Chapter 10: Debugging

After completing this chapter, you will gain a basic understanding of Host Controller Interface (HCI) commands and the WICED HCI format. You will then learn how to enable WICED HCI trace messages from the Bluetooth® stack and application. Finally, you will learn how to view them using the ClientControl and BTSpy programs.

Table of contents

## Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

10.1 Introduction

Hardware debugging on Bluetooth® applications can done using SWD just like on any other MCU application. However, if your device has an active Bluetooth® LE connection it will drop the connection when the CPU is halted in the debugger (after the Bluetooth® connection timeout). Therefore, for Bluetooth® LE applications it is more common to use WICED HCI and an application called BTSpy for debugging rather than a hardware debugger. Unlike traditional hardware debugging, WICED HCI provides debug messages while the Bluetooth® connection stays active. BTSpy allows you to "spy" on what the stack is doing during different Bluetooth® activities.

10.2 Host Controller Interface (HCI)

Before we get to BTSpy, we need to review the Bluetooth® stack and HCI in particular. You may recall this stack diagram from an earlier chapter:
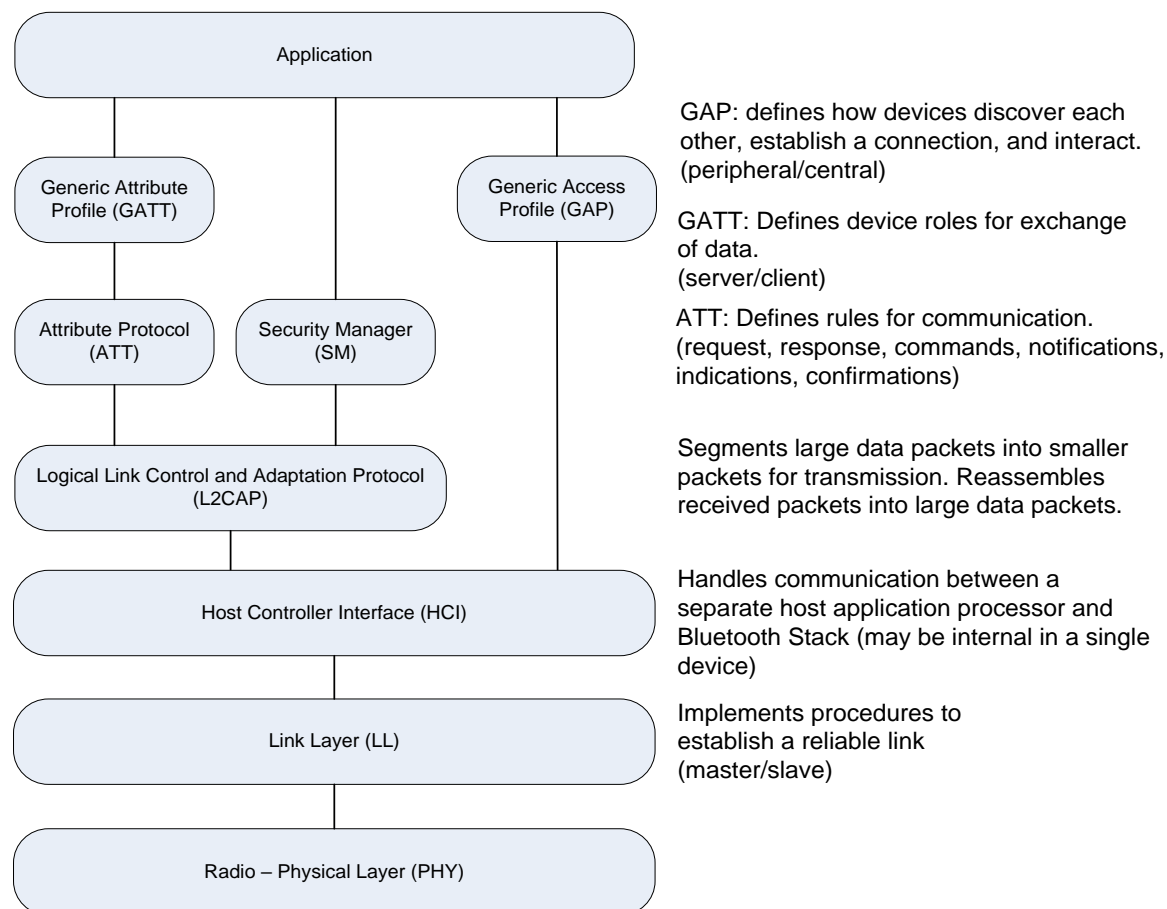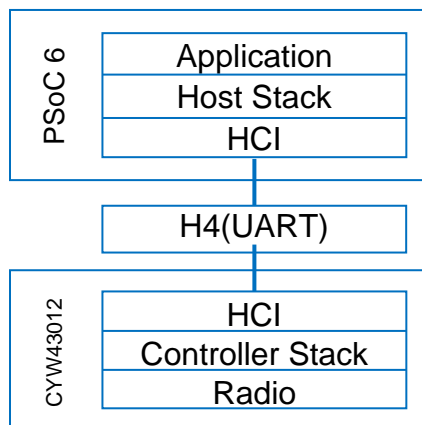

So far, we have spent a lot of time interacting with the GAP (to get connected), GATT (to exchange data) and SM (to handle security). The next block to talk about is the Host Controller Interface.

In some Bluetooth® systems, the radio is a separate chip from the one that runs the upper levels of the stack and the application. The radio chip is called the controller because it contains and controls the radio. The chip running the upper level of the stack and the application is called the host because it hosts the
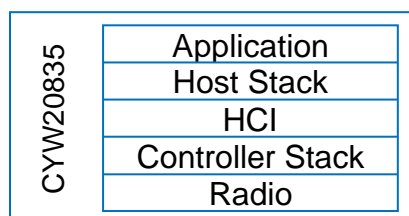
application. The interface between the host and controller is therefore called the Host Controller Interface, or HCI for short.

By standardizing the HCI, it allows different application processors to interface with Bluetooth® in a standard way. For example, the PSoC™ 6 and CYW43012 combo radio is a 2-chip solution:

```
PSoC 6   | Application   |
         | Host Stack    |
         | HCI           |

             | H4(UART)   |

CYW43012 | HCI             |
         | Controller Stack|
         | Radio           |
```

The above case is also true of large systems such as PCs where the system CPU will be the host and a separate Bluetooth® device will be the controller.

In other cases, the host and controller are integrated into one chip (like the CYW20835 that we have been using). This is common for smaller systems like IoT devices. Even though both sides are physically on the same chip, HCI persists. In that case, the HCI layer is essentially just a pass-through from the host side of the stack to the controller side of the stack.

```
CYW20835 | Application     |
         | Host Stack      |
         | HCI             |
         | Controller Stack|
         | Radio           |
```

Note: *We have been using the CYW20835 in "hosted mode" since it is running both sides of the stack. However, it can also be used in "controller mode" in which it will only run the controller side of the stack and the radio while the application and host stack are not used. In this case, an external device runs the host stack and communicates with the controller using HCI. That is, the CYW20835 can be used in a larger system such as a PC just like the previous example.*

10.3 WICED HCI

The HCI concept was extended by the Infineon software team to provide a means of mirroring the HCI traffic between the devices to allow debugging of the Bluetooth® interface while it is running. It also allows commands to be sent back to the device as well. This interface is called "WICED HCI". You can send WICED HCI messages and standard application debug messages to the same UART interface. The result is that from a PC you can view both sets of messages together.
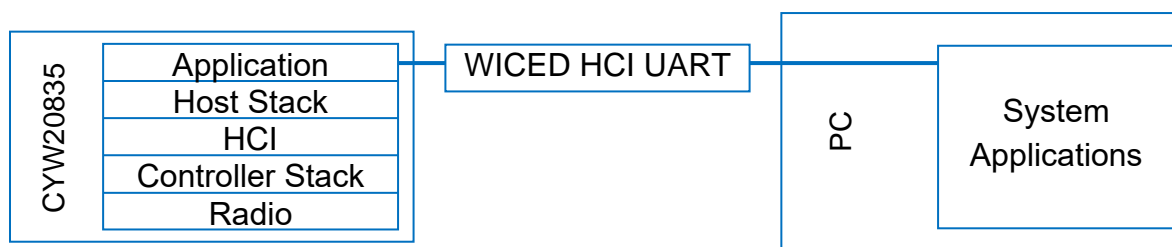
10.3.1 Message Format

WICED HCI is a packet-based format for PC applications to interact with the application running on the host. The packets have 3 standard fields plus an optional number of additional bytes for payload. The packet looks like this:

- 0x19 – the initial byte to indicate a WICED HCI packet
- 2-byte little endian Opcode consisting of a 1-byte Control Group (a.k.a. Group Code) and a 1-byte Sub-Command (a.k.a. Command Code)
- A 2-byte little endian length of the additional bytes
- An optional number of additional bytes for payload

The Control Group is one of a predefined list of categories of transactions including `Device=0x00`, `BLE=0x01`, `GATT=0x02`, etc. These groups are defined in *mtb_shared/wiced_btsdk/dev-kit/btsdk-include/<version>/hci_control_api.h*. Each Control Group has one or more optional Sub-Commands. For instance, the Device Control Group has Sub-Commands `Reset=0x01`, `Trace Enable=0x02`, etc.

The Control Group plus the Sub-Command together is called a "Command" or an "Opcode" and is a 16-bit number. For example, Device Reset = 0x0001. In the actual data packet, the Opcode is represented little endian. For example, the packet for a Device Reset is 19 01 00 00 00.

10.4 UART interfaces

As you have learned previously, there are two UARTs on the device we are using: the peripheral UART and the HCI UART.

10.4.1 Peripheral UART

The peripheral UART (PUART) is intended to be used by your application to send/receive whatever UART data you want. On the CYW920835 this UART is attached via a USB-UART bridge to your PC.

10.4.2 HCI UART

The HCI UART has four main uses:

1. Send/receive WICED HCI messages
2. Mirror HCI Commands to a PC
3. Download firmware to the device
4. Connect a controller stack to a host stack running on a separate application processor (this is not our use case)

There are 6 types of data that are typically transmitted via the two UARTs:

1. Raw data –data your application sends/receives to the PC (or a sensor) in whatever format you choose
2. Debug Traces –debugging messages from your application
3. WICED HCI messages – Packets of data in the WICED HCI format between the PC and your application
4. HCI Spy – a mirror of the standard HCI packets of data that cross HCI in the device
5. HCI Commands – packets of data to/from the controller and the host running on an application processor (again, not our use case)
6. Firmware – Firmware is downloaded via the HCI UART
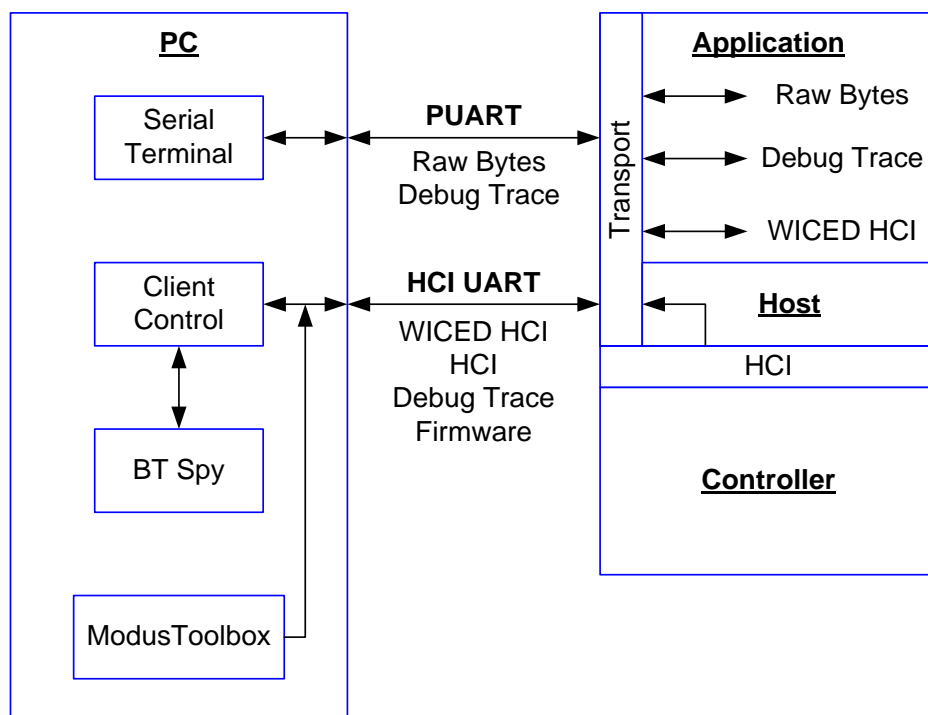
10.5   WICED HCI debug firmware architecture

10.5.1   Configuration

To send WICED HCI messages you first need to configure the transport system for the HCI UART. To do this you need to do the following things:

1. Include *wiced_transport.h* and *hci_control_api.h*.
2. Declare a pointer to a `wiced_transport_buffer_pool_t`.
3. Create a global structure of type `wiced_transport_cfg_t` which contains the HCI UART configuration including the UART configuration, size of the receive buffer, a pointer to a status handler function (if needed), and a pointer to the TX complete callback function (not usually needed).
4. Call `wiced_transport_init` (with a pointer to your configuration structure).
5. Call `wiced_transport_create_buffer_pool` to create buffers for the transport system to use.
6. Create the HCI TX handler function (not usually needed).

Note: *If you also want to receive WICED HCI messages (to send commands back to the device), you will need to set up RX buffer pools and specify a pointer to your RX handler function it in the* `wiced_transport_cfg_t` *structure. This is less commonly used so we won't cover it in this class. However, code examples exist that demonstrate using WICED HCI to send commands to the device.*

Each of the steps are shown below with examples:

10.5.1.1   Includes

There are two includes required for WICED HCI API functions:

```
#include "wiced_transport.h"
#include "hci_control_api.h"
```

10.5.1.2   Transport buffer pool pointer

Create a global pointer to a `wiced_transport_buffer_pool_t` like this:

```
static wiced_transport_buffer_pool_t* transport_pool = NULL;
```

10.5.1.3   Transport Configuration Structure

An example transport configuration structure for HCI is shown below. The value of `HCI_UART_DEFAULT_BAUD` is 3000000. In this example, there are no HCI TX or RX handlers implemented so they are specified as NULL. The RX functionality is not used, so the RX buffer size and count are set to 0. This should be specified as a global variable.

```
/* Transport Configuration for WICED HCI*/
const wiced_transport_cfg_t  transport_cfg =
{
    .type                = WICED_TRANSPORT_UART,    /**< Wiced transport type. */
    .cfg.uart_cfg        =
    {
     .mode = WICED_TRANSPORT_UART_HCI_MODE,  /**<  UART mode, HCI or Raw */
     .baud_rate = HCI_UART_DEFAULT_BAUD      /**<  UART baud rate */
    },
    .rx_buff_pool_cfg    =
        {
            .buffer_size = 0,                /**<  Rx Buffer Size */
            .buffer_count = 0                /**<  Rx Buffer Count */
        },
    .p_status_handler    = NULL,    /**< Wiced transport status handler.*/
    .p_data_handler      = NULL,    /**< Wiced transport receive data handler. */
    .p_tx_complete_cback = NULL     /**< Wiced transport tx complete callback. */
};
```

10.5.1.4   Transport Initialization and buffer pool creation

Once the structure is set up, you have to initialize the transport and create buffer pools. This is commonly done right at the top of `application_start`.

```
#define TRANS_UART_BUFFER_SIZE  1024
#define TRANS_UART_BUFFER_COUNT 2

/* Initialize the transport configuration */
wiced_transport_init( &transport_cfg );

/* Initialize Transport Buffer Pool */
transport_pool = wiced_transport_create_buffer_pool ( TRANS_UART_BUFFER_SIZE,
    TRANS_UART_BUFFER_COUNT );
```

10.5.2   Sending WICED HCI messages

Once the WICED HCI port is configured the way you want it, you need to send messages to it. There are two common sources: trace messages that are generated by the Bluetooth® stack, and debug messages generated by the application.

10.5.2.1   Bluetooth® trace messages

To configure the stack to generate Bluetooth® protocol HCI trace messages, the application must define a callback function and register it with the stack using `wiced_bt_dev_register_hci_trace`. The register function should be called in the `BTM_ENABLED_EVT` once the Bluetooth stack has been initialized. For example:

```
wiced_bt_dev_register_hci_trace(app_btspy_callback);
```

The callback function implementation must call `wiced_transport_send_hci_trace` with the data received in the callback. The callback function looks like this:

```
void app_btspy_callback(wiced_bt_hci_trace_type_t type, uint16_t length, uint8_t* p_data)
{
    wiced_transport_send_hci_trace( transport_pool, type, length, p_data );
}
```

With just those few lines of code, you will get to see everything that is going on inside the stack. This is an extremely helpful debugging tool for Bluetooth® as you will see.

10.5.2.2   Application debug messages

As you have previously seen, `WICED_BT_TRACE` can be used to generate debug messages from your application. So far, we have sent those messages to the PUART interface. If you want, you can leave that as-is so that application debug messages go to the PUART while Bluetooth® stack trace messages go to the HCI UART. However, it is often useful to have the two types of messages go to the same place. To do that, you just need to change the parameter when you call the `wiced_set_debug_uart` function. It allows you to specify the format of your message (either plain text or WICED HCI) and the destination for those messages (PUART, HCI UART or none). The possible choices are:

```
/** Debug trace message destinations. Used when calling wiced_set_debug_uart().*/
typedef enum
{
    WICED_ROUTE_DEBUG_NONE   =  0x00, /**< No traces */
    WICED_ROUTE_DEBUG_TO_WICED_UART, /**< send debug strings in formatted WICED HCI
                          messages over HCI UART to ClientControl or MCU */
    WICED_ROUTE_DEBUG_TO_HCI_UART,   /**< send debug strings as plain text to HCI UART,
                          used by default if wiced_set_debug_uart() not called */
    WICED_ROUTE_DEBUG_TO_DBG_UART,   /**< Deprecated */
    WICED_ROUTE_DEBUG_TO_PUART       /**< send debug strings as plain text to the
                          peripheral uart (PUART) */
} wiced_debug_uart_types_t;
```

Notice that some of the combinations are not valid. For instance, you cannot send WICED HCI formatted messages to the PUART. You should be careful about the enumeration names as they can be a little bit confusing.

To format application debug messages in HCI format and send them to the HCI UART, you would use:

```
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART );
```

Note: *The configuration specified in* `wiced_set_debug_uart` *is just for the* `WICED_BT_TRACE` *messages. It does not affect the Bluetooth® stack trace messages, which are always sent on the HCI UART.*

When you format debug trace messages in WICED HCI format (i.e. by specifying `WICED_ROUTE_DEBUG_TO_WICED_UART`) the messages sent by `WICED_BT_TRACE` are automatically converted to the equivalent WICED HCI message. For example, if you call `WICED_BT_TRACE("abc");` it will send: `19 02 00 03 00 41 42 43`.

- 19 = HCI Packet
- 02 = Command Code (Trace)
- 00 = Group Code (Device)
- 03 00 = Number of additional bytes (little endian)
- 41 = 'a'
- 42 = 'b'
- 43 = 'c'

That's what you would see if you connected a standard serial terminal emulator (such as Putty) to the HCI UART port. That isn't very readable, so we need a different set of programs. Namely, ClientControl and BTSpy.

10.6   ClientControl and BTSpy

Now that you are sending WICED HCI encoded messages, how do you interpret them? The answer involves two separate programs – ClientContol and BTSPy.

10.6.1   Client Control

ClientControl is a PC based program that connects to the HCI UART port and gives you a GUI to send and receive WICED HCI messages.

It is contained in the **BTSDK Host Apps for Bluetooth® Classic and Bluetooth® Low Energy** library (*btsdk-host-apps-bt-ble*) which you can add to your application using the library manager.

Once the library has been downloaded, the executable for *ClientControl* and the source code for different operating systems can be found in:

> *mtb_shared/wiced_btsdk/tools/btsdk-host-apps-bt-ble/<version>/client_control*

If you have the tool listed in the `CY_BT_APP_TOOLS` variable in the application's *Makefile*, you can run it from the command line interface by using the following command from the application's root directory:

```
make open CY_OPEN_TYPE=ClientControl
```

Note: *All template applications used for this class include ClientControl and BTSpy in the* `CY_BT_APP_TOOLS` *variable in the Makefile.*

When you open the tool, select the appropriate COM port (make sure to choose the HCI UART, not the PUART) and set the Baud Rate – the default for HCI UART is 3,000,000. Then click on Open Port.
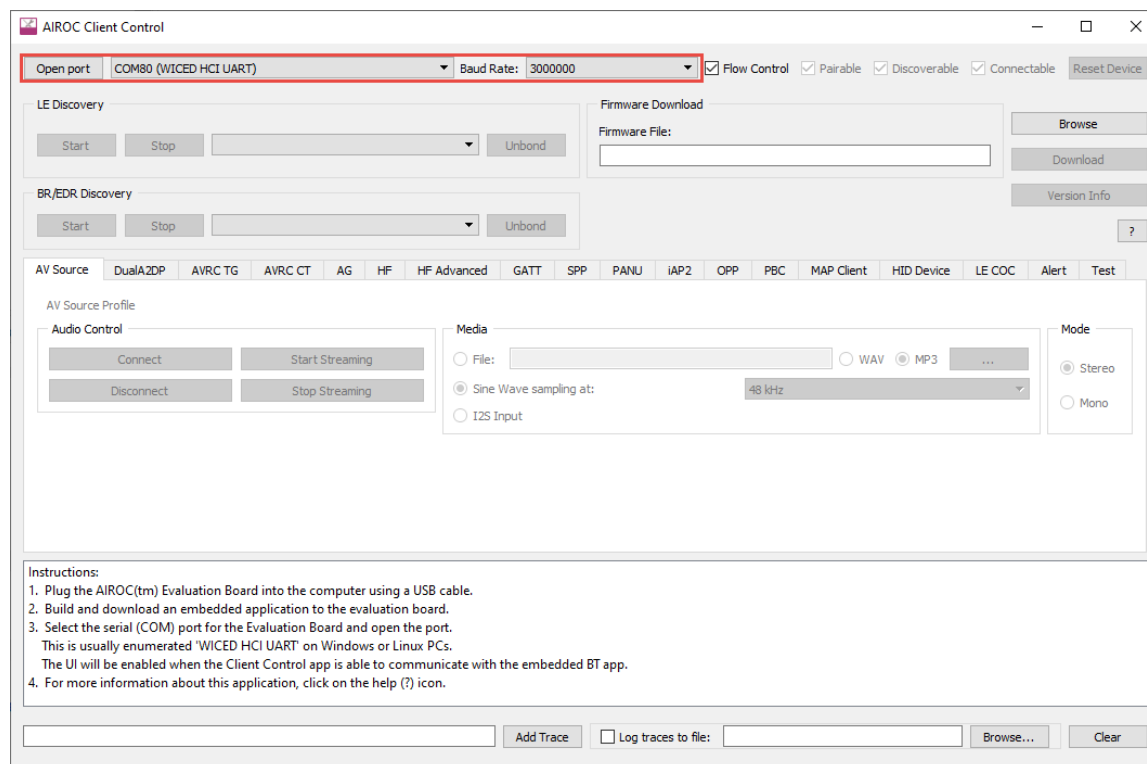
As you can see, there are a lot of tabs that allow you to send all sorts of WICED HCI commands to the device. We won't be using this functionality, but if you do want to use it, you need to set up an RX handler that receives and decodes the messages on the device and takes the appropriate action. Code examples exist that demonstrate this functionality.

Note: *Since ClientControl connects to the HCI UART port, you will need to disconnect from the port whenever you want to re-program the kit.*

10.7   BTSpy Application

ClientControl is great for decoding HCI messages and sending HCI commands, but it still isn't as user friendly as we would like for Bluetooth® stack trace messages. To fix that, we can use a program called *BTSpy*. It connects into *ClientControl* and re-formats the messages for human readability. Expanding on a picture from earlier, the data path looks like this:
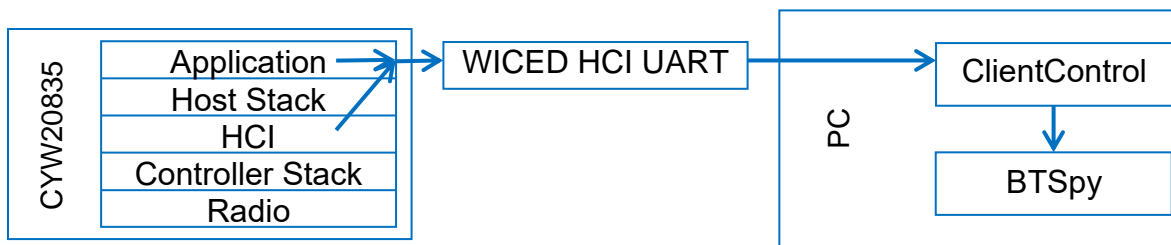
Remember that the HCI UART can carry both Bluetooth® stack trace messages as well as application debug messages, so you can see everything in one place.

The *BTSpy* application is available in the **BTSDK utilities** library (btsdk-utils). This library is included as a dependency in BTSDK BSPs, so you will automatically have access to it in your application.

As was the case with *ClientControl*, all template applications used for this class include *ClientControl* and *BTSpy* in the `CY_BT_APP_TOOLS` variable in the *makefile*. Therefore, to launch the tool from the application's root directory using the CLI, you can launch it as follows:

```
make open CY_OPEN_TYPE=BTSpy
```

The executable for *BTSpy* and the source code for different operating systems can be found at:

mtb_shared/wiced_btsdk/tools/btsdk-utils/<version>/BTSpy/<Windows|Linux64|OSX>

The process to run *BTSpy* is as follows:

1. Build and download the application with the proper configuration to the kit.
2. Run *ClientControl*.
3. In *ClientControl*, select the HCI UART port in the UI, and set the baud rate to the baud rate used by the application for the HCI UART (the default baud rate is 3000000).
4. Run *BTSpy*.
5. In *ClientControl*, open the HCI UART COM port by clicking on "Open Port".
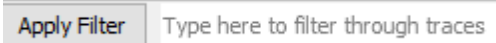
Note: *Remember that the HCI UART is the same port used for programming, so you must disconnect in ClientControl each time you want to re-program.*

Note: *If you reset the kit while the ClientControl utility has the port open, the kit will go into recovery mode (because the CTS line is asserted). Therefore, you must disconnect the ClientControl utility before resetting the kit.*

When *BTSpy* opens you will see a window like this:
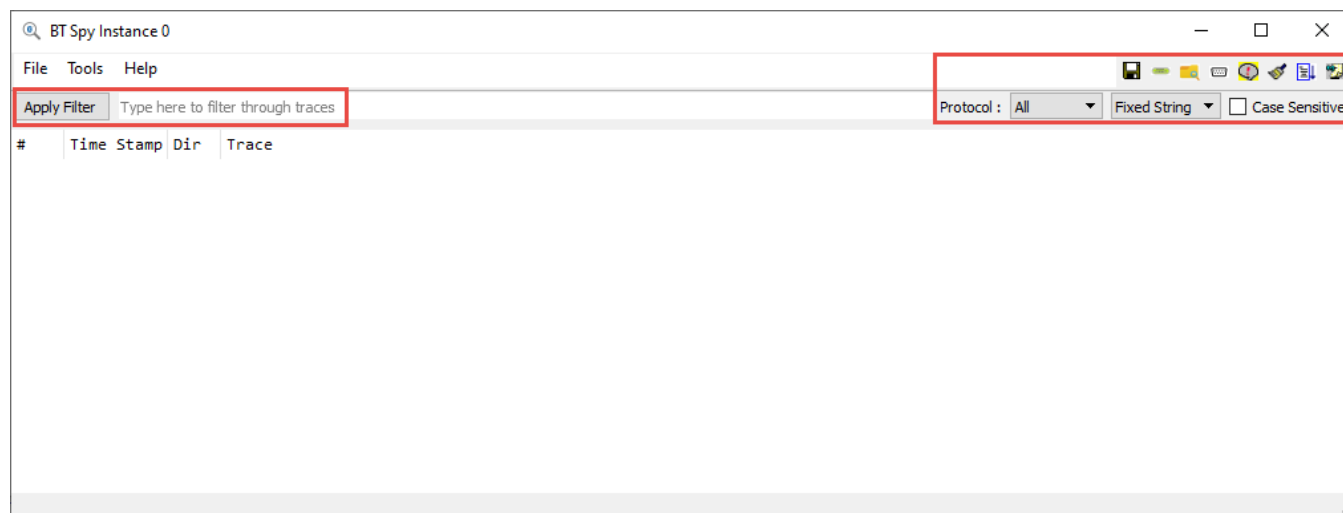
A few controls of interest are:

Command	Icon	Use
Clear Trace		Clear the log history.
Note		Allows you to enter a note that shows up in the log. The note will have a row of asterisk characters above/below it.
File Logging Options		Set up and start/stop logging messages to a file.
Open Logged File		Open a previous log file in the tool.
Protocol		Allows you to filter out messages of certain types (ALL, HCI, GENERIC, etc.).
Apply Filter		Allows you to filter traces using a text string.
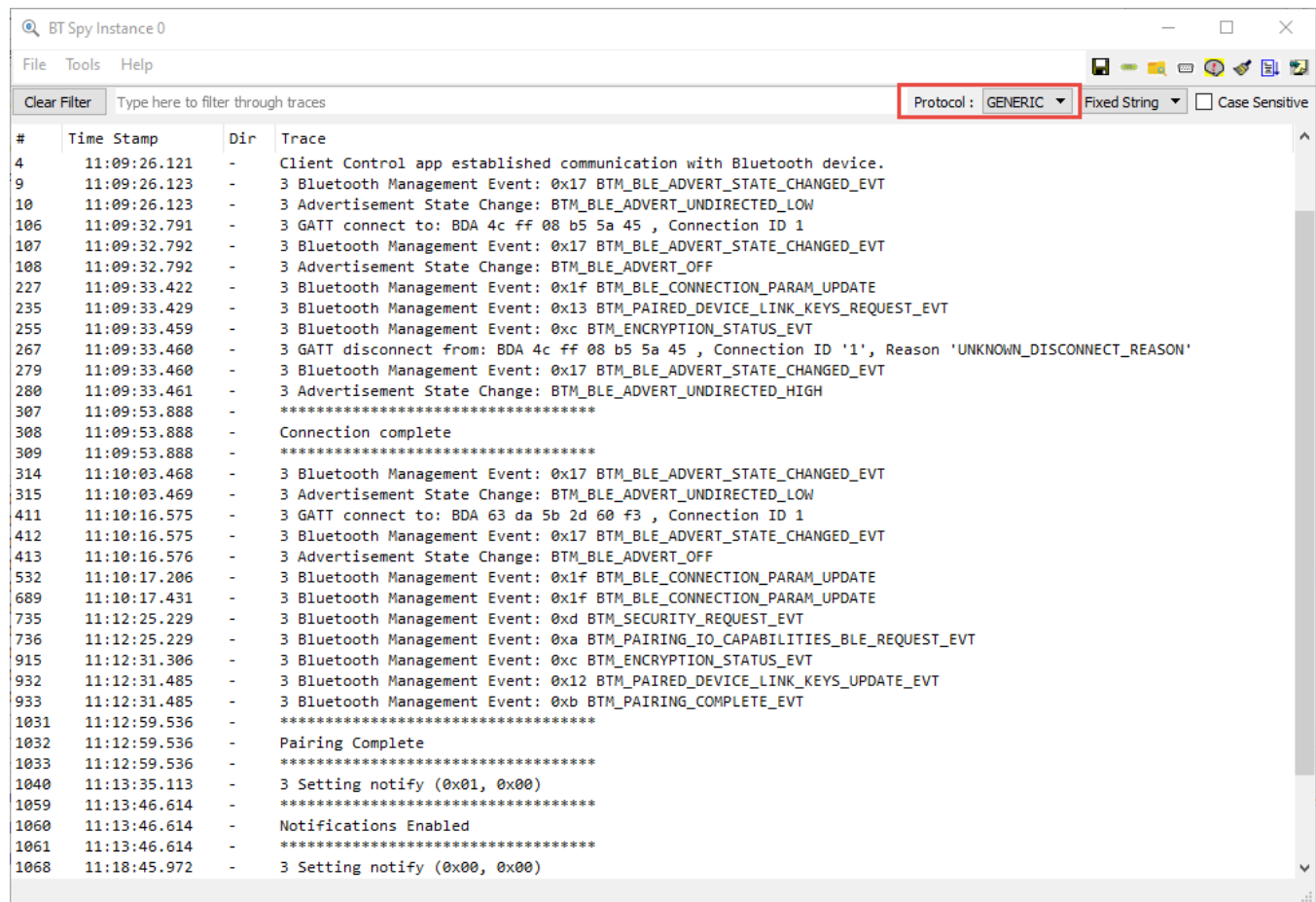
As messages are received in BTSpy, they will be color-coded:

- Green text with a light green background for incoming Bluetooth® traffic.
- Blue text with a yellow background for outgoing Bluetooth® traffic.
- Black text with a white background for application debug messages.

The following figure shows an example of part of a log that was created for a Bluetooth® peripheral that allows pairing and has notifications. The lines shown with starts above/below were added using the Note function (the stars are added automatically by the tool). As you can see, after pairing is complete, the user enables notifications from the central. To do this, the central sends a GATT write request (1) to handle 0x000C (which is the handle for the CCCD) and a value of 01 00 (which is the code to enable notifications). The peripheral responds to the write request (2). Next, the central sends a read request (3) to the same handle and the peripheral sends back a read response (4) with the new value of the CCCD of 01 00. This allows you to see exactly what is being sent back and forth across the Bluetooth® link which is extremely helpful during debugging.

If you change the protocol filter from ALL to GENERIC, you will see just the messages that are printed from the application along with the notes that were added manually.

10.8 Exercises

Exercise 1: Add WICED HCI traces to pairing application and use BTSpy to test

In this exercise, you will start from the Bluetooth® pairing exercise from a previous chapter and you will add in WICED HCI trace capability. You will then use *BTSpy* to look at the application messages and HCI messages.

Application Creation

1. Create a new application with the CYW920835M2EVB-01 BSP.

   On the application template page, use the **Browse** button to start from the completed application for the ch05_ex01_pair exercise.
   If you did not complete that exercise, the solution can be *found in Projects/key_ ch05_ex01_pair.*
   Name the new application **ch10_ex01_btspy**.

2. Open the library manager and add the **BTSDK Host Apps for Bluetooth® Classic and Bluetooth® Low Energy** library (btsdk-host-apps-bt-ble) to the application.

3. Open the Bluetooth® configurator and change the device name to `<inits>_btspy`.

4. Follow the instructions in section 10.5 to enable WICED HCI logging for both Bluetooth® stack trace messages and application debug messages.

Note: *You must:*

     a.   *Configure and initialize the HCI UART.*

     b.   *Configure the Bluetooth® stack to send HCI UART messages.*

     c.   *Route the application's debug messages to the HCI UART.*

Note: *The call to `wiced_set_debug_uart` is already in `application_start`. You will need to update its parameter to route application debug messages to the HCI UART.*

Testing

1. Program your application to the kit.

2. Run *ClientControl*. Select the port and baud rate, but <u>do not open the port yet</u>.

Note: *Remember to select the HCI UART port and set the baud rate to the default of 3000000.*

3. Run *BTSpy*.

4. Open the port from *ClientControl*.

5. Observe messages in the *BTSpy* program.

6. Open the AIROC™ Connect application. Search for and connect to your device.

Note: *If advertising times out on your device before you get to this step you will have to reset the kit. However, if you reset with the HCI UART port open in ClientControl, the kit will go into recovery mode. Therefore, to reset the kit you must:*

*a. Click **Close Port** in ClientControl*
b. Reset the kit
*c. Click **Open Port** in ClientControl*

7. Once the connection has been made, enter a note in *BTSpy* that says "Connection complete".

8. Click the GATT DB widget followed by the Unknown Service and then the Characteristic that has Read & Notify properties. This is the Counter Characteristic that counts button presses.

9. Accept the pairing request on the mobile device. Once it completes, enter a note in *BTSpy* that says "Pairing Complete".

10. Click the **Notify** button in AIROC™ Connect. Once it completes, enter a note in *BTSpy* that says "Notifications Enabled".

11. Scroll through the *BTSpy* log messages to see what is going on during each operation. You will see the notes that you added to be able to tell when each operation starts/stops.

12. Use the **Protocol** dropdown to select **GENERIC**. Now you will see just the informational messages from your application along with the notes that you added.

13. Disconnect and close AIROC™ Connect. Close *BTSpy* and *ClientControl*.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.