

Chapter 5: Using the Debugger

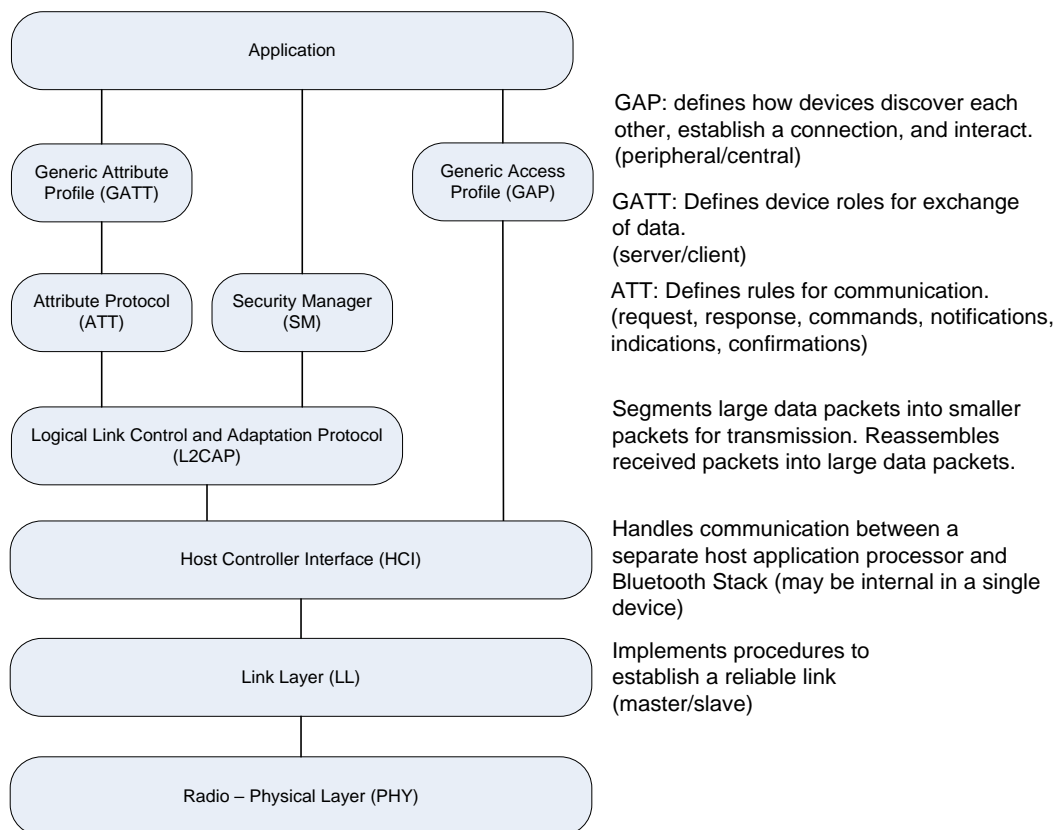
Time: 1 ½ Hours

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.....	2
5.1.1	HCI.....	2
5.2	UART DEBUGGING	4
5.2.1	ARCHITECTURE.....	4
5.2.2	DEBUGGING TRACES	5
5.2.3	WICED HCI & THE CLIENT CONTROL UTILITY	6
5.2.4	USING BTSPY & THE CLIENT CONTROL TO VIEW HCI COMMANDS	10
5.3	DEBUGGING VIA THE ARM DEBUG PORT	12
5.3.1	PROJECT CONFIGURATION	13
5.3.2	HOST DEBUG ENVIRONMENT SETUP FOR SEGGER J-LINK	17
5.3.3	HOST DEBUG ENVIRONMENT SETUP FOR OLIMEX ARM-USB-TINY-H	23
5.3.4	USING THE DEBUGGER	33
5.4	EXERCISES.....	37
	EXERCISE - 5.1 USE THE CLIENT CONTROL UTILITY	37
	EXERCISE - 5.2 RUN BTSPY	39
	EXERCISE - 5.3 (ADVANCED) RUN THE DEBUGGER	40

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

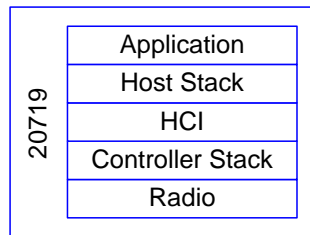
The next block to talk about is the "Host Controller Interface".

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.

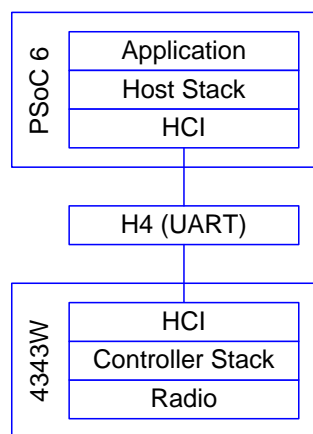
The interface between the Host and the Controller was typically UART or SPI. The data flying over that serial connection was formatted in Bluetooth SIG specific packets called "HCI Packets".

By standardizing the HCI interface, it allowed big application processors (like those existing in PCs and cellphones) to interface with Bluetooth. As time went by the Host and Controller have frequently merged into one chip (e.g. CYW20719), 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.

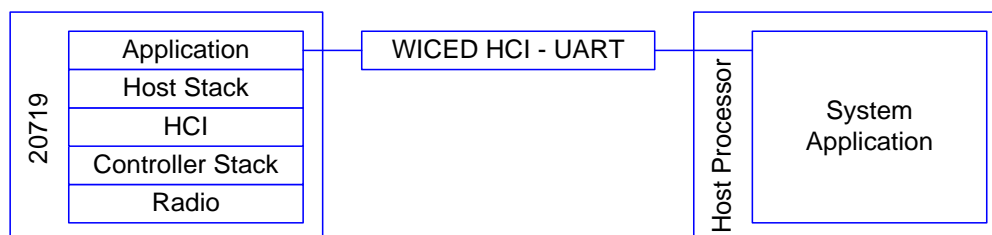
Note that even on single chip Host/Controller solutions, it is still be possible to use the chip in Controller mode with an HCI interface. For example, when the CYW20719 is used in Linux applications, it is booted in Controller mode and the Stack runs in Linux. Likewise, when the chip is put into recovery 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".



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 CYW920179 this UART is attached via a USB-UART bridge to your PC.

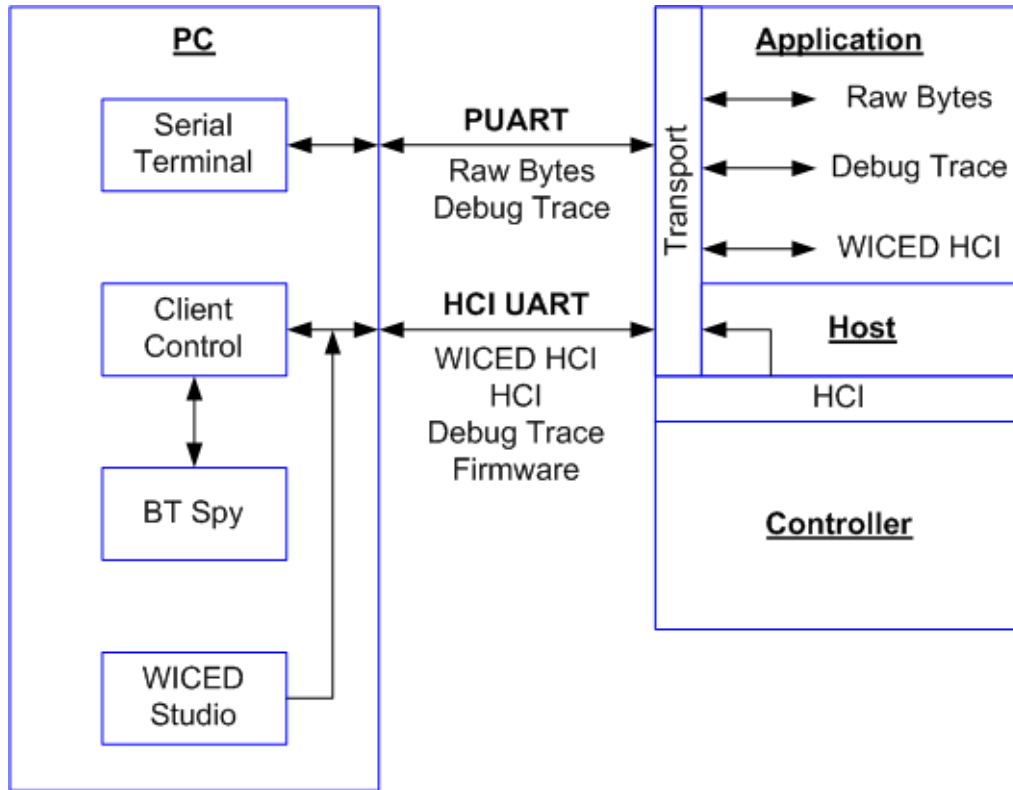
HCI UART

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

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
- A 1-byte Control Group (a.k.a. Group Code)
- 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 the WICED SDK under `include/common`. 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 = 01 00 00 00.

Transport Configuration

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 three things:

1. Create a structure of type `wiced_transport_cfg_t` which contains the HCI UART configuration, 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).
2. Call `wiced_transport_init` (with a pointer to your configuration structure).
3. Call `wiced_transport_create_buffer_pool` to create buffers for the transport system to use.

Conveniently, WICED Bluetooth Designer sets up all this code for you as shown below. Note that there is no TX complete callback function created by default.

Transport Configuration Structure

The transport configuration structure default setup from WICED Bluetooth Designer is shown below. The default baud is 3,000,000.

```

/*****
 * Transport Configuration
 *****/
wiced_transport_cfg_t transport_cfg =
{
    WICED_TRANSPORT_UART,          /**< Wiced transport type. */
    {
        WICED_TRANSPORT_UART_HCI_MODE, /**< UART mode, HCI or Raw */
        HCI_UART_DEFAULT_BAUD        /**< UART baud rate */
    },
    {
        TRANS_UART_BUFFER_SIZE,      /**< Rx Buffer Size */
        TRANS_UART_BUFFER_COUNT      /**< Rx Buffer Count */
    },
    NULL,                            /**< Wiced transport status handler.*/
    hci_control_process_rx_cmd,       /**< Wiced transport receive data handler. */
    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, and that is what WICED Bluetooth designer does.

```

/* 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 actual 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 RX handler that is created by Bluetooth Designer 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`.

There are currently two bugs in the RX handler generated by WICED Bluetooth designer that must be fixed for the handler to work properly:

1. The opcode must be declared as a `uint16_t`
2. The payload_length must be declared as a `uint16_t`

```

/* 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;
    uint8_t opcode = 0;
    uint8_t* p_payload_data = NULL;
    uint8_t payload_length = 0;

    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 Length from little-endian byte array
        opcode = LITTLE_ENDIAN_BYTE_ARRAY_TO_UINT16(p_data);
        payload_length = LITTLE_ENDIAN_BYTE_ARRAY_TO_UINT16(&p_data[sizeof(uint16_t)]);
        p_payload_data = &p_data[sizeof(uint16_t)*2];

        // TODO : Process received HCI Command based on its Control Group
        // (see 'hci_control_api.h' for additional details)
        switch ( HCI_CONTROL_GROUP(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,...);
```

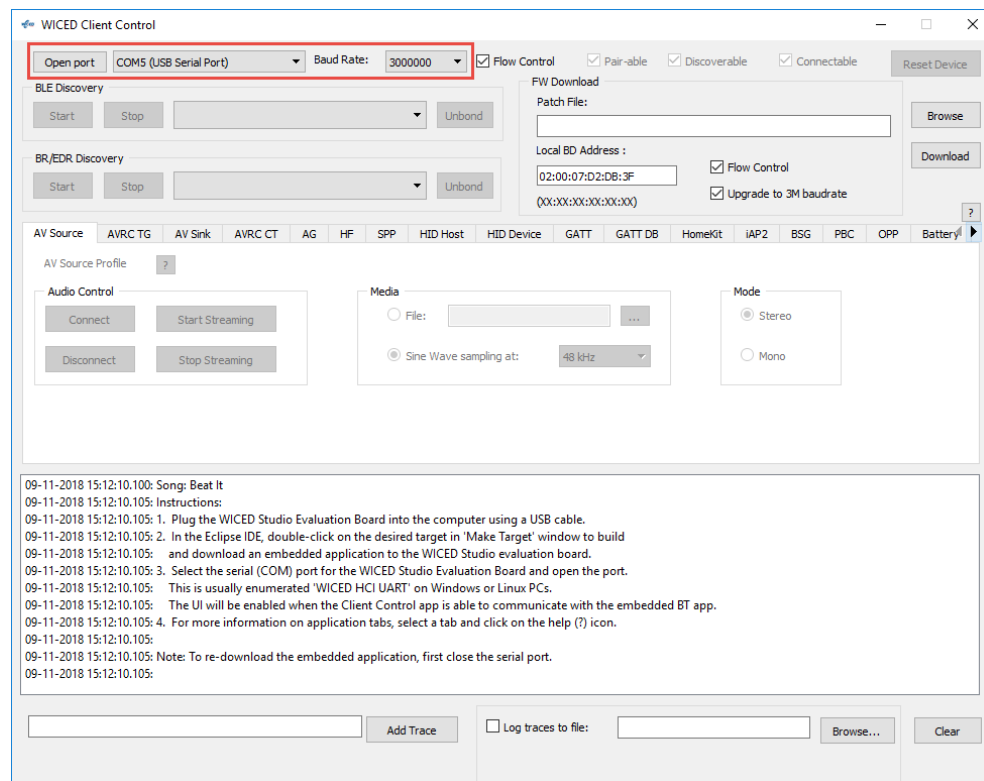
```
wiced_transport_send_data(HCI_CONTROL_EVENT_WICED_TRACE, &string, strlen(string));
```

For example, if you call `WICED_BT_TRACE("abc");` it will send `19 02 01 03 00 41 42 43`.

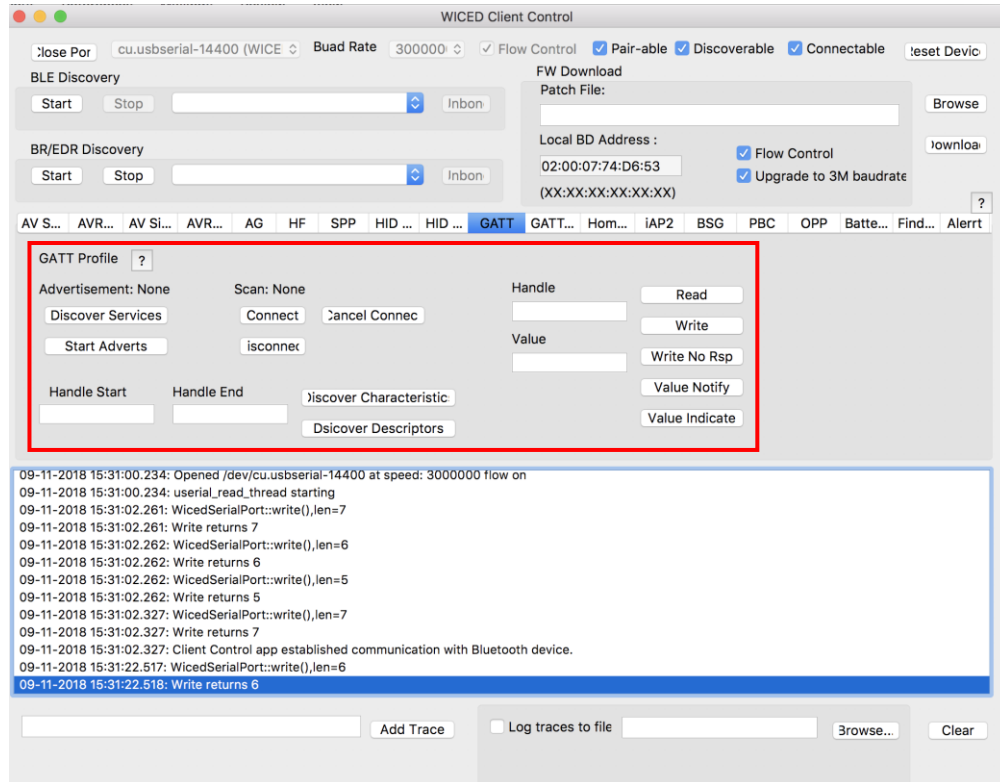
Client Control Utility (Part 1)

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 found inside the WICED Studio Project Explorer under `apps/host/client_control`. The source code is provided as well as pre-compiled executables for Windows. If you go to the `apps/host/client_control/Windows` folder, you can just double-click on `ClientControl.exe` to launch it.

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.



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

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 CYW20719.

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 CYW20719 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.

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. The callback function implementation should call the `wiced_transport_send_hci_trace()` API with the data received in the callback. This functionality is included by WICED BT Designer for you as long as the `HCI_TRACE_OVER_TRANSPORT` compile flag is set in the makefile.

Project Configuration

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

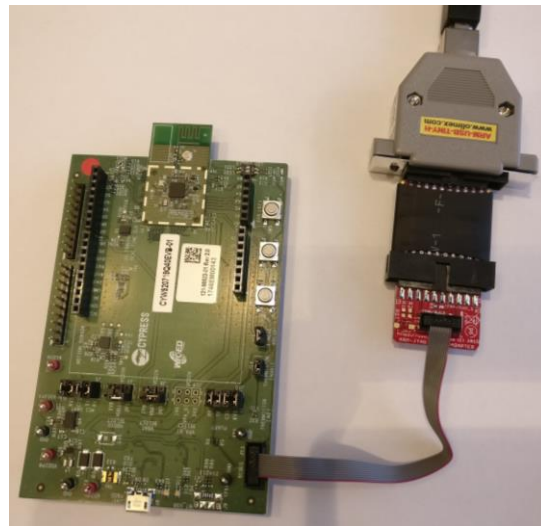
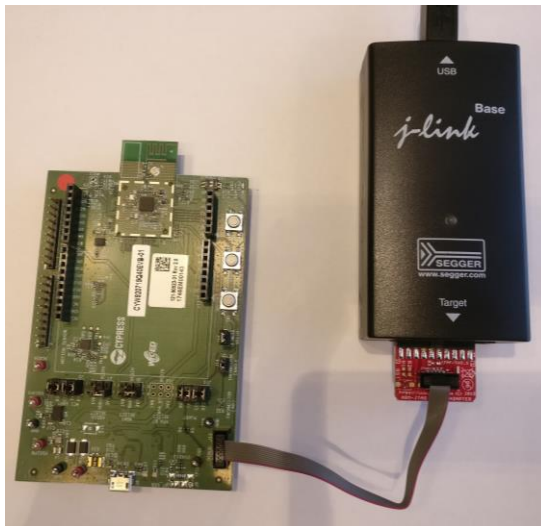
1. Setup the Transport for HCI just like in the previous section. If you created the project using WICED Bluetooth Designer, this is all done for you.
2. Build and download the application to the evaluation board.
3. Press **Reset** (SW2) on the WICED evaluation board to reset the HCI UART after downloading and before opening the port with *ClientControl*. This is necessary because the programmer and HCI UART use the same interface.
4. Run the *ClientControl* utility from `apps\host\client_control\Windows`. To open the utility from inside WICED Studio, double-click on it or right-click and select Open With > System Editor.
5. 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).
6. Run the *BTSpy* utility from `wiced_tools\BTSpy\Win32`. To open the utility from inside WICED Studio, double-click on it or right-click and select Open With > System Editor.
7. 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.
8. View trace messages in the *BTSpy* window.
 - a. Lines in black are standard WICED_BT_TRACE messages, blue are messages sent over HCI and green are messages received over HCI.

5.3 Debugging Via the ARM Debug Port

Debugging on WICED Bluetooth devices is done using an OpenOCD supported JTAG probe. In this chapter we will describe two different sets of hardware: a Segger J-Link debug probe and an Olimex ARM-USB-TINY-H debug probe with an additional ARM-JTAG-SWD adapter. In both cases, 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 tables below list the hardware required for each option:

Option 1	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 2	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:



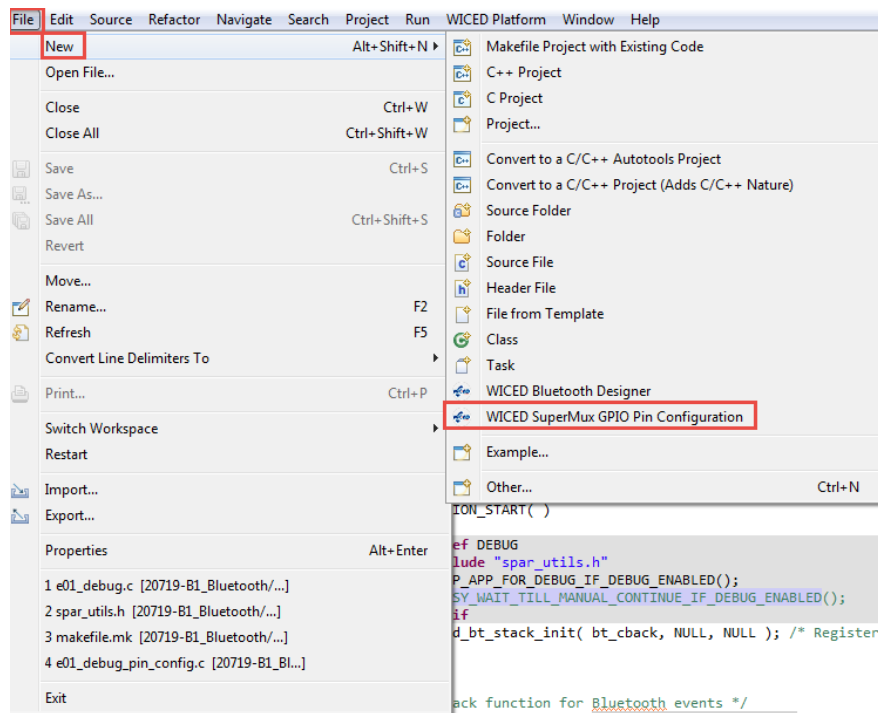
5.3.1 Project Configuration

To setup JTAG debugging, the project's pin configuration, makefile.mk, main C source file, and the Make Target must be updated. The changes are as follows:

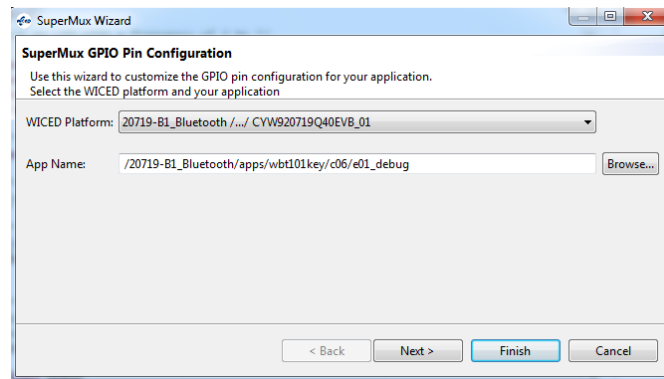
Edits to Pin Configuration

The pins used for SWD_CLK and SWD_IO must be configured for the debugger to operate. Note that these pins are dependent on the hardware being used. The pin configuration shown below is correct for the CYW920719Q40EVB-01 kit. If you are using different hardware, check the schematic to see which device pins are connected to the debug header.

The easiest way to change the pin configuration is to use the *WICED SuperMux GPIO Pin Configuration* tool. Start by clicking on the folder for the project for which you want to change the pin configuration. Then select *File -> New -> WICED SuperMux GPIO Pin Configuration*.



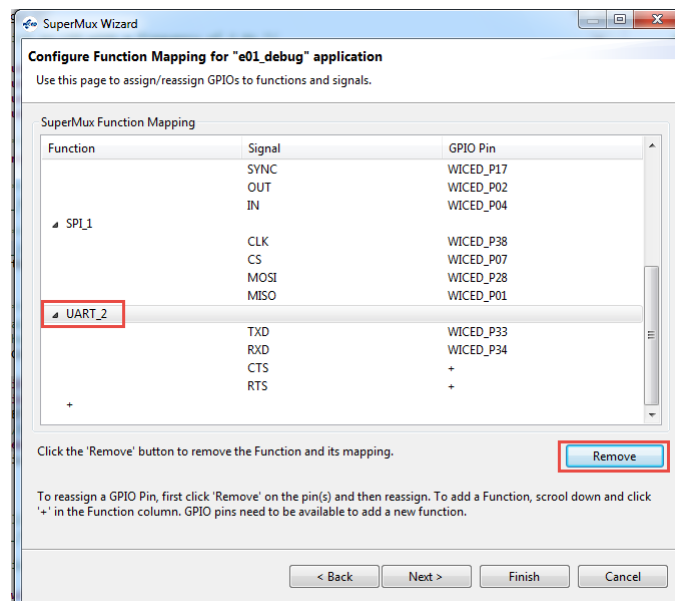
The first pin configuration window will look like this:



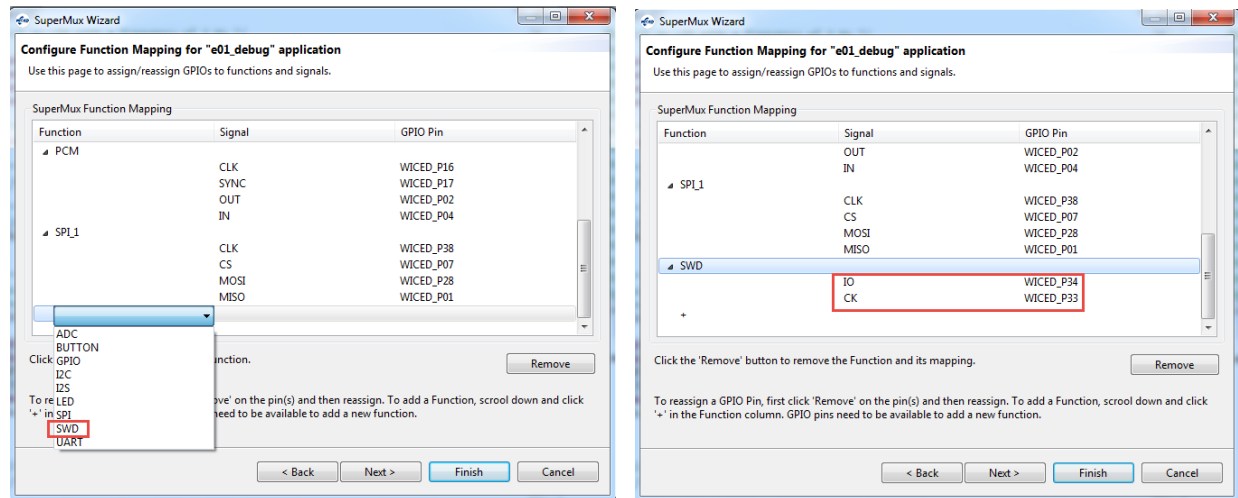
Select the WICED Platform you are using (e.g. WBT101_2_CYW920719Q40EVB_01) and that application name is correct and then click "Next". If you don't select the correct platform name you will have to cancel and start over. Leave the selected pins as-is and click on "Next" again.

From the function mapping page, scroll to the bottom, select UART_2, and click "Remove". This is necessary because the pins used for SWD (WICED_P33 and WICED_P34) are also used by the PUART. Therefore, the PUART pins need to be SWD pins to allow debugging on this kit.

(Note: You should also remove the shunts for Rx and Tx from J10 on the board. This disconnects the PUART Rx and Tx pins from the USB-UART bridge on the board.)



Next, click on the "+" sign at the bottom of the list and select SWD from the drop-down menu. Assign the SWD IO to WICED_P34 and the SWD CK to WICED_P33. Once you are done, click "Finish" (we don't need to change any GPIO pin control settings).



You will now have a <project_name>_pin_config.c file in your project folder with the pin configuration that was just created along with a <project_name>_pin_config.wsm that will be used if you re-run the wizard so that it starts with the previous configuration (backup copies of existing files will also be made). The makefile.mk is also automatically updated to include the new pin configuration file in the project.

Edits to makefile.mk

The following lines must be added to the project's makefile.mk in the C flags section.

```
ifdef DEBUG
C_FLAGS += -DDEBUG
endif
```

You may also want to move the two lines at the end related to the custom pin configuration file to be inside the ifdef DEBUG / endif construct so that the pins are only reconfigured when the make target specifies that you are debugging. For example, it might look like this:

```
ifdef DEBUG
C_FLAGS += -DDEBUG
C_FLAGS += -DSMUX_CHIP=$(CHIP)
APP_SRC += ex03_debug_pin_config.c
endif
```

Edits to the Project Source Code

In the main source file, you must include three additional header files and then call two macros as shown here:

```
#include "spar_utils.h"
#include "wiced_hal_wdog.h"
#include "tx_port.h"

#ifdef DEBUG
SETUP_APP_FOR_DEBUG_IF_DEBUG_ENABLED();
BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED();
#endif
```

These includes will typically go at the top of the file and the macro calls will typically go right at the beginning of APPLICATION_START that that execution is halted before you application begins running.

The macro BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED will cause the project to sit in a loop so the rest of the project will not execute until you run the debugger. You will have to manually set a value for the project to continue – this will be covered in detail in the section on using the debugger. If you want the project to continue prior to the debugger starting, you can remove the BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLE macro line, but you will enter debugging at an unknown point.

Note: On the CYW920719Q40EVB-01 kit, the SWD_CLK and SWD_IO pins are shared with the PUART so the PUART cannot be used while debugging. Therefore, you must disable any PUART functionality in the project when using SWD debugging. You can use #ifndef DEBUG / #endif construct to automatically turn PUART functionality on/off depending on the make target settings.

Edits to Make Target

The make target needs to be updated to set DEBUG=1 so that the lines added in makefile.mk are enabled. For example, to build a project called wbt101.ch05.ex03_debug for the CYW920719Q40EVB-01 and PSoC Analog Front End Shield combination , the Make Target would be:

```
wbt101.ch05.ex03_debug-WBT101_2_CYW920719Q40EVB-01 DEBUG=1 download
```

Programming the Project

Once the above edits are complete, execute the make target to build the project and program it to the board. The BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED macro will cause execution to wait in a loop until the JTAG debugger resumes execution. Therefore, the board will not show any activity until execution is resumed by the user from inside the debugger.

Note: If you have trouble re-programming after using debugging, try the following:

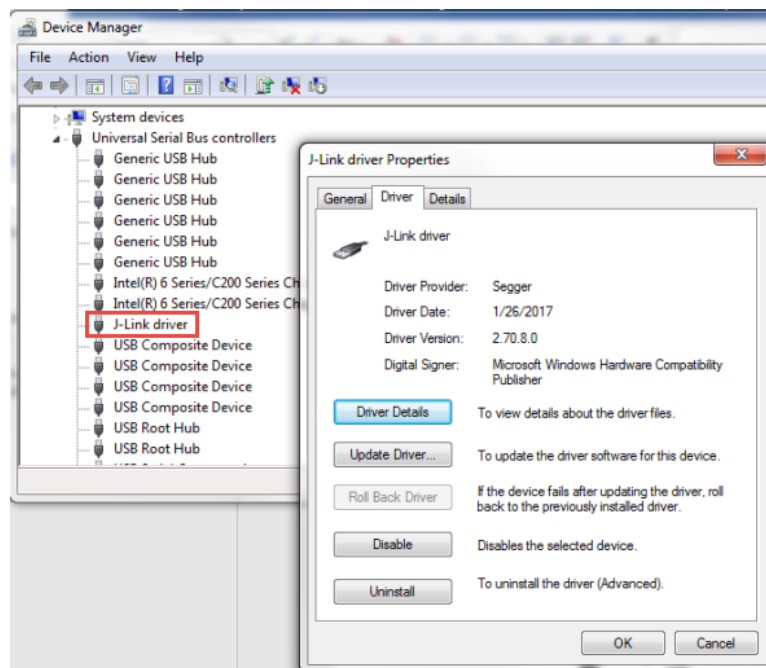
1. Hold the "Recover" button on the kit.
2. Press and release the "Reset" button.
3. Release the "Recover" button.
4. Attempt to program again.

5.3.2 Host Debug Environment Setup for Segger J-Link

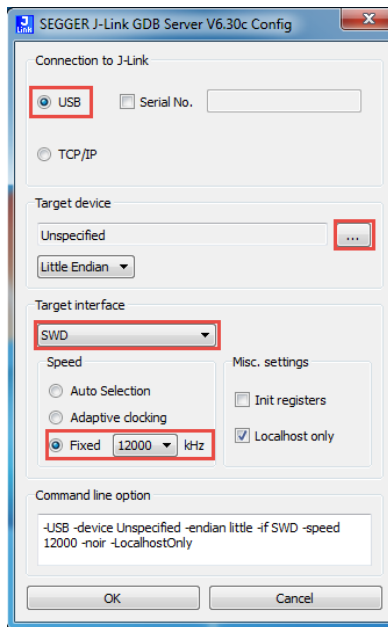
Software and Hardware Setup for J-Link

Install the Segger J-Link GDB Server. The software can be downloaded from <https://www.segger.com/downloads/jlink>. Look for "J-Link Software and Documentation Pack".

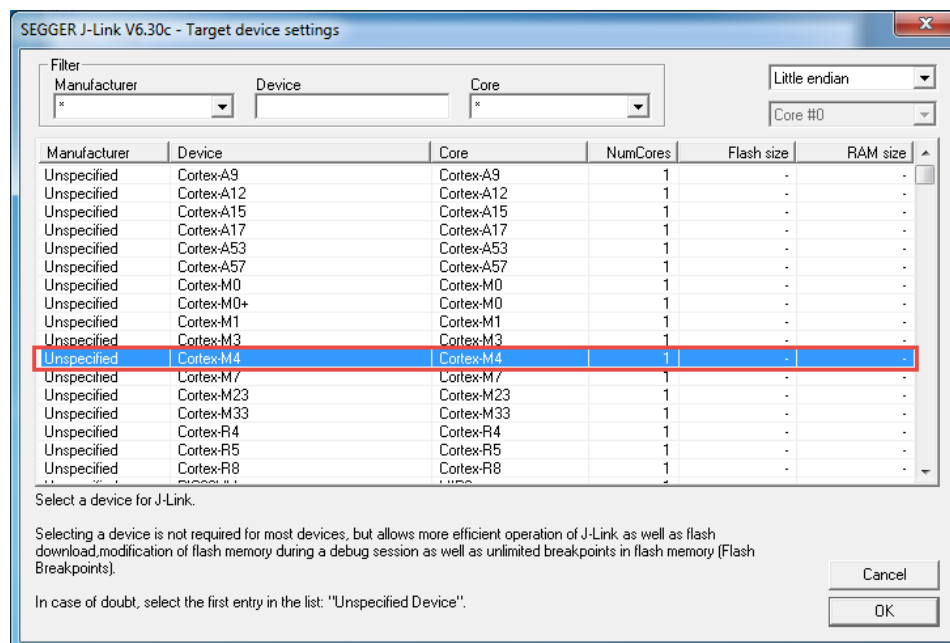
Once the software is installed, connect the J-Link to the kit's debug connector and then plug the J-Link into a USB port on the computer. When USB enumeration completes, the J-Link device should appear in the Device Manager as shown here:



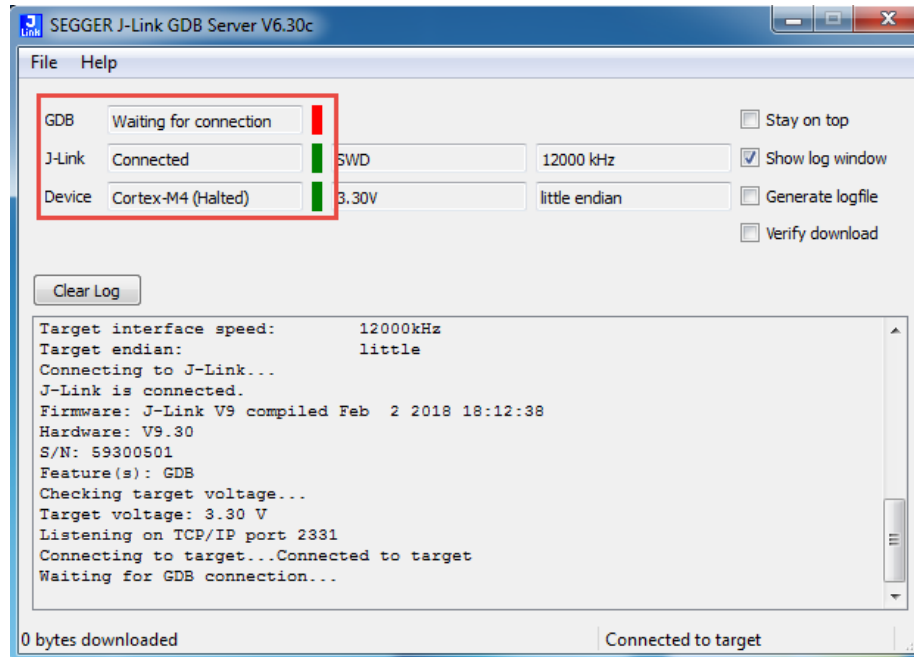
Run the Segger J-Link GDB Server program and select the USB and SWD interfaces. Set the speed to 12000 kHz.



Click the three dots next to Target device and select "Cortex-M4" from the list.



Click "OK" to close the Target Device Settings window and then click "OK" again to close the SEGGER J-Link GDB Server Config window. You may be notified of a firmware update for the connected emulator. If so, select "Yes" to update the firmware. If everything is set up properly and the WICED application has run and configured the SWD interface, then the GDB server will show that it is connected to the J-Link and the device and that it is "Waiting for GDB connection..." as shown here:

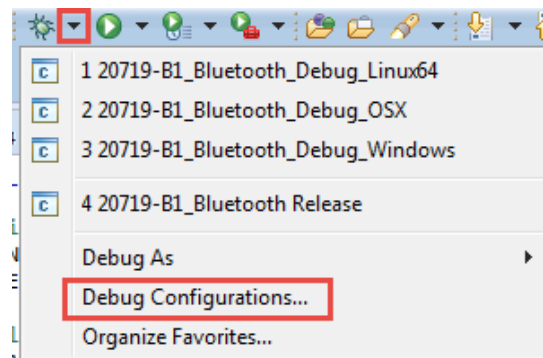


If the window does not appear as shown above or if the window closes after a few seconds, double check that the WICED kit has the correct firmware loaded and that everything is connected properly.

Once you have verified that the GDB server started properly, you can close it. WICED Studio will open the sever in command line mode when debugging starts.

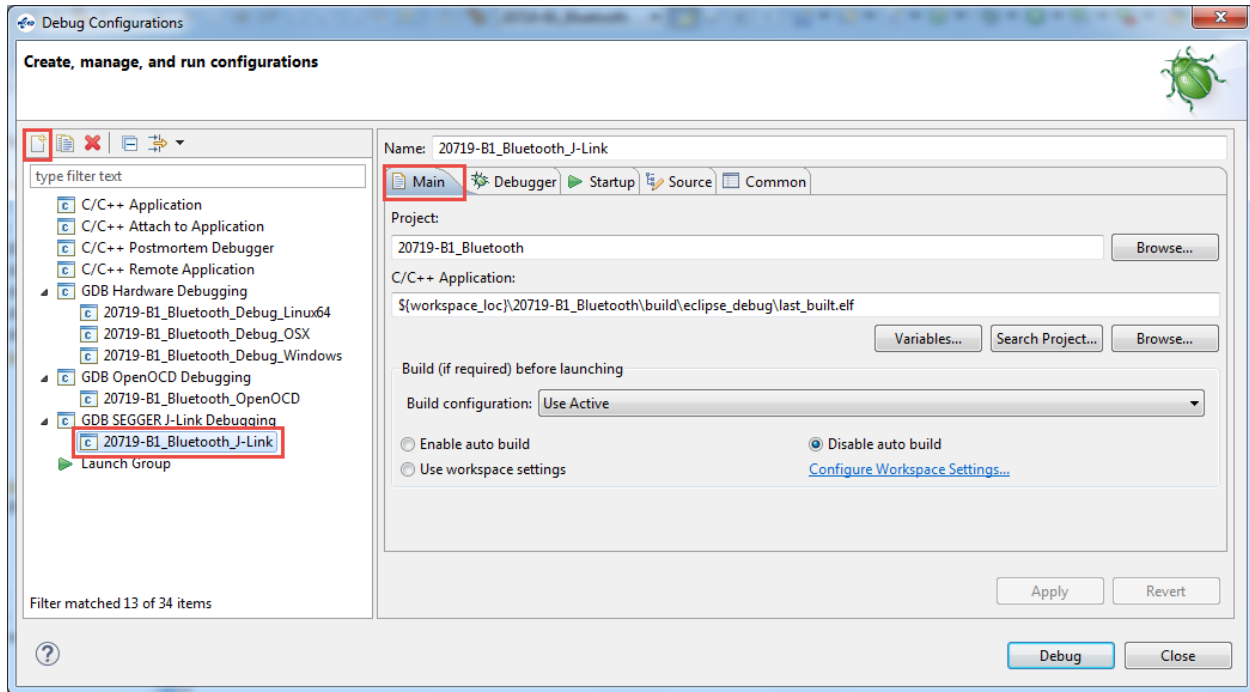
WICED Studio Setup for J-Link

From WICED Studio, click the arrow next to the Bug icon and select "Debug Configurations...".



First look for a debug configuration under *GDB SEGGER J-Link Debugging*. If there is a configuration, select it. If there isn't, select *GDB SEGGER J-Link Debugging* and click the "New" button. You can name the configuration anything you want. In the pictures below, the name is "20719-B1_Bluetooth_J-Link".

Once you have the configuration selected, set up the tabs as shown below:

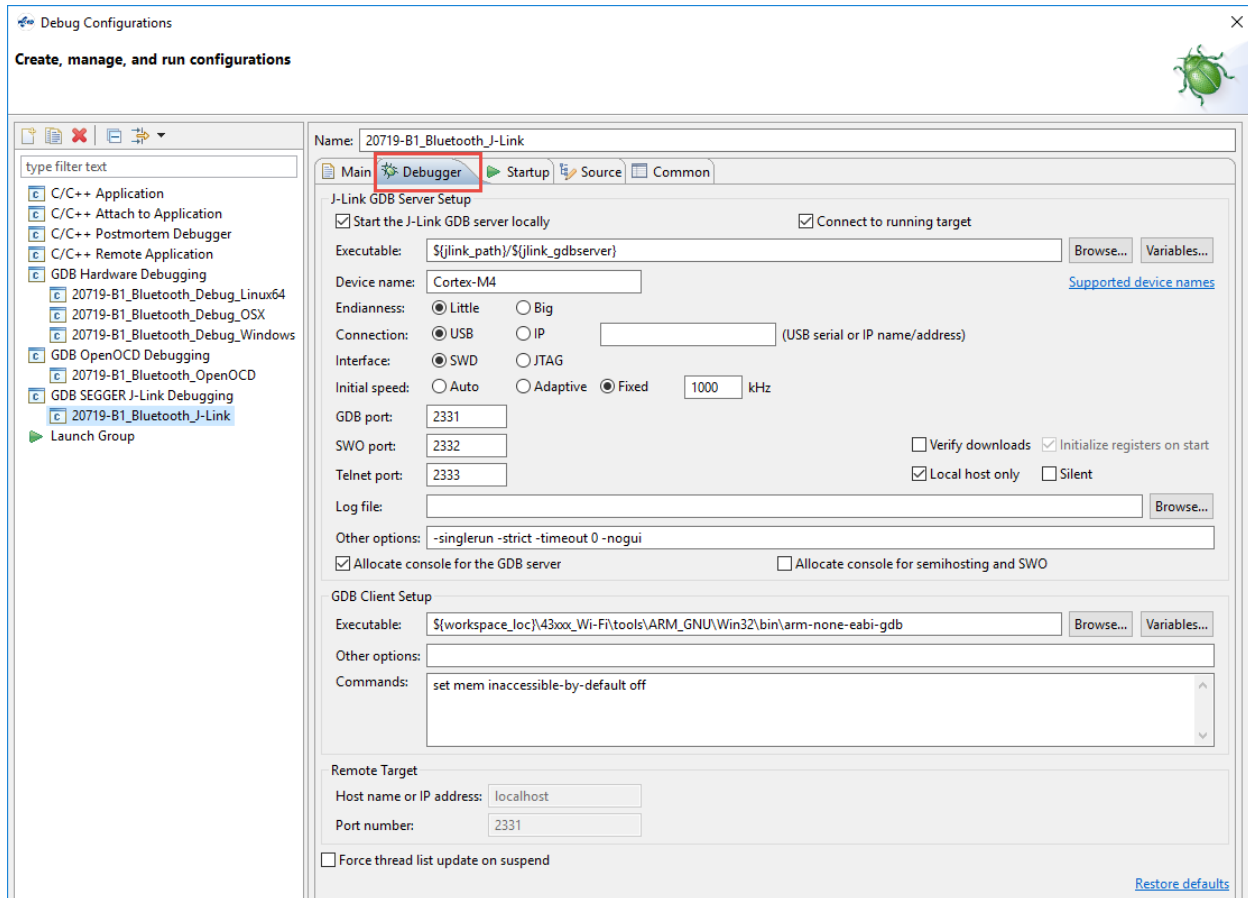


The *Project* and *C/C++ Application* string shown above are:

20719-B1_Bluetooth

\${workspace_loc}\20719-B1_Bluetooth\build\eclipse_debug\last_built.elf

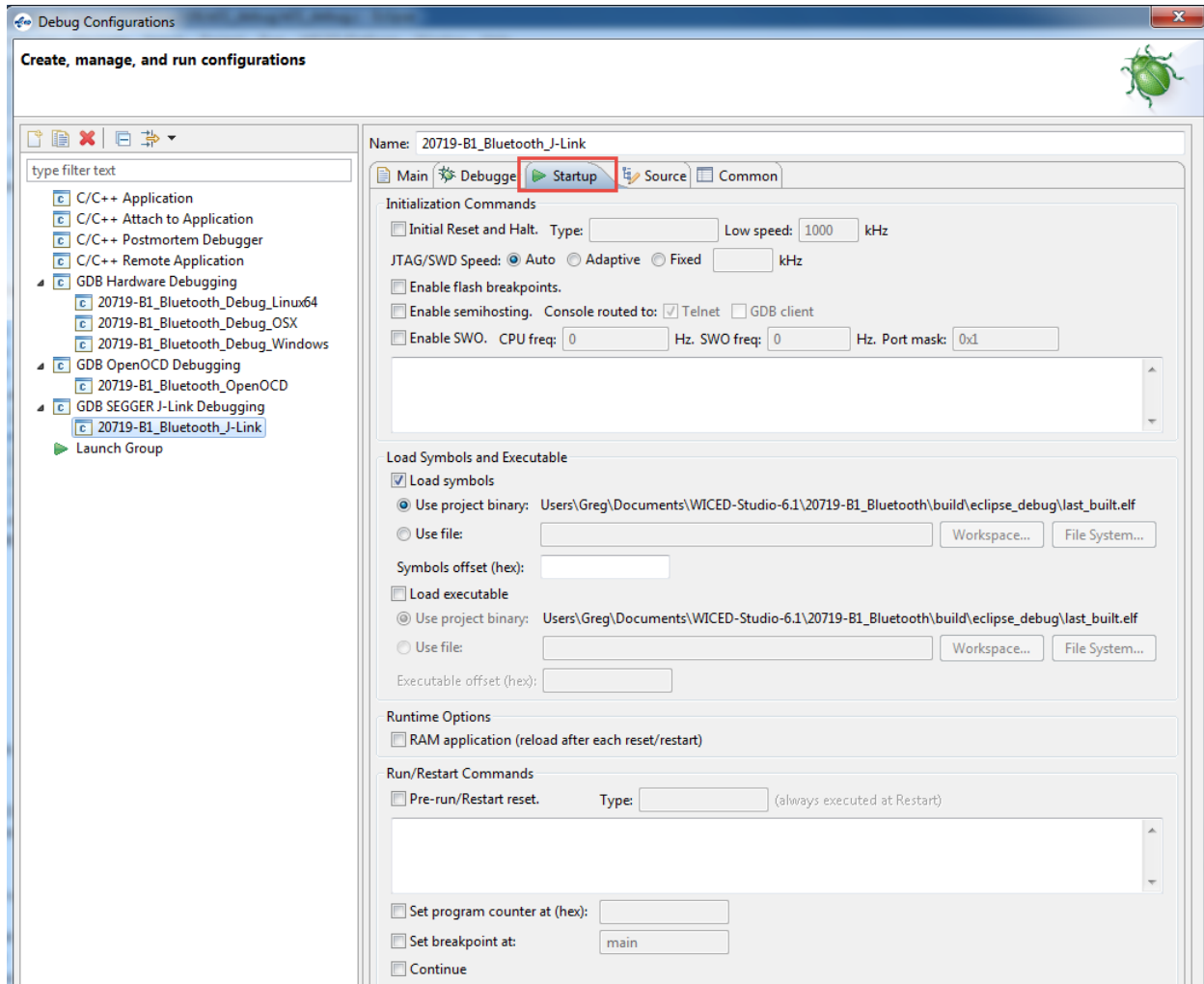
Click *Apply* on each tab as you complete it since it will affect what you see on the other tabs.



The *Device Name* and *GDB Client Setup Executable* string shown above are:

Cortex-M4

\$(workspace_loc)\43xxx_Wi-Fi\tools\ARM_GNU\Win32\bin\arm-none-eabi-gdb



Once you are done, reset the kit to make sure it is in the busy wait loop and click on "Debug". When the debugger has started running, go onto section 5.3.4 to learn how to use the debugger functions.

5.3.3 Host Debug Environment Setup for Olimex ARM-USB-TINY-H

Software and Hardware Setup for Olimex

To use the Olimex debugger, it is necessary to first download Open OCD and then install configuration files for your platform. A driver update from Zadig may be required if the drivers for the Olimex Debugger do not install properly on their own. The instructions for each of these steps is provided below along with a verification step.

Open OCD

We are using version 0.10.0.7. Other versions may not work with the script files we are using so it is recommended to use that version.

Version 0.10.0.7 for Windows 64-bit operating systems is included in the class files at:

Software_tools/Olimex/GNU_MCU_Eclipse

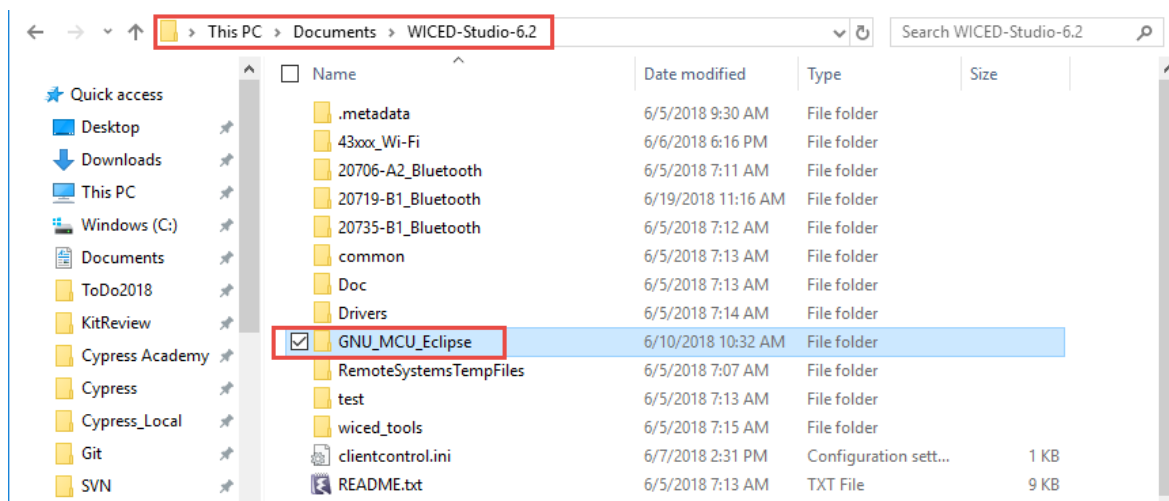
Note: In the files provided, the name *GNU MCU Eclipse* has been changed to *GNU_MCU_Eclipse* because spaces in the name can cause problems on some systems.

If you need a different platform, you can download it from the following location. Be sure to get version 0.10.0.7. Once it is downloaded replace the spaces in the folder name with underscores.

<https://github.com/gnuarmeclipse/openocd/releases>

Once you have the proper version, move the *GNU_MCU_Eclipse* folder to a location that you can find it. In these instructions, it is placed in the top WICED SDK folder which is what I would recommend.

An example of the resulting path can be seen here:



More detailed information and instructions on OpenOCD can be found at:

<https://gnuarmeclipse.github.io/openocd/install/>

Platform Specific Configuration Files

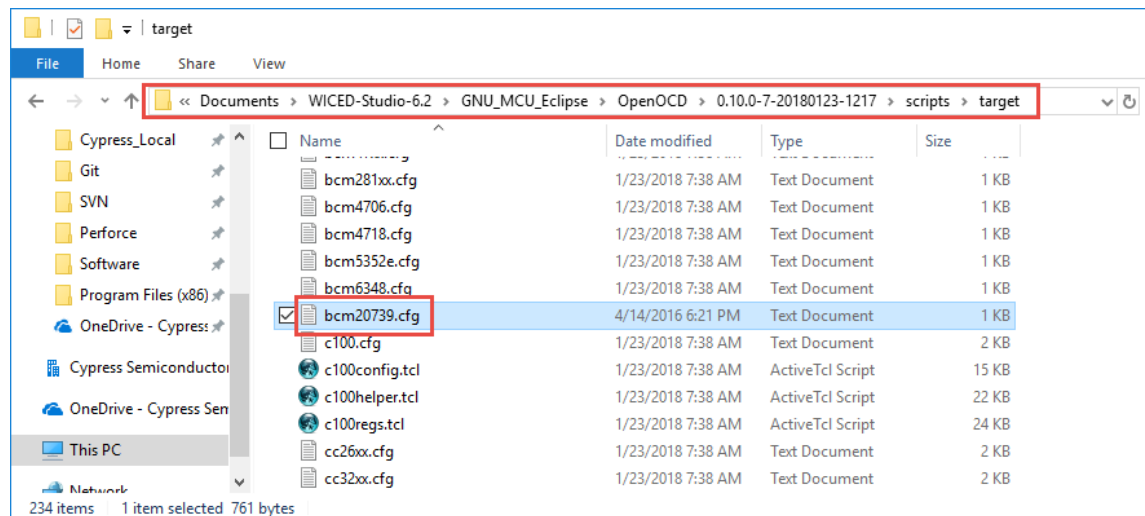
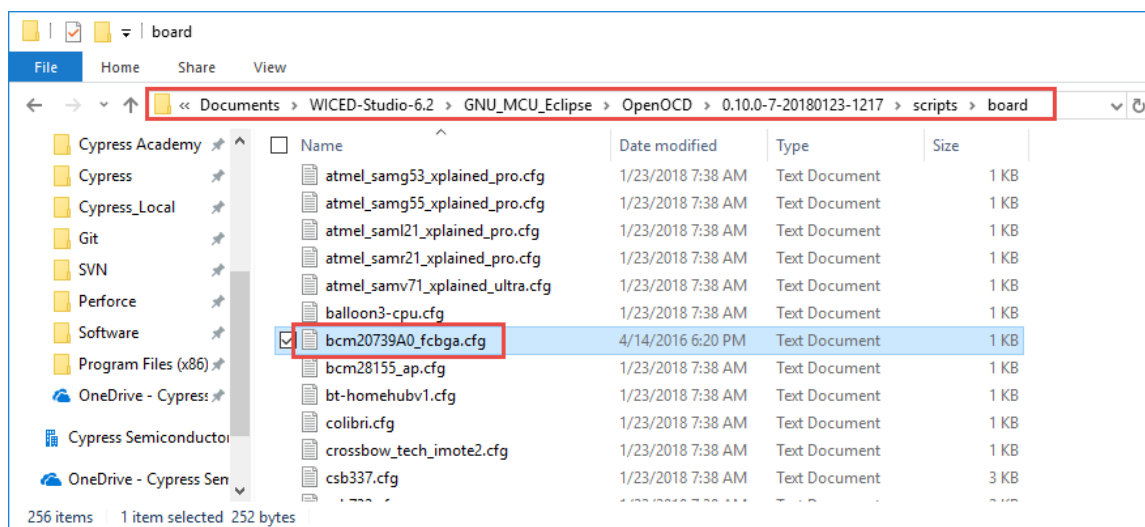
Inside the folder `GNU_MCU_Eclipse\OpenOCD<version>\scripts` there are folders called *board* and *target*. There is a board and target file required for the kit you are using. If you copied the GNU MCU Eclipse folder from the class files those two files are already included for you and you can skip this next step.

If you downloaded OpenOCD from the web, you must copy the board and target files into the appropriate folders. The required files can be found in the class files at:

Software_tools/Olimex/Debug_Config_Files/board/bcm20739A0_fcbga.cfg

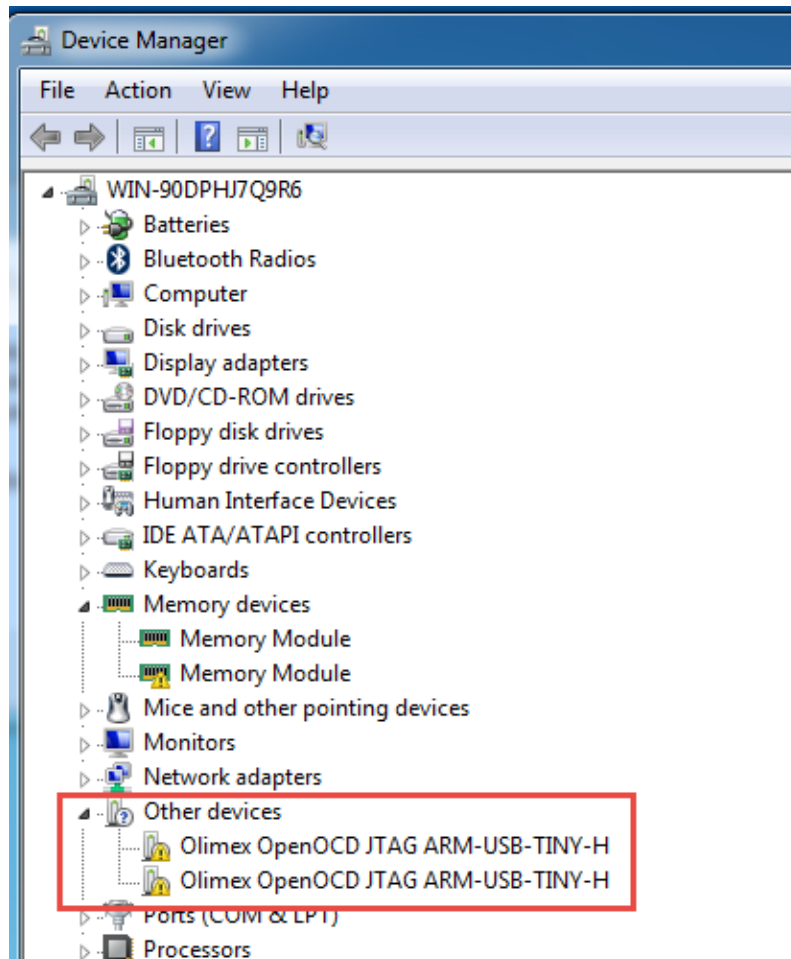
Software_tools/Olimex/Debug_Config_Files/target/bcm20739.cfg

Each file should be copied into the equivalent folder in the OpenOCD installation as shown here:



Olimex Driver Update

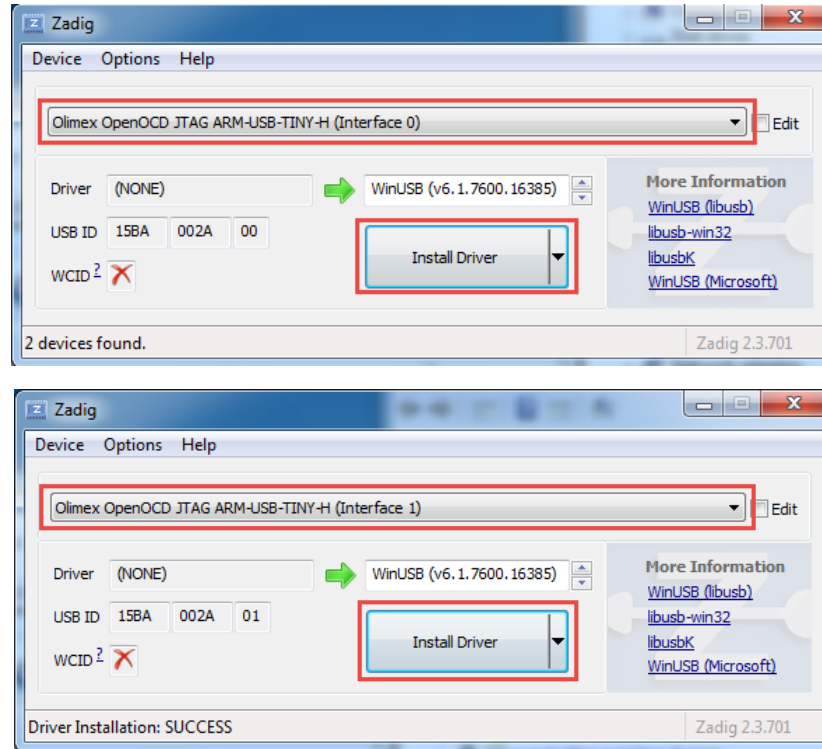
Connect the kit and Olimex Debugger to each other and to USB ports on your computer. Open the device manager and look for the Olimex Debugger. If it shows up under "Other devices" like in the following picture, then the driver will need to be updated manually:



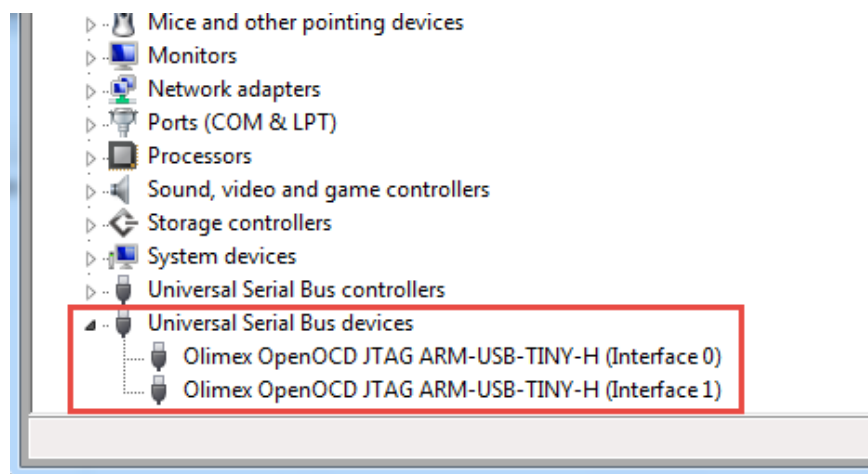
If the driver is not installed properly, run the tool zadig-2.3.exe. This can be found in the class files at:

Software_tools/Olimex/zadig-2.3.exe

Select each Olimex channel one at a time and click "Install Driver".



After installing the drivers, the Olimex Debugger should show up in the Device Manager under Universal Serial Bus devices like this:



To verify that the files are installed correctly, and the drivers are working, open a command window or power shell window in the location where you installed the OpenOCD executable. For example:

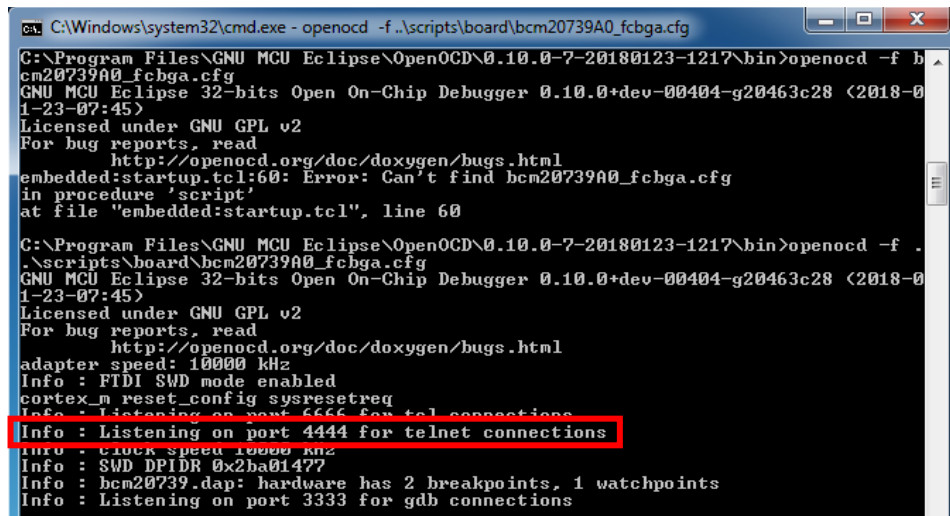
This PC\Documents\WICED-Studio-6.2\GNU_MCU_Eclipse\OpenOCD\<version>\bin

Note: To open a command window or power shell window, go to the folder in Windows explorer, hold Shift, right-click on an open area in the window, and then select "Open Command Window Here" or "Open Powershell Window Here".

From window, enter the following command:

`.\openocd -f ..\scripts\board\bcm20739A0_fcbga.cfg`

You should see a result like this:

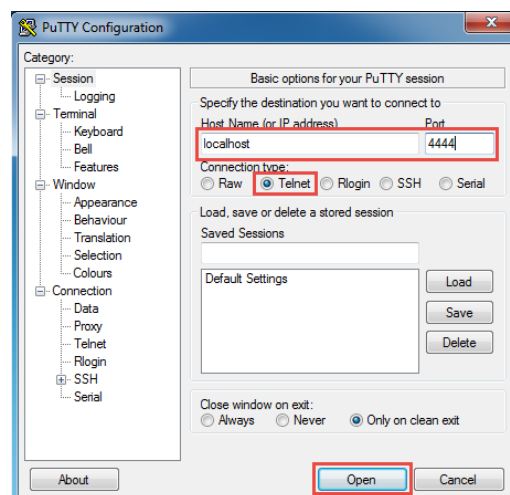


```

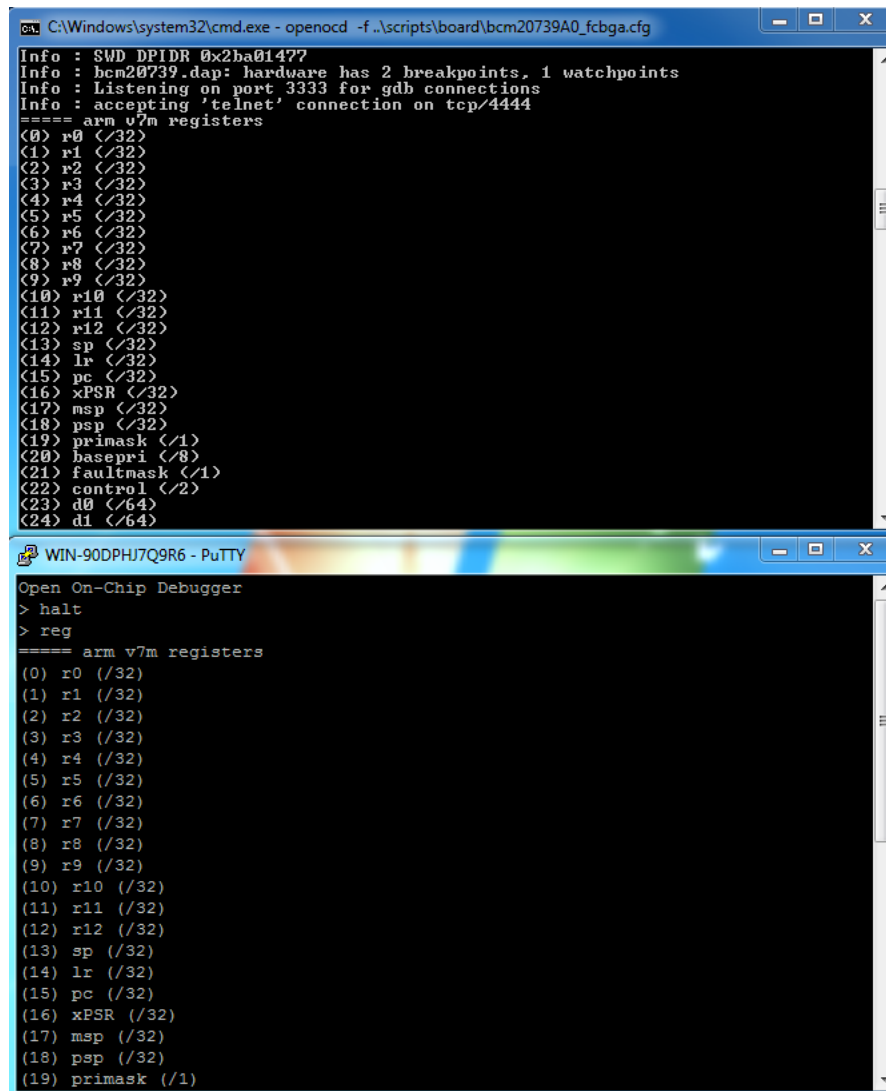
C:\Windows\system32\cmd.exe - openocd -f ..\scripts\board\bcm20739A0_fcbga.cfg
C:\Program Files\GNU MCU Eclipse\OpenOCD\0.10.0-7-20180123-1217\bin>openocd -f b
cm20739A0_fcbga.cfg
GNU MCU Eclipse 32-bits Open On-Chip Debugger 0.10.0+dev-00404-g20463c28 <2018-0
1-23-07:45>
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
embedded:startup.tcl:60: Error: Can't find bcm20739A0_fcbga.cfg
in procedure 'script'
at file "embedded:startup.tcl", line 60

C:\Program Files\GNU MCU Eclipse\OpenOCD\0.10.0-7-20180123-1217\bin>openocd -f .
.\scripts\board\bcm20739A0_fcbga.cfg
GNU MCU Eclipse 32-bits Open On-Chip Debugger 0.10.0+dev-00404-g20463c28 <2018-0
1-23-07:45>
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
adapter speed: 10000 kHz
Info : FTDI SWD mode enabled
cortex_m reset_config sysresetreq
Info : Listening on port 6666 for tel connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 10000 kHz
Info : SWD DPIDR 0x2ba01477
Info : bcm20739.dap: hardware has 2 breakpoints, 1 watchpoints
Info : Listening on port 3333 for gdb connections
  
```

Keep that window open and then open a Telnet connection to localhost port 4444. For example, Putty can be used for Tenet as shown here:



Once the Telnet window opens, enter "halt" to halt the CPU and then enter "reg" to see the register values. You will see a response both in the Telnet window and the OpenOCD window.



The image shows two overlapping terminal windows. The top window is a command prompt running OpenOCD, and the bottom window is a PuTTY window running the Open On-Chip Debugger.

Top Window (OpenOCD):

```
C:\Windows\system32\cmd.exe - openocd -f .\scripts\board\bcm20739A0_fcbga.cfg
Info : SWD DPIDR 0x2ba01477
Info : bcm20739.dap: hardware has 2 breakpoints, 1 watchpoints
Info : Listening on port 3333 for gdb connections
Info : accepting 'telnet' connection on tcp/4444
===== arm v7m registers
<0> r0 (/32)
<1> r1 (/32)
<2> r2 (/32)
<3> r3 (/32)
<4> r4 (/32)
<5> r5 (/32)
<6> r6 (/32)
<7> r7 (/32)
<8> r8 (/32)
<9> r9 (/32)
<10> r10 (/32)
<11> r11 (/32)
<12> r12 (/32)
<13> sp (/32)
<14> lr (/32)
<15> pc (/32)
<16> xPSR (/32)
<17> msp (/32)
<18> psp (/32)
<19> primask (/1)
<20> basepri (/8)
<21> faultmask (/1)
<22> control (/2)
<23> d0 (/64)
<24> d1 (/64)
```

Bottom Window (Open On-Chip Debugger):

```
Open On-Chip Debugger
> halt
> reg
===== arm v7m registers
(0) r0 (/32)
(1) r1 (/32)
(2) r2 (/32)
(3) r3 (/32)
(4) r4 (/32)
(5) r5 (/32)
(6) r6 (/32)
(7) r7 (/32)
(8) r8 (/32)
(9) r9 (/32)
(10) r10 (/32)
(11) r11 (/32)
(12) r12 (/32)
(13) sp (/32)
(14) lr (/32)
(15) pc (/32)
(16) xPSR (/32)
(17) msp (/32)
(18) psp (/32)
(19) primask (/1)
```

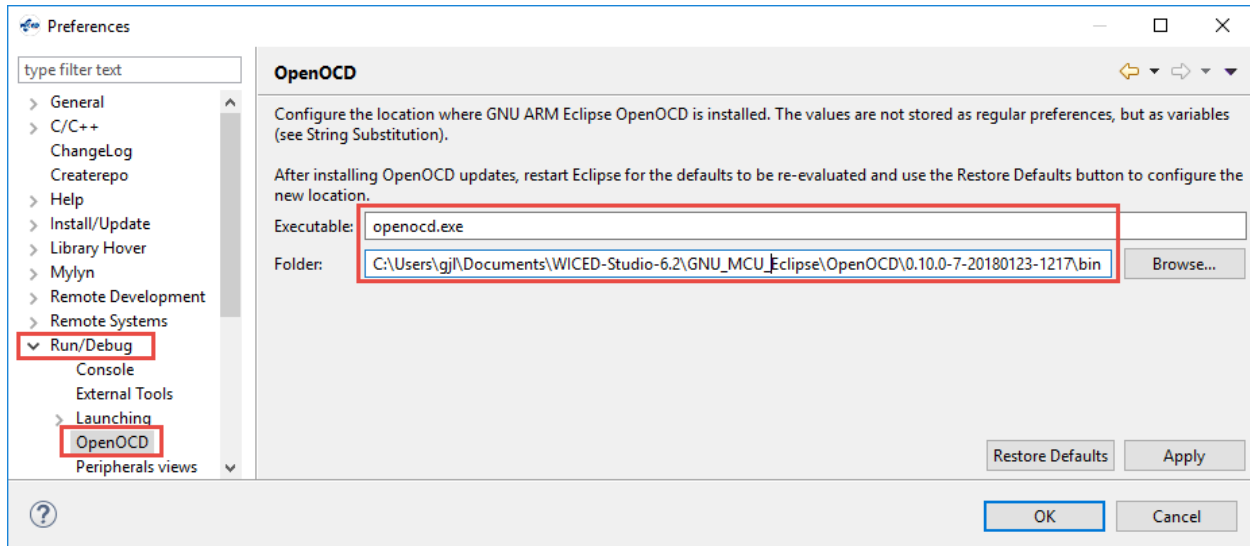
Enter the command "exit" to quit out of Telnet and hit Ctrl-C to stop the OpenOCD GDB Server. Both windows should be closed at this point.

WICED Studio Setup for Olimex

To use the Olimex Debugger inside WICED Studio it is necessary to specify the path to OpenOCD and to set up a Debug Configuration. Each is discussed separately below.

OpenOCD Path

In WICED Studio, go to *Window -> Preferences* and then select *Run/Debug -> OpenOCD*. Enter the executable and folder as shown below. The folder path must match the location where you put the OpenOCD installation.

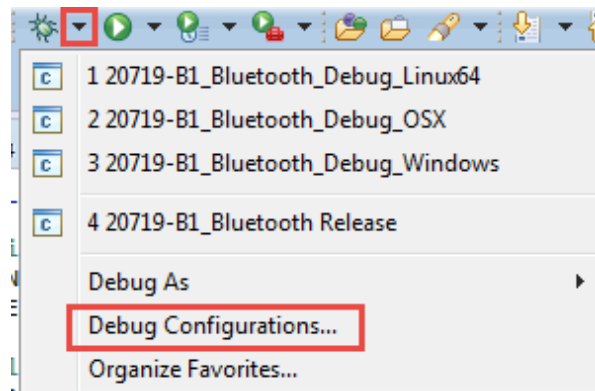


The *Folder* shown above is:

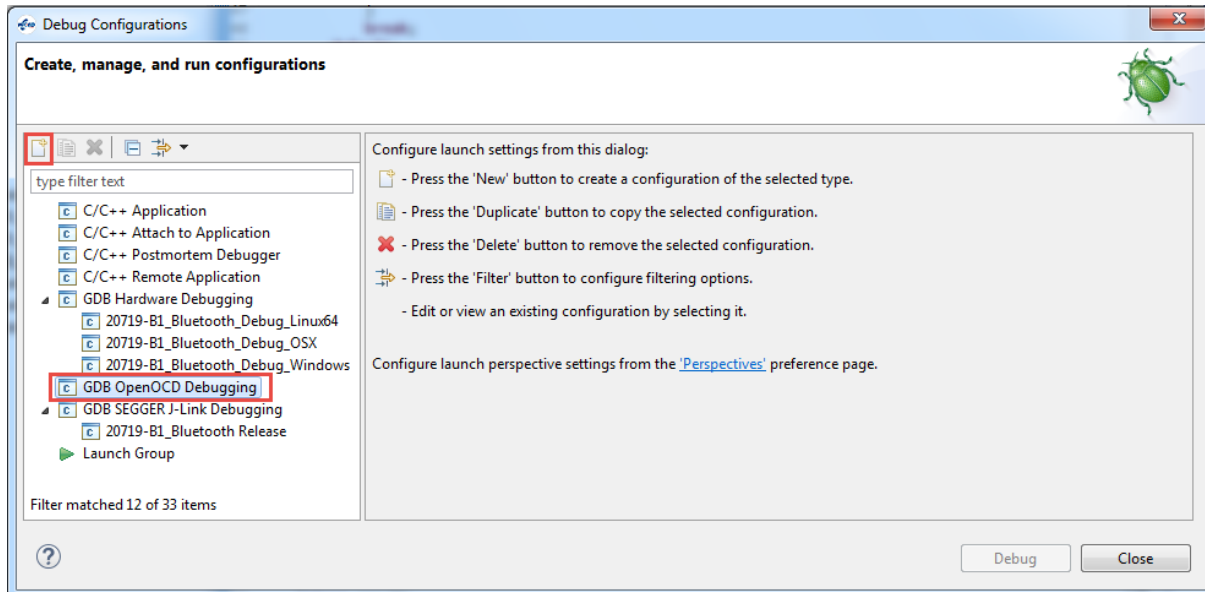
`C:\Users\gjl\Documents\WICED-Studio-6.2\GNU_MCU_Eclipse\OpenOCD\0.10.0-7-20180123-1217\bin` but it will be different in your case.

Debug Configuration

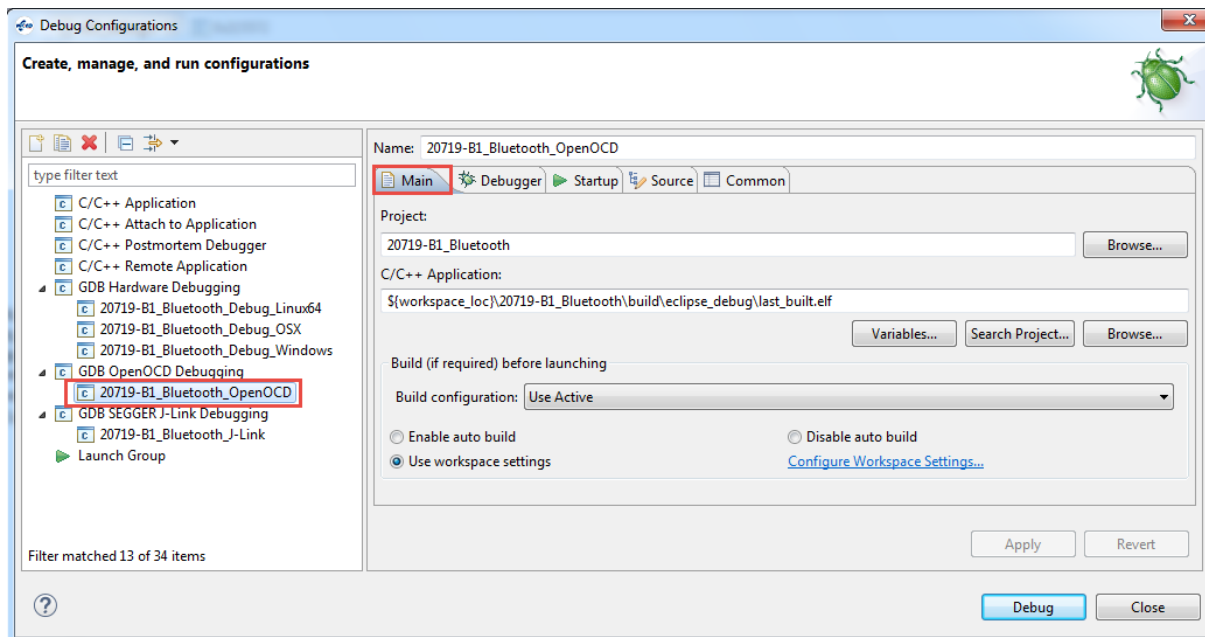
From WICED Studio, click the arrow next to the Bug icon and select "Debug Configurations...".



Select *GDB OpenOCD Debugging* and click the "New" button.



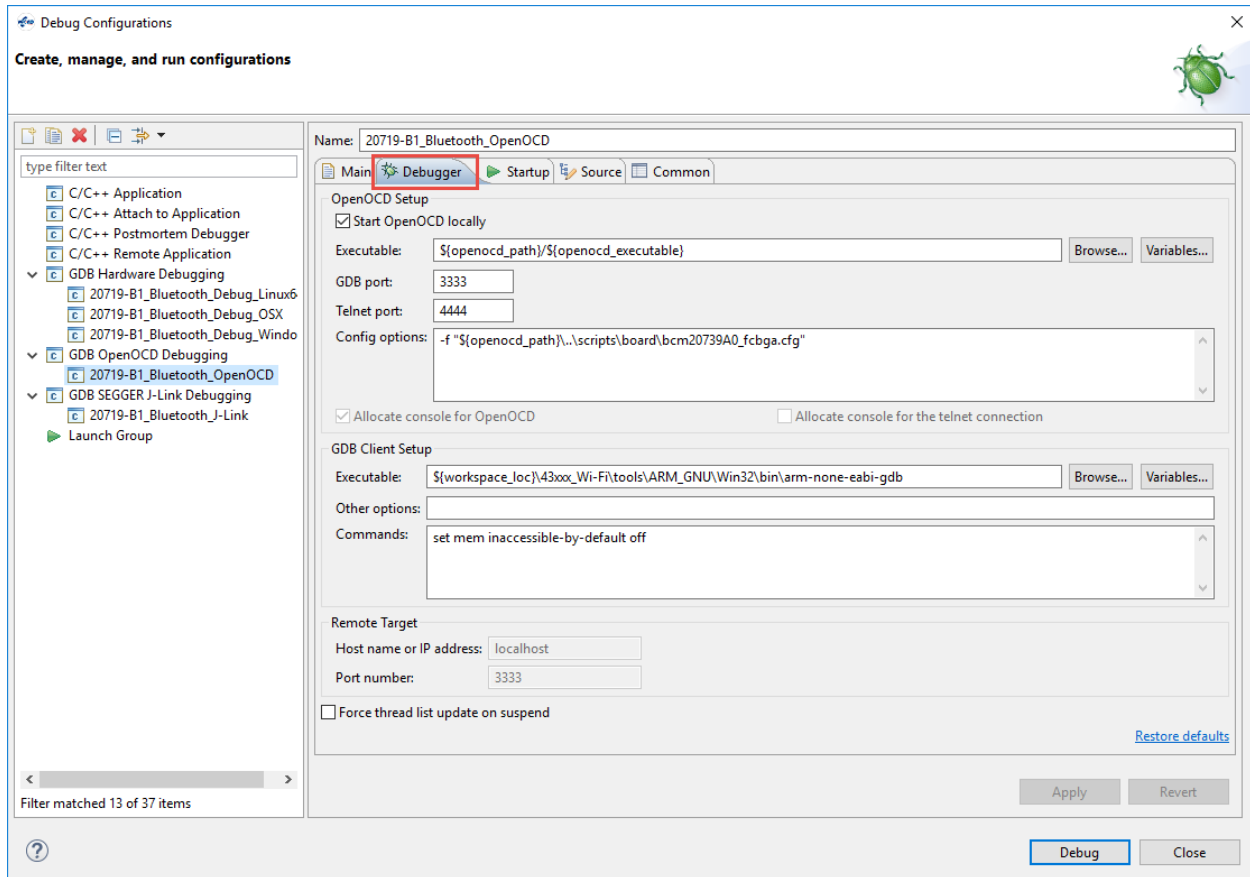
Change the name of the configuration if desired (the pictures below use the name "20719-B1_Bluetooth_OpenOCD") and set the options on each tab as shown in the following pictures.



The *Project* and *C/C++ Application* string shown above are:

20719-B1_Bluetooth
\$(workspace_loc)\20719-B1_Bluetooth\build\eclipse_debug\last_built.elf

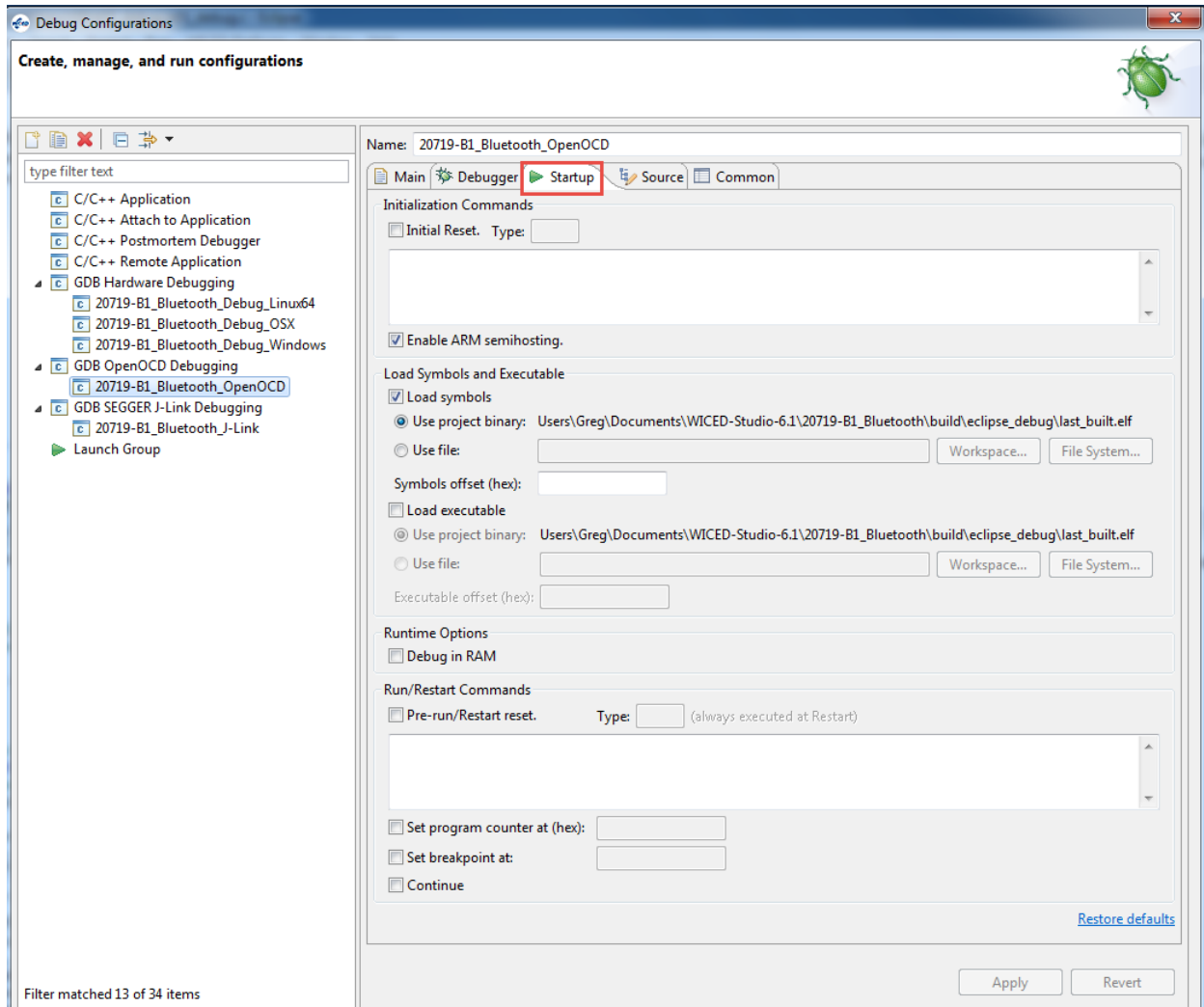
Click *Apply* on each tab as you complete it since it will affect what you see on the other tabs.



The *Config options* and *GDB Client Setup Executable* string shown above are:

`-f "${openocd_path}/../scripts/board/bcm20739A0_fcbga.cfg"`

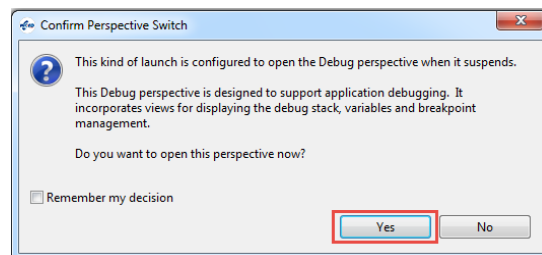
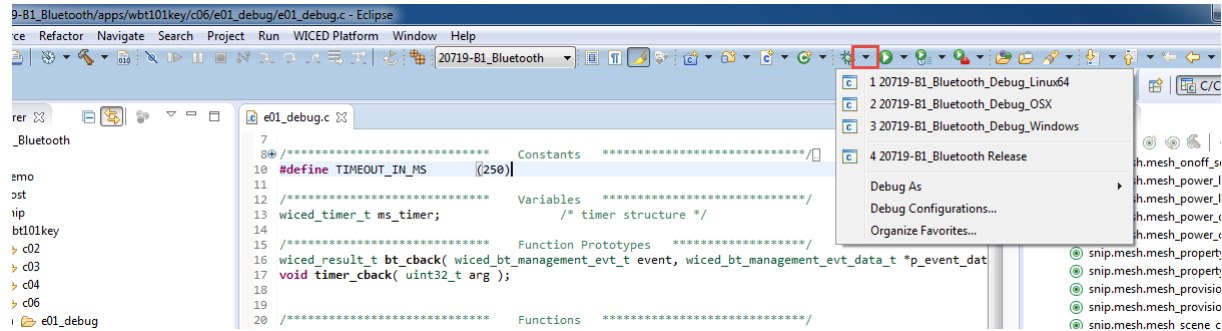
`${workspace_loc}/43xxx_Wi-Fi/tools/ARM_GNU/Win32/bin/arm-none-eabi-gdb`



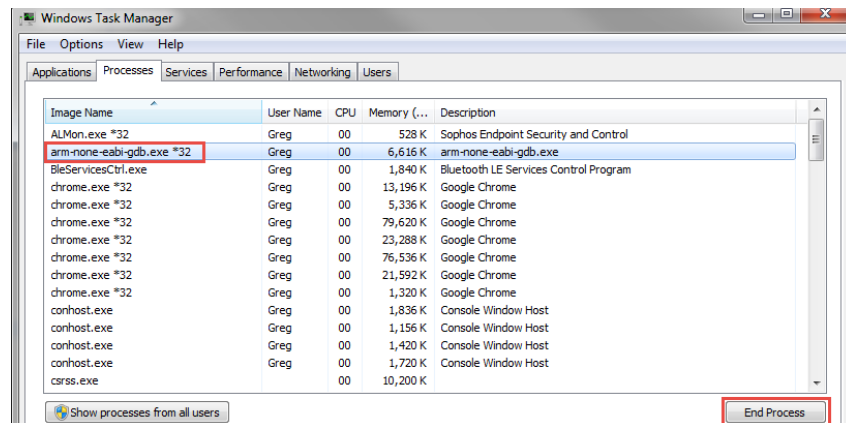
Once you are done, reset the kit to make sure it is in the busy wait loop and click on "Debug". When the debugger has started running, go onto section 5.3.4 to learn how to use the debugger functions.

5.3.4 Using the Debugger

In prior sections you have setup a debugger configuration for your hardware setup. If you have not started the debugger yet, click the arrow next to the Bug icon and select the appropriate configuration. If you get a message asking if you want to open the debug perspective, click "Yes". You can click the check box to tell the tool to switch automatically in the future.



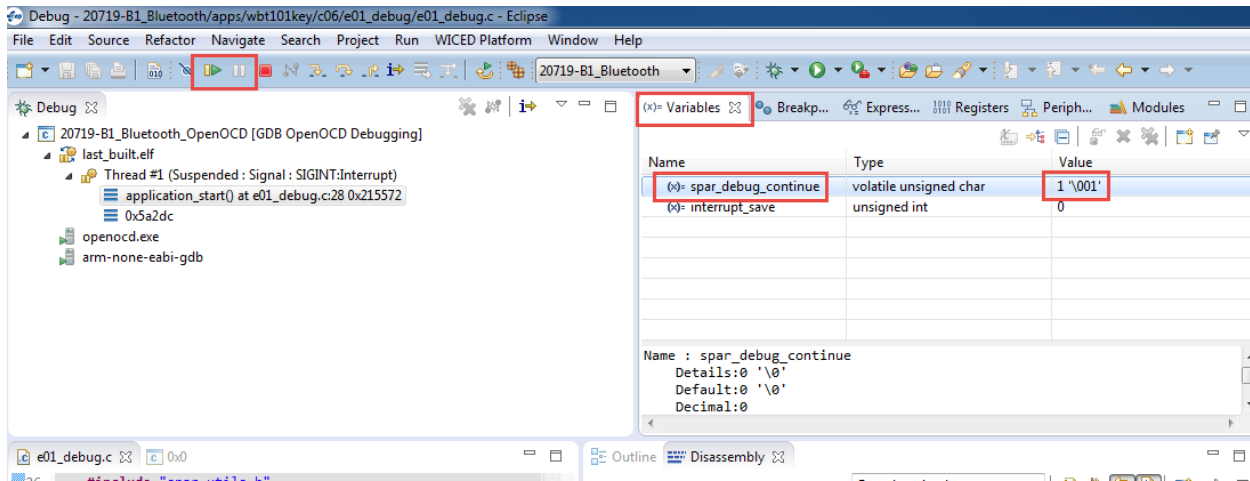
If you get an error when trying to launch the debugger you may need to terminate an existing debug process. Open the Windows Task Manager, select the Process tab, click on "Image Name" to sort by the process name and terminate all "arm-none-eabi-gdb" processes.



When the debugger starts, you will be in the "Debug Perspective".

If you included the macro `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED` in your firmware, the project will sit in that loop. Once the debugger starts, start execution and then pause to make sure

the firmware is inside the loop. In the "Variables" window, highlight the entire line for the value of `spar_debug_contine`, enter a 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.



If you want the project to continue prior to the debugger starting, you can remove the `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLE` macro line, but you will enter debugging at an unknown point.

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 (such as `02_blinkled.c`), click on the line where you want a breakpoint and press `Ctrl-Shift-B` or from the menu select `Run > Toggle Breakpoint`. If you need to see the project explorer window to open the source file, click on `"C/C++"` in the upper right corner to switch to the `C/C++` Perspective. Once you have opened the file, switch back to the `Debug` Perspective.

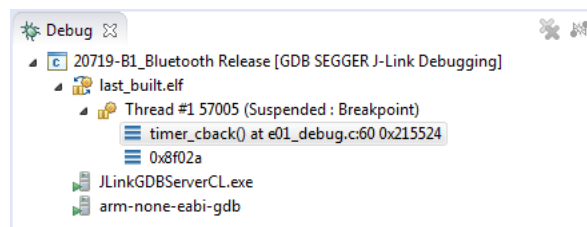
Click the "Resume" button (shown in the figure above) to resume execution. The program will halt once it reaches the breakpoint.

```

55
56 /* The function invoked on timeout of the timer. */
57 void timer_cback( uint32_t arg )
58 {
59     /* Read current set value for the LED pin and invert it */
60     wiced_hal_gpio_set_pin_output( WICED_GPIO_PIN_LED_2, !wiced_hal_gpio_g
61 }
62
63

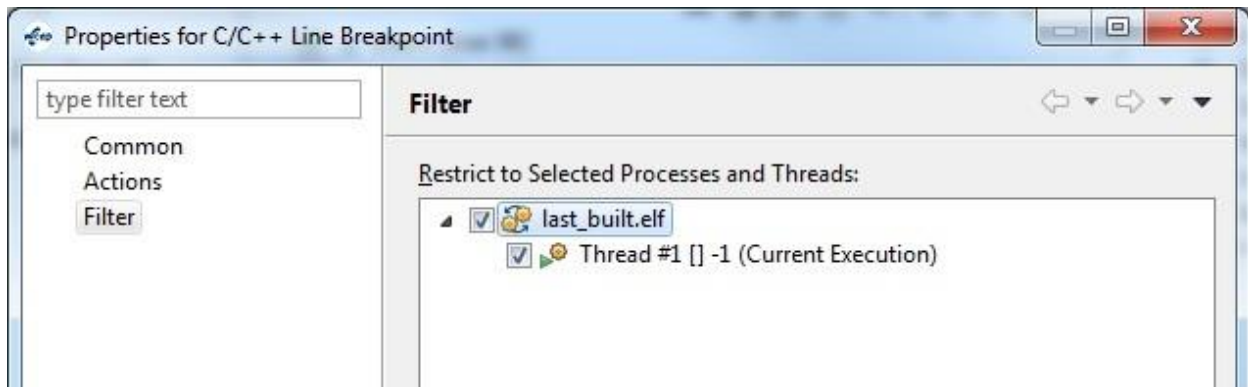
```

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.

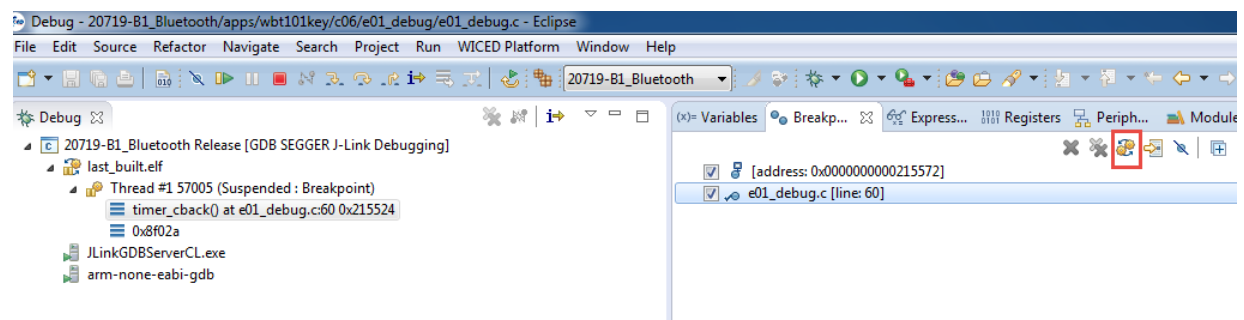


You can enable or disable breakpoints by double clicking on the green circle next to the line in the source code or from the "Breakpoints" window. If you don't see the Breakpoints tab, use the menu item "Window > Show View > Breakpoints".

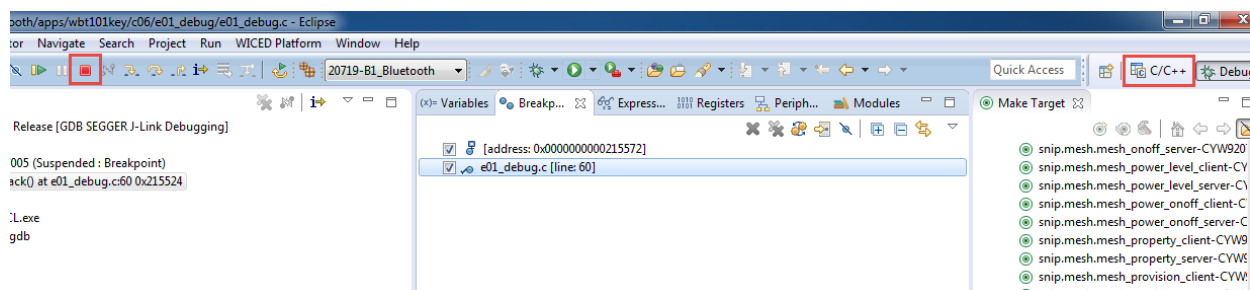
If breakpoints are created prior to starting the current debug session, they will not be associated with the current thread and will be indicated with a blue circle without a check mark. To enable the breakpoints in the current thread, right-click the desired breakpoint and select "Breakpoint Properties..." Click on "Filter" and then select the "last_built.elf" check box 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 you terminate the debugger, you will want to switch back to the C/C++ Perspective by clicking on the button at the top right corner.





A few 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 the breakpoint is not available.

5.4 Exercises

Exercise - 5.1 Use the Client Control Utility

Introduction

In this project, you will add the ability to start and stop advertising using WICED HCI messages from the Client Control utility.

Project Creation

1. Copy the folder from the class files at WBT101_Files/Templates/ch05/ex01_wicedhci into the ch05 folder for your workspace.
 - a. Hint: The template is just the solution from exercise ch04a/ex03_ble_adv so if you prefer, you can instead copy your answer to that exercise and rename things as necessary.
 - b. Hint: Change the name from *key_hci* to *use your initials instead of "key"* in the *wiced_bt_cfg.c* file.
2. Create a Make Target.
3. Verify that WICED HCI is enabled in your application.
 - a. Hint: review the steps in the Transport Configuration section of this chapter.
4. Remove *wiced_bt_start_advertisements* from the *application_init* function.
5. Modify the RX handler to handle the *_ADVERTISE Command.
 - a. Hint: Refer to the file *include/common/hci_control_api.h* to find the full name of the Command.
 - b. Hint: The switch statement by default uses the Group Code. You may want to change it in this case to use the full Opcode instead.
 - c. Hint: The payload will be 0 to stop advertisements and 1 to start advertisements.
 - d. Hint: Fix the two bugs in the function.
6. Update the Bluetooth Stack Management callback to send WICED HCI status messages when the advertising state changes.
 - a. Hint: Use the *wiced_transport_send_data* function and the same ADVERTISE Command 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. If you send a value other than 0 or 1, the Client Control utility will crash.

Testing

1. Program the project 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 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.



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 - 5.2 Run BTSPy

In this project you will use BTSPy to look at Bluetooth protocol trace messages.

Project Creation

1. Copy the folder from the class files at WBT101_Files/Templates/ch05/ex02_btspy into the ch05 folder for your workspace.
 - a. Hint: The template is just the solution from exercise ch04b/ex03_ble_pair so if you prefer, you can instead copy your answer to that exercise and rename things as necessary.
 - b. Hint: Change the name from *key_spy* to *use your initials instead of "key"* in the *wiced_bt_cfg.c* file and in *ex02_btspy.c*.
2. In the makefile, verify that the HCI_TRACE_OVER_TRANSPORT C flag is set as discussed earlier.
3. In the main C file, verify the BTSPy setup as discussed earlier. Be sure to make sure the debug interface is set to WICED_ROUTE_DEBUG_TO_WICED_UART.
4. In the *wiced_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 *apps\host\client_control\Windows* by double clicking on it.
4. 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).
5. Run the *BTSPy* utility from *wiced_tools\BTSPy\Win32* by double clicking on it.
 - 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.
6. 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 pairing 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 - 5.3 (Advanced) Run the Debugger

In this exercise you will setup and run the debugger using an Olimex ARM-USB-TINY-H. You will then use the debugger to change the value of a variable that controls an LED on the shield.

1. Copy the project ex03_debug from the templates folder.
2. Run the SuperMux pin configuration tool and setup the SWD pins.
3. Edit the make file to include debugging options.
4. Edit the C file to allow debugging.
5. Create a Make Target for the project which includes the DEBUG flag.
6. Follow the procedure to setup for Olimex debugging.
7. Program the board and start the debugger.
8. Place a break point at the delay.
9. Execute up to the break point a few times. Notice that the LED does not turn on because the variable "led" never changes.
 - a. Hint: Remember to set the value of *spar_debug_contine* to 1 to get out of the initial busy wait loop.
10. Change the value of the variable "led" and then re-start execution.
11. Note that the LED now turns on.