Chapter 5: Classic Bluetooth

Time: 3 Hours

At the end of this chapter you will understand the basics of Classic Bluetooth and how to create Classic Bluetooth projects on WICED devices.

5.1 Classic Bluetooth Introduction 3

5.2 Profiles 3

5.2.1 Advanced Audio Distribution Profile (A2DP) 3

5.2.2 Audio/Video Remote Control Profile (AVRCP) 4

5.2.3 Headset Profile (HSP) 4

5.2.4 Hands-Free Profile (HFP) 4

5.2.5 Human Interface Device Profile (HID) 4

5.2.6 Serial Port Profile (SPP) 4

5.3 Stack 5

5.4 Network 6

5.4.1 Piconets and Scatternets 6

5.4.2 Device Address (BD\_ADDR) 7

5.4.3 Piconet Clocks 7

5.4.4 Channel (Frequency) Hopping Sequence 7

5.4.5 Time slots 7

5.5 Logical Transports (Links) 9

5.5.1 Synchronous Connection-Oriented (SCO) 9

5.5.2 Extended Synchronous Connection-Oriented (eSCO) 9

5.5.3 Asynchronous Connection-Less (ACL) 10

5.5.4 Active Slave Broadcast (ASB) 10

5.5.5 Parked Slave Broadcast (PSB) 10

5.6 States and State Transitions 10

5.6.1 Inquiry 11

5.6.2 Paging 12

5.6.3 Sniff 12

5.6.4 Hold 12

5.6.5 Park 13

5.7 Packets 14

5.7.1 Access Code 14

5.7.2 Header 15

5.8 Security 15

5.8.1 Authentication (Link) Key 16

5.8.2 Encryption Key 17

5.8.3 Security Issues 17

5.9 Bonding and Paring 17

5.9.1 Legacy Pairing 18

5.9.2 Secure Simple Pairing (SSP) 18

5.10 Using Bluetooth in WICED Studio 19

5.10.1 Bluetooth Stack Initialization and Callback 19

5.10.2 Example Applications 22

5.10.3 WICED Bluetooth Designer 22

5.11 Advanced Topics 23

5.11.1 Combo BT/BLE Projects 23

5.12 Exercise(s) 24

Exercise - 5.1 Serial Port Profile (SPP) ??? 24

Exercise - 5.2 Human Interface Device Profile (HID) ??? 24

Exercise - 5.3 Headset Profile (HSP) ??? 24

Exercise - 5.4 Hands-Free Profile (HFP) ??? 24

Exercise - 5.5 Combo Bluetooth/BLE: SPP + BLE Custom CapSense ??? 24

5.13 Related Example "Apps" 25

5.14 Recommended Reading 25

# Classic Bluetooth Introduction

As you learned in the previous chapter, BLE is widely used for low power devices that can afford to send data a lower rate in regular bursts. For devices that either require a constant connection (i.e. streaming of data) or higher throughput, Classic Bluetooth is still widely used. Common examples include wireless headsets (streaming audio), hands-free phone headsets, virtual serial port connections for data transfer between devices and human interface devices (mice, keyboards, etc.).

Bluetooth operates in the 2.4 GHz ISM band (2.400 – 2.4835 GHz). In the US, Bluetooth uses 79 channels with 1 MHz spacing between channels. There is a lower guard band of 2 MHz and an upper guard band of 3.5 MHz. Therefore, the channel frequencies are from 2.402 GHz to 2.480 GHz.

Bluetooth devices employ continuous frequency hopping between channels to avoid interference from other devices that operate in the 2.4 GHz ISM band such as microwaves and cordless phones. Depending on the mode, channels are changed either 1600 times per second (normal operation) or 3200 times per second (inquiry and paging).

There are 3 data transfer rates available in Bluetooth. 1 Mbps, 2 Mbps, and 3 Mbps. Each uses a different encoding scheme as shown here:

|  |  |  |
| --- | --- | --- |
| **Mode** | **Speed** | **Modulation** |
| Basic Rate | 1 Mbps | GFSK (Gaussian Frequency Shift Keying) |
| Extended Data Rate | 2 Mbps | π/4 DQPSK (Differential Quadrature Phase Shift Keying) |
| Extended Data Rate | 3 Mbps | 8DPSK (Octal Differential Phase Shift Keying) |

# Profiles

Classic Bluetooth devices communicate with one another by using one or more of a standard set of profiles (often called services) which is maintained by the Bluetooth Special Interest Group (SIG). By using standard profiles, devices only need to determine the profile to use to start communicating rather than having to transmit the communication parameters themselves. A list of Bluetooth Profiles can be found at:

<https://en.wikipedia.org/wiki/List_of_Bluetooth_profiles>

Some of the more commonly used profiles are:

## Advanced Audio Distribution Profile (A2DP)

The A2DP profile is used for streaming multi-media audio. It is used, for example, when streaming audio from a mobile phone to a wireless headset or a car sound system. This profile is often used in conjunction with AVRCP, HSP, or HFP as described below.

The A2DP profile is designed for a unidirectional audio stream of up to 2 channel stereo. There may be more than one A2DP profile on a single device.

## Audio/Video Remote Control Profile (AVRCP)

The AVRCP profile is designed to provide a standard remote-control interface for devices such as televisions, stereo equipment, in-car navigation systems, etc.

There are several versions available depending on the functionality required, each of which is a superset of the previous version.

|  |  |
| --- | --- |
| Version | Functionality |
| 1.0 | Basic remote (play, pause, stop, etc). |
| 1.3 | 1.0 plus metadata (such as artist, track name, etc.) and player state (such as playing, stopped, etc.) |
| 1.4 | 1.3 plus multiple media player browsing including a "Now Playing" list and search capabilities. Also has support for absolute volume. |
| 1.5 | 1.4 plus corrections/clarifications to absolute volume control |
| 1.6 | 1.5 plus browsing and track information. Support for sending cover art through BIP/OBEX (Basic Imaging Profile and Object Exchange Profile) |

## Headset Profile (HSP)

The HSP provides support for headsets including two-way 64 kbit/sec audio and minimal controls for ringing, answer a call, hang up and adjust the volume.

In a typical headset, A2DP will be used when listening to music since it provides the best quality stereo connection, but HSP will be used when making a phone call since it allows two-way communication.

## Hands-Free Profile (HFP)

The HFP is commonly used to allow car hands-free kits to communicate with mobile phones. It provides relatively low-quality monaural audio to allow the user to control some features of their phone such as making calls, playing music, etc. It is often used with other profiles such as A2DP to provide high quality audio streaming.

## Human Interface Device Profile (HID)

The HID is used for devices such as mice, keyboards, and joysticks.

## Serial Port Profile (SPP)

The SPP emulates a serial cable to provide a means for setting up virtual seral ports to connect two Bluetooth devices.

Need more info here…some example real applications for SPP would be good.

# Stack

As with most complex systems, the Bluetooth stack is broken into layers as shown below.



# Network

## Piconets and Scatternets

Bluetooth devices communicate using a Master-Slave protocol. A single master can communicate with up to 7 slaves in a "piconet" in the older spec and up to 14 slaves on two different logical links (more on that later) in the newer spec. At any given time, data can be transferred between the master and one slave device (except when using broadcast mode). The master decides which slave to address. Usually the master will switch between slaves in a round-robin fashion.



Two or more piconets can connect to form a "scatternet". In a scatternet, devices can have different roles in different piconets. For example, a device can be a master in one piconet and a slave in another.



## Device Address (BD\_ADDR)

Each Bluetooth device has a unique 48-bit address. The 24 least significant bits are the lower address part (LAP), the next 8 bits are the upper address part (UAP), and the final 16 bits are the non-significant address part (NAP).

Need explanation of the 3 parts – is the LAP assigned by the company and the UAP/NAP assigned by SIG for a given company?

|  |  |  |
| --- | --- | --- |
| LSB MSB | | |
| **LAP** | **UAP** | **NAP** |
| 24 bits | 8 bits | 16 bits |
| Company assigned? | Company ID from SIG? | |

There are 64 LAP values reserved for the inquiry process (discussed later). These values must not be used for any BD\_ADDR. The reserved values are:

|  |  |
| --- | --- |
| **LAP Range** | **Purpose** |
| 0x9E8B00 | Dedicated Inquiry |
| 0x9E8B01 – 0x9E8B32 | Reserved for future use |
| 0x9E8B33 | General Inquiry |
| 0x9E8B34 - 0x9E8B3F | Reserved for future use |

## Piconet Clocks

Each device has a 28-bit timer which counts at 3.2 kHz (period = 312.5µs) to be used as a clock. Therefore, the timer wraps around every 23 hours and 18 minutes.

In a piconet, the clock from the master is called the piconet clock. All timing signals are derived from the piconet clock. The clock for each slave is its own internal timer with an offset used to synchronize it to the piconet clock.

## Channel (Frequency) Hopping Sequence

The sequence used for channel hopping is determined by the 28 least significant bits of the BD\_ADDR of the master. The current channel for the hopping sequence (a.k.a. the phase) is determined by the 27 most significant bits of the piconet clock. That is why each slave's clock must be synchronized to the piconet clock. Otherwise, the slave and master would not hop to the same channels.

Since the piconet clock is 3.2 kHz and the phase uses the upper 27 of the 28 bits of the piconet clock, that means hopping happens at a rate of 1.6 kHz (i.e. 1600 times per second or once every 625 µs).

## Time slots

Communications between master/slave on a piconet are divided into 625 µs time slots. The slots are numbered using the most significant 27 bits of the piconet clock. The master uses even numbered time slots to send packets while the slave uses odd numbered time slots. Since hopping also happens every 625 µs, each time slot uses a different channel from the hop sequence.



In addition to standard single slot packets, there are multi-slot packets of 3 or 5 slots. In the case of a multi-slot packet, the channel stays the same for the entire packet but the channel will jump to the next frequency in the sequence once that packet is finished. For example, if a 3-slot packet uses the 4th channel in the hopping sequence, the packet after the 3-slot packet will use the 7th channel. The pictures below show multi-slot packets from master to slave, but the other direction is also possible.





An adapted frequency hopping sequence may use fewer than all 79 channels (but at least 20). In that case, the master and slave slot pairs use the same channel. As with the multi-slot packets, the next channel will jump to the appropriate frequency in the sequence. For example, if the master uses the 4th channel for a single slot packet using adapted frequency hopping, the slave will also use the 4th channel and the next master packet will use the 6th channel.



The master initiates any changes to or from adapted frequency hopping or to a different adapted frequency hopping scheme by sending an AFH command. The slave(s) must then send an acknowledge (ACK).

# Logical Transports (Links)

There are five types of link supported in Bluetooth. Each is discussed in detail in the following sections.

1. Synchronous Connection-Oriented (SCO)
2. Extended Synchronous Connection-Oriented (eSCO)
3. Asynchronous Connection-Oriented (ACL)
4. Active Slave Broadcast (ASB)
5. Parked Slave Broadcast (PSB)

## Synchronous Connection-Oriented (SCO)

An SCO link is a point-to-point link between the master and a specific slave. This type of link can be used for time critical data such as voice. Some time slots are reserved for both the master and the slave. SCO packets are never re-transmitted. A master can support up to three simultaneous SCO links (either to one slave or to different slaves).

## Extended Synchronous Connection-Oriented (eSCO)

The eSCO link is similar to SCO except that re-transmission is allowed. Since it allows time for re-transmission without losing time synchronization, this type of link is useful for higher-quality streaming than a standard SCO link.

It uses a 3-bit device address (LT\_ADDR) so that all 7 devices on a piconet can be addressed. The LT\_ADDR is used with retransmitted messages so that the slaves can identify them.

eSCO links support both BR and EDR.

## Asynchronous Connection-Less (ACL)

An ACL link is used for asynchronous communication. This type of link is useful for non-time critical data. Like eSCO, it uses a 3-bit address (LT\_ADDR) so it can address all 7 devices on a piconet. Note that the LT\_ADDR for eSCO and the LT\_ADDR for ACL are not the same so a master can have 7 eSCO links and 7 ACL links.

## Active Slave Broadcast (ASB)

An ASB link is used for a master to send broadcast packets to all active slaves connected to a piconet. An acknowledgement is not necessary for ASB packets.

## Parked Slave Broadcast (PSB)

A PSB link is used for a master to send broadcast packets to all parked slaves on a piconet (more on the parked state later). There can be up to 255 parked slaves on a piconet even though a maximum 14 can be active at a time.

PSB is the only link between a master and a parked slave.

# States and State Transitions

The Bluetooth device states are shown in the figure below.



A device starts in the Standby state. In order to become Connected, it must go through ether Inquiry and then Paging (if the address is unknown), or just Paging (if the address is known). Once it is Connected, a device can Transmit and Receive data. Three different low power states (Sniff, Hold, and Park) can be used for devices that do not need to stay Connected but which don't want to go all the way back to Standby.

## Inquiry

Inquiry is used for a master to discover nearby Bluetooth devices. The master enters Inquiry mode to discover nearby Bluetooth devices. Each Bluetooth device that wants to be discoverable will occasionally enter the Inquiry Scan state to allow it to be discovered by the master.

The master transmits messages either for general or dedicated inquiry. A general inquiry message is used to find any nearby discoverable Bluetooth devices. A dedicated inquiry message is used to discover a specific group of devices.

Inquiry messages consist of just a 68-bit access code. The General Inquiry Access Code (GIAC) is generated by running a specific algorithm on the 24-bit General Inquiry LAP (0x9E8B33). The Dedicated Inquiry Access Code (DIAC) is generated by running a specific algorithm on the 24-bit Dedicated Inquiry LAP (0x9EB00).

The master sends a series of inquiry messages over 32 hop frequencies with a hop rate of 3200 times per second. Since the hop rate is 2X the normal hop rate, each 625us slot has either 2 Tx messages from the master or potentially two Rx messages from the slave.



The 32 hopping channels and sequence are generated from the LAP for general inquiry with 4 leading 0's (0x09E8B33) and the 27 most significant bits of the piconet clock.

The recommended timing relationship between the master performing an inquiry and the slave entering inquiry scan mode is as follows:

|  |  |
| --- | --- |
| Value | Description |
| 10.24 sec | Time span that master performs an inquiry |
| 10.625 ms | Minimum time slave is in the Inquiry Scan mode |
| 2.56 sec | Maximum time for slave before entering Inquiry Scan mode again |

## Paging

Paging is used for a Bluetooth master to connect to a Bluetooth slave. The master is the "paging device" and will be in the Page mode while the slave is the "paged device" and will periodically enter the Page Scan mode to look for paging messages. Like inquiry, the master sends a series of paging messages over 32 hop frequencies with a hop rate of 3200 times per second.

The 32 hopping channels and sequence are generated from the 28 least significant bits of the BD\_ADDR of the device being paged (i.e. the slave that the master wants to connect to).

The recommended timing for a slave to enter page scan mode is as follows:

|  |  |
| --- | --- |
| Value | Description |
| 10.625 ms | Minimum time slave is in the Page Scan mode |
| 2.56 sec | Maximum time for slave before entering Page Scan mode again |

## Sniff

In the Sniff state, a slave still listens but it does so at a reduced rate. This is applicable to ACL links but not to SCO or eSCO links due to the time-sensitive nature of data on those links. While not listening, a slave in Sniff may engage in activity on another piconet or it may enter a reduced power mode.

When a slave is in the Sniff state, the master can only transmit to it in specified time slots that start at sniff anchor points. These anchor points are spaced with an interval of Tsniff.



## Hold

In the Hold state, capacity is made available for other tasks such as scanning, paging, inquiry, or engaging with another piconet. The slave device can also enter a low power mode during Hold. ACL links do not support Hold mode but already established SCO or eSCO links do.

Prior to entering Hold, the master and slave agree on the length of time that the slave will remain in Hold mode.

## Park

In the Park state, the slave does not participate on the piconet channel but remains synchronized to the channel. The slave gives up its LT\_ADDR before entering Park and receives two new addresses:

PM\_ADDR: 8-bit Parked Member Address

AR\_ADDR: 8-bit Access Request Address

The PM\_ADDR is used when the master initiates an un-park procedure while the AR\_ADDR is used when the slave initiates an un-park procedure. The master can also un-park a slave by using its 48-bit BD\_ADDR.

All messages to parked slaves must be sent using broadcast packets (i.e. LT\_ADDR = 0).

To keep parked slaves synchronized, the master sends a beacon train consisting of one or more equidistant beacon slots sent periodically. The number of slots in a beacon train, interval between slots and interval between trains is sent from the master to the slave when the slave is parked. A slave is not required to wake up for every beacon train – instead it may wakeup only after several intervals.



Access windows are provided between the periodic beacon trains for a slave to request to be un-parked. Multiple windows can be provided to increase reliability.



# Packets

Bluetooth data is sent in Packets which consist of an Access Code, Header, and Payload. Some packets have just the access code and header and still other packets have just a header.

Bits are ordered in Little Endian format. That is, the LSB is the first bit sent.

Basic Rate packets look like this:

|  |  |  |
| --- | --- | --- |
| LSB MSB | | |
| **Access Code** | **Header** | **Payload** |
| 68 or 72 bits | 54 bits | 0 – 2745 bits |

Enhanced Data Rate packets look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| LSB MSB | | | | | |
| **Access Code** | **Header** | **Guard** | **Sync** | **EDR Payload** | **Trailer** |
| 68 or 72 bits | 54 bits | Time? | X bits? | 0 - 2745 bits ??? | 4 bits? |
| Basic Rate (GFSK) | |  | Enhanced Data Rate (DPSK) | | |

Note that the Access Code and Header are sent using basic rage (i.e. GFSK) and the payload is sent using enhanced data rate (i.e. DPSK). The guard time is used to allow the physical channel change modulation schemes.

## Access Code

The access code is used for synchronization, DC offset compensation, and identification. All packets on the same piconet will have the same access code.

The first 4 bits of the access code are the preamble, the next 64 bits are the sync word, and the final 4 bits are the trailer. If the packet does not contain a header then the trailer is left out. That is, packets containing only the access code will be 68 bits long while all other packets will have a 72-bit access code.

|  |  |  |
| --- | --- | --- |
| LSB MSB | | |
| **Preamble** | **Sync Word** | **Trailer** |
| 4 bits | 64 bits | 4 bits |

The preamble is either 0101 or 1010 depending on the LSB of the sync word. That is, if the LSB of the sync word is 0 then the preamble will be 0101 and vice versa. This guarantees that the first 5 bits transmitted will alternate between 0 and 1 on each bit.

Likewise, the trailer is either 1010 or 0101 depending on the MSB of the sync word. Again, this guarantees that the last 5 bits of the access code will alternate values on each bit.

The sync word is a 64-bit code derived from the 24-bit LAP.

## Header

The header is 18 bits of data but it is encoded using a 1/3 FEC (Forward Error Correction) which means each bit is sent 3 times in a row. Therefore, the header is 54 bits long.

The first 3 bits of data (9 bits transmitted) are the LT\_ADDR which is slave address. Since there are up to 7 slaves on a piconet, 3 bits is enough to specify the slave. For a packet in a master-to-slave time slot this is the destination while in a slave-to-master time slot this is the source.

The next 4 bits of data (12 bits transmitted) is the packet type. There are 16 different packet types.

The next 3 bits of data (9 bits transmitted) are FLOW, ARQN, and SEQN which are used for various flow control, acknowledgement, and sequencing requirements. These values depend on the packet type.

The final 8 bits of data (24 bits transmitted) are for Header Error Checking (HEC).

Putting it all together, we have:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| LSB MSB | | | | | |
| **LT\_ADDR** | **TYPE** | **FLOW** | **ARQN** | **SEQN** | **HEC** |
| 3 bits data | 4 bits data | 1-bit data | 1-bit data | 1-bit data | 8 bits data |
| 9 bits sent | 12 bits sent | 3 bits sent | 3 bits sent | 3 bits sent | 24 bits sent |

Details on different packet types (segment-1, -2, -3, -4, ID, NULL, POLL, FHS, SCO (HV1, 2, 3, DV), eSCO, ACL) are not discussed. Is that beyond the scope of this?

# Security

We don't talk about encryption methods or MIM. Should we copy some of that material from WW101? If so, this section will get quite a bit longer. This might have to go in a prior chapter (either a separate chapter or in the BLE chapter) since that is likely needed for BLE too.

In Bluetooth v2.0 and earlier, encryption is not required and can be turned off at any time. Turning of encryption is required for several normal operations so it is difficult to detect if encryption is disabled for a valid reason or because of an attack.

In Bluetooth v2.1, encryption is required for all non-Service Discovery Protocol connections. An Encryption Pause and Resume feature is used for all normal operations that require encryption to be disabled which enables easy identification of normal operation vs. security attacks. (???: how do you know that a security attack didn't use encryption disable to turn it off?)

Secure connections in Bluetooth use two different keys:

|  |  |  |
| --- | --- | --- |
| **Element** | **Size** | **Generation** |
| *Authentication Key* or *Link Key* | 128 bits | Generated during pairing (more on this later) |
| *Encryption Key* | 8-128 bits (8-bit granularity) | Generated from authentication key, unique for each encrypted link |

## Authentication (Link) Key

There are four types of authentication keys defined for different types of applications. These are:

### Initialization Key

The initialization key is used during the initialization process when the other keys have not been generated or when a link key has been lost. It is generated from the BD\_ADDR of the device being verified, a PIN code, the length of the PIN, and a 128-bit random number created by the verifier. The random number is sent unencrypted to the device being verified since no key has yet been generated. Therefore, the only "secret" portion to this key is the PIN code.

### Unit Key

A unit key is dependent on a single Bluetooth device. It is derived from the BD\_ADDR of the device and a 128-bit random number (this is different than the random number used for the initialization key). It is typically done once for a given device and is very rarely changed after that.

A device's unit key is sent to another device by encrypting it with the initialization key. After that, the initialization key is discarded by both devices.

For devices with limited memory or devices that must be accessible to a large group of users will typically use their own unit key. In that case, the device only needs to store a single key.

Combination Key

A combination key is dependent on information from both Bluetooth devices in a connection. Therefore, this key will be unique for each new combination of two devices. It is derived using the following process:

1. Each device generates a 128-bit random number (again, not the same random number that was used for the initialization or unit keys), XORs it with the existing link key (i.e. initialization key or unit key) and sends it to the other device.
2. Each device runs an encryption algorithm on the random number it generated and its BD\_ADDR.
3. Each device runs an encryption algorithm on the random number it got from the other device (by XORing what it received again with the existing key) and the other device's BD\_ADDR (which it got during discovery).
4. The combination key is created by XORing the results from the previous two steps. Since both devices ran the same algorithm on both sets of data, they both generate the same key.
5. Once the combination key is generated, the old link key is discarded by both devices.

### Master (Temporary) Key

A master or temporary key replaces the original link key during the current session. This is useful when a master wants to reach multiple devices simultaneously using the same key.

## Encryption Key

The encryption key can be any length from 8 – 128 bits in 8-bit increments (i.e. 1 – 16 bytes). The encryption key length is variable to: (1) accommodate requirements imposed by various countries with respect to export regulations and; (2) facilitate future increased security by increasing the key length without requiring a redesign of the encryption algorithms (which may be in hardware).

Encryption keys are valid for approximately 23.5 hours because a simple XOR attack may be able to crack the key in that amount of time. Encryption keys need to be refreshed before they expire.

## Security Issues

The PIN is by far the largest point of weakness in Bluetooth security. For most systems, the PIN is a 4-digit number and is often either 0000 or 1234. Since the PIN is the only "secret" information used to generate the initialization key, it is quite easy to eavesdrop on the initialization procedure by guessing the PIN.

However, this weakness only exists during initialization when randomly generated numbers are exchanged to create the other keys (unit key or combination key). So, if the initial bonding is done without someone eavesdropping on the connection, then the connection is secure from that point on since the other keys were created using random numbers that were sent using the initialization key only one time during initialization. If those random numbers were not stolen during initialization, everything is good.

# Bonding and Pairing

Bonding is a process by which two Bluetooth devices (a master and a slave) can recognize each other. This is critical for security reasons and is also convenient for the user – for example, if they have two devices that they frequently use together (e.g. a cellphone and a Bluetooth headset) they only need to manually connect them and possible enter a security PIN once. After that the devices can connect automatically whenever they are in range.

Bonding two Bluetooth devices is done using the pairing process. Pairing can be triggered in one of two ways:

1. Dedicated Bonding: The user requests bonding (i.e. when a new device is first connected).
2. General Bonding: The device connects to a service for the first time that requires the device identity for security purposes.

There are two sets of pairing mechanisms available: Legacy Pairing and Secure Simple Pairing (SSP) which are discussed below. The end result of bonding is the creation of the *authentication key* which is stored on both devices so that they can use it on future connections. If the key is removed from either device then the devices need to be bonded again before they can be connected.

## Legacy Pairing

In Legacy Pairing, each device must enter a PIN code which can be any UTF-8 string (ASCII) up to 16-bytes in length. There are three mechanisms for this pairing scheme:

1. *Limited Input Device*: If a device cannot enter a pin (e.g. a headset) it may use a fixed PIN such as 0000 or 1234. In this case, the fixed PIN must be entered on the device that is bonding with it (e.g. cellphone).
2. *Numeric Input Device*: If a device allows only numeric entry, any numeric PIN up to 16 digits in length can be used.
3. *Alpha-numeric Input Device*: If a device allows alpha-numeric entry, any UTF-8 string up to 16 characters can be used as the PIN.

## Secure Simple Pairing (SSP)

In Bluetooth v2.1, Secure Simple Pairing was introduced. It is required by Bluetooth v2.1 but a v2.1 device can still use legacy pairing to allow connections to v2.0 or earlier devices. SSP uses public key cryptography. Depending on the authentication mechanism used, SSP may help protect against Man In the Middle (MIM) attacks. There are four types of authentication for SSP:

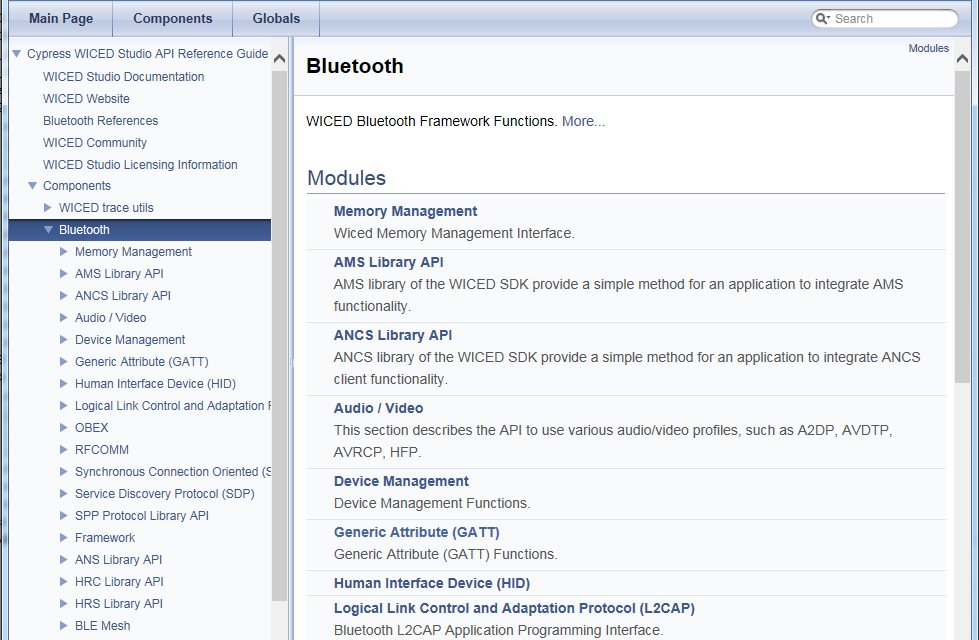
1. *Just Works*: This method works with no user interaction but a device may prompt the user to confirm pairing. Typically used with devices with limited IO capabilities (e.g. headsets). This is more secure than using a fixed PIN but does not provide MIM protection.
2. *Numeric Comparison*: In this method, a 6-digit number is displayed on both devices being paired. The user compares the numbers, and if they are identical, then the user confirms pairing on one of the devices. This method provides MIM protection as long as the user actually verifies that the numbers are identical before confirming pairing.
3. *Passkey Entry*: This method can be used when one device has a display and one device has numeric entry capability, or when both devices have numeric entry capability. In the first case, the device with the display shows a 6-digit number which is entered on the device with numeric entry. In the second case, the user enters the same 6-digit number on both devices. This method provides MIM protection.
4. *Out Of Band (OOB)*: This method uses an external means to exchange information used in the pairing process. For example, near-field communication (NFC) may be used to exchange a Passkey. This method provides MIM protection only if it is present in the OOB mechanism used.

# Using Bluetooth in WICED Studio

Now that you have learned the basics of Classic Bluetooth, let's look at how to implement the firmware in WICED Studio.

## Documentation

The relevant documentation for Bluetooth functions are in the WICED SDK documentation under Components🡪Bluetooth.

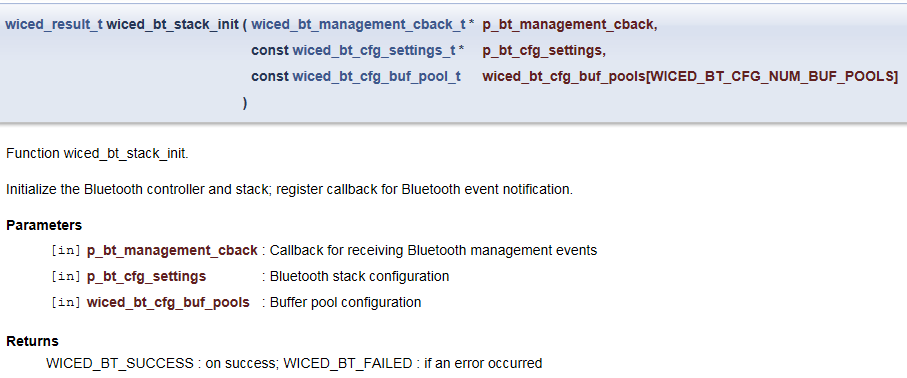


## Bluetooth Stack Initialization and Callback

This section is the same as the BLE chapter. Repeat, put a reference, or move to a common chapter?

As you have seen in prior exercises, Bluetooth applications start at *application\_start()* which does initialization that is required before starting the Bluetooth stack and then starts the stack using *wiced\_bt\_stack\_init()*.

Documentation for this function can be found in Components -> Bluetooth -> Framework

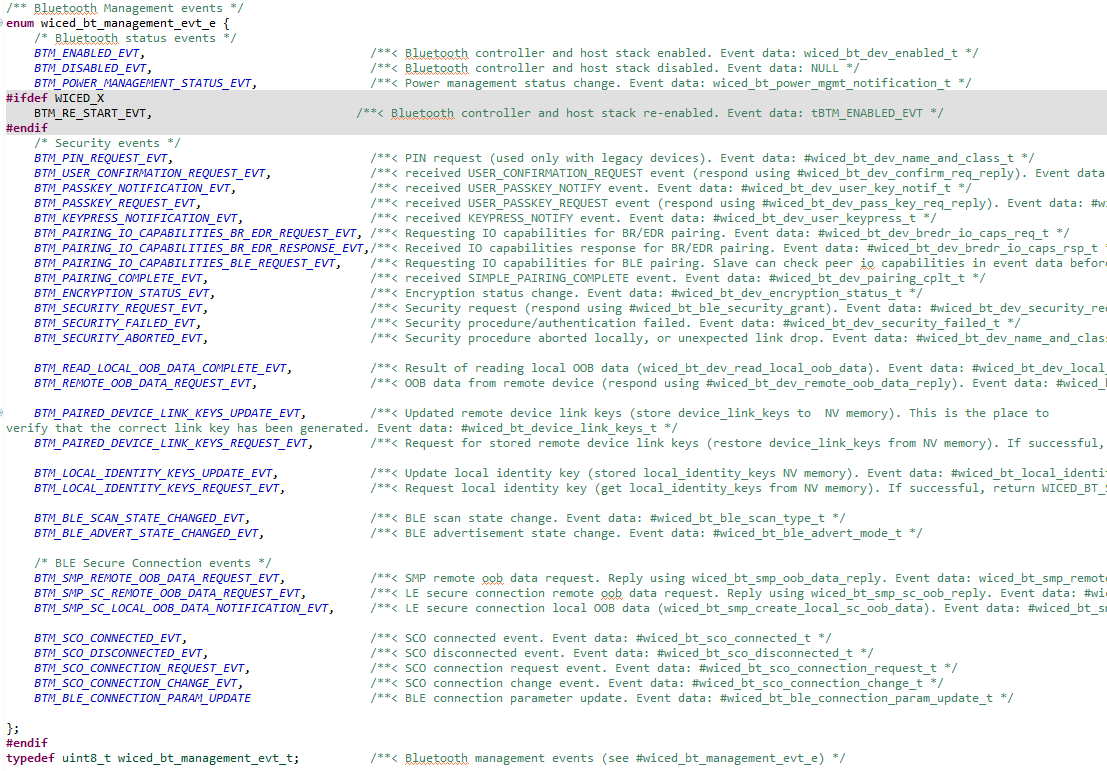


As you can see, there are three arguments. Each of these is discussed separately below.

### Callback Function

The first argument to *wiced\_bt\_stack\_init()* is a pointer to the callback function that is called by the Bluetooth stack for various events. This function typically handles the bulk of the application's functionality.

The list of callback events can be found in wiced\_bt\_dev.h and is shown here:

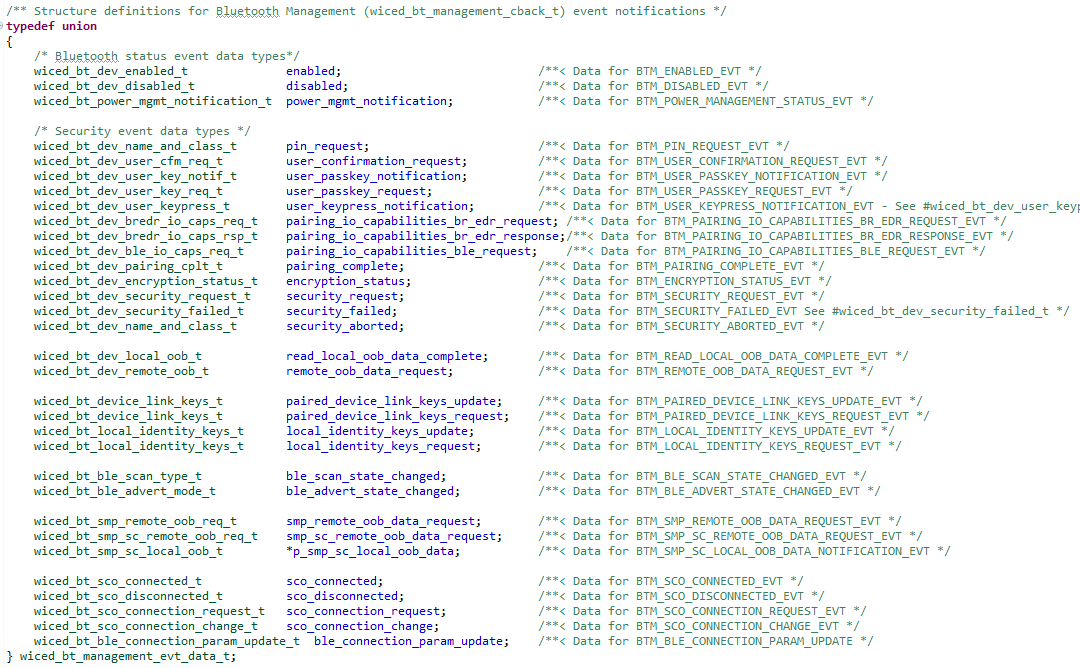


Applications will not typically implement all callback events. The events required will depend on the application.

All applications should have the BTM\_ENABLED\_EVT event implemented. This event occurs after the *wiced\_bt\_stack\_init()* function completes stack initialization and is the application's notification that it can continue with initialization that needs to be done after the stack has started and then start normal operation.

Add details of some states here? Which ones?

In addition to the event itself, the callback function also provides data that goes along with that event (if any). The type of the data provided depends on the event that caused the callback. The list of event data types can be found in wiced\_bt\_dev.h and is shown here:



### Stack Configuration Settings

The second argument to *wiced\_bt\_stack\_init()* is a pointer to configuration settings for the Bluetooth stack. The configuration is usually kept in a separate source file called *wiced\_bt\_cfg.c*. This file is specified in the application's makefile. This file can (and should) be created by copying an existing file from another project or by using the *WICED Bluetooth Designer* (more on that later).

Add details on the stack configuration here? How much detail?

### Buffer Pool Settings

The third and final argument to *wiced\_bt\_stack\_init()* is an array of configuration settings for the buffer pools. Like the stack configuration, this is also usually specified in the wiced\_bt\_cfg.c file and will be created the same way (i.e. copying from another project or using *WICED Bluetooth Designer*).

Should have more detailed explanation of the buffer pools?

## Example Applications

The WICED Studio SDK contains a wealth of example snip and demo applications to illustrate how to use various Classic Bluetooth profiles and connection strategies. It is always recommended to start a new application based on one of the provided examples or by using *WICED Bluetooth Designer* (discussed in the next section) to simplify and speed up development.

Some of the example applications provided in the WICED Studio SDK are described below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Application Name** | **Connection/Pairing**  **Method** | **Profiles Used** | **Description** |
| snip.bt.ad2p\_sink |  |  |  |
| snip.bt.spp |  |  |  |
| demo.hci\_pbap\_client |  |  |  |
| demo.bt\_keyboard |  |  |  |
| demo.hid\_device |  |  |  |
| demo.hid\_host |  |  |  |
| demo.watch |  |  |  |

Most of these applications have corresponding sample implementations of MCU functionality executing on a PC, in the Client Control application, demonstrating WICED HCI control of the device over UART. You will get to experiment with this in some of the exercises.

Many apps that are in the 20706 SDK are not in the 20719 SDK (like demo.audio.headset, demo.audio.hci\_audio\_remote\_control, demo.audio.hci\_handsfree). Are there supposed to be equivalent apps (i.e. with those profiles) for the 20719?

There are some apps listed in the demo README.txt that are not there such as headset, headset\_pro, and btspeaker\_pro. Will these be in the production release?

TBD: Add details on each application from the table above.

TBD: Do we want SPP details here like the setup of the callback functions? Have a separate section for this info or include it with the example?

## WICED Bluetooth Designer

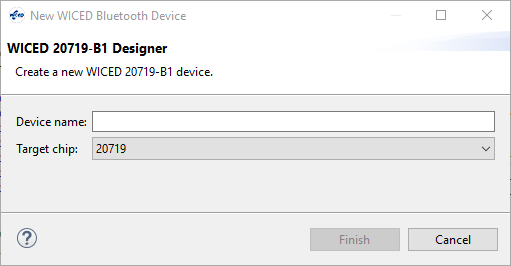
GJL: There is a lot of overlap with the BLE section, especially in the first 2 pages. Do we want to repeat that information?

Issue: once you mve the project,

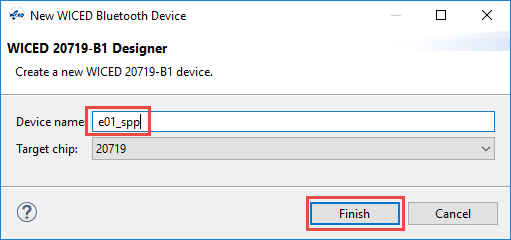
WICED Bluetooth Designer is a utility that helps set up a new Bluetooth application for Classic, BLE, or a combination of both. It creates C source files including a top-level file, a wiced\_bt\_cfg.c file, a makefile and a Make Target for the application. The top-level source file will contain initialization code, the Bluetooth callback function, and other necessary functionality based on the options chosen in the tool. Other C and header files may be created depending on the options. For example, it will create C and header files for the GATT database if the project has one.

A .wic file is created which contains the WICED Bluetooth Designer configuration. Open that file if you want to re-run the tool again but keep in mind that previous files will be over-written (after a backup copy is made) when you re-generate the code.

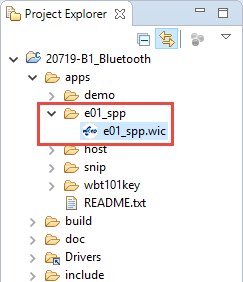
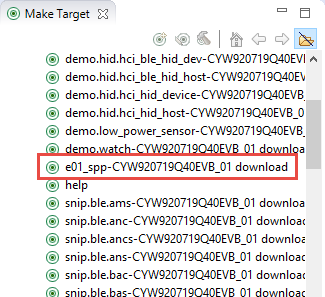
To run the tool, go to "*File -> New -> WICED Bluetooth Designer*". The first window to appear will look like this:



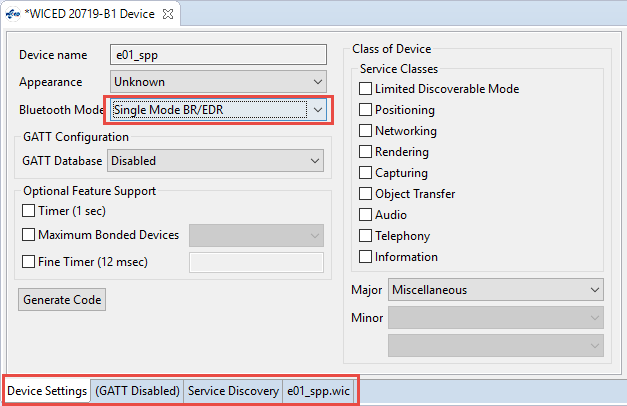
Enter a *Device name* and click "Finish". The name entered will be used for the name of the project as well as the name of the Bluetooth device. Either or both names can be changed later, but they both use this name to start out.



A folder with the name of the project is created under the *apps* folder containing the .wic file and a Make Target for the project is created. You will have to update the platform name in the Make Target if you are using a different platform such as a base board and shield combination. The project can be moved once the tool has generated the code. For example, it could be moved down two levels to a folder called *wbt101\ch05*. If you move the project, remember to edit the make target to reflect the new location.

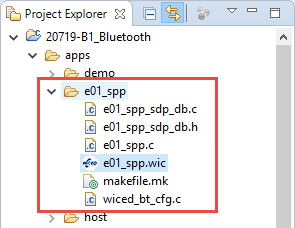
The center panel will show the .wic file as a configuration window with tabs along the bottom as shown here:



For a Classic Bluetooth project, select the Bluetooth Mode to be Single Mode BR/EDR as shown above. The GATT Database will automatically be set to Disabled and the Service Discovery tab will be enabled.

In the *Service Discovery* tab, you can add one or more services such as *Serial Port*, *Human Interface Device*, etc. Once you click the "+" sign to add a service, you can configure the service, add *Service Attributes*, and configure the attributes. GJL: need more detail here.

Once everything is setup the way you want it, go back to the Device Settings tab and click "Generate Code". When that step is done, the project folder will look like the following.



At this point, you can move the project if desired and then edit the files to add your own application's functionality. The main project C file has sections marked with the text "TODO" which give you some hints as to where customization may be required.

As stated earlier, you can re-run the tool by opening the .wic file from the project. Note that if you moved the project, you will no longer be able to re-run the tool using that configuration file unless you move the project back under the apps folder first. When you re-generate code, all the existing generated files will be backed up first (with the extension .bak\_1) before creating the new ones, but remember that any custom code you added will need to be manually added back into the new files.

# Advanced Topics

TBD

## Combo BT/BLE Projects

TBD

# Exercise(s)

Do we have the right hardware resources for these activities? Do we need a different shield?

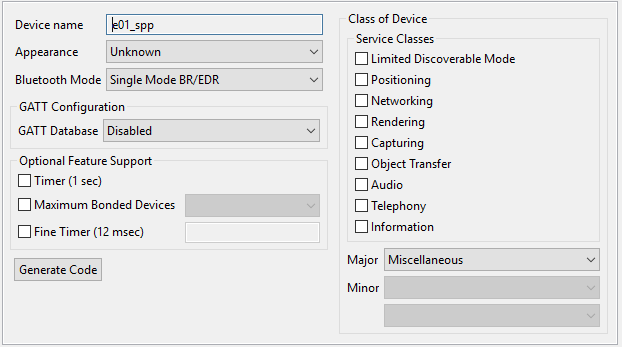
* 1. Serial Port Profile (SPP) ???

TBD

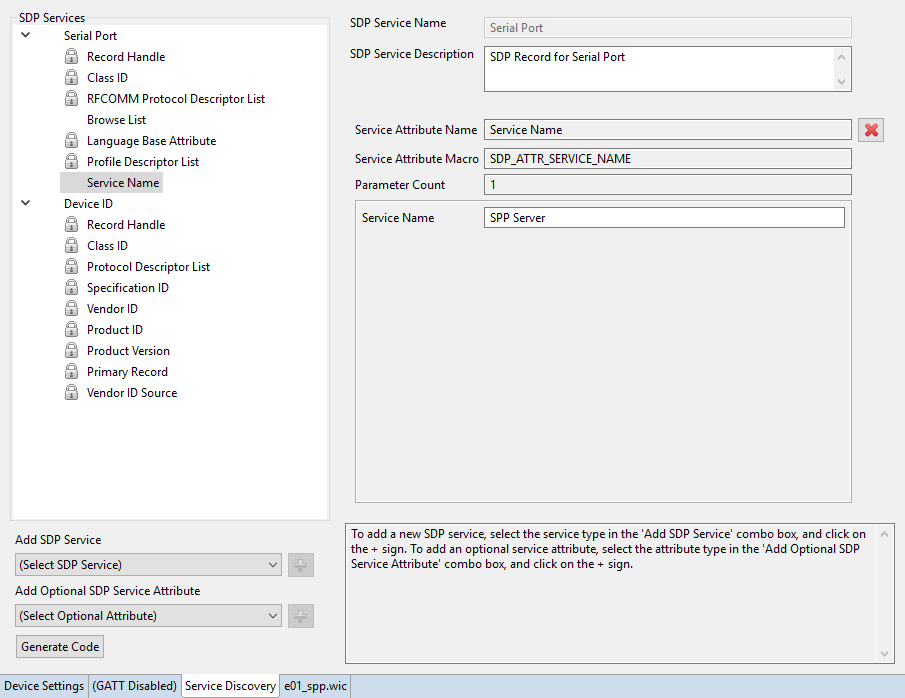
Create this first using Bluetooth designer and then compare to the existing spp/snip project?

Steps:

1. New Bluetooth Designer project
2. Setup configuration



1. Setup services and attributes



? Do we want/need the device ID service?

Service name: SPP Server

Vendor ID: 0x000F

Product ID: 0x0401

Product Version: 0x0001

1. Generate code
2. In makefile:
   1. Add $(NAME)\_COMPONENTS := spp\_lib.a
   2. Comment out the HCI\_TRACE\_OVER\_TRANSPORT so that PUART is used for debug messages
3. In wiced\_bt\_cfg.c:
   1. Change wiced\_set\_debug\_uart to use PUART (just change comments)
   2. Update buffer pool sizes. Large: 1056,3, Extra-large: 1056,2
4. In C file:
   1. Add includes for wiced\_bt\_spp.h and wiced\_bt\_sdp.h to C file
   2. (optional) Setup EID data and call wiced\_bt\_dev\_write\_eir – not actually needed. Why does SPP snip have this?
   3. Add global for the spp\_handle:

uint16\_t spp\_handle;

* 1. Add wiced\_bt\_spp\_reg\_t spp\_reg defines and structure (global):

**#define** SPP\_RFCOMM\_SCN 1

**#define** MAX\_TX\_BUFFER 1017

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 \*/

};

* 1. Add functions for the following (note – prototypes need to be placed before the spp\_reg structure since it defines the callbacks):
     1. void spp\_connection\_up\_callback(uint16\_t handle, uint8\_t\* bda) – save handle passed in to the global spp\_handle
     2. void spp\_connection\_down\_callback(uint16\_t handle) – reset spp\_handle to 0
     3. wiced\_bool\_t spp\_rx\_data\_callback(uint16\_t handle, uint8\_t\* p\_data, uint32\_t data\_len) – deal with data received
        1. e.g. for a loopback function, add wiced\_bt\_spp\_send\_session\_data(handle, p\_data, data\_len) to send back the received data
  2. Can also call wiced\_bt\_spp\_send\_session\_data other places to send whatever data you want. Just need the handle, pointer to the data, and length of data to send
  3. Add call to wiced\_bt\_spp\_startup(&spp\_reg) at the start of the app init function (before calling wiced\_bt\_sdp\_dp\_init) – this gives a pointer to the spp\_reg structure wi6.th the callback names
  4. Note: can add IO capabilities NONE if you don't want to have to enter the code

Note: Because we are not saving the key in NVRAM, you must remove the device from the PC and re-pair every time you power cycle or reset the kit (including reprogramming the FW). If not, the PC will think it is still using the old key and it won’t be able to connect.

* 1. Human Interface Device Profile (HID) ???

TBD

* 1. Headset Profile (HSP) ???

TBD

* 1. Hands-Free Profile (HFP) ???

TBD

* 1. Combo Bluetooth/BLE: SPP + BLE Custom CapSense ???

TBD

# Recommended Reading