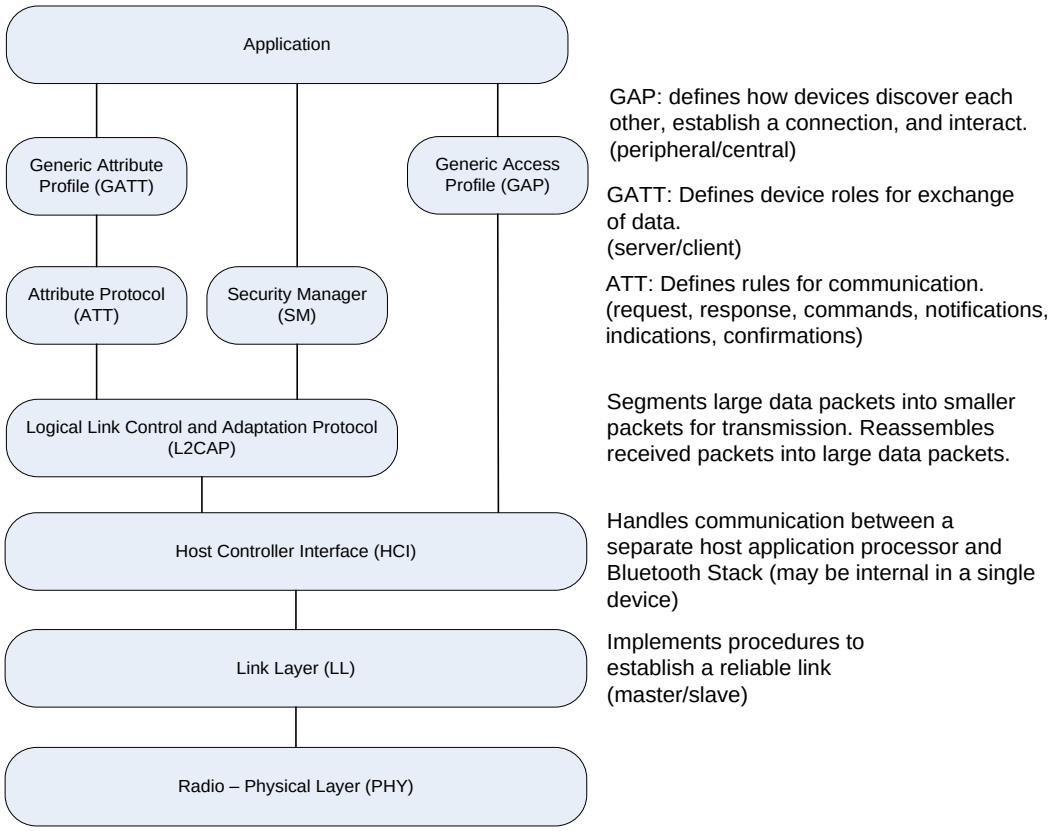
Chapter 5: Debugging

At the end of this chapter you should understand how to use the WICED debugging hardware including the UARTs (PUART and HCI UART) as well as the ARM debugging port. You should also have a basic understanding of WICED Debug Trace, HCI Commands, WICED HCI commands as well as using a Terminal, Client Control, BTSPY and the Eclipse Debugger.

5.1 WICED Chips & the Architecture of HCI

In many complicated systems, hierarchy is used to manage the complexity. WICED Bluetooth is no different. The WICED Bluetooth Stack is called a Stack because it is a set of blocks that have well defined interfaces. Here is a simple picture of the software system that we have been using. You have been writing code in the block called "Application". You have made API calls and gotten events from the "Attribute Protocol" and you implemented the "Generic Attribute Profile" by building the GATT Database. Moreover, you advertised using GAP and you Paired and Bonded by using the Security Manager.



5.1.1 HCI

The next block to talk about is the "Host Controller Interface".

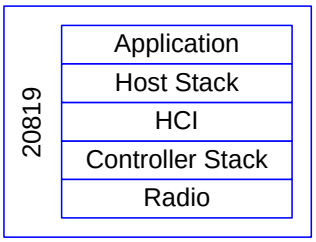
For technical and cost reasons, when Bluetooth was originally created the Radio was a separate chip from the one that was running the Application. The Radio chip took the name of Controller because it was the Radio and Radio Controller, and the chip running the Application was called the Host because it was hosting the Application.

The interface between the Host and the Controller was typically UART or SPI. The data flying over that serial connection was formatted in Bluetooth SIG specific packets called "HCI Packets".

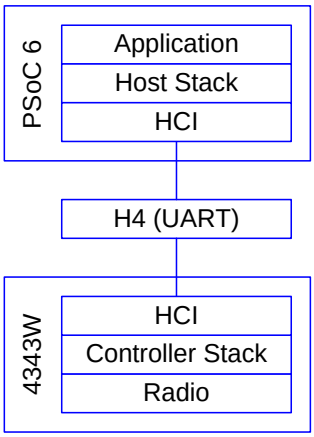
By standardizing the HCI interface, it allowed big application processors (like those exiting in PCs and cellphones) to interface with Bluetooth. As time went by the Host and Controller have frequently merged into

one chip (e.g. CYW20819), however the HCI interface persists even though both sides may be physically on the same chip. In this case, the HCI layer is essentially just a pass-through.

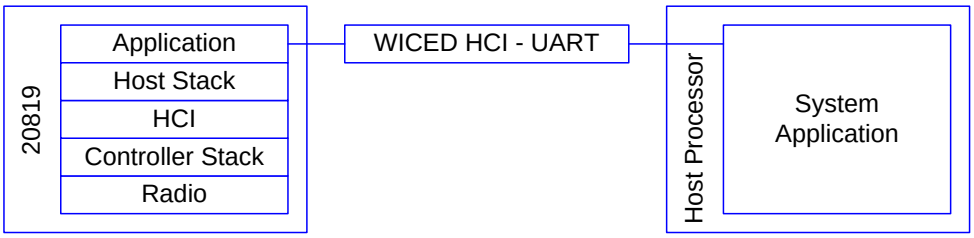
Note that even on single chip Host/Controller solutions, it is still be possible to use the chip in Controller mode with an HCI interface. For example, when the CYW20819 is used in Linux applications, it is booted in Controller mode and the Stack runs in Linux. Likewise, when the chip is put into recovery mode to do programming, it boots as a Controller.



In some devices, the WICED Bluetooth Stack can be split into a "Host" and a "Controller" part. For example, the PSoC 6 and 4343W Combo Radio is a 2-chip solution that looks like this:



The HCI concept was extended by the WICED Software team to provide a means of communication between the application layer of two chips. They call this interface "WICED HCI". The WICED HCI messages can do many things including providing a mechanism to control or observe the HCI messages inside the BT device.



5.2 UART Debugging

5.2.1 Architecture

There are typically two UARTs in the WICED Bluetooth chips: The Peripheral UART and the HCI UART.

Peripheral UART

The Peripheral UART (PUART) is intended to be used by your Application to send/receive whatever UART data you want. On the CYW920819 this UART is attached via a USB-UART bridge to your PC.

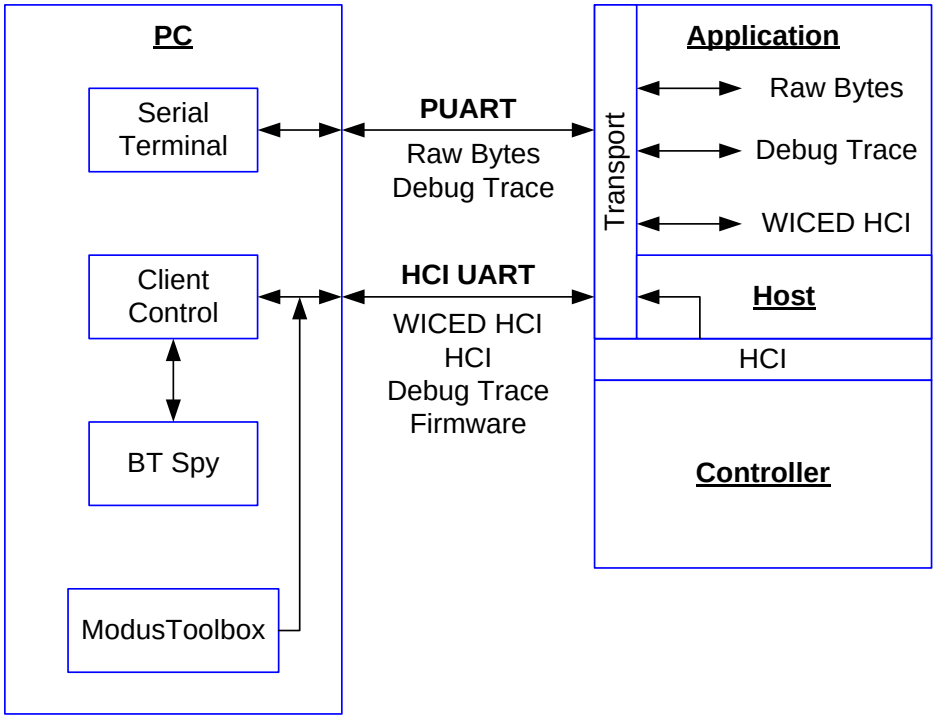
HCI UART

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 Controller Stack for programming
4. Connect the Controller Stack to a Host Stack running on a PC or an Application Processor (not our use case since we have an embedded Host stack)

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 sensor) in whatever format you choose
2. Debug Traces –debugging messages from your Application
3. WICED HCI messages – Packets of data in the Cypress WICED HCI format between the PC and your Application
4. HCI Spy – a mirror of the packets of data that cross HCI in the WICED chip
5. HCI Commands – packets of data to/from the Controller and the Host running on a PC or an Application Processor (when the chip is in Controller mode)
6. Firmware – your downloaded firmware via HCI formatted packets from the PC



5.2.2 Debugging Traces

Throughout this book, we have been using the API WICED_BT_TRACE to print out debugging messages in plain text to the PUART. It turns out that you can print your messages in two formats (plain text or WICED HCI) and you can print them to one of three places (PUART, HCI UART or None). When you called the function wiced_set_debug_uart with a parameter of wiced_debug_uart_types_t you specify a combination of destination UART and formatting.

```
/** 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 print WICED HCI messages to the PUART. You should be careful about the enumeration names as they can be a little bit confusing.

Not that this configuration is just for the WICED_BT_TRACE messages. WICED HCI messages are sent/received using a different mechanism that we will discuss next. Often you may want to send WICED_BT_TRACE messages to the PUART while using the HCI UART for WICED HCI messages, as you'll see.

5.2.3 WICED HCI & the Client Control Utility

WICED HCI

You will try this out in Exercise - 5.2.

WICED HCI is a packet-based format for PC applications to interact with the Application in a WICED Bluetooth device. WICED HCI 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 hci_control_api.h. This can be found in wiced_btsdk/dev-kit/btsdk-include. 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 = 19 01 00 00 00.

Transport Configuration

You will use this information in Exercise - 5.1 and Exercise - 5.2

To send and receive WICED HCI messages you 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 size of the receive and transmit buffers, a pointer to a status handler function, a pointer to the RX handler function and a pointer to the TX complete callback function (if 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 RX handler function and TX handler function (if needed).

Each of these are shown below with examples.

Includes

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

Transport Buffer Pool Pointer

Add a global pointer to a wiced_transport_buffer_pool_t like this:

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

Transport Configuration Structure

An example transport configuration structure for HCI is shown below. The default baud is 3,000,000. In this example, there is no HCI TX handler implemented so it is specified as NULL.

```
/******************************************************************
 * Transport Configuration
 ******************************************************************/
#define TRANS_UART_BUFFER_SIZE  1024
#define TRANS_UART_BUFFER_COUNT 2

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 = TRANS_UART_BUFFER_SIZE,              /**<  Rx Buffer Size */
        .buffer_count = TRANS_UART_BUFFER_COUNT             /**<  Rx Buffer Count */
    },
    .p_status_handler    = NULL,                            /**< Wiced transport status handler.*/
    .p_data_handler      = hci_control_process_rx_cmd,      /**< Wiced transport receive data handler. */
    .p_tx_complete_cback = NULL                             /**< Wiced transport tx complete callback. */
};
```

Transport Init and Buffer Pools

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

```
    /* 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 );
```

RX Handler

Your application is responsible for handling the HCI commands sent to your application. When a new command is sent, your RX handler function will be called. This function should just make sure that a legal packet has been sent to it, and then do the right thing.

The example RX handler below just verifies the packet is legal, extracts the Command (i.e. Opcode), optional data length, and optional payload. It then prints out a message and sends back a message to the host formatted as a WICED HCI command called HCI_CONTROL_EVENT_COMMAND_STATUS with a data value of HCI_CONTROL_STATUS_UNKNOWN_GROUP.

```
/* Handle Command Received over Transport */
uint32_t hci_control_process_rx_cmd( uint8_t* p_data, uint32_t len )
{
    uint8_t  status = 0;
    uint8_t  cmd_status = HCI_CONTROL_STATUS_SUCCESS;
    uint16_t opcode = 0;
    uint8_t* p_payload_data = NULL;

    WICED_BT_TRACE("hci_control_process_rx_cmd : Data Length '%d'\n", len);

    // At least 4 bytes are expected in WICED Header
    if ((NULL == p_data) || (len < 4))
    {
        WICED_BT_TRACE("Invalid Parameters\n");
        status = HCI_CONTROL_STATUS_INVALID_ARGS;
    }
    else
    {
        // Extract OpCode and Payload data from little-endian byte array
        opcode = (uint16_t)( ((p_data)[0] | ((p_data)[1] << 8)) );
        p_payload_data = &p_data[sizeof(uint16_t)*2];

        // TODO: Process received HCI Command based on its Opcode
        // (see 'hci_control_api.h' for additional details)
        switch (opcode)
        {
        default:
            // HCI Control Group was not handled
            cmd_status = HCI_CONTROL_STATUS_UNKNOWN_GROUP;
            wiced_transport_send_data(HCI_CONTROL_EVENT_COMMAND_STATUS, &cmd_status,
                                      sizeof(cmd_status));
            break;
        }
    }

    // When operating in WICED_TRANSPORT_UART_HCI_MODE or WICED_TRANSPORT_SPI,
    // application has to free buffer in which data was received
    wiced_transport_free_buffer( p_data );
    p_data = NULL;

    return status;
}
```

WICED HCI TX

Inside of your application you can send WICED HCI messages to the host by calling wiced_transport_send_data with the 16-bit opcode, a pointer to the optional data and the length of the data. In the situation where you have the WICED_BT_TRACE setup to send WICED HCI Trace messages, when you call the WICED_BT_TRACE API all it does is format your data, then it calls this:

sprintf(string,…);

wiced_transport_send_data(HCI_CONTROL_EVENT_WICED_TRACE, &string, strlen(string));

For example, if you call *WICED_BT_TRACE("abc");* it will send *19 02 00 03 00 41 42 43*

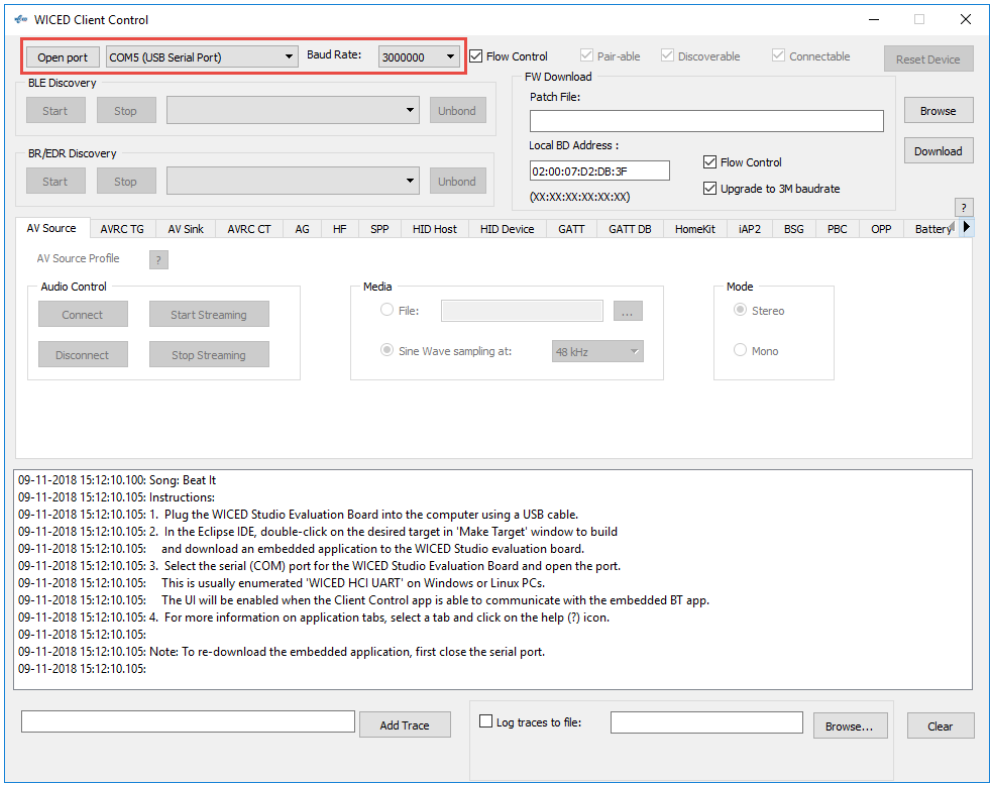
19 = HCI Packet
02 = Command Code (Trace)
00 = Group Code (Device)
03 00 = Number of additional bytes (little endian)
41 = 'a'
42 = 'b'
43 = 'c'

Client Control Utility (Windows, Linux, MacOS)

The *Client Control* utility 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 can be launched from the Tools section in the Quick Panel. The executable along with the source code can be found in the wiced_btsdk at:

wiced_btsdk/tools/btsdk-host-apps-bt-ble/client_control

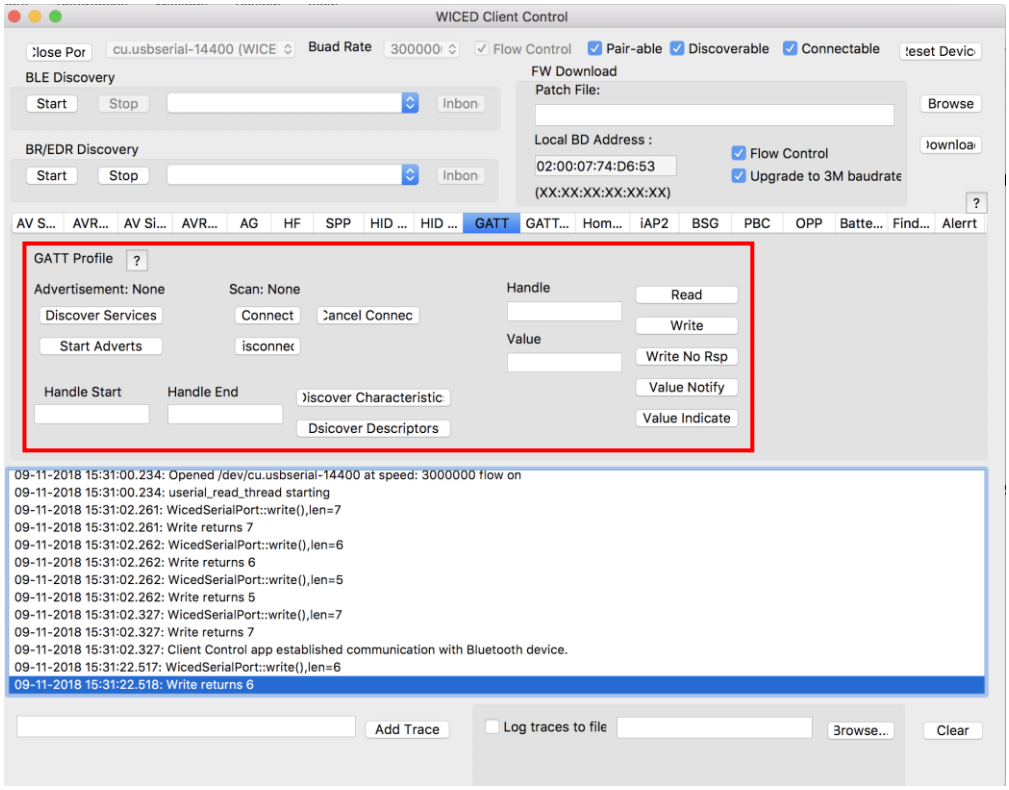
Once you open the tool, select the appropriate COM port (make sure to choose the HCI UART, not the PUART) and set the Baud Rate – the default for HCI UART is 3,000,000. Then click on Open Port to make the connection.



Note: BTSpy shows up in the Tools section of the Quick Panel because of this line in the application's makefile:

```
CY_BT_APP_TOOLS=BTSpy ClientControl
```

The GUI has buttons and text boxes that make sense for each WICED HCI Control group. For example, in the picture below the “GATT Profile” tab is selected which gives you logical access to the commands in the GATT_COMMAND control group such as Start Adverts, Discover Services, Connect, etc. Return messages will show up in the log window at the bottom.



5.2.4 Using BTSpy & the Client Control to view HCI commands

You will try this out in Exercise - 5.1.

When you are trying to figure out what in the world is going on between your Host API calls and the Controller, BTSpy is a very useful utility. Note that these calls may be "virtual". That is, they may be calls between the Host and Controller Stacks on a single device such as the CYW20819.

Like CilentControl, the BTSpy utility can be launched from the Tools section in the Quick Panel and is available for Windows, Linux, and MacOS.

The executable and source code are available in the wiced_btsdk. The path is:

wiced_btsdk/tools/btsdk-utils/BTSpy

Introduction

The *BTSpy* trace option provides an advantage over the UART options. Namely, applications may configure the Stack to generate Bluetooth protocol trace messages showing all activity between the host and the controller over the virtual HCI interface embedded in the CYW20819 device. The Bluetooth protocol trace messages will be encoded into WICED HCI message packets and sent along with application trace messages from *WICED_BT_TRACE()*, which can be displayed by the *BTSpy* utility. You can then view the application trace messages in sequence with the corresponding Bluetooth protocol trace messages.

BTSpy connects into the Client Control Utility that we discussed previously. So, the setup is exactly the same with one exception. Namely, you will want to reroute debug trace messages to WICED_ROUTE_DEBUG_TO_WICED_UART instead of to WICED_ROUTE_DEBUG_TO_PUART. That way, you will see both your application trace messages and messages from the virtual HCI port in the same window.

Application Configuration

To configure the stack to generate Bluetooth protocol trace messages, applications must define a callback function and register it with the stack using the *wiced_bt_dev_register_hci_trace()* API. This 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 should call the *wiced_transport_send_hci_trace()* API 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 );
}
```

Viewing Traces

To view application and Bluetooth protocol traces in *BTSpy*:

1. Setup the Transport for HCI just like in the previous section. Namley:
    a. Include header files
    b. Add a global pointer to a buffer pool.
    c. Setup the Transport Configuration Structure.

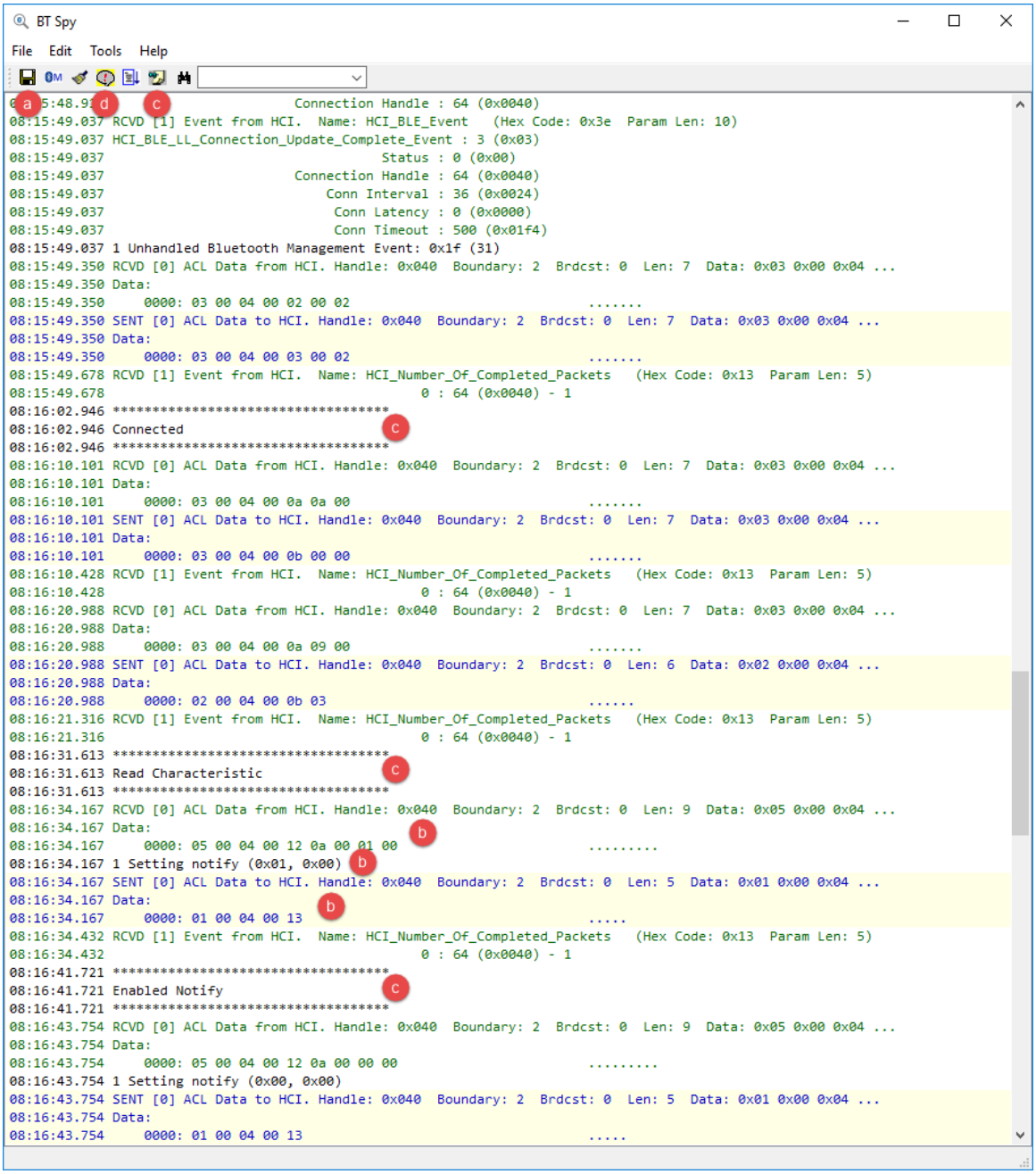
i. Typically, when using BTSpy to monitor traffic you will not also be using ClientControl to send WICED HCI commands to the chip. In that case, you can set the data_handler callback function in the transport configuration structure to NULL and you can leave out the callback function (e.g. hci_control_process_rx_command) entirely. The configuration structure would look like this:

```
/******************************************************************
 * Transport Configuration
 ******************************************************************/
#define TRANS_UART_BUFFER_SIZE  1024
#define TRANS_UART_BUFFER_COUNT 2

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 = TRANS_UART_BUFFER_SIZE,              /**<  Rx Buffer Size */
    .buffer_count = TRANS_UART_BUFFER_COUNT             /**<  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. */
};
```

d. Initialize Transport and Create Buffer Pools.

2. Create and register a function for Bluetooth protocol trace messages.
3. Build and download the application to the evaluation board.
4. Press **Reset** (SW2) on the WICED evaluation board to reset the HCI UART after downloading and before opening the port with *ClientControl*. This is necessary because the programmer and HCI UART use the same interface.
5. Run the *ClientControl* utility.
6. In the *ClientControl* utility, 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).
7. Run the BTSpy utility.
8. In the ClientControl utility, open the HCI UART COM port by clicking on "Open Port".
    a. Note: The HCI UART is the same port used for programming, so you must disconnect each time you want to re-program.
    b. 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.
9. View trace messages in the *BTSpy* window. See the figure below for an example of what the BTSpy output looks like.
    a. The trace output can be saved to a file.
    b. Lines in black are standard WICED_BT_TRACE messages, blue are messages sent over HCI and green are messages received over HCI.
    c. Items surrounded by ********** are notes added by the user.
    d. Items can be filtered.


BT Spy
File Edit Tools Help
a
d
c
Connection Handle : 64 (0x0040)
08:15:49.037 RCVD [1] Event from HCI. Name: HCI_BLE_Event (Hex Code: 0x3e Param Len: 10)
08:15:49.037 HCI_BLE_LL_Connection_Update_Complete_Event : 3 (0x03)
08:15:49.037 Status : 0 (0x00)
08:15:49.037 Connection Handle : 64 (0x0040)
08:15:49.037 Conn Interval : 36 (0x0024)
08:15:49.037 Conn Latency : 0 (0x0000)
08:15:49.037 Conn Timeout : 500 (0x01f4)
08:15:49.037 1 Unhandled Bluetooth Management Event: 0x1f (31)
08:15:49.350 RCVD [0] ACL Data from HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 7 Data: 0x03 0x00 0x04 ...
08:15:49.350 Data:
08:15:49.350 0000: 03 00 04 00 02 00 02 ....
08:15:49.350 SENT [0] ACL Data to HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 7 Data: 0x03 0x00 0x04 ...
08:15:49.350 Data:
08:15:49.350 0000: 03 00 04 00 03 00 02 ....
08:15:49.678 RCVD [1] Event from HCI. Name: HCI_Number_Of_Completed_Packets (Hex Code: 0x13 Param Len: 5)
08:15:49.678 0 : 64 (0x0040) - 1
08:16:02.946 **********
08:16:02.946 Connected
c
08:16:02.946 **********
08:16:10.101 RCVD [0] ACL Data from HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 7 Data: 0x03 0x00 0x04 ...
08:16:10.101 Data:
08:16:10.101 0000: 03 00 04 00 0a 0a 00 ....
08:16:10.101 SENT [0] ACL Data to HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 7 Data: 0x03 0x00 0x04 ...
08:16:10.101 Data:
08:16:10.101 0000: 03 00 04 00 0b 00 00 ....
08:16:10.428 RCVD [1] Event from HCI. Name: HCI_Number_Of_Completed_Packets (Hex Code: 0x13 Param Len: 5)
08:16:10.428 0 : 64 (0x0040) - 1
08:16:20.988 RCVD [0] ACL Data from HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 7 Data: 0x03 0x00 0x04 ...
08:16:20.988 Data:
08:16:20.988 0000: 03 00 04 00 0a 09 00 ....
08:16:20.988 SENT [0] ACL Data to HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 6 Data: 0x02 0x00 0x04 ...
08:16:20.988 Data:
08:16:20.988 0000: 02 00 04 00 0b 03 ......
08:16:21.316 RCVD [1] Event from HCI. Name: HCI_Number_Of_Completed_Packets (Hex Code: 0x13 Param Len: 5)
08:16:21.316 0 : 64 (0x0040) - 1
08:16:31.613 **********
08:16:31.613 Read Characteristic
c
08:16:31.613 **********
08:16:34.167 RCVD [0] ACL Data from HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 9 Data: 0x05 0x00 0x04 ...
08:16:34.167 Data:
b
08:16:34.167 0000: 05 00 04 00 12 0a 00 01 00 ....
08:16:34.167 1 Setting notify (0x01, 0x00)
b
08:16:34.167 SENT [0] ACL Data to HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 5 Data: 0x01 0x00 0x04 ...
08:16:34.167 Data:
b
08:16:34.167 0000: 01 00 04 00 13 .....
08:16:34.432 RCVD [1] Event from HCI. Name: HCI_Number_Of_Completed_Packets (Hex Code: 0x13 Param Len: 5)
08:16:34.432 0 : 64 (0x0040) - 1
08:16:41.721 **********
08:16:41.721 Enabled Notify
c
08:16:41.721 **********
08:16:43.754 RCVD [0] ACL Data from HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 9 Data: 0x05 0x00 0x04 ...
08:16:43.754 Data:
08:16:43.754 0000: 05 00 04 00 12 0a 00 00 00 ....
08:16:43.754 1 Setting notify (0x00, 0x00)
08:16:43.754 SENT [0] ACL Data to HCI. Handle: 0x040 Boundary: 2 Brdcst: 0 Len: 5 Data: 0x01 0x00 0x04 ...
08:16:43.754 Data:
08:16:43.754 0000: 01 00 04 00 13 .....

5.3 Debugging Via the ARM Debug Port

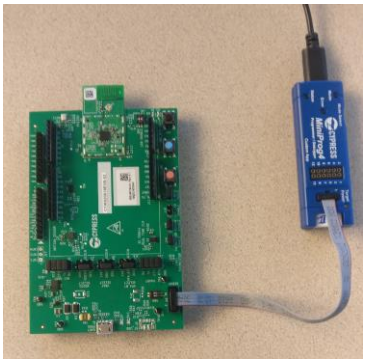
You will use this information in Exercise - 5.3.

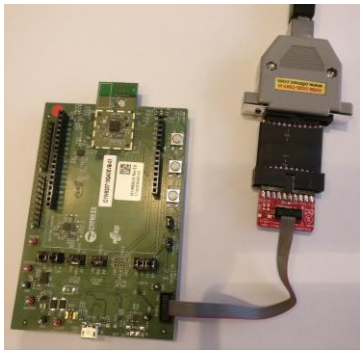
Hardware debugging on WICED Bluetooth devices can done using an OpenOCD supported probe. However, if your device has an active BLE connection it will drop the connection when the CPU is halted in the debugger (after the connection timeout). Therefore, for BLE applications it is more common to use BTSpy for debugging rather than a hardware debugger.

In this section we will describe two different sets of hardware: a Cypress MiniProg4, and a Segger J-Link debug probe. It is also possible to use an Olimex ARM-USB-TINY-H debug probe with an additional ARM-JTAG-SWD adapter but that is not supported out of the box. For the J-Link and Olimex, an adapter to convert the 20-pin connector to the small footprint 10-pin connector may be needed depending on the debug connector available on the kit. The table below lists the hardware required for each option:

Option 1	**Manufacturer**	**Part Number**	**Description**
MiniProg4	Cypress	CY8CKIT-005	Program and Debug Kit
Option 2	**Manufacturer**	**Part Number**	**Description**
Segger J-Link	Segger	8.08.00 J-Link Base	Debug probe
	Olimex	ARM-JTAG-20-10	20-10 pin adapter
Option 3	**Manufacturer**	**Part Number**	**Description**
Olimex	Olimex	ARM-USB-TINY-H	Debug probe
	Olimex	ARM-JTAG-SWD	SWD – JTAG adapter
	Olimex	ARM-JTAG-20-10	20-10 pin adapter

Pictures showing the debugging setup for each configuration are shown here:





The process to setup and run the debugger is described in detail in a guide titled "Hardware Debugging for CYW207xx and CYW208xx". This guide is available from the Eclipse, so that information is not repeated here. To access the guide from the Eclipse IDE, do the following:

> **Quick Panel > Documentation > WICED Bluetooth SDK Documentation > CYW208XX API Reference > WICED Hardware Debugging for Bluetooth Kits**

Alternately, this file can be found at:

> **Help > ModusToolbox General Documentation Index > ModusToolbox Documentation Index > Bluetooth Documentation > CYW208XX > WICED Hardware Debugging for Bluetooth Kits**

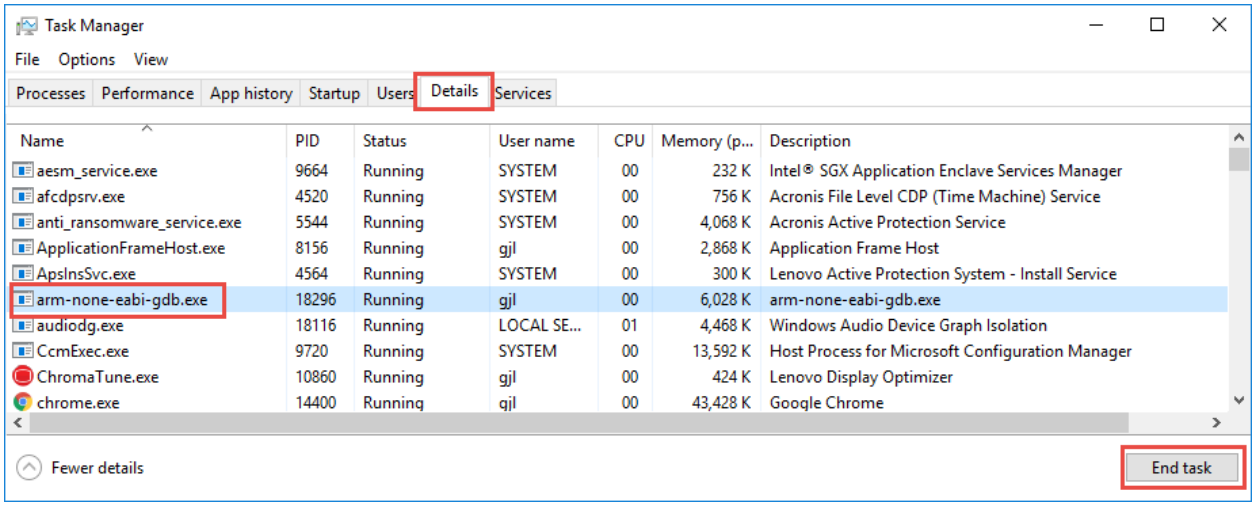
A few important notes:

1. Setting ENABLE_DEBUG to 1 in the makefile does three things located in the file at wiced_btsdk/dev-kit/baselib/20819A1/WICED/common/spar_utils.h:
    a. Configures the SWD pins.
        i. The CYW920819EVB-02 kit uses P2 for SWD_CLK and P3 for SWD_IO. If your hardware uses different pins than the ones specified in this file you will need to modify it.
    b. Disables the watchdog timer.
    c. Enables a busy-wait loop so that the application doesn't start up until after the user attaches to the debug port and changes a variable to continue. This allows the user to debug right from the application start point.
        i. Because the device is in a busy-wait loop at power up, the board will not show any activity after programming or reset until the user breaks out of the loop from inside the debugger.
2. There is a DIP switch (SW9) that controls routing of P2 and P3 to the debug header. The switches must be in the OFF position to enable the debug header connections.
    a. P2 and P3 route to Arduino header pins when the switches are in the ON position.
3. Since the debugger uses P2 (SWD_CLK) and P3 (SWD_IO) for SWD, those pins may not be used for anything else in your application.
4. The debugger attaches to a running target. Therefore, you must do a build/program step first before running the appropriate Debug Attach launch.
5. If you have trouble re-programming after using debugging, try the following:
    - Hold the "Recover" button on the kit.
    - Press and release the "Reset" button.
    - Release the "Recover" button.
    - Attempt to program again.

6. The Olimex debug probe works the same as MiniProg4 except for 1 additional step to change the default adapter setting in the file wiced_btsdk/dev-kit/baselib/20819A1/platforms/CYW20819A1_openocd.cfg. The change requited is to uncomment 2 lines for the Olimex adapter config and comment out 1 line for the KitProg3 adapter config (in this case, KitProg3 and MiniProg 4 are equivalent). The applicable section of the file is shown here:

```
…
# adapter config
# Olimex arm-usb-tiny-h
#source [find interface/ftdi/olimex-arm-usb-tiny-h.cfg]
#source [find interface/ftdi/olimex-arm-jtag-swd.cfg]
# Cypress MiniProg4
source [find interface/kitprog3.cfg]
…
```

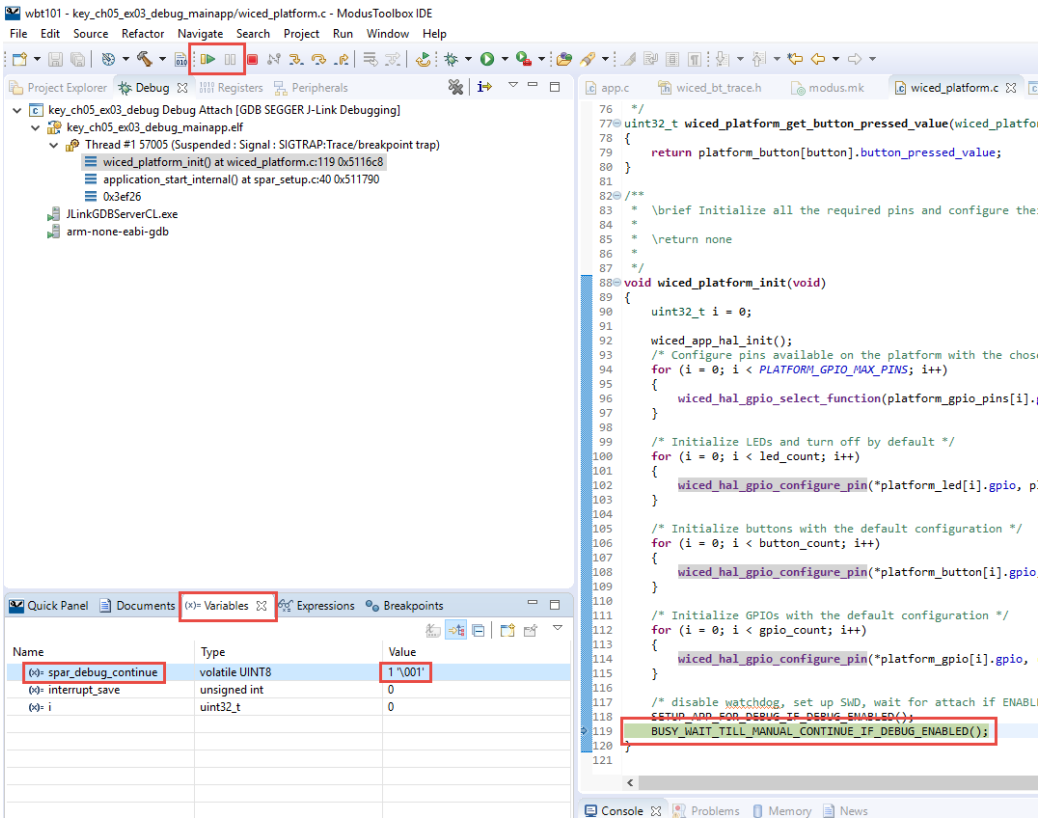
7. If you get an error message when launching the debugger about another instance already running, open the task manager (Ctrl-Alt-Del) and end any tasks named arm-none-eabi-gdb.exe.



5.3.1 Using the Debugger

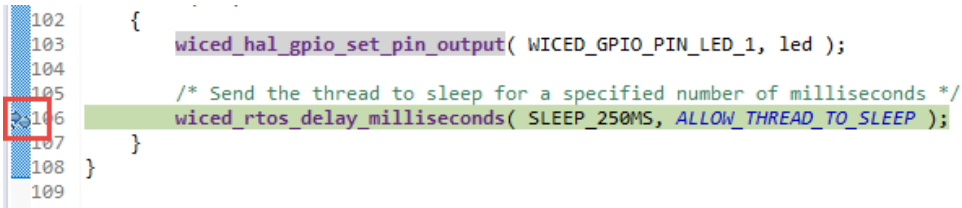
You will use this information in Exercise - 5.3.

Once you have configured the firmware, programmed the board, and started the debugger, the application will sit in the `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED` loop. Start execution (if it isn't already running) and then pause to make sure the firmware is inside the loop. In the "Variables" tab in the Quick Panel, erase the entire value for *spar_debug_contine*, enter a new value of 1 and press Enter. Once you have changed the value of *spar_debug_continue*, you can resume execution and the program will go beyond the busy wait loop.

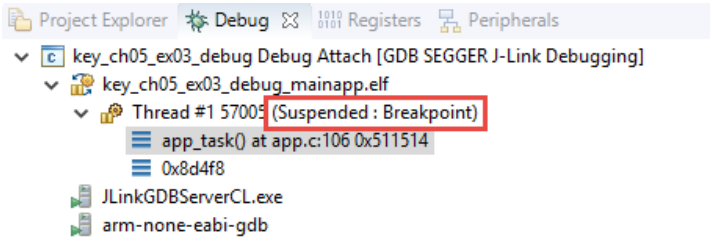


When program execution is paused, you can add breakpoints to halt at specific points in the code. To add a breakpoint, open the source file, click on the line where you want a breakpoint and double-click in the bar to the left of the code window or right-click in that area and select "Toggle Breakpoint". An enabled breakpoint is a blue circle with a black checkmark next to it.

Click the "Resume" button (shown in the figure above) to resume execution. The program will halt once it reaches the breakpoint.

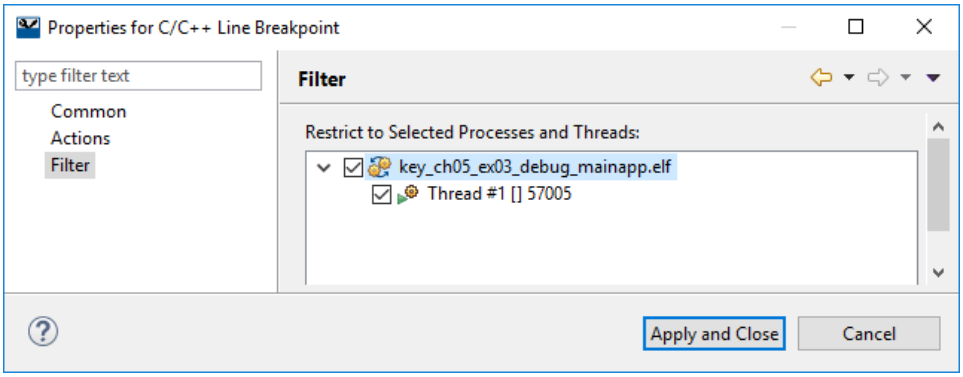


Once a thread suspends due to a breakpoint you will see that line of code highlighted in green as shown above and you will see that the thread is suspended due to the breakpoint in the debug window as shown below.

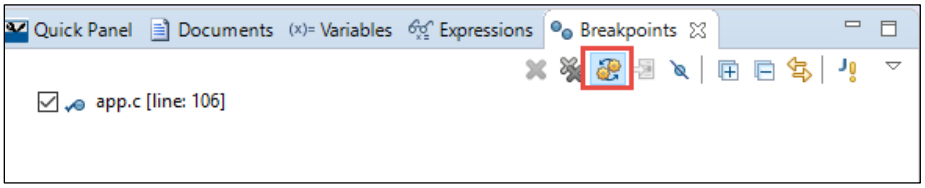


You can enable or disable breakpoints by double clicking on the green circle next to the line in the source code or from the "Breakpoints" tab in the Quick Panel.

If breakpoints are created prior to starting the current debug session, they may not be associated with the current thread and will be indicated with a blue circle without a check mark. To enable the breakpoints in the current thread, right-click the desired breakpoint and select "Breakpoint Properties…" Click on "Filter" and make sure the boxes are checked as shown below.

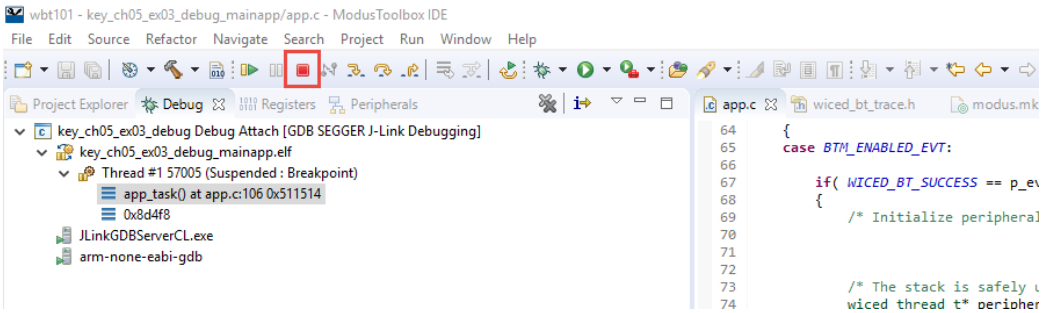


If you do not see any breakpoints in the Breakpoints window, turn off the "Show Breakpoints Supported by Selected Target" button as shown below.



Click the red "Terminate" button to stop debugging. Once debugging stops, you will likely want to switch back to the Project Explorer tab instead of the Debug tab. Alternately, you can use the menu item Window >

Perspective > Reset Perspective to reset the ModusToolbox perspective to the default window sizes and placement.



A few additional notes on debugging:

1. Single stepping through code requires a free breakpoint.
2. There are two hardware breakpoints available.
3. Only hardware breakpoints can be used for ROM opcodes.
4. The device uses both ROM and RAM for firmware. For example, many WICED API functions will call into ROM code for support.
5. Source code and symbols are not provided for ROM and some patch library areas. Source code debugging will be limited to the code for which you have sources.
6. Typically, the step command is ignored by GDB if a breakpoint is not available.

5.4 Exercises

Exercise - 5.1 Run BTSpy

In this application you will use BTSpy to look at Bluetooth protocol trace messages. This is covered in section 5.2.4

Application Creation

1. Create a new application called **ch05_ex01_btspy** using the template in templates/ch05_ex01_btspy.
2. The BT_DEVICE_ADDRESS is already set to random in the makefile.
3. Open the Bluetooth Configurator.
    a. Change the device name to <inits>_btspy.
    b. Save edits and close the configurator.
4. Review the steps in the Transport Configuration section of this chapter and add in the necessary code to support WICED HCI.
    a. Hint: Add includes for the two header files, create a variable for transport_pool, create/setup the transport_cfg structure, and in application_start, initialize transport and create buffer pools.
    b. Hint: You can specify NULL for the receive data handler since we won't deal with any incoming HCI commands in our application. The function hci_control_process_rx_cmd can be eliminated completely since you will not be responding to HCI commands from Client Control in this exercise.
5. Create a BTSpy callback function and register it in the BTM_ENABLED_EVT event.
6. Route the debug messages to WICED_ROUTE_DEBUG_TO_WICED_UART instead of WICED_ROUTE_DEBUG_TO_PUART. This will mean that Bluetooth trace messages and WICED_BT_TRACE messages will go to the HCI UART using WICED HCI formatting.
7. In the app_bt_cfg.c file, change the low duty cycle advertisement duration to 0 so that advertising doesn't stop after 60 seconds.

Programming and Setup

1. Build and download the application to the kit.
2. Press **Reset** (SW2) on the WICED evaluation board to reset the HCI UART after downloading and before opening the port with *ClientControl*. This is necessary because the programmer and HCI UART use the same interface.
3. Run the *ClientControl* utility from the Quick Panel. Select the HCI UART port and set the baud rate to the baud rate used by the application for the HCI UART (the default baud rate is 3000000). Do NOT open the port yet.
4. Run the *BTSpy* utility from the Quick Panel. If asked, click "Allow".
    a. Hint: If your computer has a Broadcom WiFi chip, it will show lots of messages that are from your computer instead of from your kit. To disable these messages, you will need to temporarily disable Bluetooth on your computer.
5. In the ClientControl utility, open the HCI UART COM port by clicking on "Open Port".

Testing

1. If you want to capture the log from BTSpy to a file, click on the "Save" button (it looks like a floppy disk). Click Browse to specify a path and file name, click on "Start Logging", and then click on "OK".
    a. Hint: If you want to include the existing log window history in the file (for example if you forgot to start logging at the beginning) check the box "Prepend trace window contents" before you start logging.
2. Use CySmart to connect to the device and observe the messages in the BTSpy window.
3. Once paring is complete, click the "Notes" button (it looks like a Post-It note), enter "Pairing Completed" and click OK. Observe that a note is added to the trace window.
4. Read the Button Characteristic and observe the messages.
5. Add a note that says " Button Characteristic Read Complete"
6. Disconnect from the device.
7. Once you are done logging, click on the "Save" button, click on "Stop Logging", and then click on "OK".

Exercise - 5.2 (Advanced) Use the Client Control Utility to Send HCI Commands

Introduction

In this application, you will add the ability to start and stop advertising using WICED HCI messages from the Client Control utility. This is covered in section 5.2.3

Application Creation

1. Create a new application called **ch05_ex02_hci** using the template in templates/ch05_ex02_hci.
2. The BT_DEVICE_ADDRESS is already set to random in the makefile.
3. Open the Bluetooth Configurator.
    a. Change the device name to <inits>_hci.
    b. Save edits and close the configurator.
4. Review the steps in the Transport Configuration section of this chapter and add in the necessary code to support WICED HCI.
5. Create an RX handler and add a case to the opcode switch statement to handle the *_ADVERTISE command.
    a. Hint: Refer to the file include/common/hci_control_api.h to find the full name of the opcode for the *_ADVERTISE command.
    b. Hint: The payload will be 0 to stop advertisements and 1 to start advertisements.
    c. Hint: Use the function wiced_bt_start_advertisements with either BTM_BLE_ADVERT_OFF or BTM_BLE_ADVERT_NONCONN_HIGH depending on whether the first byte of the payload is 0 or 1.
6. (Advanced): Update the Bluetooth Stack Management callback to send a WICED HCI status message when the advertising state changes.
    a. Hint: Use wiced_transport_send_data with the same ADVERTISE code used above. Note that you must send a value of either 0 (stopped) or 1 (started) even though the callback will send other values depending on the type of advertising (e.g. 3 for high duty connectable). If you send a value other than 0 or 1, the Client Control utility will crash.

Testing

1. Program the application to the kit.
    a. Hint: If you have the port open in Client Control, you will have to close the port before being able to program again because programming and Client Control both use the WICED HCI port.
2. Open the Client Control program from the Quick Panel and connect to the WICED HCI port.
3. Open CySmart and scan for devices. Note that your device does not appear.
4. In Client Control, switch to the GATT tab and click on Start Adverts. Look at the return message if you implemented the status message.
5. Verify that your device now appears in CySmart.
6. In Client Control click on Stop Adverts and look at the return message. Stop and Re-start the scan in CySmart. Note that your device no longer appears.

Exercise - 5.3 (Advanced) Run the Debugger

This is covered in section 5.3

In this exercise you will setup and run the debugger using a MiniProg4 debug probe. If you want to use a J-Link or Olimex, follow the instructions in the Hardware Debugging guide for additional required steps. You will then use the debugger to change the value of a variable that controls an LED on the shield.

1. Read the "Hardware Debugging for CYW207xx and CYW208xx" guide.
2. Create a new application called **ch05_ex03_debug** using the template in templates/ch05_ex03_debug.
3. Open the makefile and set the value for ENABLE_DEBUG to 1.
4. Verify that the two switches in SW9 are in the OFF position on your kit.
    a. Hint: SW9 is the 2 position DIP switch near the 10-pin debug header. The 2 switches should be pushed to the side next to the debug header.
5. If you have not already, connect your kit and debugger to each other and then to USB ports on your computer.
6. Program the application to the kit (<appname> Program).
7. Launch the debugger (<appname> Attach_KitProg3).
    a. Hint: KitProg3 = MiniProg4 for the purpose of debugging.
8. Place a break point at the wiced_rtos_delay_milliseconds call in the app_task thread in app.c.
9. Pause execution. You should be in the BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED() loop.
10. Set the value of *spar_debug_contine* to 1 to get out of the initial loop.
    a. Hint: Erase the entire value and then enter a new value of 1.
    b. Hint: The Variables tab is next to the Quick Panel tab in the lower left window.
11. Execute a few more times and notice that execution suspends at the breakpoint you added.
    a. Hint: It may take a long time to get to the breakpoint the first few times through the loop. Be patient.
12. Observe that the LED does not turn on because the variable "led" never changes.
13. Change the value of the variable "led" and then re-start execution.
    a. Hint: The LED is active low.
14. Note that the LED now turns on.
15. Stop the debugger by pressing the stop button.
16. Use the menu item Window > Perspective > Reset Perspective to get back the usual tabs. You can also switch tabs manually back to Project Explorer (upper left) and Quick Panel (lower left).