a

```

In spp.h, I provide a function prototype for the public interface to the server.

```

1 #pragma ONCE
2
3 void spp_start();
4

```

In spp.c you first need add #defines to get all the required functions.

```

1 #include "spp.h"
2 #include "wiced.h"
3 #include "stddef.h"
4 #include "bt_types.h"
5 #include "wiced_bt_spp.h"
6 #include "wiced_bt_trace.h"

```

Then you declare a variable called `spp_handle` to hold the current handle of the SPP connection. You also need to include forward declarations for the SPP handler functions.

The structure `wiced_bt_spp_reg_t` holds all the configuration information for the SPP Server.

```
static uint16_t          spp_handle;

#define SPP_RFCOMM_SCN    1
#define MAX_TX_BUFFER 1017

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);

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 */
};
```

To startup the SPP server call the startup function with the configuration you defined above.

```
32
33 void spp_start()
34 {
35     // Initialize SPP library
36     wiced_bt_spp_startup(&spp_reg);
37 }
```

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
 */
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
 */
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.

```

56 |
57 | /*
58 |  * Process data received over EA session. Return TRUE if we were able to allocate buffer to
59 |  * deliver to the host.
60 |  */
61 | static wiced_bool_t spp_rx_data_callback(uint16_t handle, uint8_t* p_data, uint32_t data_len)
62 | {
63 |     int i;
64 |     // wiced_bt_buffer_statistics_t buffer_stats[4];
65 |
66 |     // wiced_bt_get_buffer_usage (buffer_stats, sizeof(buffer_stats));
67 |
68 |     // 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,
69 |     // buffer_stats[1].current_allocated_count, buffer_stats[1].max_allocated_count,
70 |     // buffer_stats[2].current_allocated_count, buffer_stats[2].max_allocated_count,
71 |     // buffer_stats[3].current_allocated_count, buffer_stats[3].max_allocated_count);
72 |
73 |     // wiced_result_t wiced_bt_get_buffer_usage (&buffer_stats, sizeof(buffer_stats));
74 |
75 |     WICED_BT_TRACE("%s handle:%d len:%d %02x-%02x\n", __FUNCTION__, handle, data_len, p_data[0], p_data[data_len - 1]);
76 |
77 |     for(i=0;i<data_len;i++)
78 |         WICED_BT_TRACE("%c", p_data[i]);
79 |     WICED_BT_TRACE("\n");
80 |
81 |
82 | #if LOOPBACK_DATA
83 |     wiced_bt_spp_send_session_data(handle, p_data, data_len);
84 | #endif
85 |     return WICED_TRUE;
86 | }
87 |

```

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 is sent using the wiced_bt_spp_send_session_data function. That 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.

The last thing that you need to do is initialize and start the SPP server in your <appname>_app_init function:

```

146 |
147 |     spp_start();
148 |

```

5A.8 Exercises

Exercise - 5A.1 Create a Serial Port Profile Project

Project Creation

For this example, you will need to:

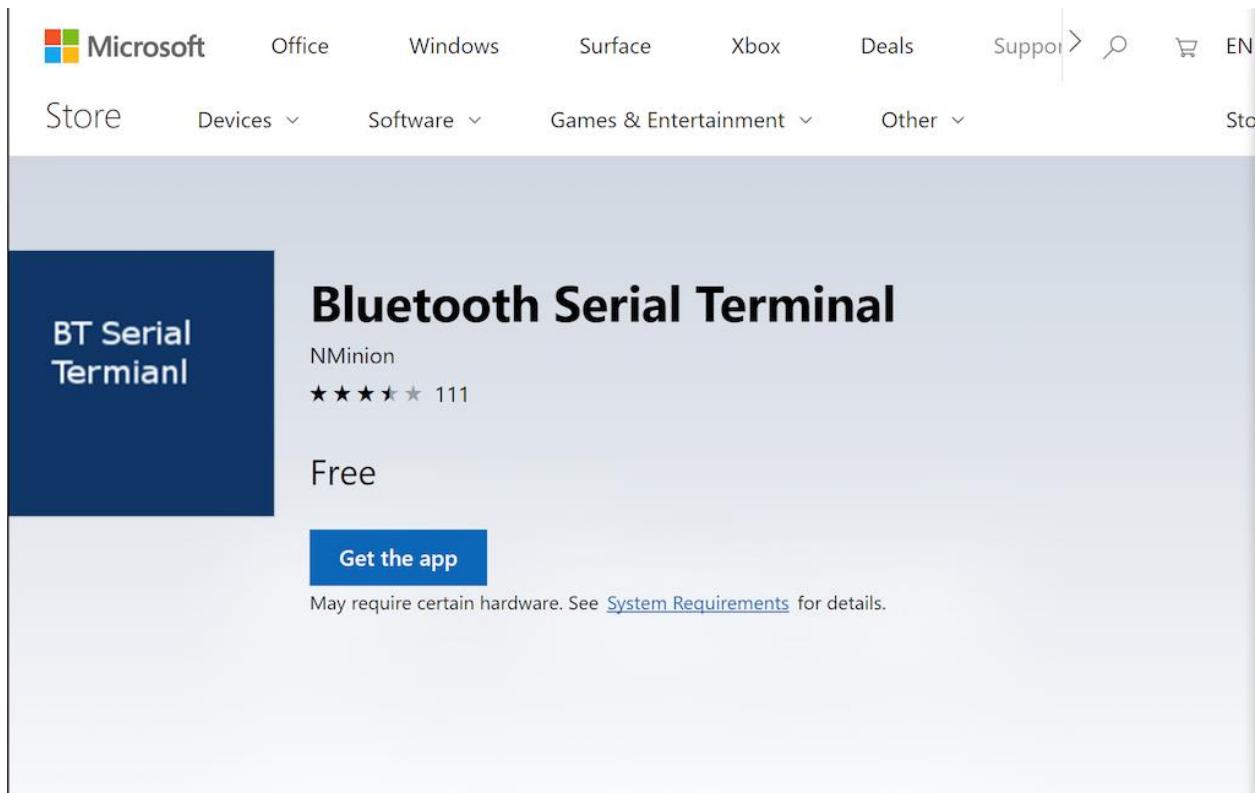
1. Create a new project using the BT Designer as documented in section 5A.5
 - a. Hint: You can move the project to ch05a and rename it to ex01_spp.
2. Update the project to handle pairing as documented in section 5A.7.3
3. Update the project to handle bonding as documented in section 5A.7.4
4. Update the project with the SPP functionality as documented in section 5A.7.5

Testing

Once your project seems to be working, you can attach to it using Windows, Mac or Android.

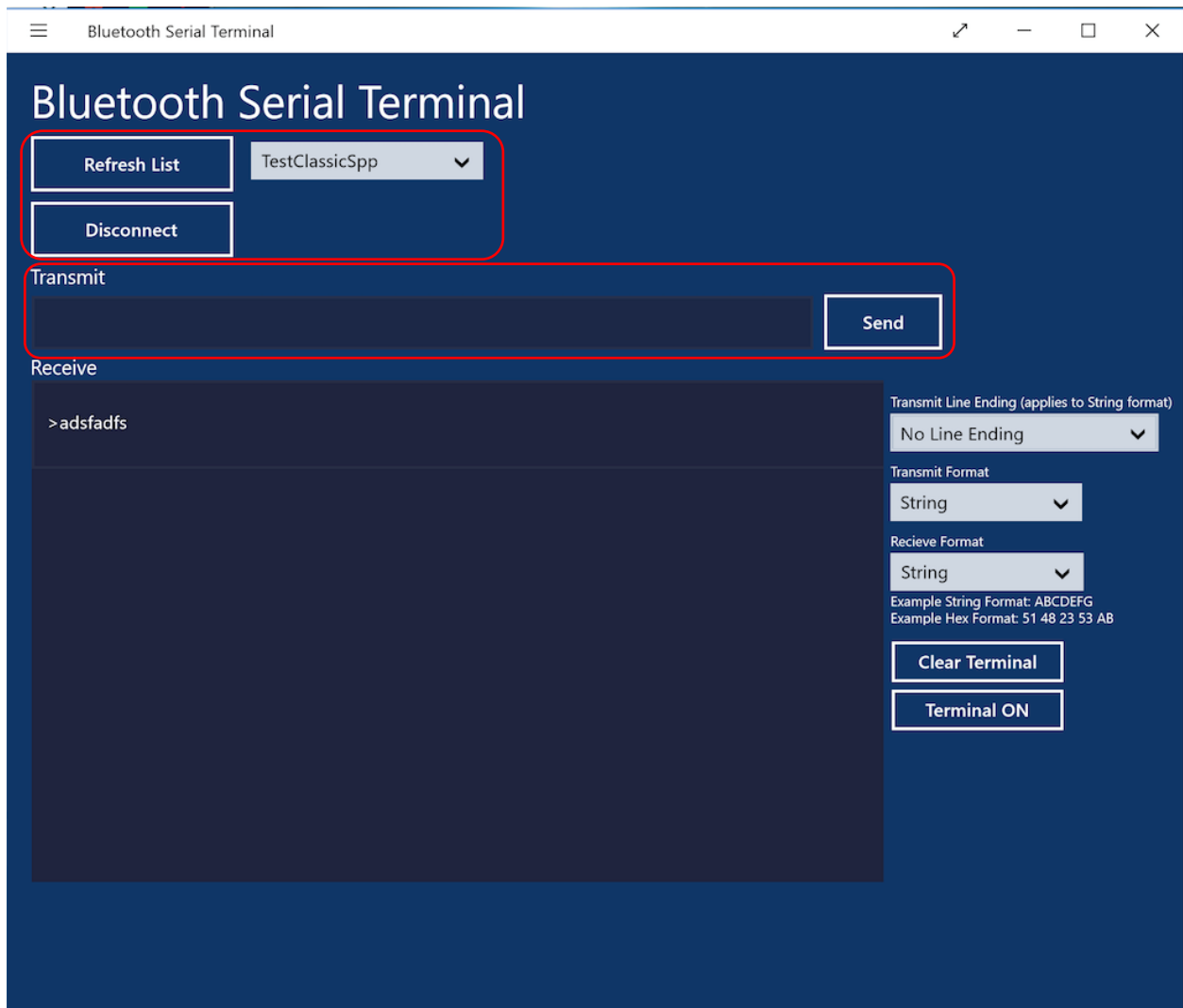
PC Instructions

For Windows 10 the easiest thing to do is install the “Bluetooth Serial Terminal” from the Microsoft App Store.



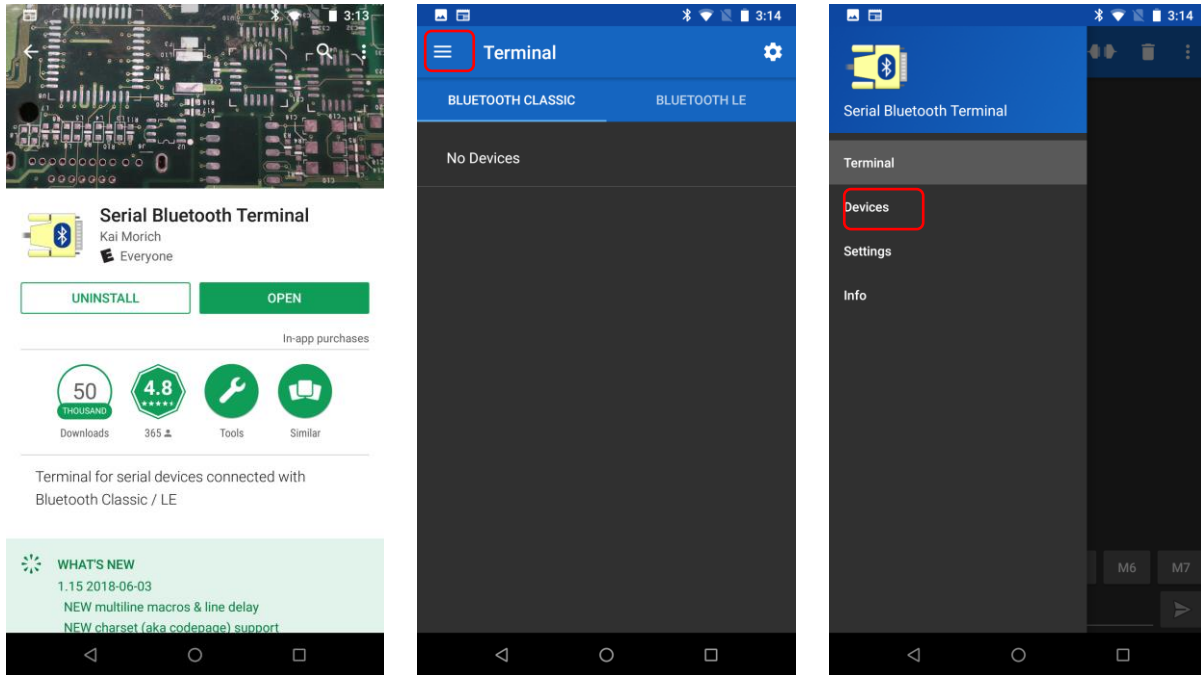
Once that is done, click “Refresh List” until you can see your project - in this case “TestClassicSpp”. Then Press “Connect”. Now you can type strings in the Transmit window and click on Send to send them to

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

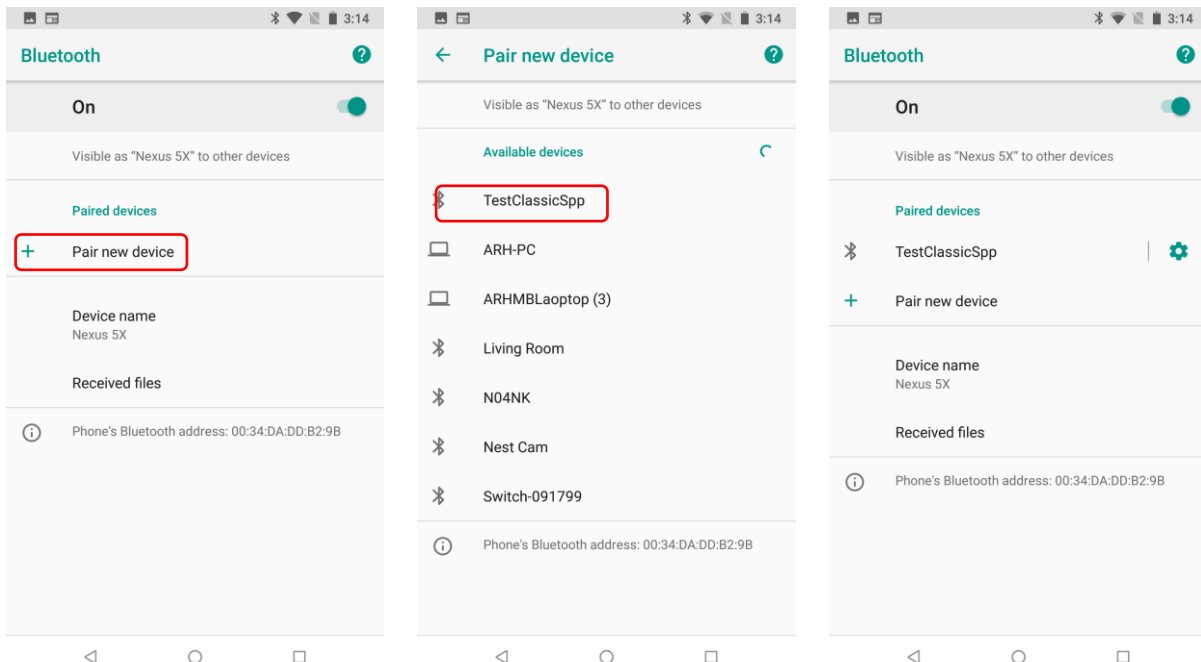


Android Instructions

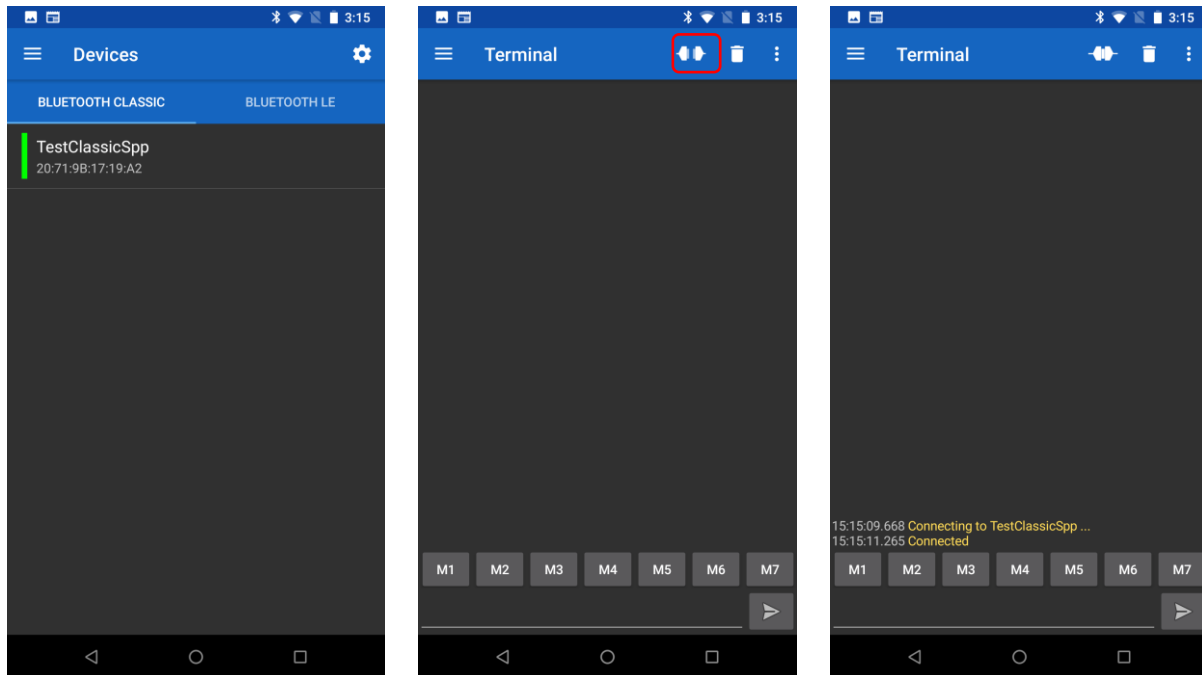
On an Android phone, you can install “Serial Bluetooth Terminal” from the Google Play Store. When you run the App, you will need to pair with your development kit. To do that open the menu in the upper left, then tap on devices.



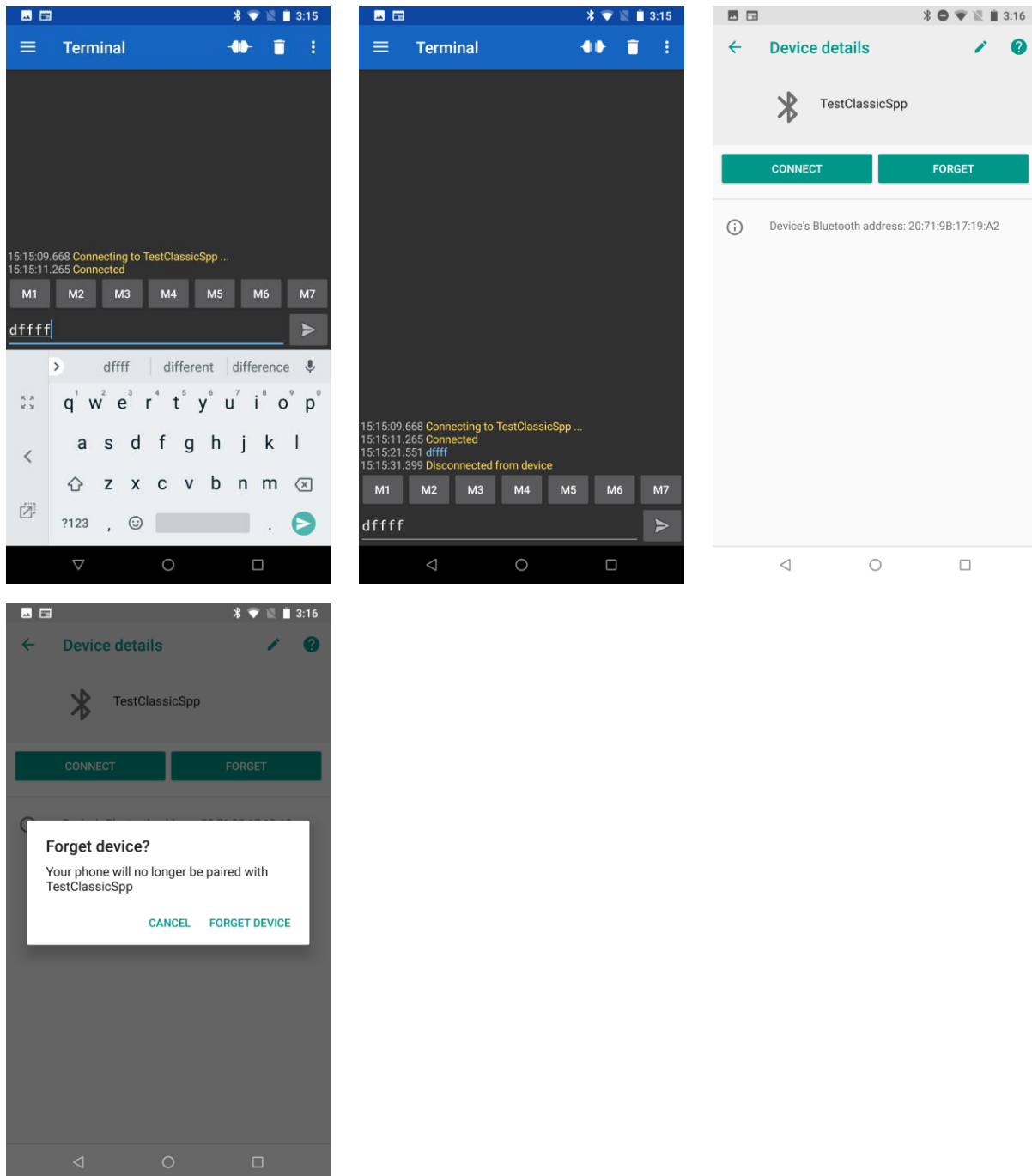
Click on “+ Pair new device”, then find your development kit in the list and tap on it to add it.



Once the device has been added, press the back arrow and you will see that your device is paired. Then press the back arrow again to see the blank terminal window. Next, tap the plug icon near the upper right to open the SPP server connection to your development kit.

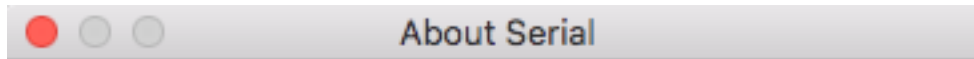


Now you can send data to the SPP server. When you press the plug again, it will disconnect. You can then go back to the Device screen and forget (aka delete the Bonding information) for your project.



Mac Instructions

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



Serial

Version 1.3.7 (MAS)

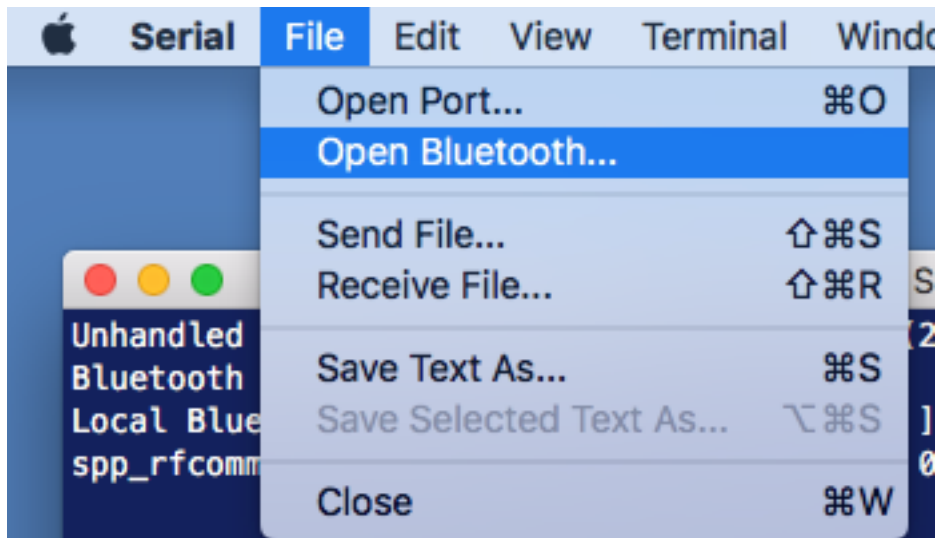


www.decisivetactics.com

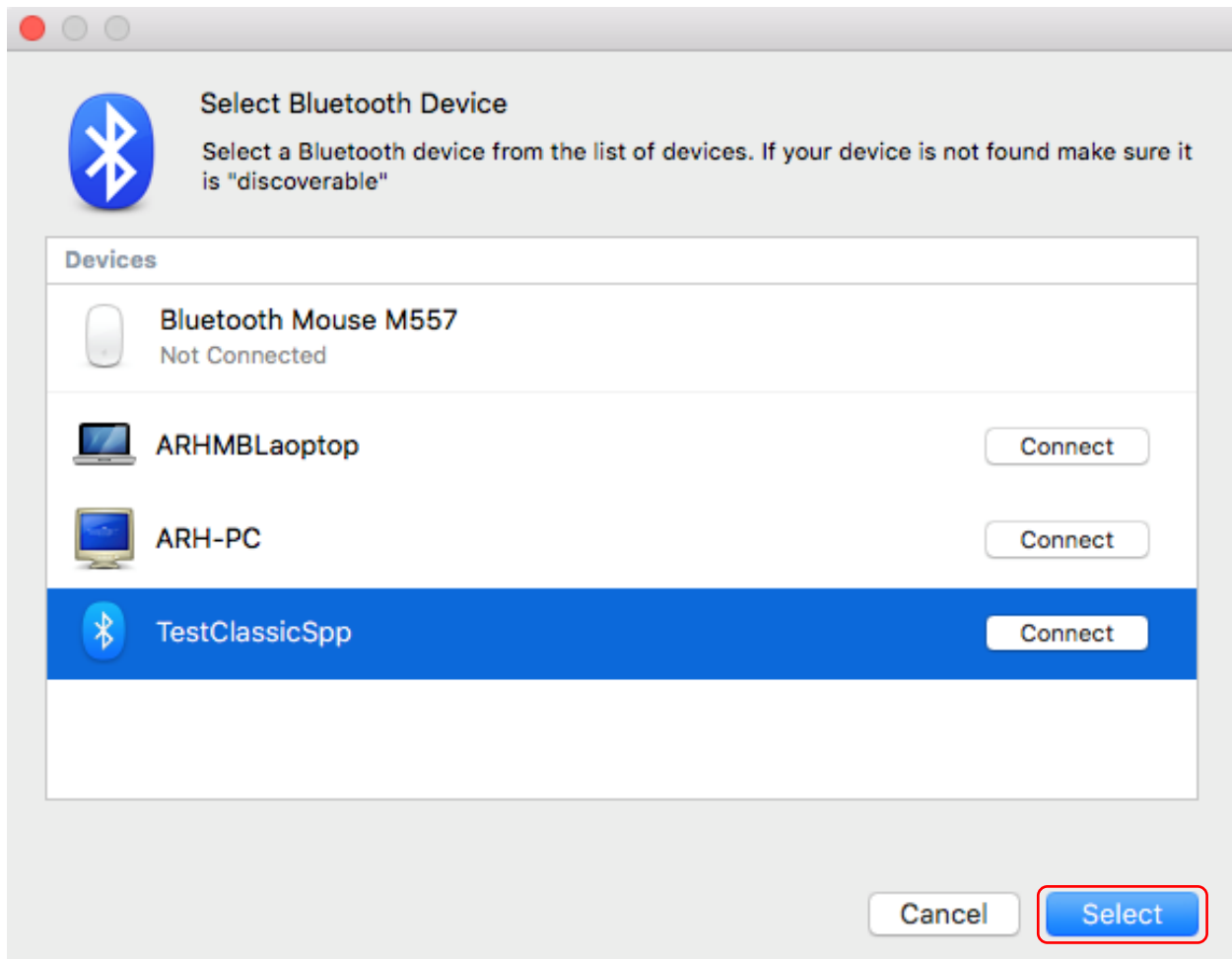
Copyright © 2014-2018 Decisive Tactics, Inc. All Rights Reserved. This product is protected by U.S. and international copyright and intellectual property laws.

[Acknowledgements](#)

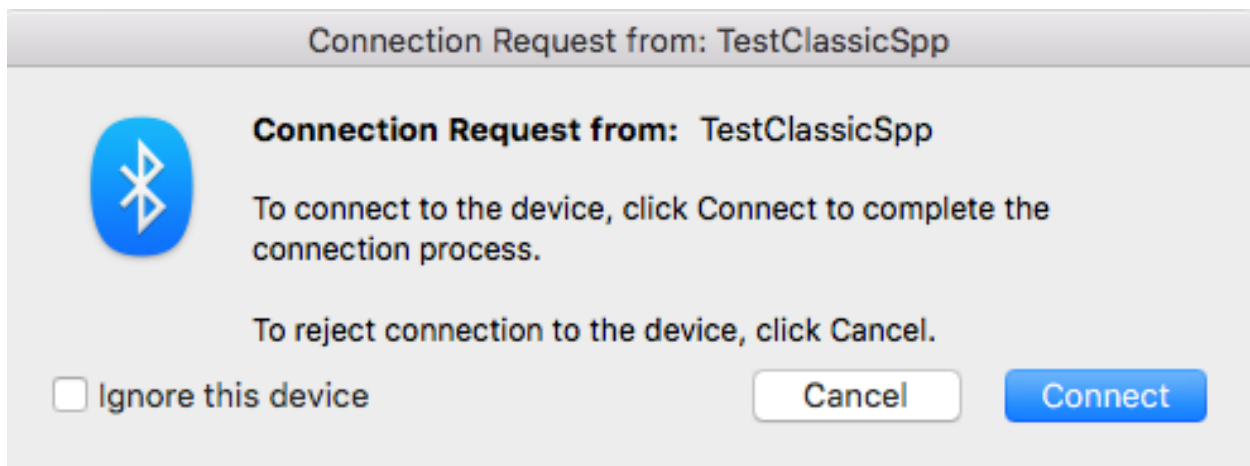
Once you have programmed the development kit you need to connect to the board. In the Serial program choose File → Open Bluetooth.



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



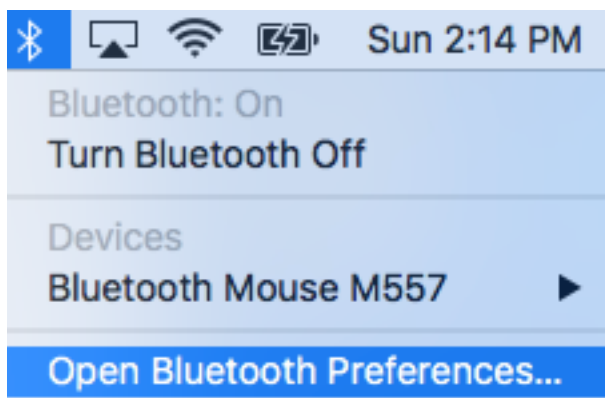
You will be asked to confirm the connection.



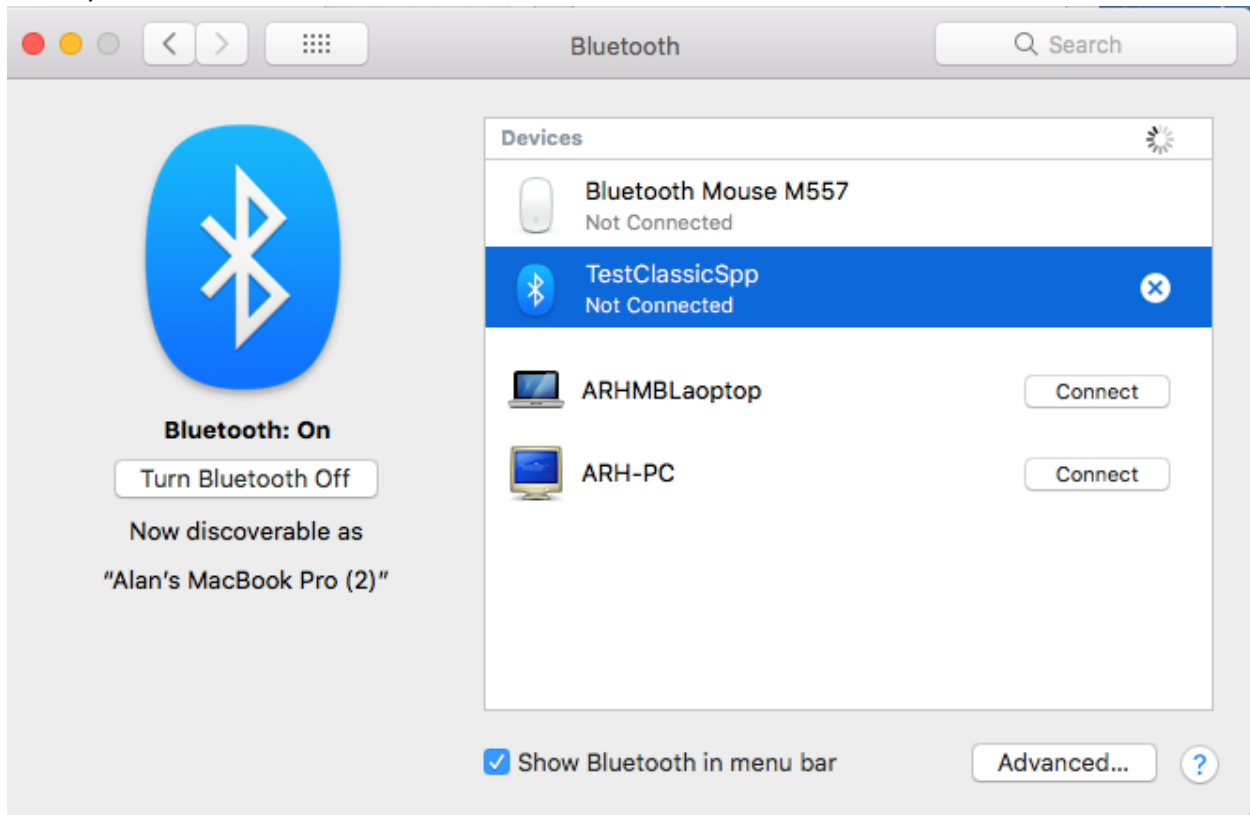
Once it is connected, everything you type will appear in the console window of the WICED Development kit. Below you can see that I typed “asdf”.

```
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
```

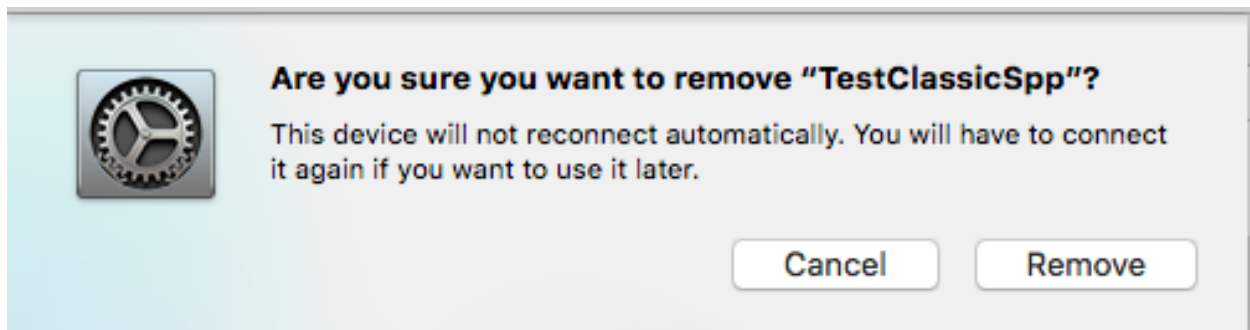
To unpair your development kit, select the Bluetooth symbol and pick “Open Bluetooth Preferences”



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 - 5A.2 Add UART Transmit

Update Exercise - 5A.1 to include a UART where you can type on the “keyboard” of the WICED development kit and it will send that output to the Master SPP.

Exercise - 5A.3 Improve Security by Adding IO Capabilities (Display)

In this exercise, we are going to take the previous exercise and add passkey validation. The WICED device will display the confirmation value at the appropriate time. You will need to confirm value on the phone for pairing to occur.

To make this work you need to make two changes:

1. When you get the event `BTM_PAIRING_IO_CAPABILITIES_BR_EDR_REQUEST_EVT` you will need to change the structure to indicate display functionality (Hint: open declaration).
2. When you get the event `BTM_USER_CONFIRMATION_REQUEST_EVT` you will need to display the value either on the serial port or on the U8G display.

Exercise - 5A.4 Improve Security by Adding IO Capabilities (Yes/No)

In this exercise, we are going change the previous exercise to add confirmation on the WICED device as well. In this case, a value will be displayed on both ends of the connection (WICED device and the phone). The user needs to compare the two values and then confirm that the values are the same on each device. Note: the phone may require user input or it may automatically confirm – this is up to the phone application.

You will add the capability for the user to confirm that the numeric comparison value is correct on the WICED device using the two mechanical buttons on the shield to indicate a “Yes” and a “No”.

To make this work you need to:

1. Copy and Modify Exercise - 5A.2.
2. Update `BTM_PAIRING_IO_CAPABILITIES_BR_EDR_REQUEST_EVT` to notify the Master that you have Yes/No and Display capability.
3. 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.
 - a. 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.

When the button interrupt happens send `wiced_bt_dev_confirm_req_reply()` and then disable the two mechanical button interrupts. Look at the function declaration to determine what information you need to send with the reply.

Exercise - 5A.5 Add Multiple Device Bonding Capability

You will need to make two changes to your current SPP implementation.

1. Handle saving multiple link keys into the NVRAM. Decide on the maximum number of saved link keys (8 is the magic number). 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 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`

